FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Framework for Side Channel Analysis on Flash Memory

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software und Information Engineering

eingereicht von

Markus Hannes Fischer
Matrikelnummer 1029057

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Projektass. Dipl.-Ing. Markus Kammerstetter BSc.

Wien, 25. Februar 2015

Markus Hannes Fischer

Markus Kammerstetter

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Framework for Side Channel Analysis on Flash Memory

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Markus Hannes Fischer
Registration Number 1029057

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Projektass. Dipl.-Ing. Markus Kammerstetter BSc.

Vienna, 25th February, 2015

Markus Hannes Fischer	Markus Kammerstetter

Erklärung zur Verfassung der Arbeit

Markus Hannes Fischer
Hugo-Meisl-Weg 11/1, 1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 25. Februar 2015

Markus Hannes Fischer

Kurzfassung

Flash-Speicher finden sich heute in nahezu jedem elektronischen Gerät. Besonders in Mobilgeräten sind sie aufgrund ihres niedrigen Energieverbrauchs und hoher Leistung sehr verbreitet, z.B. Speicher in Mobiltelefonen oder Solid State Disks (SSD) in Laptops.

Die grundsätzliche Funktionsweise von Flash-Speichern ist allgemein bekannt. Im Speziellen jedoch versuchen sich die Hersteller von der Konkurrenz abzuheben, indem sie eigene Algorithmen in den Speicher-Controllern (SPC) implementieren, die den Speicher verwalten und zusätzliche Features bieten. Die Implementierungsdetails sind nicht öffentlich und können daher nicht überprüft werden.

Um die Funktionsweise von SPCs durch einen Black-Box-Ansatz zu rekonstruieren und ganz allgemein herausfinden zu können, welche Daten in einem Flash-Speicher verarbeitet werden, ohne dabei direkten Datenzugriff zu haben, sind Seitenkanal-Attacken eine gute Lösung.

Um eine solche Attacke durchzuführen, benötigt man einen Mess-Aufbau bestehend aus Treiber-, Mess-, Erfassungs- und Analyse-Komponente.

Diese Arbeit befasst sich mit den benötigten Eigenschaften der jeweiligen Komponenten und präsentiert eine Implementation, die als Ausgangspunkt für komplexere Daten-Analysen verschiedener Flash-Speicher dienen soll.

Die Ergebnisse zeigen, dass man mit diesem Aufbau unter Verwendung von entsprechendem Labor-Equipment verlässlich Daten über die Leistungsaufnahme des Flash-Speicher-Chips erheben kann. Außerdem weisen die erhaltenen Daten darauf hin, dass es eine ausreichende Korrelation zwischen der Leistungsaufnahme eines Flash-Speichers und den verarbeiteten Daten gibt.

Abstract

Flash memory is ubiquitous in today's electronics. In particular, mobile devices rely on them because of their low energy consumption and high performance, e.g. storage in cellular phones and Solid State Disks (SSDs) in laptops.

The functional concepts are well researched and publicly known. However, vendors try to be ahead of the competition by implementing custom algorithms in the Memory Controllers (MCs) that manage the memory and provide additional features. The implementation details of the MCs are not public and can therefore not be verified.

To reverse engineer a MC through a non-invasive black box approach and, more generally, to be able to retrieve data being processed on a Flash Memory Chip (FMC) without direct data access, Side Channel Attacks (SCAs) are a good approach.

To mount these attacks, a setup consisting of driver, measurement, acquisition and analysis components is needed.

This work provides an analysis of the required properties of these components and presents an implementation that can be used as a base for advanced power analysis attacks and further research into different kinds of flash memory.

The results show that power consumption data can be retrieved reliably using this framework based on relatively simple lab equipment. Furthermore, the acquired data suggests a sufficient correlation between a flash memory chip's power consumption and the data being processed.

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Motivation

Flash memory is ubiquitous in today's electronics. Computers use Solid State Disks (SSDs) to store non volatile data and speed up Input/Output (I/O) operations. USB flash drives are used on a daily basis to quickly move files between computers or even as security tokens. Mobile devices use Flash Memory Chips (FMCs) to access data despite vibrations caused by movement and reduce power consumption. Micro Controllers (μCs) come with on-board FMCs to store program code and data in case of a power loss.

Therefore, research into FMCs gives a broad range of applications not only in an academic field, but also in every day usage.

1.2 Problem Statement

The basic principles on which FMCs are built on are common knowledge. Books like *Inside NAND Flash Memories*[1] and *Flash Memories*[2] give deep insight into the functional concepts and developments of flash memory technology.

However, a modern FMC does not only consist of the actual memory, but also of the Memory Controller (MC) that sits between the storage and the device (flash translation layer) performing I/O. This MC is a key component of the product and implements complex features that result in a better usability and reliability of the FMC. Features implemented, amongst others, may include data encryption, compression or performance and durability increasing algorithms. [1]

One of the most important features of the MCs is wear leveling. Flash cells only support a finite amount of write/erase cycles because of the underlying physical constraints. This loss of durability after a certain number of write and erase operations is called wear and causes data to not be stored and retrieved reliably anymore. Systems tend to access certain memory regions more frequently than others increasing the wear on them. [1]

For instance, consider Operating System (OS) and document files. OS files will be altered much more frequently than e.g. a text file containing a CV.

Manufacturers invest significant research into perfecting their MC algorithms and keeping their implementations secret in order to succeed in competition with other vendors.

However, this closed source approach has obvious drawbacks:

- In case of a hardware failure, the MC might not be working anymore but the data would still be stored in the memory cells. Even if the raw data can be recovered, the storage layout is unknown and the actual data cannot be recovered.
- Implementation flaws cannot be found by the public. It is more likely that bugs go unnoticed if their effects do not occur frequently, as the source code cannot be checked by third parties investigating the undesired effects.
- Innovation is restricted because users and third party developers cannot customize and adapt the product according to their needs.

1.3 Aim of the Work

From a security point of view forensic analysis and identifying implementation flaws are of great interest.

To allow a general approach not tied to a specific vendor or flash type, a black box approach is used. This means that the output of the MC for a known input is analyzed in order to discover how the target processes the input. In case of the MC being the target, the output is the data written to the memory cells.

However, the memory cells are not directly accessible in a non-invasive manner. Therefore, we must find a way to bypass the MC and read the cell data directly.

One possible approach is shown by Samyde, Skorobogatov, Anderson, *et al.* in „On a New Way to Read Data from Memory“[3]. The data extraction is performed by depackaging the chip and using laser pulses to extract the stored information from the cells. [3] This bypass of the MC may be thwarted by utilizing protective packaging and tamper detection sensors.

Another approach is analyzing information that is leaked by the memory while it is operating. This leakage is referred to as a Side Channel (SC). Possible SCs include execution time, power consumption, electromagnetic radiation, noise and temperature.

This bachelor thesis focuses on analyzing the power consumption of FMCs to infer the data and the operation that was performed on the FMCs.

If this information can be retrieved reliably through the power analysis, it will be the base for reverse engineering MCs and their proprietary implementations.

The goal of the thesis is to design and implement a framework to perform side channel measurements on FMCs that is flexible and easy to adapt to different types of flash chips and various analysis methods.

CHAPTER 2

Technical Background

2.1 Side Channels

2.1.1 General Idea

From a top-down view a machine usually reads some input, performs some operations on it and then returns the output. This is true for a desktop computer reacting to keyboard input by displaying different programs, a micro controller sampling an analog signal and outputting a digital bit stream or a memory storing data. The storage does not deliver a different output immediately, but when data is being read again.

Input and output are called main channels. They are where I/O operations happen and how the user interacts with the machine.

However, if we take a closer look at the machine we notice that there are other channels too, providing information on the machine's state. For example, it is possible to see if a computer is in stand-by or performing heavy duty operations by checking its power consumption. Another side channel can be the heat produced by the PC or the Electromagnetic Radiation (EMR) being emitted. Response times may also leak information on the kind of processing which is being performed. A basic password check that returns immediately once the first character of the input does not match the password can be easily defeated by trying every character until the response time increases a little (correct match, the machine checks the next byte). This reduces the number of needed attempts in order to guess the correct password considerably.

The focus of this thesis is to build a framework to analyze the time needed and the power consumed by the device, in this case FMCs, for different operations.

As a result, this work focuses on power analysis.

2.1.2 Simple Power Analysis

Simple Power Analysis (SPA) is the most basic way of analyzing the power SC. The power consumed by the device is measured over time. Using this information it is often possible

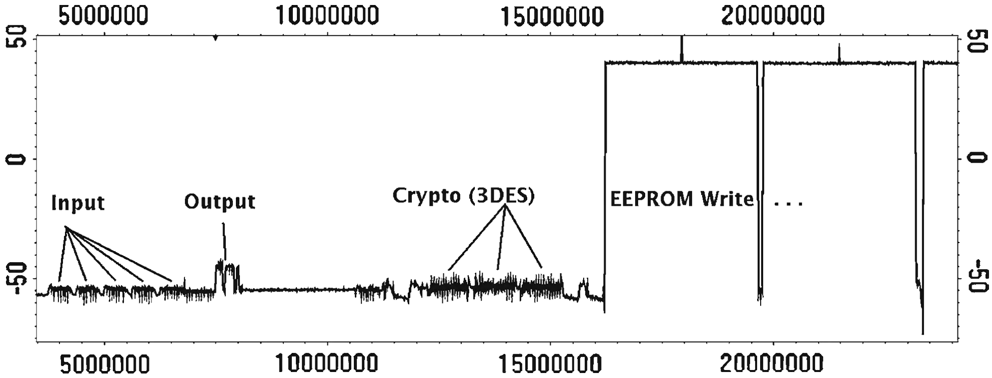

Figure 2.1: Power trace of a smart card running a pseudo-random number generation operation using 3DES and an EEPROM [4].

to distinguish between different operations inside the device. For instance, different execution paths can be found by analyzing the time required for the next step to occur. Once a difference has been found, the correlation between input and different execution paths may reveal sensitive information on the inner workings of the device. It may also be possible to reconstruct the data processed, although being significantly harder due to the noise in the power traces. [4]

2.1.3 Differential Power Analysis

The idea behind Differential Power Analysis (DPA) is to take two power traces which were generated with a specific difference in input e.g. the 5th bit of the input was set to 0 in the first measurement and it was set to 1 in the second. Then the difference between the two traces is calculated. If there is a correlation between the power consumption and the value of the 5th input bit, it will show in the difference as a spike. Since all other parameters are unchanged the expectation is that the power traces are the same everywhere else. [4]

To rule out other variables such as e.g. a timer randomly triggering another execution path, it is best to generate a big set of data with random or at least varying input and then use a selector function to pick the traces which are different in the analyzed variable e.g. 5th bit of the input. The measurements selected this way are then averaged and are subtracted from another. [4]
An example can be seen in Figure 2.2.

To gather a representative amount of power traces that also exhausts all present variables, DPA requires a very high number of measurements. If the amount of traces is limited due to time or other resources, Correlation Power Analysis (CPA) can be used to maximize the output. In comparison to a standard DPA, CPA compares the available traces not to each other based on a selection function, but against a leakage model that approximates power consumption for the input and performed operations. Generally,

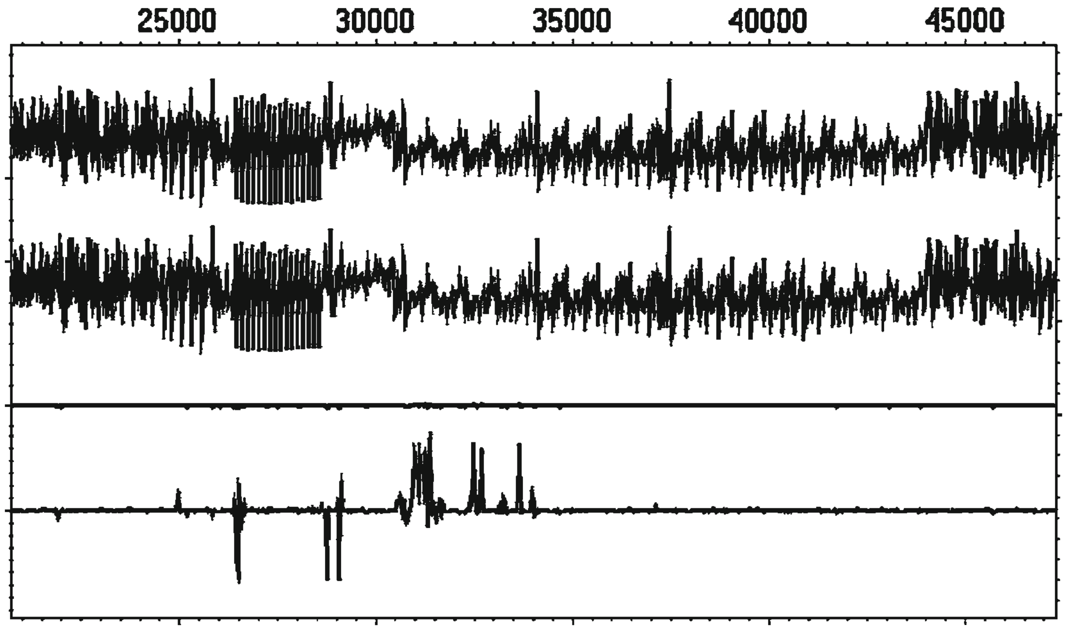

Figure 2.2: The top trace is the averaged trace of all measurements where the LSB is 1, the middle trace is the averaged trace of all measurements where the LSB is 0 and the bottom trace is the difference between them [4].

CPA is most efficient when the attacker has some architectural knowledge of the target and the model is designed accordingly. Many implementations of this attack rely on the Hamming weight or Hamming distance to predict the leakage. [4], [5]

However, if no correlation can be identified using CPA, it does not necessarily mean that there is no SC present, but it may also be that the model is not appropriate for the target [4].

2.2 Flash Memory

2.2.1 General Working Principles

Flash memory cells are typically based on the design of a Metal Oxide Semiconductor (MOS) transistor, but have a gate that is completely isolated (the Floating Gate (FG)). This design is referred to as a floating gate transistor and can be seen in Figure 2.3.

The FG is capacitively coupled to the Control Gate (CG) and acts as the storage for the memory cell. Electrons injected into the FG though the CG remain there and affect the conductivity of the transistor. [7]

To read the value of the memory cell, a fixed voltage is applied between CG and source. Depending on the charge that is stored in the FG, the current flowing between source and drain (reading current I_{read}) will either be zero or in the range of several

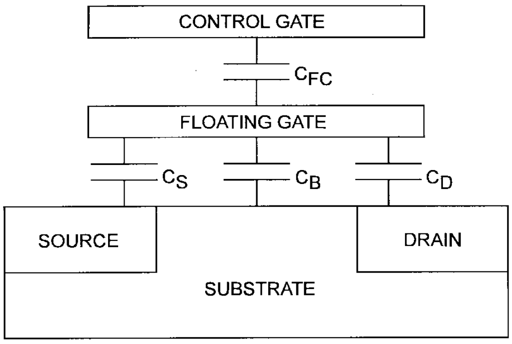

Figure 2.3: Layout of a floating gate transistor [6].

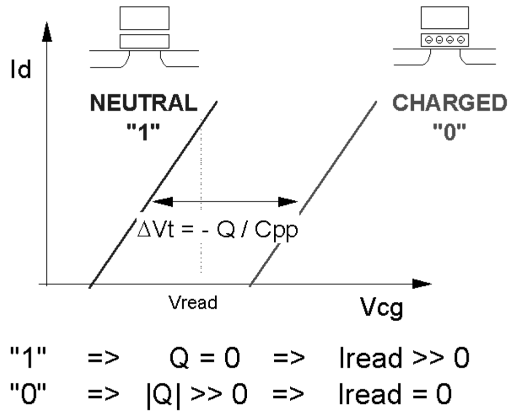

Figure 2.4: Offset between threshold voltage V_T for FG without (left) and with charge Q (right) [7].

micro amperes. If the cell is in the neutral state (no charge is stored in the FG) the transistor will show a large reading current and the cell is defined to be logically "1". If there are electrons in the FG, the reading current will be zero and the cell is defined as logically "0". [7]

The voltage required to make the transistor conductive (V_T) is applied between gate and source. The offset of V_T changes proportionally to the charge Q that is stored in the FG. This change is labeled as ΔV_T. [7]

However, the transistor curve of V_T remains the same. By analyzing the offset, a reading voltage (V_{read}) can be found that results in the previously described characteristics of I_{read}, see Figure 2.4 [7].

In order to program and erase flash memory cells, high voltages are needed. The erase level is around 20V and programming level approximately 18V [1], [2].

Mobile devices usually run on 3V or 5V and do not supply such high voltages. This is why FMCs have to generate high voltage levels themselves via charge pumps integrated on the chip. On modern flash devices, negative voltages are used for erasing which are also generated using charge pumps. [7]

The basic working principle of charge pumps is charging a capacitor (or multiple) in parallel and then reconnecting it in serial to achieve voltage doubling (multiplication). In order to achieve high efficiency and smooth higher voltages, various circuits have been developed and the actual implementation is manufacturer dependent.

Note that it takes time for the capacitor(s) to charge and that the supply of high voltage may be exhausted by long write or erase operations which may make additional charging time visible in the overall time needed by the memory chip to complete the operation.

2.2.2 NAND Memory

In NAND flash memory the storage cells are connected in series. The chain is connected to the bit line carrying the value of the read cell and ground with two selection transistors. This layout allows for high density design and page oriented programming and reading. [6]

In general, NAND memory is used for high-density storage applications. The design supports only block oriented access i.e. consecutive data is read and written very efficiently and multiple blocks can be erased with high speed. [6]

2.2.3 NOR Memory

In NOR flash memory the storage cells are connected to ground directly. This allows for random access to every cell and results in a simplified addressing and I/O. [7]

NOR memory is used if random access is required and reading operations are prevailing. While programming and erasing is slow in NOR, reading and random access reading in particular are fast. [1], [6]

2.2.4 Single and Multi Level Cells

The development of flash memory began with chips that were able to store one bit per cell called Single Level Cell (SLC). [6]

To increase the amount of storage, more flash cells have to be placed on the chip. While continuing advances in miniaturization allow more cells on a chip, at some point physical and technological constraints prevent adding more on the same die area. [6]

To be able to store more data on a chip without adding more flash cells, the obvious solution lies in storing more data in the already existing cells. This is the idea of Multi Level Cells (MLCs). Instead of having two levels to store one bit, four voltage levels are introduced allowing two bits to be stored. Even more bits can be stored in a cell by allowing more voltage levels. [1], [6] The downside of MLCs is that the periphery circuits occupy more space as they need to be able to program and measure voltage levels more accurately. This also requires higher programming voltages resulting in bigger charge pumps. [1]

Since the programming and reading operations differ between SLC and MLC in terms of voltage level and timing (two bits are written at once) we expect to see these differences in the power traces, too.

CHAPTER 3

State of the Art

3.1 Flash Memory

When it comes to FMCs a significant amount of research focuses on measuring and extending lifetime and reliability. The following literature suggests new techniques on the MC level which equals the logical layer:

In „Write Endurance in Flash Drives: Measurements and Analysis“, Boboila and Desnoyers empirically measure, analyze and provide methods of predicting the write endurance of different FMCs. They even reverse engineer some flash drives to give more insight into how manufacturers try to extend memory life. [8]

Desnoyers puts the focus on measuring performance and correlating it with the wear the memory has experienced in „Empirical Evaluation of NAND Flash Memory Performance“ [9].

Gupta, Pisolkar, Urgaonkar, *et al.* [10] and Chen, Luo, and Zhang[11] explore the idea of using value locality in order to reduce write operations, save storage and thereby not only increasing the lifetime but also performance. They use hash functions to identify data chunks already stored in memory and prevent redundant writes effectively implementing content de-duplication. [10], [11]

An improved wear leveling algorithm is introduced by Murugan and Du in „Rejuvenator: A Static Wear Leveling Algorithm for NAND Flash Memory with Minimized Overhead“. The improvement comes from identifying more heavily used data and storing it in less worn memory areas. [12]

Other research aims to improve the FMC's hardware which equals the physical layer:

In „Graphene Flash Memory“, Hong, Song, Yu, *et al.* report on the integration of graphene into FG transistors. They suggest that it allows for lower programming voltages and higher data retention times. [13]

In „Developments in Nanocrystal Memory“, Chang, Jian, Chen, *et al.* examine alternative methods of constructing the FG structures and different materials that can be used [14].

Baeg, Khim, Kim, *et al.* explore the applicability of top-gated organic field effect transistors to NAND memory in „High-Performance Top-Gated Organic Field-Effect Transistor Memory using Electrets for Monolithic Printed Flexible NAND Flash Memory“. The efficient charge trapping and detrapping in the electret layer turn out to give superior memory characteristics. [15]

Finally, Grupp, Davis, and Swanson try to project the future usage of FMC in SSDs. In „The Bleak Future of NAND Flash Memory“ they conclude that while storage capacity will continue to increase the durability and performance will stagnate or even degrade. [16]

Due to the increased use of FMC especially in mobile devices it has become an interesting target for forensic analysis.

Most attempts of accessing the device's flash storage are through a programming or debugging interface and not the memory directly. Breeuwsma, De Jongh, Klaver, *et al.*, Willassen describe using the JTAG port of a μC that is connected to the memory in order to extract the data. Additionally, they show how to read information off the flash storage using memory chip programmers. [17], [18]

However, note that data being read by a chip programmer still goes through the MC and is not the raw data saved in the flash cells.

Even though it is often said that data reminiscence does not exist in FMC, Skorobogatov shows in „Data Remanence in Flash Memory Devices“ that this is not true. If erase operations are not performed correctly data can be reconstructed even after up to 100 erase cycles. [19]

3.2 Side Channel Attacks

Research on Side Channel Attacks (SCAs) is very focused on cryptographic applications and retrieving the used secret keys. Roche, Lomné, and Khalfallah use a combination of fault injection and SCAs to efficiently extract the key used in an AES implementation in „Combined Fault and Side-Channel Attack on Protected Implementations of AES“ [20].

In „Black-Box Side-Channel Attacks Highlight the Importance of Countermeasures“, Moradi, Kasper, and Paar show how the bitstream encryption of Xilinx FPGAs, used to protect the firmware when stored in memory, can be broken by extracting the secret key with just a single start-up being analyzed [21].

Due to the dramatic success of SCAs, researchers started investigating counter measures.

Güneysu and Moradi provides a guideline on how to thwart power SCAs on FPGAs using noise generation, clock randomization and memory scrambling in „Generic Side-Channel Countermeasures for Reconfigurable Devices“ [22].

While power SCAs are the most common other SCs are also used in practical attacks.

In „Side-Channel Analysis of Cryptographic RFIDs with Analog Demodulation“, Kasper, Oswald, and Paar build a setup that uses an electro magnetic probe to measure the RFID smart card communication field and an analog filter for isolating and amplifying the SC signal [23].

Backes, Dürmuth, Gerling, *et al.* show that an acoustic SCA mounted against a dotmatrix printer can recover up to 72% of the words printed. In „Acoustic Side-Channel Attacks on Printers“, they use a sophisticated machine learning approach that works fully automatically after an initial training phase. [24]

3.3 Side Channel Attacks on Flash Memory

Semi-invasive methods that use lasers have been shown to allow manipulation the memory operations. Samyde, Skorobogatov, Anderson, *et al.* show how the raw data stored in a flash cell can be extracted using optical or electromagnetic probing in „On a New Way to Read Data from Memory“ [3].

In „Optical Fault Masking Attacks“, Skorobogatov shows how write and erase protection of a chosen memory region is achieved by pointing a laser at it. This is called optical fault masking. [25]

Existing research on the power consumption of FMCs focuses on finding models to predict and simulate power consumption for certain workloads of the systems the memory is embedded in. Olivier, Boukhobza, and Senn in „Toward a Unified Performance and Power Consumption NAND Flash Memory Model of Embedded and Solid State Secondary Storage Systems“, and Mohan, Bunker, Grupp, *et al.* in „Modeling Power Consumption of NAND Flash Memories Using Flashpower“ present frameworks for estimating power consumption to aid design of memory hierarchies in [26], [27].

To the best of my knowledge, SCAs targeting the power SC of FMC have not yet been researched.

CHAPTER 4

Methodology

4.1 Design

The goal of the thesis is to design and implement a framework to perform side channel measurements on FMC that is flexible and easy to adapt to different types of flash chips and various analysis methods.

To achieve good adaptability and clean interfaces, a modular component approach was chosen which makes use of standardized communication interfaces and enforces clear separation of duty. The main components identified in every power SCA are:

- Target: The device to be analyzed.
- Driver: The component instructing the target to perform the action of interest and trigger the measurement at the appropriate time.
- Measurement: The device measuring power consumption and providing an interface to transmit the data to the acquisition.
- Acquisition: The component that instructs the driver to generate the measurements of interest and retrieves the data from the measuring instrument.
- Preprocessing: Noise is reduced and only raw data of the sections of interest is kept for further analysis.
- Analysis: This is where the data analysis is performed. The type used may be a SPA, DPA or an alternative algorithm.

The presented structure can be applied to any type of SC measurement and not only to power analysis.

4.2 Driver

In case of FMC, the driver needs to support different types of NAND and NOR chips.

For NANDs, the I/O commands are standardized for the majority of manufacturers. As of this writing, the ONFI Workgroup[1] lists major flash manufacturers like Intel, Micron and SK Hynix as members that follow the ONFI specifications.

Similar to the ONFI group for NAND, JEDEC [2] has introduced the common flash interface (CFI) standard for NOR FMCs. Again, major manufacturers like Intel, Micron and SK Hynix are members of the JEDEC.

A general purpose driver should implement the two standards to have a fundamental support for different NAND and NOR chips. However, special operations may be implemented in a non-standard, vendor specific way and therefore the driver should always be tuned to the chips actually used.

Communication between the driver and the memory chip depends on the chip design. For receiving data from the acquisition, any form of I/O may be used. Since the data transferred will likely be of little size a serial bus supported by the acquisition is the most obvious solution. The trigger of the measurement can be easily implemented by a signal pin that changes level or sends an impulse when data recording should start.

4.3 Measurement

The measuring instrument is one of the most important components. Every effort made here to reduce signal noise and acquire high resolution data makes further analysis more likely to succeed and easier to implement.

SC analysis usually requires high-frequency and high-resolution measurements. If the sampling frequency is too low, important shifts in the measured variable may simply go undetected. Low resolution might be improved by amplifying the signal before or in the process of measuring.

There is more to it than just a good measuring instrument though. Special care has to be taken to ensure that there is no systematic error in the measurement setup. An example would be two different power supplies for the FMC, one for output amplification and one for memory logic and the amplifier's power drain being measured, rather than the one of the chip's logic.

The setup can also play a key role in preventing and reducing signal noise, e.g. an EMR SCA should always be performed inside a Faraday cage to keep out the environmental EMR.

When it comes to power SCs the interesting variable is the power consumption at a fixed supply voltage i.e. the current drain. Since oscilloscopes only measure voltages over time, the usual approach is to measure the voltage drop on a resistor that is connected in series to the target's supply voltage.

[1] `http://www.onfi.org/`

[2] `http://www.jedec.org/`

The challenging part is finding a resistor value high enough to get a good voltage drop and low enough not to affect the target device. A rough estimate can be found by calculating

$$R_{SC} = \frac{V_{CCmax} - V_{CCmin}}{I_{max}}$$

where R_{SC} is the value of the side channel resistor, V_{CCmax} the maximum and V_{CCmin} the minimum supply voltage of the target and I_{max} the maximum current drain of the device. However, note that fine tuning may be needed and can affect the outcome of the measurements.

4.4 Acquisition

The task of this component is to set up the driver to let the target perform the desired operation and then retrieve the data off the measurement device.

The communication between the driver and the acquisition devices depends on the driver and the implemented commands. However, communication between acquisition and measurement component should be standardized as far as possible to support usage of different measurement instruments.

The acquisition component is the lowest level on which user input can be accepted. Depending on the level of automation and tasks at hand, the framework may also perform preprocessing and analysis on the traces without any further user interaction. However, if the goal is exploratory research, it is useful to also expose the lower levels of the process giving the user a chance to try various processing and analysis methods on the data.

For testing the driver and measurement devices, an interactive and verbose User Interface (UI) may be preferred as it provides quick feedback and the flexibility to "test on the fly". However, once the lower components are working and the actual data analysis starts, a script-able UI is preferably. Not only does it allow easily repeatable measurements, but also unattended bulk data acquisition over a long period of time.

4.5 Preprocessing

In this pre-stage to the analysis, several operations may be performed to improve the reliability and accuracy of the data analysis.

A possible preprocessing stage may consist of the following steps:

- A simple heuristic ensures that invalid power traces are removed from the data set.
- Several repeated measurements of the same target operation are synchronized to match the begin and the end of the target's operation.
- The synchronized power traces are averaged to reduce noise.
- The filtered signal is split into different parts to improve efficiency of the analysis.

This stage is where this thesis's focus ends. It is left to future researchers to build their analysis algorithms to process the data acquired through this framework.

4.6 Analysis

The data analysis is performed on the preprocessed data in order to further understand the target's operation or data being processed. A simple analysis could compare local minima and maxima, the mean power consumption or the needed time until completion for different inputs. More advanced analysis methods like DPA may use simulated power models and statistical tests for analyzing the measurements.

CHAPTER 5

Implementation

5.1 Design

The implementation design closely resembles the components identified in Chapter 4.

The driver is implemented on an Atmel μC, an oscilloscope with a differential probe is used as measurement instrument and a computer is used to perform acquisition, preprocessing and analysis. Figure 5.1 visualizes the components and their communication with each other. Details are described in the following sections.

5.2 Target

Three flash memory chips were chosen as representatives for different flash types:

- SLC NAND: Samsung K9F1G08U0C: 128MB memory, 8 bit data bus
- MLC NAND: Hynix H27UAG8T2BTR-BC: 2048MB memory, 8 bit data bus
- NOR: Spansion S29GL128P: 16MB memory, 8 or 16 bit data bus

The chips are soldered onto sample boards ordered on Ebay[1]. The sample boards include the recommended capacitors and resistors and also connect the I/O pins to plug connectors that allow easy board swapping. A photo can be found at Figure 5.2.

The memory's supply voltage is provided by a bench top power supply. A GW Instek PSP-405 was already in the lab and more than sufficient for the task.

[1] http://www.ebay.com/

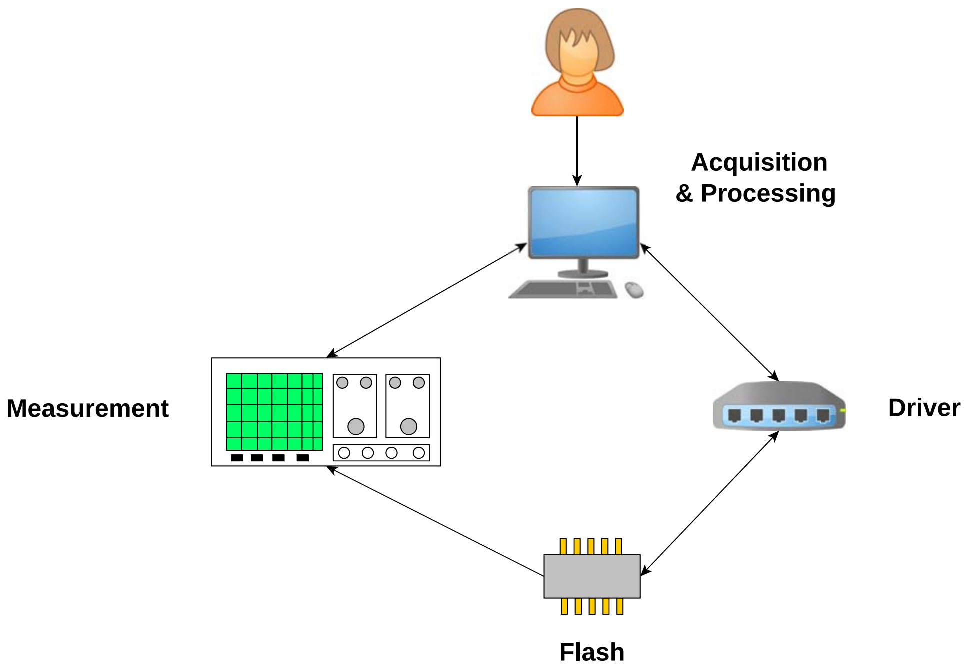

Figure 5.1: The components involved in this implementation.

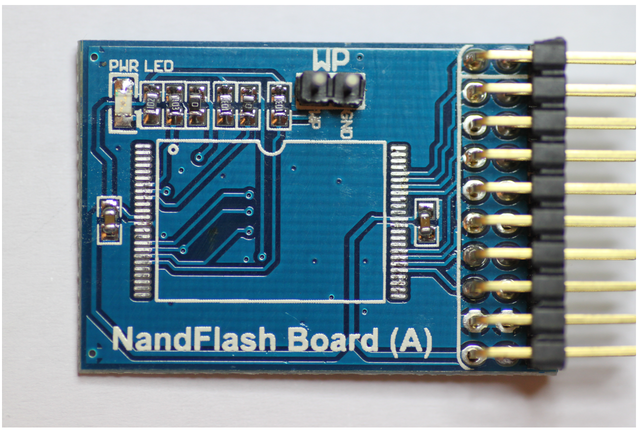

Figure 5.2: The NAND sample board without the memory chip on it.

5.3 Driver

5.3.1 Micro Controller

The driver was implemented on a XMEGA-A1 XPLAINED prototyping board with an ATxmega128A1 μC. The first important feature for the driver purpose is an integrated UART-to-USB gateway that allows a computer to connect directly via USB and use a virtual COM port to communicate with the μC. The second reason for this board are four digital I/O ports with eight pins each to connect the FMCs to.

The μC was programmed with an Atmel AVR Dragon[2] and avrdude which is included in the WinAVR[3] development bundle.

5.3.2 Flash Memory Adapters

Bridging between the driver's XPLAINED board and the flash sample boards is done using prototyping Printed Circuit Boards (PCBs). Since the sockets of the NOR and NAND sample boards are very different, two separate adapters were built.

The connection to the flash chips is made using the sample board plugs. This enables quick chip changing and easy setup.

The driver is connected to the adapter board by a 40 pin flat cable that is split into four 10 pin cables that connect to the driver's four I/O ports.

In addition, the adapter PCBs provide two cables to connect to the drivers supply voltage and ground. Another cable carries the trigger signal from the driver for the oscilloscope. A photo can be found at Figure 5.3.

5.3.3 Software

Design

The software was written in C using the AVR LibC [4] library. The main developing OS was Windows 7 and the programming tools and libraries are installed through the development package WinAVR. The driver implementation consists of the following:

- main.c: Deals with the computer communication, UI and system initialization.
- flash.c: Implements all FMC related tasks such as writing and reading data, as well as triggering measurements when appropriate.
- uart.c: Contains an initialization function and implementations of putchar and getchar for the UART bus.
- misc.c: General I/O and conversion functions.

[2] `http://www.atmel.com/tools/avrdragon.aspx`
[3] `http://sourceforge.net/projects/winavr/`
[4] `http://www.nongnu.org/avr-libc/`

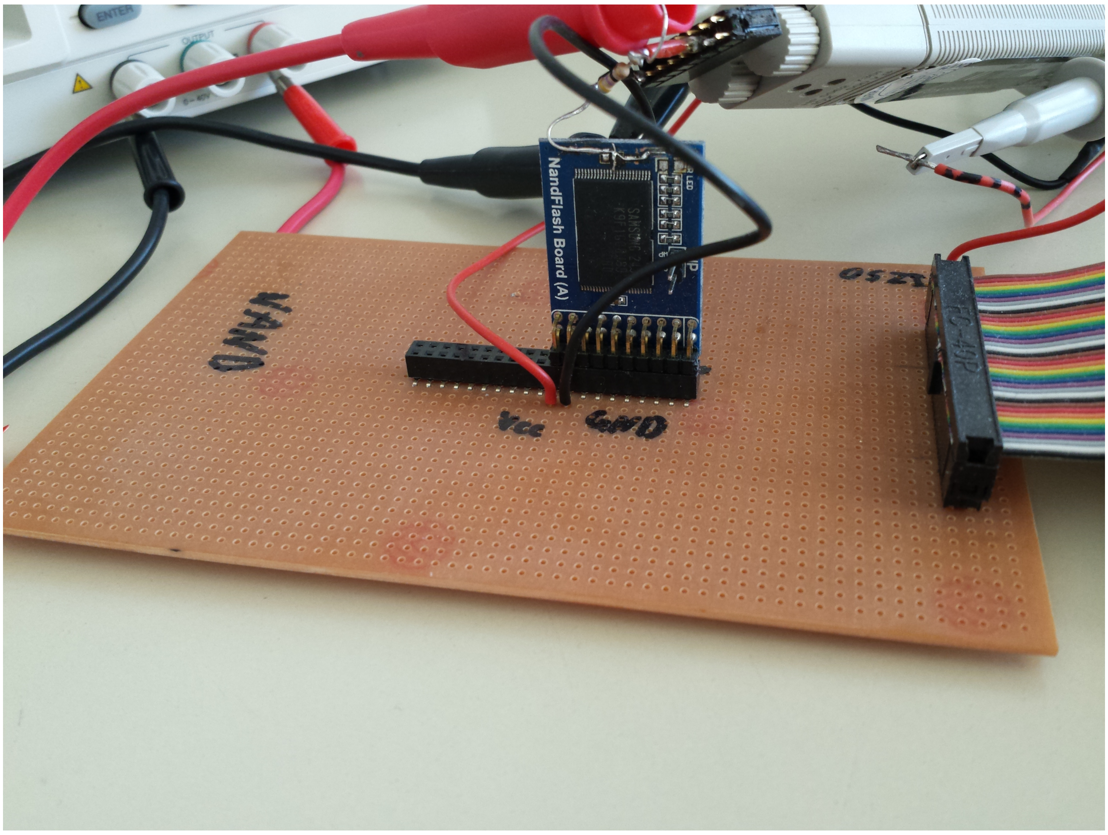

Figure 5.3: The NAND adapter board connecting the driver and the memory chip. The chip currently installed is the Samsung K9F1G08U0C. Oh the right-hand side one can see the 40 pin cable connecting to the driver. Above is the oscilloscope's probe connecting to the black striped trigger cable. On top of the NAND sample board one can see the SC resistor R_{SC}, the differential probe measuring the voltage drop and the power supply cable.

Interface

For this driver implementation the flash operations of interest are read and write. Since FMC needs to erase blocks before being able to write to them again, this function also needs to be exposed to the acquisition device, in this case the computer.

The type of flash chip can be configured to any of the supported ones without reprogramming the μC. The pin configuration of the driver is changed accordingly to match the pin layout of the flash adapters described earlier.

For debugging purposes, there are also commands to toggle the measurement pin and run a memory read/write self-check which also retrieves the manufacturer data.

The complete list of commands is printed if an unknown command is entered:

```
Commands:
r <startAddress (Dec or 0xHex)> [BytesToRead (Dec or 0xHex) - default is
    ↪ 20]
w <startAddress (Dec or 0xHex)> <Data (ascii or hex string with 0x prefix
    ↪  like 0x1122AAFF)*>
d [address (Dec or 0xHex) in sector to delete - no address == chip erase]
t <nand_s|nand_h|nor|undef>
x - toggles OSZI pin
test - runs diagnostics
```

Memory addresses may either be decimal without any prefix or hex if prefixed with a "0x".

In an early implementation, the driver only supported ASCII strings to be written to the FMC. However, this turned out to be too restrictive as C string functions usually use 0x00 as string terminator and not suitable therefore to e.g. write a series of 0x00 bytes to memory. Also other bytes that are ASCII control characters turned out to be difficult to handle for terminal applications.

Because of this, the current implementation does not only support ASCII strings but also allows the user to write arbitrary bit patterns by entering a data string that starts with "0x" followed by a series of two hex digits describing one byte to be written each. For instance `"w 0 0x00000000FF"` will write four bytes with no bits set followed by one byte with all bits set to memory starting at address zero. These strings are referred to as hex-strings.

Memory

Both chosen NAND chips follow the ONFI specifications so their I/O differs just with respect to their storage capacity and therefore their addressing. They use the same commands and the same eight pins for the data bus and five additional lines being read and write enable, address and command latch enable and the busy signal.

The NOR chip also uses 8 pins for the data bus but also has an extra address bus which means more pins are used than for the NAND memory. In total, 8 data and 18 address bus pins, plus two pins for write and output enable and one pin for the busy signal are used.

NAND memory supports block writing and reading, while NOR flash operates byte oriented. To match different ways of setting addresses and writing data, several functions were introduced to build an abstraction of an uniform access:

- void fl_setAddress(uint32_t addr): For NANDs, the address is written by sequential commands. For NORs simply the address pins are set accordingly.
- void fl_setCommand(uint8_t cmd): Writes a command to a NAND and does nothing for a NOR.
- void fl_setData(uint8_t data): Pushes the data into the NAND's data buffer or sets the NOR's data pins.

- uint8_t fl_getData(): Reads one byte from the memory's data bus.

Building on the basic operations above more complex functions were implemented:

- bool fl_WriteData(uint32_t addr, uint8_t data): The function takes one byte of data and writes it to the specified address. For NANDs, calling this function repeatedly is an inefficient way to write sequential data, for NORs this is perfectly fine. The return value is true on success and false on error.

- Status fl_Write(uint32_t addr, char* buf): This function is optimized for writing sequential data (strings) to memory. These strings may be regular ASCII strings or the afore mentioned hex-strings. The return value indicates success or what kind of error has occurred.

- uint8_t fl_ReadData(uint32_t addr): This is the counterpart to fl_WriteData. It reads one byte from the specified memory address. Again, it is not efficient to read sequential data from NANDs using this function.

- uint32_t fl_Read(uint32_t addr, char* buf, uint32_t readCount): The counterpart to fl_Write reads a specified amount of bytes from the given address and saves the result into buf. The string returned is not plain ASCII, but a hex-string. The return value is the number of bytes read from memory.

The source code can be found in flash.c in the Appendix.

Triggering

With both types of memory, the same output pin is used to trigger the oscilloscope with a rising edge. The time of the triggering is hard-coded into the driver's software. For the purpose of this thesis, the interesting operation is when the data is actually written to the flash cells. For NORs, this is with every byte written but for NANDs the stage where the bytes are loaded into the data buffer can be ignored. Once the persist command is issued the oscilloscope should start the measurement.

Another question to be answered in future research is the possibility to retrieve the processed data not only on write but also on read operations.

Therefore the trigger commands were placed in the fl_Read and fl_Write functions. In this implementation the oscilloscope trigger pin goes high before starting the operations of interest and goes low again after they have finished. This already gives some clues to the timing of the power trace displayed on the oscilloscope.

5.4 Measurement

The measurement instrument used is an Agilent Technologies DSO-X 3014A oscilloscope. It provides a sampling frequency of 200MHz and four input channels. The measurement

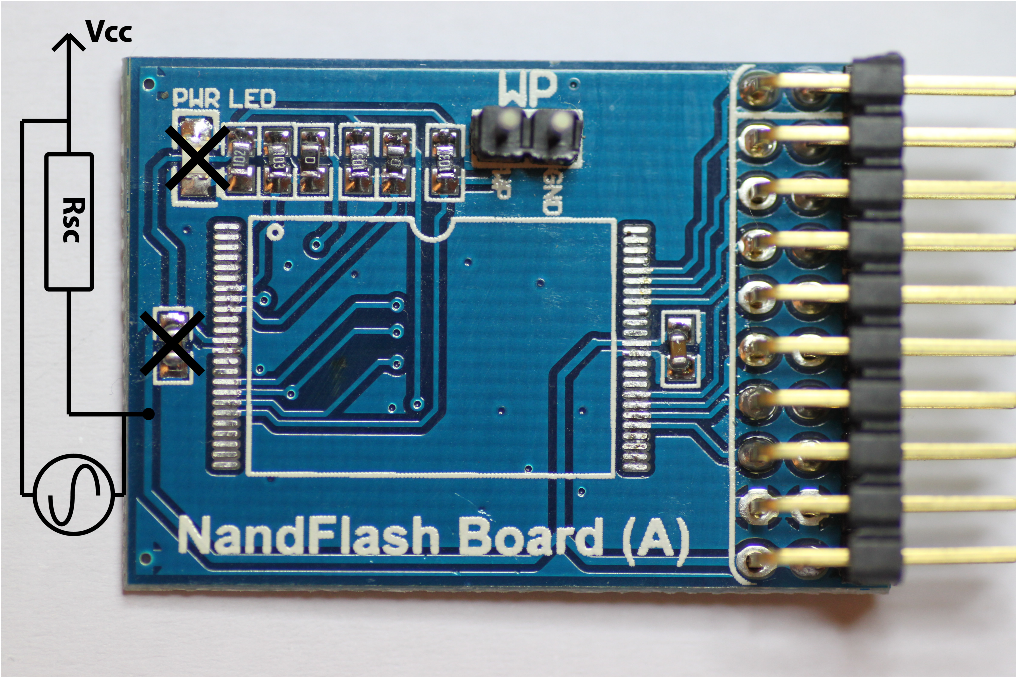

Figure 5.4: The modifications applied to the NAND sample board.

data can be retrieved using the Virtual Instrument Software Architecture (VISA) I/O API.

To measure the power used by the chip a resistor is put in series to the memory's supply voltage. To reduce interferences it is best to measure the signal as close to the chip as possible. Furthermore, power smoothening components should be removed just like additional parts not needed for operation.

Following these guidelines the power LED and decoupling capacitor were removed from the sample boards. The SC resistor R_{SC} was directly soldered onto the supply conductor. The modifications can be seen in Figure 5.4.

To measure the voltage drop on R_{SC} as accurately as possible and without any mathematical signal manipulation, a differential probe was used. The Hewlett Packard 1142A Probe Controller with a HP 1141A Differential Probe was already available in the lab and therefore chosen. To allow simple disconnect of the probe, a socket was soldered onto the resistor into which the probe can be plugged.

5.5 Acquisition

5.5.1 Interface

Python 3.4[5] was chosen as programming language because it offers libraries for many different I/O types, works cross-platform and permits high flexibility as a scripting language. For rendering the power trace plots the Python module matplotlib[6] was used. The implementation can be found in the file measure.py in the Appendix.

Following the recommendation from Section 4.4 two different UIs were implemented:

- Interactive: Used when experimenting with the setup and debugging.
- Scripted: Easy to gather bulk data for later analysis.

The interactive mode is used if the script is started without any command line arguments i.e.

```
> python3.4 measure.py
```

If an instruction is entered incorrectly a help message is printed showing all implemented commands:

```
Commands:
  i <nand_s | nand_h | nor> - initialize flash controller to Samsung nand,
    ↪  Hynix nand or nor chip
  w <# of measurements> <string to be written>
  r <# of measurements> <string to be written>
  d - toggles debug mode
  o [<mVPerDiv> <nsPerDiv> <mVTriggerLevel> <measurePoints>] - sets values
    ↪  for oszi
    to default or given values
  e - exits the program
  everything else prints this command list
```

Further details on the workings of the commands are given in Section 5.5.4.

To start the scripted interface the first argument to measure.py must be the instruction file i.e.

```
> python3.4 measure.py batch_measure.txt
```

The syntax for such a file is as follows:

- A line starting with ";" is a comment line and ignored
- Empty lines are ignored
- Configuration lines start with "config:" and follow the form of config:chip;mVPerDiv;nsPerDiv;mVTriggerLevel;measurePoints e.g. "config:nand_s;50;10000;2000;50000"

[5] https://www.python.org/

[6] http://matplotlib.org/

- measurements are defined with
  <w|r> <# measurements> <string used for measurement> e.g.
  - `"w 10 example data"` will take 10 measurements of writing the string "example data" without a terminating '\0' character to the flash chip.
  - `"r 50 0x7465737400"` will write ASCII "test" followed by a null terminator to the memory and then perform 50 measurements of reading the string.

The scripted mode only allows the commands "r" and "w" with the same parameters as the interactive mode. An example instruction file can be found in batch_measure.txt in the Appendix.

5.5.2 Driver Communication

The driver is connected to the PC's USB port. The driver's UART-to-USB bridge installs itself as a virtual COM port on the PC which allows easy serial communication. Stock Python already includes the *serial* module which was used for driver I/O. The commands issued by the acquisition have to conform to those implemented by the driver. The communication with the driver is hidden but can be seen by activating the debug mode. By default debug is deactivated in scripted mode and activated in interactive mode. However, the interactive user can toggle it using the "d" command. The output of the driver is filtered for keywords such as "Error" to detect any problems and "data:" to retrieve the output. The input strings are always converted to hex-strings (introduced in Section 5.3.3) before being sent to the driver. To ensure proper setup of the driver the "test" command is executed after every new flash chip configuration in the Python script. Measurements can begin after the command has finished successfully.

5.5.3 Oscilloscope Communication

The Agilent oscilloscope features an Ethernet interface and supports the VISA API. Necessary drivers and the VISA library are bundled in the Agilent IO Libraries Suite[7].

The VISA API should work across different oscilloscope models, but checking the device's programming manual is highly recommended.

In order to use the Agilent VISA library with Python, the PyVISA[8] module was installed.

measure.py will connect to the VISA device immediately if it only finds one. If there are more VISA devices available the user will be prompted to choose one.

Upon successful connection the oscilloscope's ID is queried and displayed. Then the device's self-check is run before resetting it to its default settings.

If no errors occurred the oscilloscope initialization is complete and the memory chip can be configured.

[7] `http://www.agilent.com/find/iosuite`
[8] `https://github.com/hgrecco/pyvisa`

5.5.4 Acquisition Process

To begin the SC measurement, the memory chip has to be connected to the driver via an adapter board. The driver is linked to the computer via USB and is assumed to install as COM3 port. The last step is to make sure that the computer and the oscilloscope are in the same network.

The measure.py script will show errors if one component is not reachable. Assuming that all components are connected correctly calling the script without any arguments will start the interactive mode.

The steps to perform a measurement are:

1. On startup the oscilloscope is connected and initialized.
2. Configure a FMC using the "t" command. This triggers the driver's self test for the chip.
3. Optionally change the default measurement settings like resolution using the "o" command. By default they are 50mV/div, 10μsec/div, 2V trigger level and 50000 measurement points.
4. Issue a measure command i.e. "r" or "w".

Every measurement gets a unique measurement-ID. It consists of the command, the string to be used, the string's length and a time stamp, all separated by one space each. If the string has more than 10 characters, only the first 10 are taken and three dots are appended.

The procedure of both measure commands is as follows:

1. Erase the FMC with the driver's "d" command.
2. Write the string to memory starting at address 0x00.
3. Read the characters from 0x00 and make sure they match the written string.
4. A new line is added to the "measure.log" file containing the measurement id, a time stamp, the memory type, the command, the string as hex-string and as ASCII string.
5. Set the oscilloscope with the configured parameters.
6. For the specified number of measurements repeat:
   a) Arm the oscilloscope to wait for the trigger signal.
   b) The next step depends on the issued command:
      - "r": The driver will continue to read the string from 0x00 and make sure it matches the test-string.

    - "w": The driver will continue writing to memory at increasing addresses to avoid overwriting previously written data.

c) Retrieve data from the oscilloscope:

    i. Query waveform data from the oscilloscope.

    ii. Strip the header information and save stripped data as "<measurement-ID>#<measurement-number>_stripped.txt" into the measurements folder.

    iii. Query preamble from the oscilloscope and save it as "<measurement-ID>#<measurement-number>_preamble.txt" into the measurements folder.

    iv. Adjust the waveform data with the information from the preamble to get correct timing and voltage data.

    v. Save the corrected waveform as Comma Separated Value (CSV) file with the name "<measurement-ID>#<measurement-number>.csv" into the measurements folder.

    vi. Plot the power trace, save it as "<measurement-ID>#<measurement-number>.png" into the measurements folder and add it as a subplot to the measurement's plot overview.

d) If the script was started in scripted mode it prints a success message and continues with the next measurement. Once all are done it exits. If it was started in interactive mode, the measurement's overview plot is shown, followed by a prompt for the next command.

5.6 Preprocessing

5.6.1 Interface

Like the acquisition component the preprocessing was implemented using Python 3.4. The implementation can be found in the file process.py in the Appendix.

Since the preprocessing does not need any user input, the interface was kept very simple and the task is fully automatized.

```
> python3.4 process.py
```

When the script is called it searches for files ending with "#0.csv". Then all measurements are removed where the same filename is found but instead of ending with "#0.csv" ends in " mean.csv". This results in a list of all measurements that have not yet been processed. The preprocessing process is then performed on this list.

5.6.2 Preprocessing Process

The goal of the implemented preprocessing is reducing signal noise and producing a clean power trace.

The first step for achieving this is synchronizing the different measurements to avoid corrupting the power traces when averaging them.

In an earlier implementation local maxima and minima were used to find points in the traces and to synchronize them on. However, the results showed that due to noise and too many inconsistencies between the measurements the number of identified points and their locations varied too much to be of any use.

The current implementation uses a very simple and efficient method of synchronizing and averaging the power traces:

1. The maximum values for each measurement are calculated.
2. The median from these maxima is taken and divided by 4. This is the threshold value.
3. For each measurement the offset from the first data point to the point exceeding the threshold value is calculated.
4. Starting from these offsets the power traces are averaged using the arithmetic mean function.
5. In the last step the averaged power trace is trimmed by finding the minimum of all offsets and then removing the data points from 0 to this minimum offset.

The now processed data is saved to the measurement directory as "<measurement-id> mean.csv" and a plot of the data is saved as "<measurement-id> mean.png".

If only one measurement was processed the resulting plot is displayed to the user, if more were processed the script simply exits.

5.7 Analysis

The data analysis can now be performed using any tool of choice. The raw power trace data and the preprocessed data is available as in CSV file format. This data can then be imported into regular office tools like Microsoft Excel[9], special mathematics software like Mathworks Matlab[10] and R[11], or programming languages like Java[12] and Python. Analysis methods can be as simple as looking at the plotted power traces, comparing the time it takes for different operations to complete or comparing the total power consumed. The number of voltage spikes may even reveal what data is processed. More complex analysis may include simulated power models and statistical tests which can be used in a DPA.

[9] http://products.office.com/en-us/excel
[10] http://en.mathworks.com/products/matlab/
[11] http://www.r-project.org/
[12] https://www.java.com/en/

CHAPTER 6

Results and Validation

6.1 Setup

The devices used were:

- Computer (all software is for x64 architecture)
  - Windows 7 SP1
  - Agilent IO Libraries v. 16.3.17914.4
  - WinAVR v. 20100110
  - Python v. 3.4
  - PyVISA v. 1.5.dev1
  - matplotlib v. 1.3.1
- Oscilloscope: Agilent Technologies DSO-X 3014A
- Probe: Hewlett Packard 1142A Probe Controller with a HP 1141A Differential Probe
- Driver: Atmel XMEGA-A1 XPLAINED with an ATxmega128A1 μC
- Programmer: Atmel AVR Dragon
- Flash memory chip: Samsung K9F1G08U0C SLC-NAND
- Power supply: GW Instek PSP-405

A photo of the setup can be found at Figure 6.1.

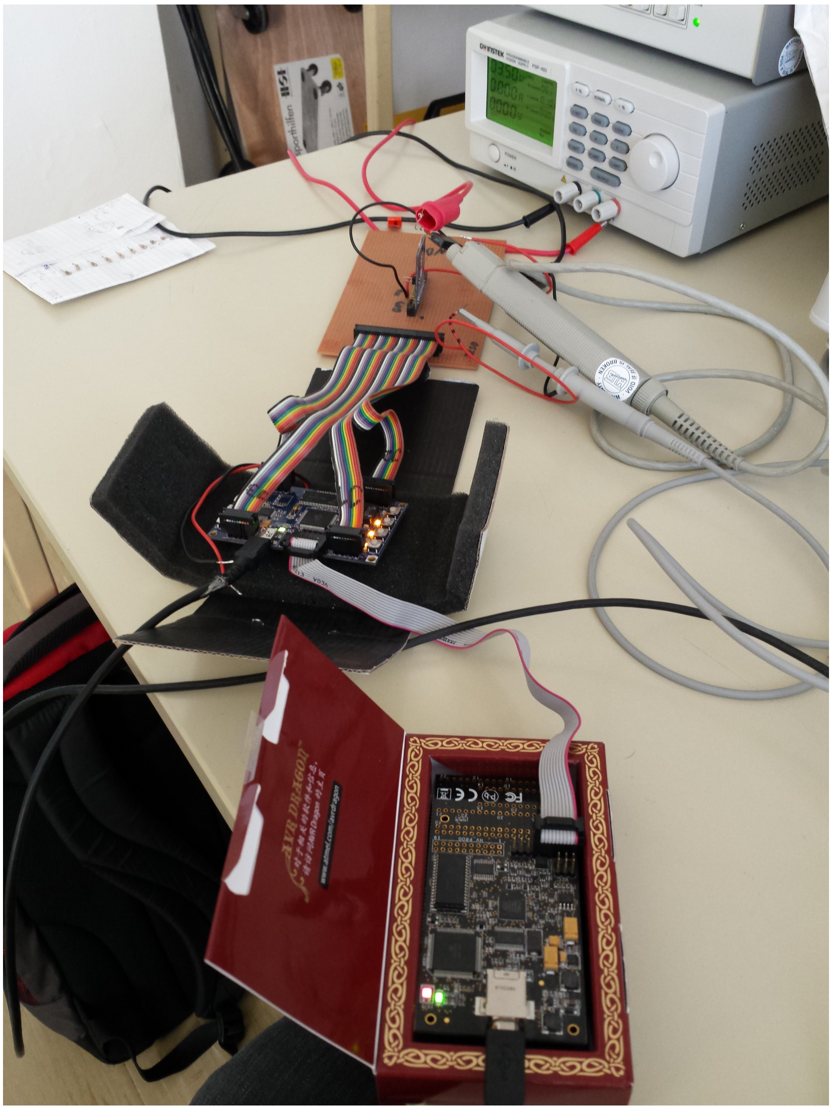

Figure 6.1: Lab setup for the demonstration. In the front is the programmer which is connected to the driver. The flat rainbow cable connects to the adapter board on which the memory chip sits. From the chip there is a gray bulky device going to the right of the picture, this is the differential probe. The thin gray probe next to it is the trigger. In the background is the power supply.

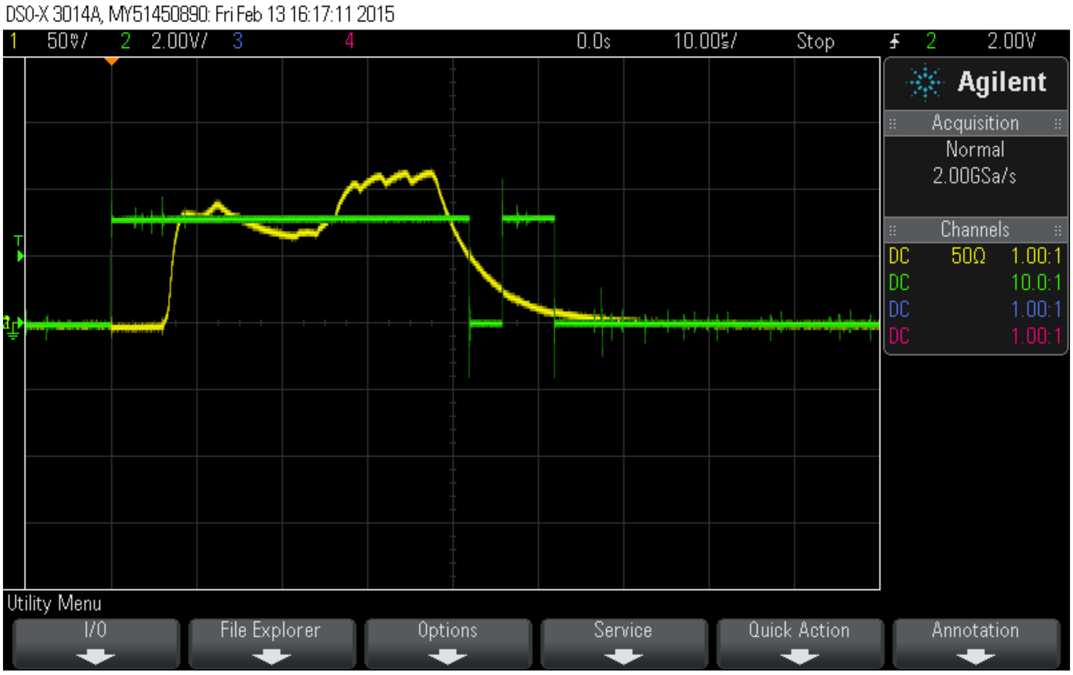

Figure 6.2: Oscilloscope's display of power trace (yellow) and trigger signal (green) from a write operation of the string "example measurement" to a NAND memory.

The NAND's supply voltage was set to 3.5V and the resistor used for the SC measurements was approximated to be

$$R_{SC} = \frac{V_{CCmax} - V_{CCmin}}{I_{max}} = \frac{4.6V - 2.7V}{35mA} \approx 54\Omega$$

Empirically, a resistor $R_{SC} = 46,1\Omega$ was chosen.

6.2 Demonstration

For the demonstration, the string "example measurement" was written three times to the NAND chip and the power traces recorded. In Figure 6.2 one of the measurements can be seen. Assuming that everything is connected properly the steps to get this measurement would be:

```
> python3.4 measure.py
> t nand_s
> w 3 example measurement
```

This will create a file "measure.log" with a log entry for the measurement. For each of the three traces recorded the following files will be crated with a suffix of "#0" to "#2": A "_preamble.txt" file, a "_stripped.txt" file with the raw data, a CSV file with the adjusted data and a ".png" file with the plotted power trace.

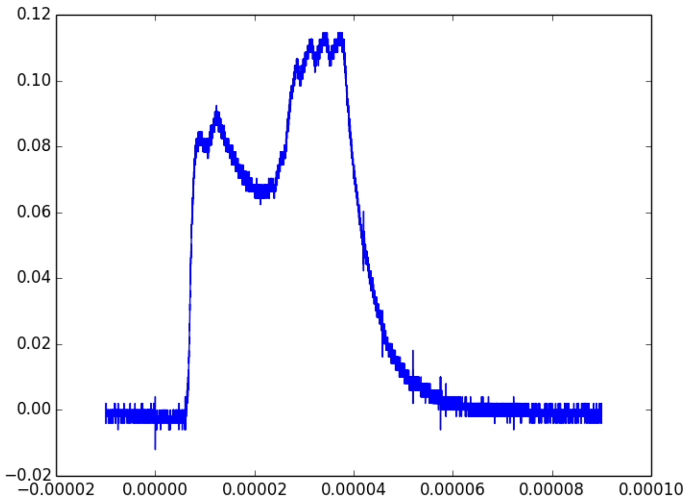

Figure 6.3: Plot of the acquired data for a write of the string "example measurement" to a NAND memory. file: "w 'example me...' 19 2015.02.13-16_02_06#0.png".

For example the file "w 'example me...' 19 2015.02.13-16_02_06#0.png" can be seen in Figure 6.3

Once the measurement is completed, the preprocessing can be run:

```
> python3.4 process.py
```

With only three measurements the smoothing effect is not big but one can clearly see the effects of synchronization and trimming of the preprocessing stage in Figure 6.4.

6.3 Data Analysis

A quick and not representative analysis of the sample data generated using the implemented SC framework only focused on comparing write operations of different length and Hamming weight.

The instructions file for the measurements performed is:

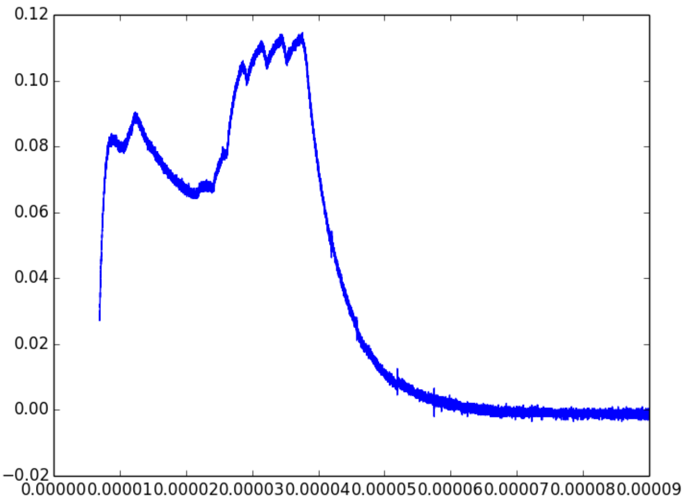

Figure 6.4: Plot of the preprocessed data for a write of the string "example measurement" to a NAND memory. file: "w 'example me...' 19 2015.02.13-16_02_06 mean.png".

```
; try different lengths
w 10 0x5B
w 10 0x5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B
w 10 0x5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B
    ↪ 5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B
    ↪ 5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B
    ↪ 5B

; try different bits
w 10 0x00000000000000000000
w 10 0x01010101010101010101
w 10 0xffffffffffffffffffff
```

Each measurement was repeated 10 times and subsequently preprocessed. The analysis is a very simple form of DPA.

The assumption is that the length of the input correlates to the amount of power consumed and time taken. To explore this assumption the difference between measurements of writing a character once and 20 times is compared. In particular the difference between "0x5B" ("[" in ASCII) and "0x5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B" is computed.

A Python script was implemented that subtracts two power traces from one another and displays a plot where both measurements are plotted over another and the difference below. This script is called analysis.py and can be found in the Appendix.

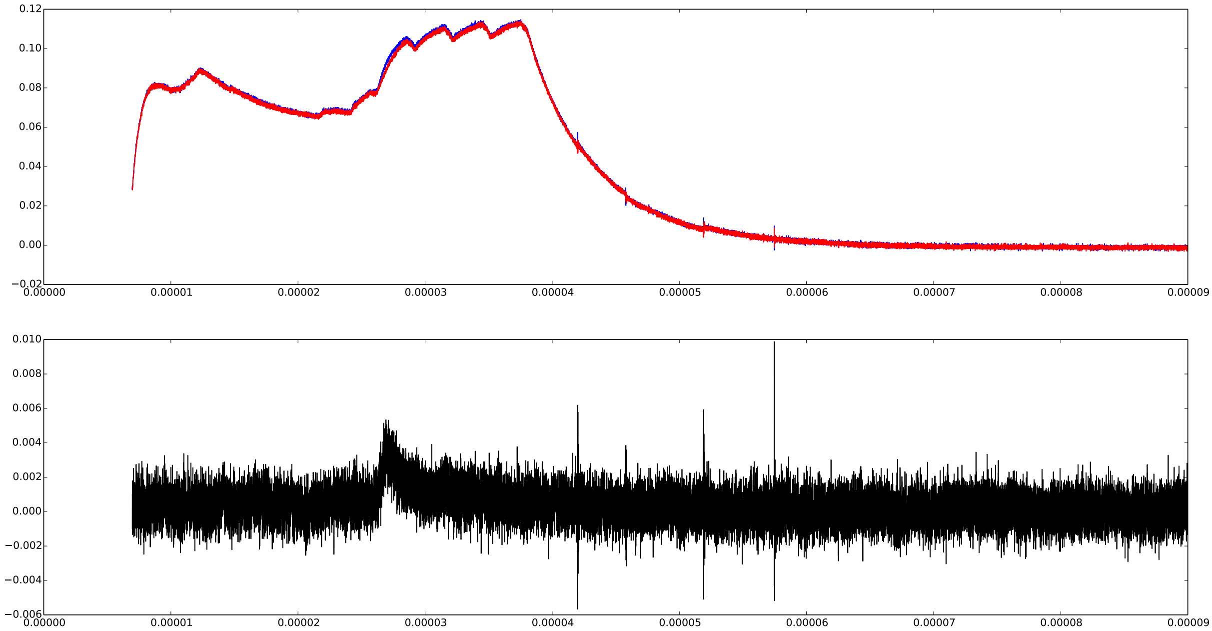

Figure 6.5: Analysis result of 20 "[" characters and 1 "[" character written to the NAND memory.

```
> python analysis.py "./measurements/w '0x5B5B5B5B...' 20
   ↪ 2015.02.13-13_06_23 mean.csv" "./measurements/w '0x5B'
   ↪ 2015.02.13-12_51_44 mean.csv" res_20-1.png
```

The result in Figure 6.5 shows that there is definitely a difference between the two power traces. The degree of correlation should be further investigated. The difference between 100 "[" characters and 1 character written can be seen in Figure 6.6.
Supporting our assumption the difference in power consumption increased with the amount of characters written. This is a strong indication for correlation of power consumption and string length.

The second assumption is that the number of unset bits written corresponds with the consumed power and programming time. The idea is based on the working principle of FMC that has all bits set when the chip was cleared. The flash cells will not be altered by programming a bit with a 1 but a 0 will take power and time to program.
Figure 6.7 shows the difference between programming 10 successive bytes with no bits set (0x00) and 10 bytes with all bits set (0xff).
Again a difference is visible and the correlation should be further investigated.
To see just how little of a difference in Hamming weight can be seen Figure 6.8 shows the result for writing 10 successive null bytes (0x00) and 10 bytes where just one bit is set (0x01).
In this case no clear difference can be seen. This may be due to too much noise or because the assumption was wrong.

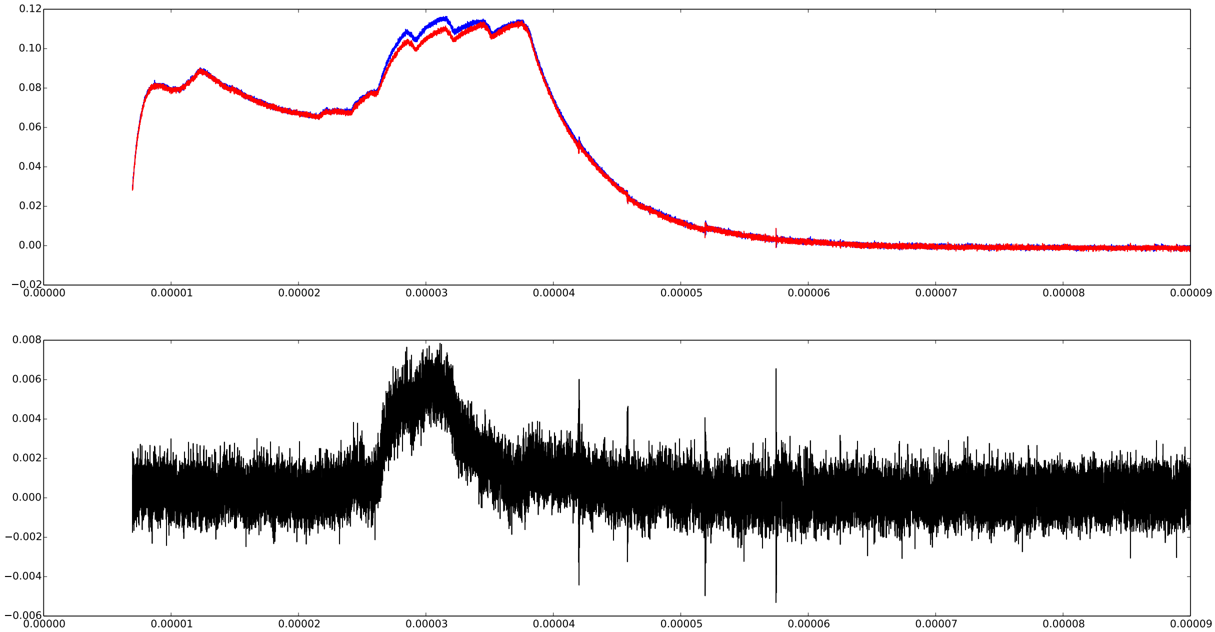

Figure 6.6: Analysis result of 100 "[" characters and 1 "[" character written to the NAND memory.

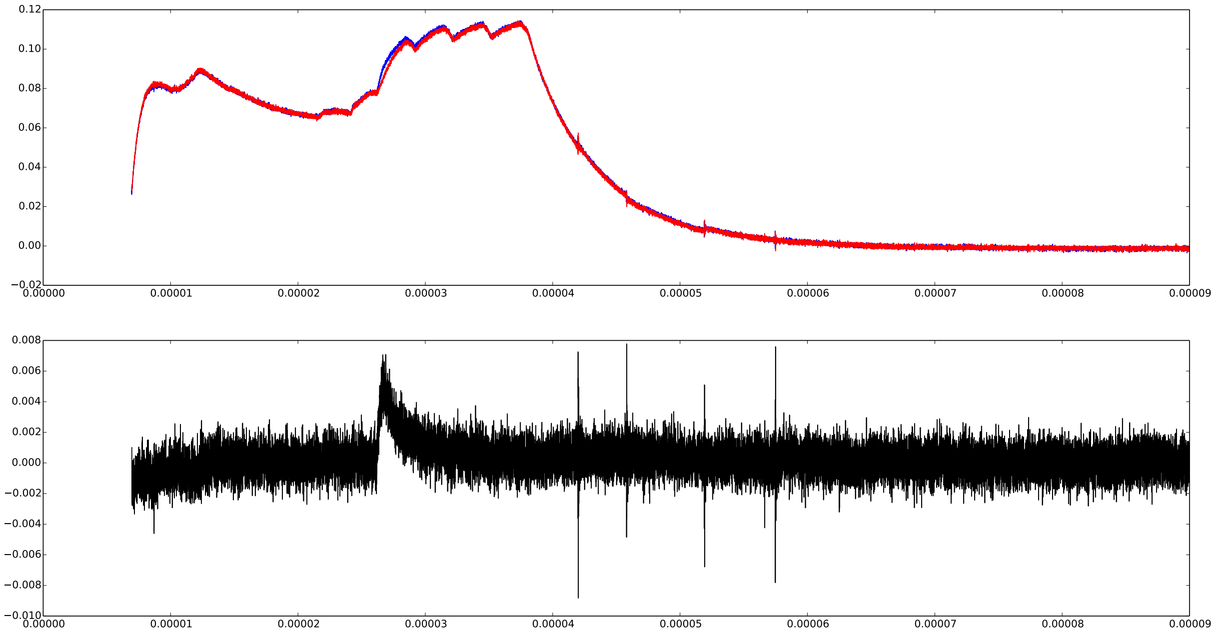

Figure 6.7: Analysis result of 10 0x00 bytes and 10 0xff bytes written to the NAND memory.

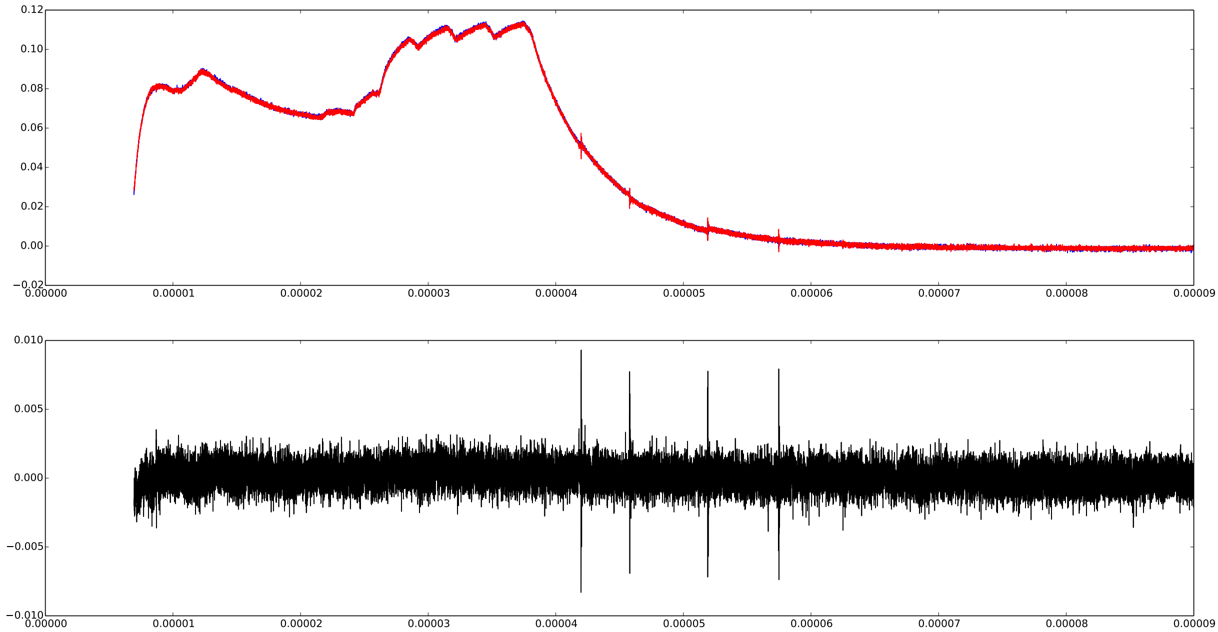

Figure 6.8: Analysis result of 10 0x00 bytes and 10 0x01 bytes written to the NAND memory.

In all cases above, the sample set was very limited. Instead of writing the same bytes over and over, random data should be used that only differs in one respect e.g. the string length or the value of the 5th bit.
However, the shown differences in power consumption are indications that the assumptions stated may be provable with more research.

6.4 Discussion

6.4.1 Automation and User Interface

The presented implementation automates the acquisition process and also attempts to ensure correct setup and data processing. However, preprocessing and data analysis must be triggered manually. It is no "one click" SC framework that automates the complete procedure.
This is due to the exploratory state of research. Once different analysis techniques have been developed to produce meaningful results they can be integrated into the framework and the components can then be linked together to provide a "one click" solution. Until then, the current implementation allows easy swapping of components and testing new approaches.
The UI is purely command line oriented, the reason being simplicity of implementation. It also allows scripting sequences of operations using e.g. shell scripts. The current target

users are researchers and developers which can be expected to work with a Command Line Interface (CLI). However, if the goal was to introduce a wider public to this technology a graphical UI would be more appropriate.

6.4.2 Measurement

The measurement tools used in this implementation were not specifically chosen for SCAs. They were selected on a "best available" basis.
One major issue that should be improved is the limited sampling frequency of the oscilloscope and its finite memory lowering resolution for power traces over a prolonged amount of time. More advanced instruments offer sampling frequencies of above 5GHz whereas the used oscilloscope only reaches up to 200MHz.
The differential probe was suited quite well for the used oscilloscope but the method of connecting it to the power supply of the chip focused more on practicability than on a perfect measurement. Results could be improved by placing the resistor very close to the memory chip and soldiering the probe directly to it.
The SC resistor used was chosen empirically without trying a representative amount of alternatives. Different qualities and types of resistors were ignored entirely.

6.4.3 Acquisition

The saved power traces start just before the persisting operation in the FMC is initiated. The power consumption during the data transmission is not recorded. While the persistence is probably more interesting to analyze, the filling of the buffer may also reveal information about the processed data and the memory's inner workings.
The current driver implementation only supports measurements of read and write operations. Not only may the delete operation be of interest, but FMCs may offer advanced operations such as data encryption before saving it or different access methods. These operations were not implemented because the focus of the framework is on read and especially write operations.

6.4.4 Analysis

Even though a very simple data analysis method was presented in Section 6.3 this thesis has its focus on the measurement and acquisition. The data processing and analyzing should be investigated further.
It would be nice if an analysis framework were developed that provides core components and tools for advanced data processing but still allows adaptation and customization for exploratory research.

CHAPTER 7

Summary

First, Chapter 1 gives an introduction and motivation to why this work is a useful contribution to research. It describes that it is often not possible to know what data is really written to the FMC and how it can be different from the data that was sent to the device's MC.
A brief description of FMC in general and the different types that can be found in today's devices is given in Chapter 2.
Then Chapter 3 presents the current state of FMCs and side channel attacks in research. It emphasizes that the focus of SCAs is on extracting encryption keys from μCs and no comparable attempts to analyze the power SC of FMCs has been attempted yet.
The analysis and design of components needed to mount a power SCA on FMCs is presented in Chapter 4. Six key components are identified and described.

A practical implementation of these components is shown in Chapter 5. Using a μC, an oscilloscope with a differential probe and a computer, a framework is set up that allows generation and acquisition of power traces from FMCs. The component diagram can be found at Figure 5.1. The framework is designed to support FMC following certain I/O standards and three chosen chips are used for reference. UIs are implemented with the different needs in mind that arise from the scenarios of performing exploratory research and building a big set of measurements for further data analysis. The framework includes a driver with adapter boards for NANDs and NORs, an oscilloscope and two Python scripts, one for generating the power traces through the driver and retrieving the data from the oscilloscope and one script for synchronizing and averaging multiple measurements to reduce signal noise.
Sample measurements and a simple DPA like data analysis are presented in Chapter 6. Using a simple form of DPA it is shown that the strings written to one of the NAND chips do affect the power consumption of the FMC. The two variables analyzed are string lengths and the number of unset bits in a byte (Hamming weight).

Concerning the framework, further research should be directed towards implementing a variety of standards and FMCs with their specific operations.

Also, the used equipment should be reviewed to improve measurement quality and potentially find correlations that were missed in this work.
In terms of applicability of the presented framework major efforts should be put into designing reliable and automatic power analysis methods. Once accessed data can be retrieved from the power traces reliably, the next step would be to analyze devices that have FMC integrated into them e.g. SSDs and to reverse engineer the algorithms implemented in MCs.
A very interesting idea is that firmware is also stored on FMC in many devices. If one could retrieve the data read during the device's startup and operation, it might be possible to reconstruct the firmware of the device. This has the potential of becoming a major issue for manufacturers trying to protect their firmware by encrypting communication between MC and the processor but not encrypting the stored firmware.

Appendix

Source Code

Controller

main.c

```c
#include "defines.h"
#include "uart.h"
#include "flash.h"
#include "misc.h"
#include <stdio.h>
#include <avr/io.h>
#include <string.h>
#include <stdbool.h>
#include <stdlib.h>
#define BUF_LEN 1024
FILE uartStream;

static void init() {
  LED_PORT.DIR = 0xff;  //set LED port to be output
  LED_PORT.OUT = 0xff;  //low active -> turn LEDs off

  LED_PORT.OUTCLR = 0x01; //Status display 1st LED

#if F_CPU == 32000000
  setClockTo32MHz();
#endif
  LED_PORT.OUTCLR = 0x02; //Status display 2nd LED
  uartStream = uartInit();
  stdin=stdout=&uartStream;
  LED_PORT.OUTCLR = 0x04; //Status display 3rd LED
  if (fl_Init(Undef)) {
    LED_PORT.OUTCLR = 0x08; //done init 4th LED
    printf("System started!\nInit success!\n");
  } else {
    printf("System started!\nInit error! Please set type via t-command!\n");
  }

}
```

```

int main(void) {
  char inBuf[BUF_LEN];
  char outBuf[BUF_LEN];
  char* cmd;
  char* addrStr;
  char* dataStr;
  uint32_t addr;
  uint32_t i;
  Status stat;
  bool toggle=false;
  long tmpHexLong=0;
  char* tmpHexBuf="0xAA";

  init();

  for (;;) {

    //Read out the received data
    printf("> ");
    in(inBuf, BUF_LEN);
    cmd = strtok(inBuf, " ");
    addrStr = strtok(NULL, " ");
    dataStr = strtok(NULL, NULL );

    printf("cmd: %s\naddrStr: %s\ndataStr: %s\n", cmd, addrStr, dataStr);

    if (strcmp(cmd, "test") == 0) {
      puts("TEST received!");

      fl_Test();

      LED_PORT.OUTCLR = 1 << PIN4_bp;
      LED_PORT.OUTSET = 1 << PIN5_bp;

    } else if (strcmp(cmd, "w") == 0) { //write command
      if (addrStr == NULL || dataStr == NULL ) {
        puts(
            "Error! Usage:\nw <startAddress (Dec or 0xHex)> <Data (ascii)
                ↪ *>\n");
        continue;
      }

      addr = (uint32_t) strtol(addrStr, NULL, 0);
      stat = fl_Write(addr, dataStr);

      if (stat == Success) {
        printf("Success writing to 0x%08lx\n", addr);
      } else {
        printf("Error(%i) writing to 0x%08lx\n", stat, addr);
      }

    } else if (strcmp(cmd, "r") == 0) { //read command
```

```
      if (addrStr == NULL ) {
        puts(
            "Error! Usage:\nr <startAddress (Dec or 0xHex)> [BytesToRead (Dec) - default is 20]\n");
        continue;
      }
      addr = (uint32_t) strtol(addrStr, NULL, 0);
      if (dataStr == NULL )
        i = 20;
      else
        i = (uint32_t) strtol(dataStr, NULL, 0);

      if (i > BUF_LEN/2-2-1) {  //every byte needs 2 hex digits - 2 for "0x" header - 1 for \0
        printf("Error! Can read max %i bytes!\n", BUF_LEN - 1);
        continue;
      }

      i = fl_Read(addr, outBuf, i);

      printf("Read %ld bytes from 0x%08lx\n", i, addr);

      for (int a = 0; a < i; a++) {
        memcpy(tmpHexBuf+2, outBuf+2+a*2, 2);
        tmpHexLong=strtol(tmpHexBuf, NULL, 0);

        printf("0x%08lx: %c (0x%02x)\n", addr + a, (unsigned char) tmpHexLong, (unsigned char) tmpHexLong);
      }
      printf("data: %s\n", outBuf);

    } else if (strcmp(cmd, "d") == 0) { //delete command

      if (addrStr == NULL ) {
        puts("starting chip erase...");
        stat = fl_Erase(-1);
        if (stat == Success) {
          printf("Chip erase success!\n");
        } else {
          printf("Error(%i) erasing chip!\n", stat);
        }
      } else {
        puts("starting sector erase...");
        addr = (uint32_t) strtol(addrStr, NULL, 0);
        stat = fl_Erase(addr);
        if (stat == Success) {
          printf("Sector erase success!\n");
        } else {
          printf("Error(%i) erasing sector 0x%08lx!\n", stat, addr);
        }
      }

    } else if (strcmp(cmd, "t") == 0) { //type command
```

```

          if (strcmp(addrStr, "nand_s") == 0) {
            fl_Init(Nand_S);
            puts("Set type to Samsung nand!\n");
          } else if (strcmp(addrStr, "nand_h") == 0) {
            fl_Init(Nand_H);
            puts("Set type to Hynix nand!\n");
          } else if (strcmp(addrStr, "nor") == 0) {
            fl_Init(Nor);
            puts("Set type to nor!\n");
          } else {
            fl_Init(Undef);
            puts("Set type to undef!\n");
          }

        } else if (strcmp(cmd, "x") == 0) { //toggle command
          printf("Toggle is %i\n",toggle);
          fl_Oszi(toggle);
          toggle=!toggle;

        } else {
      out("unknown command: ");
      puts(inBuf);
      puts(
          "Commands:\n"
              "r <startAddress (Dec or 0xHex)> [BytesToRead (Dec or 0xHex) - default is 20]\n"
              "w <startAddress (Dec or 0xHex)> <Data (ascii or hex string with 0x prefix like 0x1122AAFF)*>\n"
              "d [address (Dec or 0xHex) in sector to delete - no address == chip erase]\n"
              "t <nand_s|nand_h|nor|undef>\n"
              "x - toggles OSZI pin\n"
              "test - runs diagnostics\n"
          );

      LED_PORT.OUTCLR = 1 << PIN5_bp;
      LED_PORT.OUTSET = 1 << PIN4_bp;
    }

  }

  return 0;
}
```

flash.h

```
#ifndef MY_LITTLE_CUSTOM_FLASH_H
#define MY_LITTLE_CUSTOM_FLASH_H

#include <stdint.h>
#include <stdbool.h>
```

```c

typedef enum {
  Nand_S, //Samsung K9F1G08U0D
  Nand_H, //Hynix H27UAG8T2BTR
  Nor,
  Undef
} FlashType;

typedef enum {
  Success = 0, Busy, Error, TimeOut
} Status;

bool fl_Init(FlashType type);
Status fl_Write(uint32_t addr, char* buf);
bool fl_WriteData(uint32_t addr, uint8_t data);
uint32_t fl_Read(uint32_t addr, char* buf, uint32_t readCount);
uint8_t fl_ReadData(uint32_t addr);
void fl_Test();
bool fl_Reset();
Status fl_Erase(uint32_t addr);
Status fl_GetStatus(uint32_t Timeout);
void fl_Oszi(bool enable);

#endif
```

flash.c

```c
#include "flash.h"
#include "defines.h"
#include "misc.h"
#include <avr/io.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <util/delay.h>

#define NOR_BlockErase_Timeout          ((uint32_t)0x00A00000)
#define NOR_ChipErase_Timeout           ((uint32_t)0x30000000)
#define NOR_Program_Timeout             ((uint32_t)0x00001400)

#define NAND_BlockErase_Timeout          ((uint32_t)10000000)
#define NAND_Program_Timeout             ((uint32_t) 5000000)
#define NAND_Read_Timeout               ((uint32_t)   200000)

#define OSZI(val) bitWrite(&PORTD.OUT, 5, (val))

#define NOR_AM1(val) bitWrite(&PORTR.OUT, 1, (val & 0x01))
#define NOR_A7_A0(val) PORTF.OUT = ((val) & 0xff)
#define NOR_A11_A8(val) do{PORTC.OUTCLR=0xf0; PORTC.OUT |= ((val) & 0xf)<<4;} while (false)
#define NOR_A16_A12(val) do{PORTD.OUTCLR=0x1f; PORTD.OUT |= ((val) & 0x1f);} while (false)
```

```
#define NOR_RE(val) bitWrite(&PORTC.OUT, 0, (val))
#define NOR_WE(val) bitWrite(&PORTC.OUT, 1, (val))
#define NOR_DATA PORTA
#define NOR_BUSY (PORTR.IN & 0x01)

#define NAND_DATA PORTF
#define NAND_RE(val) bitWrite(&PORTA.OUT, 0, (val))
#define NAND_WE(val) bitWrite(&PORTA.OUT, 1, (val))
#define NAND_AE(val) bitWrite(&PORTA.OUT, 3, (val))
#define NAND_CE(val) bitWrite(&PORTA.OUT, 4, (val))
#define NAND_BUSY ((PORTA.IN >> 2) & 0x01)

static FlashType t = Undef;
static void fl_setAddress(uint32_t addr);
static void fl_setData(uint8_t data);
static uint8_t fl_getData();
static void fl_setCommand(uint8_t cmd);
static bool fl_isValidBlock(uint32_t addr);
bool fl_testReadWrite(uint32_t addr, char* testString, uint32_t stringLen)
    ↪ ;

bool fl_Init(FlashType type) {
  t = type;

  if (t == Nand_S || t == Nand_H) {   //NAND BOARD
    bitWrite(&PORTD .DIR, 5, 1);  //OSZI Pin

    // latched on the rise edge
    NAND_DATA.DIR = 0xff;      //D0 to D7
    NAND_DATA.PIN0CTRL = PORT_OPC_PULLUP_gc;
    NAND_DATA.PIN1CTRL = PORT_OPC_PULLUP_gc;
    NAND_DATA.PIN2CTRL = PORT_OPC_PULLUP_gc;
    NAND_DATA.PIN3CTRL = PORT_OPC_PULLUP_gc;
    NAND_DATA.PIN4CTRL = PORT_OPC_PULLUP_gc;
    NAND_DATA.PIN5CTRL = PORT_OPC_PULLUP_gc;
    NAND_DATA.PIN6CTRL = PORT_OPC_PULLUP_gc;
    NAND_DATA.PIN7CTRL = PORT_OPC_PULLUP_gc;

    bitWrite(&PORTA .DIR, 0, 1);  //Read Enable
    bitWrite(&PORTA .DIR, 1, 1);  //Write Enable
    bitWrite(&PORTA .DIR, 2, 0);  //Busy
    bitWrite(&PORTA .DIR, 3, 1);  //Address Latch Enable
    bitWrite(&PORTA .DIR, 4, 1);  //Command Latch Enable

    //initialize outputs
    NAND_DATA.OUT = 0;
    NAND_RE(1);
    NAND_WE(1);
    NAND_AE(0);
    NAND_CE(0);

  }

```

```
  if (t == Nor) {            //NOR BOARD
    bitWrite(&PORTD .DIR, 5, 1);  //OSZI Pin

    //OUTPUTS
    PORTF.DIR = 0xff;       //A0 to A7
    PORTC .DIR |= 0xf0;       //A8 to A11
    PORTD .DIR |= 0x1f;       //A12 to A16 as output
    bitWrite(&PORTR.DIR, 1, 1);   //A-1 (LSB of address)
    bitWrite(&PORTC .DIR, 0, 1);  //Output Enable (low to enable)
    bitWrite(&PORTC .DIR, 1, 1);  //Write Enable (low to enable)

    //IN-OUT
    NOR_DATA .DIR = 0xff;     //D0 to D7
    NOR_DATA .PIN0CTRL = PORT_OPC_PULLUP_gc;
    NOR_DATA .PIN1CTRL = PORT_OPC_PULLUP_gc;
    NOR_DATA .PIN2CTRL = PORT_OPC_PULLUP_gc;
    NOR_DATA .PIN3CTRL = PORT_OPC_PULLUP_gc;
    NOR_DATA .PIN4CTRL = PORT_OPC_PULLUP_gc;
    NOR_DATA .PIN5CTRL = PORT_OPC_PULLUP_gc;
    NOR_DATA .PIN6CTRL = PORT_OPC_PULLUP_gc;
    NOR_DATA .PIN7CTRL = PORT_OPC_PULLUP_gc;

    //INPUTS
    bitWrite(&PORTR.DIR, 0, 0); //Ready (busy on low, done after rising edge)
    PORTR.PIN0CTRL = PORT_OPC_PULLUP_gc;

    //initialize outputs
    OSZI(0);
    NOR_AM1(0);
    NOR_A7_A0(0);
    NOR_A11_A8(0);
    NOR_A16_A12(0);
    NOR_RE(1);
    NOR_WE(1);
    NOR_DATA .OUT = 0;

  }
  fl_Oszi(false);
  return fl_Reset();
}

/**
 * Performs needed write cycles to store buf on the flash memory starting at address
 * addr.
 * Returns status returned by flash memory.
 */
Status fl_Write(uint32_t addr, char* buf) {
  Status stat = Error;
  uint32_t buflen = strlen(buf);
  uint32_t i;

```

```
  if (strncmp("0x", buf, 2)==0){
    buflen=hexString2chars(buf);
    if (buflen==0){
      printf("ERROR! Could not convert hex-string to bytes!\n");
      return Error;
    }
  }

  if (t == Nand_S || t == Nand_H) {
    if (!fl_isValidBlock(addr)) {
      printf("ERROR! Invalid Block!\n");
      return Error;
    }
    fl_setCommand(0x80);
    fl_setAddress(addr);
    //fl_Oszi(false);
    //_delay_us(1);
    //fl_Oszi(true);    //1st action: write data to buffer
    //printf("fl_Write...\n");
    for (i = 0; i < buflen; i++) {
      fl_setData(buf[i]);
      //printf("0x%02x\n", buf[i]);
    }

    fl_Oszi(false);
    _delay_us(1);
    fl_Oszi(true);    //2nd action: persist data

    fl_setCommand(0x10);

    stat = fl_GetStatus(NAND_Program_Timeout );

    fl_Oszi(false);   //done
    return stat;

  }

  if (t == Nor) {
    stat = Success;

    //perform write
    for (i = 0; i < buflen; i++) {
      fl_Oszi(false);
      _delay_us(1);
      fl_Oszi(true);    //1st action: set up

      fl_WriteData(0xAAA, 0xAA);  //unlock 1
      fl_WriteData(0x555, 0x55);  //unlock 2
      fl_WriteData(0xAAA, 0xA0);  //program setup

      fl_Oszi(false);
      _delay_us(1);
      fl_Oszi(true);    //2nd action: persist
```

```
      fl_WriteData(addr + i, buf[i]); //program
      stat |= fl_GetStatus(NOR_Program_Timeout );

      fl_Oszi(false);   //done

      if (stat != Success) {
        return stat;
      }
    }
    return stat;
  }

  return fl_GetStatus(NOR_Program_Timeout | NAND_Program_Timeout ); //Plan C
}

/**
 * Performs a write cycle writing data to addr.
 */bool fl_WriteData(uint32_t addr, uint8_t data) {
  if (t == Undef)
    return false;

  if (t == Nand_S || t == Nand_H) {
    if (!fl_isValidBlock(addr)) {
      printf("ERROR! Invalid Block!\n");
      return Error;
    }

    fl_setCommand(0x80);
    fl_setAddress(addr);

    fl_setData(data);

    fl_setCommand(0x10);

    return fl_GetStatus(NAND_Program_Timeout ) == Success;

  }

  if (t == Nor) {
    //set address
    fl_setAddress(addr);
    //set data
    fl_setData(data);
    NOR_WE(0);
    NOR_WE(1);

  }

  return true;
}
```

```
/**
 * Sets the Address to write data to.
 */
static void fl_setAddress(uint32_t addr) {

  if (t == Nand_S) {
    NAND_AE(1);

    //first address cycle (Column address A0-A7)
    fl_setData(addr & 0xff);

    //second address cycle (Column address A8-A11)
    fl_setData((addr >> 8) & 0xf);

    //third address cycle (Row address A12-A19)
    fl_setData((addr >> 12) & 0xff);

    //fourth address cycle (Row address A20-A27)
    fl_setData((addr >> 20) & 0xff);

    NAND_AE(0);

  }

  if (t == Nand_H) {
    NAND_AE(1);

    //first address cycle (Column address A0-A7)
    fl_setData(addr & 0xff);

    //second address cycle (Column address A8-A13)
    fl_setData((addr >> 8) & 0x3f);

    //third address cycle (Page address A14-A21)
    fl_setData((addr >> 14) & 0xff);

    //fourth address cycle (Plane address A22,  Block address A23 - A29)
    fl_setData((addr >> 22) & 0xff);

    //fifth address cycle (Block address A30-A31)
    fl_setData((addr >> 30) & 0x3);

    NAND_AE(0);
  }

  if (t == Nor) {
    //set address
    //printf("LSB: %x, PortF: %x, PortC: %x, PortD: %x\n", addr & 0x1,(
        ↪ addr >> 1) & 0xff, ((addr >> 8) & 0xf) << 4, (addr >> 12) & 0
        ↪ x1f);

    NOR_AM1(addr & 0x1);
    NOR_A7_A0(addr >> 1);//don't forget the A-1 (A0 minus 1) in byte mode!
```

```
    NOR_A11_A8(addr >> 9);
    NOR_A16_A12(addr >> 13);

  }
}

/**
 * Sets the data to write
 */
static void fl_setData(uint8_t data) {

  if (t == Nand_S || t == Nand_H) {
    NAND_DATA.DIR = 0xff;
    NAND_DATA.OUT = data;
    NAND_WE(0);
    NAND_WE(1);
  }

  if (t == Nor) {
    NOR_DATA .DIR = 0xff; //Set as input
    NOR_DATA .OUT = data;
  }
}

/**
 * Reads readCount-1 bytes characters from memory into buf, where starting address is addr.
 * buf is always null-terminated -> buf[readCount-1]='\0'.
 * The number of read characters is returned.
 * On error, -1 is returned.
 */
uint32_t fl_Read(uint32_t addr, char* buf, uint32_t readCount) {
  buf[0] = '\0';
  uint32_t i = -1;

  if (t == Nand_S || t == Nand_H) {
    fl_setCommand(0x00);
    fl_setAddress(addr);

    fl_Oszi(false);
    _delay_us(1);
    fl_Oszi(true);     //1st action: reading

    fl_setCommand(0x30);

    uint32_t timeout = NAND_Read_Timeout;
    while ((NAND_BUSY != 0) && (timeout > 0)) {
      timeout--;
    }

    timeout = NAND_Read_Timeout;

    while ((NAND_BUSY == 0) && (timeout > 0)) {
```

```
      timeout--;
    }

    fl_Oszi(false);
    _delay_us(1);
    fl_Oszi(true);     //2nd action: fetching data

    if (timeout == 0) {
      printf("Reading timed out!!!!!\n");
      buf[0] = '\0';
      return -1;
    } else {

      for (i = 0; i < readCount; i++) {
        buf[i] = fl_getData();
      }
      buf[i] = '\0';
    }

    fl_Oszi(false);    //done

  }

  if (t == Nor) {

    for (i = 0; i < readCount; i++) {
      buf[i] = fl_ReadData(addr + i); //read data
    }
    buf[i] = '\0';

  }

  chars2hexString(buf, i);

  return i;
}

/**
 * Performs a read cycle returning the data stored at addr.
 */
uint8_t fl_ReadData(uint32_t addr) {
  uint8_t res = 0;
  if (t == Nand_S || t == Nand_H) {
    fl_setCommand(0x00);
    fl_setAddress(addr);

    fl_Oszi(false);
    _delay_us(1);
    fl_Oszi(true);     //1st action: reading

    fl_setCommand(0x30);

    uint32_t timeout = NAND_Read_Timeout;
```

```
    while ((NAND_BUSY != 0) && (timeout > 0)) {
      timeout--;
    }

    timeout = NAND_Read_Timeout;

    while ((NAND_BUSY == 0) && (timeout > 0)) {
      timeout--;
    }

    fl_Oszi(false);
    _delay_us(1);
    fl_Oszi(true);     //2nd action: fetching data

    res = fl_getData();

    fl_Oszi(false);    //done

  }

  if (t == Nor) {
    fl_setAddress(addr);

    fl_Oszi(false);
    _delay_us(1);
    fl_Oszi(true);     //1st action: reading

    NOR_RE(0);

    res = fl_getData();
    fl_Oszi(false);    //done

    NOR_RE(1);
  }

  return res;
}

/**
 * Sets data pins as input and returns data pin value.
 */
static uint8_t fl_getData() {
  uint8_t res = 0;

  if (t == Nand_S || t == Nand_H) {
    NAND_DATA.DIR = 0x00; //Set as input
    NAND_DATA.OUT = 0xff;
    NAND_RE(0);
    res = NAND_DATA.IN;
    res = NAND_DATA.IN;
    NAND_RE(1);

  }
```

```

  if (t == Nor) {
    NOR_DATA .DIR = 0x00; //Set as input
    NOR_DATA .OUT = 0xff;
    res = NOR_DATA .IN;
  }

  return res;
}

/**
 * Writes command cmd to command latch of nand.
 * Does nothing for nor and undefined.
 */
static void fl_setCommand(uint8_t cmd) {
  if (t == Nand_S || t == Nand_H) {
    NAND_CE(1);
    fl_setData(cmd);
    NAND_CE(0);
  }

}

/**
 * Checks whether block of nand is valid.
 * Returns true on valid block and false on invalid block, aswell as not nand type.
 */
static bool fl_isValidBlock(uint32_t addr) {
  uint32_t page, block, tmp;
  uint8_t val;

  if (t == Nand_S) {
    block = addr / ((2 * 1024 + 64) * 64L);
    page = (addr - block * (2 * 1024 + 64) * 64) / (2 * 1024 + 64);

    tmp = block * (2 * 1024 + 64) * 64 + 0L * (2 * 1024 + 64) + 2 * 1024;
    val = fl_ReadData(tmp);
    if (val != 0xff) {  //1st Byte in the spare area of 1st page
      printf(
          "Calculated block: %ld\npage: %ld first page (0x%08lx: 0x%02x)\n",
          block, page, tmp, val);
      return false;
    }

    tmp = block * (2 * 1024 + 64) * 64 + 1L * (2 * 1024 + 64) + 2 * 1024;
    val = fl_ReadData(tmp);
    if (val != 0xff) {  //1st Byte in the spare area of 2nd page
      printf(
          "Calculated block: %ld\npage: %ld second page (0x%08lx: 0x%02x)\n",
          block, page, tmp, val);
```

```
      return false;
    }

    return true;
  }

  if (t == Nand_H) {
    block = addr / (8 * 1024 + 448) / 256;
    page = (addr - block * (8 * 1024 + 448) * 256) / (8 * 1024 + 448);

    tmp = block * (8 * 1024 + 448) * 256 + 0L * (8 * 1024 + 448) + 8 *
    ↪ 1024;
    val = fl_ReadData(tmp);
    if (val != 0xff) {  //1st Byte in the spare area of 1st page
      printf(
          "Calculated block: %ld\npage: %ld first page (0x%08lx: 0x%02x)\n
          ↪ ",
          block, page, tmp, val);
      return false;
    }

    tmp = block * (8 * 1024 + 448) * 256 + 255L * (8 * 1024 + 448)
        + 8 * 1024;
    val = fl_ReadData(tmp);
    if (val != 0xff) {  //1st Byte in the spare area of last page
      printf(
          "Calculated block: %ld\npage: %ld last page (0x%08lx: 0x%02x)\n"
          ↪ ,
          block, page, tmp, val);
      return false;
    }

    return true;
  }

  return false;
}

void fl_Test() {
  uint32_t buflen=1024;
  char inBuf[buflen];
  char outBuf[buflen];
  uint32_t addr, i, slen;
  uint8_t val;
  int error=0;

  if (!fl_Reset()) {
    printf("Could not reset chip!\n");
  }

  if (t == Nand_S) {
    printf("### TESTING SAMSUNG NAND ###\n");
    fl_setCommand(0x90);
```

```
    fl_setAddress(0x00);
    printf("Manufacturer (0xEC): 0x%02X\n", fl_getData());
    printf("Device Code (0xF1): 0x%02X\n", fl_getData());
    printf("ID1 (0x00): 0x%02X\n", fl_getData());
    printf("ID2 (0x15): 0x%02X\n", fl_getData());
    printf("ID3 (0x40): 0x%02X\n", fl_getData());

    printf("CHECKING FOR BAD BLOCKS...\n");

    for (uint32_t block = 0; block < 1024; block++) {
      addr = block * (2 * 1024 + 64) * 64 + 0L * (2 * 1024 + 64)
          + 2 * 1024;
      val = fl_ReadData(addr);
      if (val != 0xff) {  //1st Byte in the spare area of 1st page
        printf("Invalid Block %ld at 0x%08lx\t\t(Start=0x%02x)\n",
            block, addr, val);
        continue;
      }

      addr = block * (2 * 1024 + 64) * 64 + 1L * (2 * 1024 + 64)
          + 2 * 1024;
      val = fl_ReadData(addr);
      if (val != 0xff) {  //1st Byte in the spare area of last page
        printf("Invalid Block %ld at 0x%08lx\t\t(End=0x%02x)\n", block,
            addr, val);
      }

    }
    printf("DONE!\n");

    printf("CHECKING ADDRESSES IN BLOCKS...\n");
    for (uint32_t block = 0; block < 1024; block++) {
      addr = block * (2 * 1024 + 64) * 64 + 5L * (2 * 1024 + 64) + 123;
      val = fl_ReadData(addr);
      if (!fl_isValidBlock(addr)) {//1st Byte in the spare area of 1st page
        printf("Invalid Block at address 0x%08lx\n", addr);
      }
    }
    printf("DONE!\n");

  }

  if (t == Nand_H) {
    printf("### TESTING HYNIX NAND ###\n");
    fl_setCommand(0x90);
    fl_setAddress(0x00);
    printf("Manufacturer (0xAD): 0x%02X\n", fl_getData());
    printf("Device Code (0xD5): 0x%02X\n", fl_getData());
    printf("ID1 (0x94): 0x%02X\n", fl_getData());
    printf("ID2 (0x9A): 0x%02X\n", fl_getData());
    printf("ID3 (0x74): 0x%02X\n", fl_getData());
    printf("ID3 (0x42): 0x%02X\n", fl_getData());
```

```
    printf("CHECKING FOR BAD BLOCKS...\n");

    for (uint32_t block = 0; block < 1024; block += 8) {
      addr = block * (8 * 1024 + 448) * 256 + 0L * (8 * 1024 + 448)
          + 8 * 1024;
      val = fl_ReadData(addr);
      if (val != 0xff) {  //1st Byte in the spare area of 1st page
        printf("Invalid Block %ld at 0x%08lx\t\t(Start=0x%02x)\n",
            block, addr, val);
      }

      addr = block * (8 * 1024 + 448) * 256 + 255L * (8 * 1024 + 448)
          + 8 * 1024;
      val = fl_ReadData(addr);
      if (val != 0xff) {  //1st Byte in the spare area of last page
        printf("Invalid Block %ld at 0x%08lx\t\t(End=0x%02x)\n", block,
            addr, val);
      }

    }
    printf("DONE!\n");

    printf("CHECKING ADDRESSES IN BLOCKS...\n");

    for (uint32_t block = 0; block < 1024; block += 8) {
      addr = block * (8 * 1024 + 448) * 256 + 15L * (8 * 1024 + 448)
          + 911;
      val = fl_ReadData(addr);
      if (!fl_isValidBlock(addr)) {//1st Byte in the spare area of 1st page
        printf("Invalid Block at address 0x%08lx\n", addr);
      }
    }
    printf("DONE!\n");

  }

  if (t == Nor) {

    fl_WriteData(0xAAA, 0xAA);  //unlock 1
    fl_WriteData(0x0555, 0x55); //unlock 2
    fl_WriteData(0x0AAA, 0x90); //enter autoselect
    printf("Manufacturer ID(0x01): 0x%02X\n", fl_ReadData(0x00));
    printf("Device ID1(0x7E): 0x%02X\n", fl_ReadData(0x02));
    printf("Device ID2(0x21): 0x%02x\n", fl_ReadData(0x1C));
    printf("Device ID3(0x01): 0x%02x\n", fl_ReadData(0x1E));
    fl_Reset();

    printf("CHECKING BLOCK DELETE...\n");
    addr = 0x1fffa;
    fl_Erase(addr); //Erase sector
    fl_Write(addr, "0123456789");
```

```
    fl_Read(addr, inBuf, 11);
    printf("Before delete: %s\n", inBuf);

    fl_Erase(0x0);  //Erase sector 0
    fl_Read(addr, inBuf, 11);
    printf("After delete of sector 0x0: %s\n", inBuf);

    fl_Write(addr, "0123456789");
    fl_Erase(0xffff); //Erase sector 0
    fl_Read(addr, inBuf, 11);
    printf("After delete of sector 0xffff: %s\n", inBuf);

    fl_Write(addr, "0123456789");
    fl_Erase(0x10000);  //Erase sector 0
    fl_Read(addr, inBuf, 11);
    printf("After delete of sector 0x10000: %s\n", inBuf);

    fl_Write(addr, "0123456789");
    fl_Erase(0x20000);  //Erase sector 1
    fl_Read(addr, inBuf, 11);
    printf("After delete of sector 0x20000: %s\n", inBuf);

  }

  printf("\n\nWriting test data...\n");
  addr = 0x0;

  printf("\nTesting printable ascii chars...\n");
  slen=(buflen-2)/2;
  for (i = 0; i < slen; i++) {
    outBuf[i] = 32 + i % 95; //printable ascii chars
  }
  if (fl_testReadWrite(addr, outBuf, slen)) {
    printf("TEST SUCCESS!\n");
  } else {
    printf("TEST FAILED!\n");
    error=1;
  }

  printf("\nTesting complete byte range...\n");
  for (i = 0; i < slen; i++) {
    outBuf[i] = 1 + (i%0x100); //ascii chars
  }
  if (fl_testReadWrite(addr, outBuf, slen)) {
    printf("TEST SUCCESS!\n");
  } else {
    printf("TEST FAILED!\n");
    error=2;
  }

  printf("\nTesting evil string #1...\n");
  uint32_t k = 0;
  for (i = 0; i < slen; i++) {
```

```c
    if (i % 4 == 0) {
      outBuf[i] = '0' + k % 10;
      k++;
    } else {
      outBuf[i] = 'o';
    }

  }
  if (fl_testReadWrite(addr, outBuf, slen)) {
    printf("TEST SUCCESS!\n");
  } else {
    printf("TEST FAILED!\n");
    error=3;
  }

  printf("\nTesting evil string #2...\n");
  memset(outBuf, 'o', 80);
  strcpy(outBuf+80, "0123456789abcdefghijklmnopqrstuvwxyz0123456789");
  memset(outBuf+80+46, 'o', 80);
  if (fl_testReadWrite(addr, outBuf, 2*80+46)) {
    printf("TEST SUCCESS!\n");
  } else {
    printf("TEST FAILED!\n");
    error=4;
  }

  fl_Erase(addr);

  printf("\n\nDONE!\n");

  if (error==0)
    printf ("ALL TESTS PASSED SUCCESSFULLY!");
  else
    printf("AT LEAST TEST %i FAILED! CHECK REPORT ABOVE!",error);
}

bool fl_testReadWrite(uint32_t addr, char* testString, uint32_t stringLen) {

  uint32_t i;
  Status stat;
  uint8_t val;
  bool result = true;

  char inBuf[2+2*stringLen];

  printf("Testing Read Write with string (%ld chars):\n%s\n", stringLen,
      testString);
  stat = fl_Erase(addr);
  if (stat != Success) {
    printf("Error(%i) erasing address 0x%08lx\n", stat, addr);
    return false;
  }
```

```

    chars2hexString(testString, stringLen);

    stat = fl_Write(addr, testString);
    if (stat != Success) {
      printf("Error(%i) writing to address 0x%08lx\n", stat, addr);
      return false;
    }

    fl_Read(addr, inBuf, stringLen);
    printf("Multi read:\n%s\n", inBuf);

    if (hexString2chars(inBuf)!=  stringLen){
      printf("Error! hexString2chars returned wrong number of bytes!");
      return false;
    }

    printf("Single read:\n");
    for (i = 0; i < stringLen; i++) {
      val = fl_ReadData(addr + i);
      printf("%c", val);

      if (((char) val) != testString[i]) {
        printf("\nMISSMATCH WITH SINGLE READ!!\n");
        printf("%8ld: %c 0x%02x\texpected: %c 0x%02x\n", i, val, val,
            testString[i], testString[i]);
        result = false;
      }
      if (inBuf[i] != testString[i]) {
        printf("\nMISSMATCH WITH MULTI READ!!\n");
        printf("%8ld: %c 0x%02x\texpected: %c 0x%02x\n", i, inBuf[i], inBuf[
           ↪ i],
            testString[i], testString[i]);
        result = false;
      }
    }

    putchar('\n');

    return result;
  }

  bool fl_Reset() {
    if (t == Nand_S || t == Nand_H) {
      fl_setCommand(0xff);
    }

    if (t == Nor) {
      fl_WriteData(0x0, 0xF0);
    }
    return fl_GetStatus(100) == Success;
  }

```

```
Status fl_Erase(uint32_t addr) {
  if (t == Nand_S || t == Nand_H) {

    if (addr == -1) {
      for (uint32_t block = 0; block < 1024; block++) {
        if (t == Nand_S) {
          addr = block * (2 * 1024 + 64) * 64 + 0L * (2 * 1024 + 64)
              + 2 * 1024;
        }
        if (t == Nand_H) {
          addr = block * (8 * 1024 + 448) * 256
              + 0L * (8 * 1024 + 448) + 8 * 1024;
        }

        if (fl_Erase(addr) != Success) {
          printf("Warning! Could not erase block %ld\n", block);
        }

      }
      return Success;

    } else {
      if (!fl_isValidBlock(addr)) {
        printf("ERROR! Invalid Block!\n");
        return Error;
      }

      fl_setCommand(0x60);

      NAND_AE(1);
      //set block addresses
      if (t == Nand_S) {
        //third address cycle (Row address A12-A19)
        fl_setData((addr >> 12) & 0xff);

        //fourth address cycle (Row address A20-A27)
        fl_setData((addr >> 20) & 0xff);
      }

      if (t == Nand_H) {
        //third address cycle (Page address A14-A21)
        fl_setData((addr >> 14) & 0xff);

        //fourth address cycle (Plane address A22,  Block address A23 - A29)
        fl_setData((addr >> 22) & 0xff);

        //fifth address cycle (Block address A30-A31)
        fl_setData((addr >> 30) & 0x3);
      }

      NAND_AE(0);
    }
```

```
    fl_setCommand(0xd0);
    return fl_GetStatus(NAND_BlockErase_Timeout );
  }

  if (t == Nor) {
    //Sector border is from 0x1ffff to 0x20000
    fl_WriteData(0xAAA, 0xAA);  //unlock 1
    fl_WriteData(0x555, 0x55);  //unlock 2
    fl_WriteData(0xAAA, 0x80);
    fl_WriteData(0xAAA, 0xAA);
    fl_WriteData(0x555, 0x55);
    if (addr == -1) {
      fl_WriteData(0xAAA, 0x10);
      return fl_GetStatus(NOR_ChipErase_Timeout );
    } else {
      fl_WriteData(addr, 0x30); //erase sector
      return fl_GetStatus(NOR_BlockErase_Timeout );
    }

  }

  return fl_GetStatus(NOR_ChipErase_Timeout | NAND_BlockErase_Timeout );
      ↪ //Plan C
}

Status fl_GetStatus(uint32_t Timeout) {
  Status status = Busy;
  uint32_t timeout = Timeout;

  if (t == Nand_S || t == Nand_H) {
    while ((NAND_BUSY != 0) && (timeout > 0)) {
      timeout--;
    }
    timeout = Timeout;

    while ((NAND_BUSY == 0) && (timeout > 0)) {
      timeout--;
    }
    uint8_t reg; //status register content
    while (status == Busy && Timeout > 0) {
      Timeout--;
      fl_setCommand(0x70);
      reg = fl_getData(); //get status register content

      if ((reg & 0x1) == 1) {
        status = Error;

      } else if (((reg >> 6) & 0x1) == 1) {
        status = Success;
      } else {
        status = Busy;
      }
```

```c

    }

    if (timeout == 0) {
      status = TimeOut;
    }
  }

  if (t == Nor) {
    uint8_t val1 = 0x00, val2 = 0x00;

    /* Poll on NOR memory Ready/Busy signal -----------------------------------*/
    while ((NOR_BUSY != 0) && (timeout > 0)) {
      timeout--;
    }

    timeout = Timeout;

    while ((NOR_BUSY == 0) && (timeout > 0)) {
      timeout--;
    }

    /* Get the NOR memory operation status ------------------------------------*/
    while ((Timeout != 0x00) && (status != Success)) {
      Timeout--;

      /* Read DQ6 and DQ5 */
      val1 = fl_getData();
      val2 = fl_getData();

      /* If DQ6 did not toggle between the two reads then return NOR_Success */
      if ((val1 & 0x0040) == (val2 & 0x0040)) {
        return Success;
      }

      if ((val1 & 0x0020) != 0x0020) {
        status = Busy;
      }

      val1 = fl_getData();
      val2 = fl_getData();

      if ((val1 & 0x0040) == (val2 & 0x0040)) {
        return Success;
      } else if ((val1 & 0x0020) == 0x0020) {
        return Error;
      }
    }

    if (Timeout == 0x00) {
```

```
      status = TimeOut;
    }

    /* Return the operation status */

  }

  return status;
}

void fl_Oszi(bool enable){
  if (enable){
    OSZI(1);
  }else{
    OSZI(0);
  }
}
```

uart.h

```
#ifndef MY_LITTLE_CUSTOM_UART_H
#define MY_LITTLE_CUSTOM_UART_H

#include <stdio.h>

int uartPutchar(char c, FILE *stream);
int uartGetchar(FILE *stream);

FILE uartInit();

#endif
```

uart.c

```
#include "uart.h"
#include "defines.h"
#include <avr/io.h>
#include <stdio.h>
#include <stdbool.h>

//based on http://www.appelsiini.net/2011/simple-usart-with-avr-libc

FILE uartInit() {
  USART_PORT.DIRSET = PIN3_bm;    //Pin 3 (TXC0) as output.
  USART_PORT.DIRCLR = PIN2_bm;    //Pin 2 (RXC0) as input.

  USART.CTRLC = (uint8_t) USART_CHSIZE_8BIT_gc | USART_PMODE_DISABLED_gc
      | false; //8 Data bits, No Parity, 1 Stop bit

  //Set baud rate
```

```
  USART.BAUDCTRLA = (USART_BSEL & 0xff);
  USART.BAUDCTRLB = (USART_BSCALE << USART_BSCALE_gp)
      | ((USART_BSEL >> 8) & 0x0f);

  //Set USART to asynchronous (UART)
  //0x00 asynchronous
  //USART.CTRLC = (USART.CTRLC & (~USART_CMODE_gm)) | 0x00;

  //Enable both RX and TX
  USART.CTRLB |= USART_RXEN_bm;
  USART.CTRLB |= USART_TXEN_bm;

  FILE stream = FDEV_SETUP_STREAM(uartPutchar, uartGetchar, _FDEV_SETUP_RW
    ↪ );

  return stream;
}

int uartPutchar(char c, FILE *stream) {
  loop_until_bit_is_set(USART.STATUS, USART_DREIF_bp);
  USART.DATA = c;
  if (c=='\n')
    uartPutchar('\r', NULL);
  return 0;
}

int uartGetchar(FILE *stream) {
  loop_until_bit_is_set(USART.STATUS, USART_RXCIF_bp);
  char c = USART.DATA;
  if (c == '\r')
    c = '\n';

  return c;
}
```

misc.h

```
#ifndef MY_LITTLE_CUSTOM_MISC_H
#define MY_LITTLE_CUSTOM_MISC_H

#include <stdint.h>
#include <stdbool.h>
#include <stdlib.h>


void setClockTo32MHz();
void in(char* buf, uint16_t len);
void out(char* buf);
void bitWrite(volatile unsigned char* reg, uint8_t pos, bool val);
uint32_t hexString2chars(char* buf);
void chars2hexString(char* buf, uint32_t len);

```

```
#endif
```

misc.c

```
#include "misc.h"
#include "defines.h"
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>

void setClockTo32MHz() {
  unsigned char n, s;

  // Save interrupts enabled/disabled state
  s = SREG;
  // Disable interrupts
  cli();

  // Internal 32/1 = 32MHz  RC oscillator intern
  // Enable the internal 32 MHz RC oscillator
  OSC.CTRL |= OSC_RC32MEN_bm;

  // System Clock prescaler A division factor: 1
  // System Clock prescalers B & C division factors: B:1, C:1
  // ClkPer4: 32000,000 kHz
  // ClkPer2: 32000,000 kHz
  // ClkPer:  32000,000 kHz
  // ClkCPU:  32000,000 kHz
  n = (CLK.PSCTRL & (~(CLK_PSADIV_gm | CLK_PSBCDIV1_bm | CLK_PSBCDIV0_bm))
    ↪ )
      | CLK_PSADIV_1_gc | CLK_PSBCDIV_1_1_gc;
  CCP = CCP_IOREG_gc;
  CLK.PSCTRL = n;

  // Enable the autocalibration of the internal 32 MHz RC oscillator
  OSC.CTRL |= OSC_RC32MEN_bm;

  // Wait for the internal 32 MHz RC oscillator to stabilize
  while ((OSC.STATUS & OSC_RC32MRDY_bm) == 0)
    ;

  // Select the system clock source: 32 MHz Internal RC Osc.
  n = (CLK.CTRL & (~CLK_SCLKSEL_gm)) | CLK_SCLKSEL_RC32M_gc;
  CCP = CCP_IOREG_gc;
  CLK.CTRL = n;

  // Disable the unused oscillators: 2 MHz, internal 32 kHz, external
    ↪ clock/crystal oscillator, PLL
```

```
  OSC.CTRL &=
      ~(OSC_RC2MEN_bm | OSC_RC32KEN_bm | OSC_XOSCEN_bm | OSC_PLLEN_bm);

  // Peripheral Clock output: Disabled
  PORTCFG.CLKEVOUT = (PORTCFG.CLKEVOUT & (~PORTCFG_CLKOUT_gm))
      | PORTCFG_CLKOUT_OFF_gc;

  // Restore interrupts enabled/disabled state
  SREG = s;

}


void in(char* buf, uint16_t len) {
  uint16_t i;
  char c;

  for (i = 0, c = getchar(); i < len && c != '\n' && c != EOF; i++, c =
      getchar()) {
    buf[i] = c;
    putchar(c);
  }

  if (c == '\n')
    putchar(c);

  buf[i] = '\0';
}

void out(char* buf) {

  for (uint16_t i = 0; buf[i] != '\0'; i++) {
    putchar(buf[i]);
  }
}

void bitWrite(volatile unsigned char* reg, uint8_t pos, bool val) {
  *reg = (*reg & ~(1<<pos)) | (val<<pos);
}

uint32_t hexString2chars(char* buf){
  long tmpHexLong=0;
  char* tmpHexBuf="0xAA";
  size_t buflen = strlen(buf);
  size_t i=0;

  if (strncmp("0x", buf, 2)!=0){
    return 0; //error
  }

  if(buflen%2!=0){
    return 0;
  }
```

```
  buflen=(buflen-2)/2;

  for (i=0; i<buflen; i++){
    memcpy(tmpHexBuf+2, buf+2+i*2, 2);
    tmpHexLong=strtol(tmpHexBuf, NULL, 0);
    buf[i] = (char) tmpHexLong;
  }
  buf[i]='\0';

  return i;

}

void chars2hexString(char* buf, uint32_t len){
  uint32_t i = len-1;
  char tmpHexBuf[3]="AA";

  if (strncmp("0x", buf, 2)==0){  //already hex-string
      return ;
  }

  buf[2+2*len]='\0';

  for (i=0; i<len; i++){
    sprintf(tmpHexBuf, "%02x", (unsigned char) buf[len-1-i]);
    memcpy(buf+2+(len-1-i)*2, tmpHexBuf, 2);
  }

  memcpy(buf, "0x", 2);

  return;
}
```

Acquisition

measure.py

```
import binascii
import datetime
import math
import os
import sys
import time

import csv
import matplotlib.pyplot as plt
import serial
import visa


MEASUREPATH = "./measurements";
```

```
if not os.path.exists(MEASUREPATH):
      os.makedirs(MEASUREPATH)

def init():
  ser = serial.Serial('COM3', 9600, timeout = 1)
  oszi = ""
  rm = visa.ResourceManager()
  nmbrDevices = len(rm.list_resources())
  if nmbrDevices == 0:
    raise IOError("Could not find oszi!")
  elif nmbrDevices == 1:
    oszi = rm.get_instrument(rm.list_resources()[0])
  else:
    print("Found VISA devices:")
    while not oszi:
      for dev in rm.list_resources():
        print("\t", dev)
        select = input("Want to use this device? (y/n): ")
        if select.lower() == "y":
          oszi = rm.get_instrument(dev)
          break
      if not oszi:
        print("No more devices found! Please choose one!")


  oszi.timeout = 10
  print("Oszi ID: " + oszi.ask("*IDN?"));

  print("Running oszi self test...")
  oszi.write("*TST?")
  tmp = ""
  while tmp == "":
    tmp = oszi.read()

  if int(tmp) != 0:
    raise  IOError("ERROR Oszi self test failed!")

  print("Resetting oszi to default...")
  oszi.write("*RST")

  print("INIT COMPLETE!")

  return ser, rm, oszi


def readSerial(ser):
  global debug
  output = []
  errorCount = 0
  while True:

    line = bytes.decode(ser.readline(), errors = "ignore")

```

```
    if debug:
      print("\nDebug - received serial:\n\t" + line)

    if (line == ""):
      if (errorCount > 10):
        output.append("ERROR! Received only empty lines!")
      else:
        errorCount += 1


    if line.endswith("> "):
      break

    line = line.lstrip().rstrip()
    output.append(line)

  return "\n".join(output)


def sendSerial(ser, line):
  global debug
  ser.write(line + "\r")
  if debug:
    print("Debug - sent serial: " + line)

  return


def cmdSerial(ser, line):
  sendSerial(ser, line)
  output = readSerial(ser)

  return output

def setOszi(oszi, mVPerDiv = 50, nsPerDiv = 10000, mVTriggerLevel = 2000, measurePoints = 50000):

  oszi.write(":CHANnel1:DISPlay ON")
  oszi.write(":CHANnel1:LABel \"Signal\"")
  oszi.write(":CHANnel1:IMPedance FIFTy")
  oszi.write(":CHANnel1:SCALe " + str(mVPerDiv) + "mV");
  oszi.write(":CHANnel2:DISPlay ON")
  oszi.write(":CHANnel2:LABel \"Trigger\"")
  oszi.write(":CHANnel2:IMPedance ONEMeg")
  oszi.write(":CHANnel2:SCALe 2V")
  oszi.write(":TIMebase:REFerence LEFT ")
  oszi.write(":TIMebase:SCALe " + str(nsPerDiv) + "E-9")
  oszi.write(":TRIGger:EDGE:SLOPe Positive")
  oszi.write(":TRIGger:EDGE:SOURce CHANnel2")
  oszi.write(":TRIGger:EDGE:LEVel " + str(mVTriggerLevel) + "E-3")

  oszi.write(":WAVeform:FORMat ASCii")
  oszi.values_format = ascii
```

```python
    oszi.write(":WAVeform:SOURce CHAN1")
    oszi.write(":WAVeform:POINTS " + str(measurePoints))

    return

def armOszi(oszi):

    oszi.write(":SINGle")
    time.sleep(1)

    return

def fetchData(measurementID, oszi, fig):

    times = []
    vals = []

    print("Reading oszi data")

    line = "init"
    oszi.write(":WAVeform:DATA?")
    while (line[-1:] != "\n" and line != ""):
      print('.', end = "", flush = True)
      line = oszi.read_raw()
      line = line.decode('ascii')
      vals.append(line)

    print("")

    if (line == ""):
      print("ERROR: Received empty line!")

    vals = "".join(vals)[:-1]

    # strip header information
    if vals[0] == "#":
        l = vals[1]
        vals = vals[2 + int(l):]

    vals = vals.split(",")


    with open (MEASUREPATH + "/" + measurementID + "_stripped.txt", "w",
        ↪ newline = "") as f:
      writer = csv.writer(f, "excel", delimiter = ";")
      data = []
      data.append([str(v).replace(".", ",") for v in vals])
      writer.writerows(data)


    pamble = oszi.ask(":WAVeform:PREamble?").split(",")
    fmt, typ, points, count, xinc, xorig, xref, yinc, yorig, yref = [float(x
        ↪ ) for x in pamble]
```

```
  points = int(points)

  f = open(MEASUREPATH + "/" + measurementID + "_preamble.txt", "w")
  for x in pamble:
      _ = f.write(str(x) + "\n")

  f.close()


  for i in range(len(vals)):
    vals[i] = (float(vals[i]))
    time = (i - xref) * xinc + xorig;
    times.append(time)

  with open (MEASUREPATH + "/" + measurementID + ".csv", "w", newline = "" ) as f:
    writer = csv.writer(f, "excel", delimiter = ";")
    writer.writerow(["val", "time"])
    data = []
    data.append([str(f).replace(".", ",") for f in vals])
    data.append([str(f).replace(".", ",") for f in times])
    writer.writerows(zip(*data))

  # Save plot as image file
  tmp = plt.figure()
  plt.plot(times, vals)
  tmp.savefig(MEASUREPATH + "/" + measurementID + ".png")  # ".pdf"
  plt.close(tmp)

  plt.figure(fig.number)

  plt.plot(times, vals)

  if (len(vals) != points):
    print("ERROR! INCORRECT AMOUNT OF POINTS RETURNED!");

  print(measurementID + " DONE!")


  return

def flashInit(ser, flashType):
  global debug

  cmdSerial(ser, "t " + flashType)
  output = cmdSerial(ser, "test")

  if "TEST FAILED!" in output:
    if debug:
      print(output)
    print(">>>> ERROR RUNNING INIT TEST!")

    return
```

```python

  print("System initialized successfully!")

  return

def flashReadWrite(ser, oszi, chip, action, string, nrMeasurements, mVPerDiv = 50, nsPerDiv = 10000, mVTriggerLevel = 2000, measurePoints = 50000, showPlots = True):
  print("Clearing chip...")
  output = cmdSerial(ser, "d")
  if "Error" in output:
    print("ERROR deleting chip!")
    return

  print("Testing write...")
  output = cmdSerial(ser, "w 0 " + string)
  if "Error" in output:
    print("ERROR writing test data!")
    return


  stringlen = len(string)
  if string.startswith("0x"):
    stringlen = int((stringlen - 2) / 2)
    expect = "data: " + string.lower()
  else:
    expect = "data: 0x" + binascii.hexlify(string.encode()).decode()

  output = cmdSerial(ser, "r 0 " + str(stringlen))
  if not output.endswith(expect):
      got = output[output.rfind("data: "):]
      print("ERROR checking test data!\nexpected: " + expect + "\ngot:" + got)
      return


  errorCount = 0
  if len(string) > 10:
    measurementID = action + " '" + string[:10] + "...' " + str(stringlen) + " " + datetime.datetime.now().strftime("%Y.%m.%d-%H_%M_%S")
  else:
    measurementID = action + " '" + string + "' " + str(stringlen) + " " + datetime.datetime.now().strftime("%Y.%m.%d-%H_%M_%S")

  logEntry = (
        measurementID +
        "\t\t" + datetime.datetime.now().strftime("%Y.%m.%d-%H_%M_%S") +
        "\t\t" + chip +
        "\t\t" + action)

  if string.startswith("0x"):  # hex-string comes before plain text
    logEntry += (
      "\t\t" + string +
```

```
      "\t\t")
    logEntry = logEntry.encode() + binascii.unhexlify(string[2:])
  else:
    logEntry = (
      logEntry.encode() +
      b"\t\t0x" + binascii.hexlify(string.encode()) +
      b"\t\t" + string.encode())

  with open(MEASUREPATH + "/measure.log", "ab") as log:
    log.write(logEntry + b"\n")

  # setup Oszi
  setOszi(oszi, mVPerDiv, nsPerDiv, mVTriggerLevel, measurePoints)

  time.sleep(3)
  print("Starting measurement...")
  figWidth = math.floor(math.sqrt(nrMeasurements))
  if (figWidth * figWidth % nrMeasurements == 0):
    figHeight = figWidth
  else:
    figHeight = figWidth + 1

  if figWidth * figHeight < nrMeasurements:
    figWidth += 1

  fig = plt.figure()

  for i in range(nrMeasurements):

    # load oszi
    armOszi(oszi)

    if action == "r":
      output = cmdSerial(ser, "r 0 " + str(stringlen))
      if not output.endswith(expect):
        errorCount += 1
        got = output[output.rfind("data: "):]
        print("ERROR reading data on #" + str(i) + "\nexpected: " + expect + "\ngot:" + got)
        if errorCount > 2:
          print("ERROR more than 3 invalid reads, aborting...")
          return

    if action == "w":
      addr = (i + 1) * stringlen
      output = cmdSerial(ser, "w " + str(addr) + " " + string)
      if "Error" in output:
        errorCount += 1
        print("ERROR writing data on #" + str(i) + ": " + output)
        if errorCount > 2:
          print("ERROR more than 3 invalid writes, aborting...")
          return

```

```python
    # read & save data from Oszi
    fig.add_subplot(figHeight, figWidth, i + 1)
    fetchData(measurementID + "#" + str(i), oszi, fig)

  print("\nMeasurements completed successfully!")
  if showPlots:
    plt.show()

  return

def printHeader():
  print("###################################################")
  print("# Side Channel Analysis of Flash Memory          #")
  print("# Data Grabber version 0.1                       #")
  print("# Markus Hannes Fischer   1029057                #")
  print("###################################################")

  return


def printHelp():
  print("Commands:")
  print("\ti <nand_s | nand_h | nor> - initialize flash controller to Samsung nand, Hynix nand or nor chip")
  print("\tw <# of measurements> <string to be written>")
  print("\tr <# of measurements> <string to be written>")
  print("\td - toggles debug mode")
  print("\to [<mVPerDiv> <nsPerDiv> <mVTriggerLevel> <measurePoints>] - sets values for oszi\n\t\tto default or given values")
  print("\te - exits the program")
  print("\teverything else prints this command list")

  return


################### MAIN ######################
def main():
  global debug

  debug = True


  printHeader()

  ser, rm, oszi = init()

  if len(sys.argv) > 1:   # scripted mode
    debug = False
    if len(sys.argv) > 2:
      MEASUREPATH = str(sys.argv[2])

    with open(str(sys.argv[1])) as fin:
      lines = fin.read().splitlines()
```

```
    linecount = 0
    for l in lines:
      linecount += 1
      if l.startswith(";") or not l:
        continue

      elif l.startswith("config:"):
        chip = str(l[7:].split(";")[0])
        mVPerDiv, nsPerDiv, mVTriggerLevel, measurePoints = [int(x) for x
            ↪ in l[7:].split(";")[1:]]
        if chip not in ["nand_s", "nand_h", "nor"]:
          print("Invalid chip config found in config on line " + str(
              ↪ linecount) + "!")

        flashInit(ser, chip);
        setOszi(oszi, mVPerDiv, nsPerDiv, mVTriggerLevel, measurePoints)
      else:
        tmp = l.split(" ")

        action = tmp[0]
        if action not in ["r", "w"]:
          print("Invalid operation found on line " + str(linecount) + "!")
          continue

        count = tmp[1]
        if not count.isdigit():
          print("Invalid number of measurements found on line " + str(
              ↪ linecount) + "!")
          continue

        count = int(count)
        string = " ".join(tmp[2:])

        flashReadWrite(ser, oszi, chip, action, string, count, mVPerDiv,
            ↪ nsPerDiv, mVTriggerLevel, measurePoints, False)


  else:  # interactive mode

    while True:
      if debug:
        print("DEBUG ON", end = " ")

      line = input("> ")

      cmd = line.split(" ")

      if debug:
        print("Debug - you entered: ", cmd)

      if cmd[0] == "i":
        if len(cmd) != 2 or cmd[1] not in ["nand_s", "nand_h", "nor"] :
```

```
          printHelp()
          continue

        chip = cmd[1]
        flashInit(ser, chip);


      elif cmd[0] == "r" or cmd[0] == "w":
        if (len(cmd) < 3):
          printHelp()
          continue
        action = cmd[0]
        count = cmd[1]
        if not count.isdigit():
          printHelp()
          continue


        string = " ".join(cmd[2:])
        count = int(count)
        flashReadWrite(ser, oszi, chip, action, string, count)

      elif cmd[0] == "e":
        print ("Exiting...")
        break

      elif cmd[0] == "d":
        debug = not debug
        if debug:
          print("Debug is now ON!")
        else:
          print("Debug is now OFF!")

      elif cmd[0] == "o":
        if len(cmd) != 5 and len(cmd) != 1:
          printHelp()
          continue
        if len(cmd) == 1:
          setOszi(oszi)
        else:
          try:
            mVPerDiv, nsPerDiv, mVTriggerLevel, measurePoints = [float(x) for x in cmd[1:]]
          except:
            print("Could not convert all values to floats!")
            continue

          setOszi(oszi, mVPerDiv, nsPerDiv, mVTriggerLevel, measurePoints)
        print("Oszi set!")

      else:
        printHelp()

```

```
    oszi.close()
    return


if __name__ == "__main__":
    main()
```

batch_measure.txt

```
; test batch measurement file
; ";" is a comment line and ignored
; empty lines are ignored
; config lines start with "config:" e.g.
; config:chip;mVPerDiv;nsPerDiv;mVTriggerLevel;measurePoints
; measurements are instructed as <w|r> <# measurements> <string used for
    ↪ measurement> e.g.
; w 10 measure this
; r 50 this will take some time

config:nand_s;50;10000;2000;50000

; test length for normal (half of the bits set on different positions)
    ↪ bytes
w 10 0x5B
w 10 0x5B5B5B
w 10 0x5B5B5B5B5B
w 10 0x5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B
w 10 0x5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5
    ↪ B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B
    ↪ 5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B

; combine bit wandering lower nibble & checking hemming weight
w 10 0x00000000000000000000
w 10 0x01010101010101010101
w 10 0x02020202020202020202
w 10 0x03030303030303030303
w 10 0x04040404040404040404
w 10 0x05050505050505050505
w 10 0x06060606060606060606
w 10 0x07070707070707070707
w 10 0x08080808080808080808
w 10 0x09090909090909090909
w 10 0x0a0a0a0a0a0a0a0a0a0a
w 10 0x0b0b0b0b0b0b0b0b0b0b
w 10 0x0c0c0c0c0c0c0c0c0c0c
w 10 0x0d0d0d0d0d0d0d0d0d0d
w 10 0x0e0e0e0e0e0e0e0e0e0e
w 10 0x0f0f0f0f0f0f0f0f0f0f

; bit wandering upper nibble
w 10 0x10101010101010101010
w 10 0x20202020202020202020
```

```
w 10 0x40404040404040404040
w 10 0x80808080808080808080
w 10 0xf0f0f0f0f0f0f0f0f0f0
w 10 0xffffffffffffffffffff

; effect of different byte location
w 10 0x5B5B5B5B5B5B5B5B5B5B
w 10 0x005B5B5B5B5B5B5B5B5B
w 10 0x5B005B5B5B5B5B5B5B5B
w 10 0x5B5B005B5B5B5B5B5B5B
w 10 0x5B5B5B5B5B5B5B5B5B00

; effect of number of successive 00 bytes
w 10 0x00ffffffffffffffffff
w 10 0x0000ffffffffffffffff
w 10 0x000000ffffffffffffff
w 10 0x0000000000000000ffff

;check if ff bytes are just skipped
w 10 0x00ff00ff00ff00ff00ff
w 10 0x0000000000
```

Preprocessing

process.py

```
import csv
import datetime
import locale
import math
import os
import re
import time

from scipy.signal import argrelextrema
import matplotlib.pyplot as plt
import numpy as np



MEASUREPATH = "./measurements"

def readData(measurement, files):

  vals = []
  times = []

  for file in files:
    with open (MEASUREPATH + "/" + file, "r", newline = "") as f:
      reader = csv.reader(f, "excel", delimiter = ";")
      header = next(reader)
      if (header[0] != "val" or header[1] != "time"):
```

```
            raise Exception("csv heading is not in format <val; time>!")

        v = []
        t = []

        for row in reader:
            v.append(float(row[0].replace(",", ".")))
            t.append(float(row[1].replace(",", ".")))

        vals.append(v)
        times.append(t)

    return (measurement, times, vals)


def filterData(times, vals):
    offsets = []

    npTimes = np.array(times)
    npVals = np.array(vals)

    meanVals = npVals[0]
    meanTimes = npTimes[0]
    thresh = np.median([np.max(vals[i]) for i in range(len(vals))]) / 4

    for i in range(len(npVals)):
        a = 0
        try:
            while npVals[i][a] < thresh:
                a += 1
        except:  # threshold is never reached -> faulty measurement
            print("ERROR: Could not find threshold for measurement #" + str(i) + "!\nDiscarding measurement")
            a = -1
            continue
        finally:
            offsets.append(a)

        print("offset for #" + str(i) + ": " + str(a))

        meanVals[a:] = [(meanVals[b] + npVals[i][b]) / 2 for b in range(a, len(npVals[0]))]
        meanTimes = [(meanTimes[b] + npTimes[i][b]) / 2 for b in range(len(npTimes[0]))]

    # truncate means
    trunc = min(offsets)
    meanVals = meanVals[trunc:]
    meanTimes = meanTimes[trunc:]

    return (meanTimes, meanVals, offsets)
```

```python

def plotData(times, vals, meanTimes, meanVals, offsets):

  fig1 = plt.figure()
  fig2 = plt.figure()

  for i in range(len(vals)):
    if (offsets[i] < 0):  # skip invalid measurements
      continue

    plt.figure(fig1.number)

    fig1.add_subplot(5, 2, i)
    plt.plot(times[i], vals[i])

    plt.figure(fig2.number)
    fig2.add_subplot(2, 1, 1)
    plt.plot(times[i][offsets[i]:], vals[i][offsets[i]:])


  plt.figure(fig2.number)
  fig2.add_subplot(2, 1, 2)
  plt.plot(meanTimes, meanVals)

  plt.show()

  return

def writeData(measurement, times, vals):
  with open (MEASUREPATH + "/" + measurement + " mean.csv", "w", newline = "") as f:
    writer = csv.writer(f, "excel", delimiter = ";")
    writer.writerow(["val", "time"])
    data = []
    data.append([str(f).replace(".", ",") for f in vals])
    data.append([str(f).replace(".", ",") for f in times])
    writer.writerows(zip(*data))

    tmp = plt.figure()
    plt.plot(times, vals)
    plt.savefig(MEASUREPATH + "/" + measurement + " mean" + ".png")  # ".pdf"
    plt.close(tmp)

  return

def main():
  vals = []
  times = []
  meanVals = []
  meanTimes = []
  measurement = ""
  offsets = []
```

```python
  files = []


  for (dirpath, dirnames, filenames) in os.walk(MEASUREPATH):
      files.extend(filenames)

  measurements = [s[:-6] for s in files if re.search(".*#0\\.csv", s)]
  measurements = [f for f in measurements if f+" mean.csv" not in files] #remove already averaged measurements
  measurements = [(m, [f for f in files if f.startswith(m + "#") and f.endswith(".csv")]) for m in measurements ]

  for i in range(len(measurements)):
    print("processing " + measurements[i][0] + "\t#measurements: " + str(len(measurements[i][1])))

    measurement = measurements[i][0]
    files = measurements[i][1]

    (measurement, times, vals) = readData(measurement, files)
    (meanTimes, meanVals, offsets) = filterData(times, vals)
    writeData(measurement, meanTimes, meanVals)

  if len(measurements) == 1:
    plotData(times, vals, meanTimes, meanVals, offsets)


  return

if __name__ == "__main__":
  main()
```

Analysis

analysis.py

```python
import csv
import sys
import matplotlib.pyplot as plt

def readData(file):

  with open (file, "r", newline = "") as f:
    reader = csv.reader(f, "excel", delimiter = ";")
    header = next(reader)
    if (header[0] != "val" or header[1] != "time"):
      raise Exception("csv heading is not in format <val; time>!")

    v = []
    t = []

    for row in reader:
```

```
      v.append(float(row[0].replace(",", ".")))
      t.append(float(row[1].replace(",", ".")))

  return [t,v]



def writeData(filename, times, vals):
  with open (filename + ".csv", "w", newline = "") as f:
    writer = csv.writer(f, "excel", delimiter = ";")
    writer.writerow(["val", "time"])
    data = []
    data.append([str(f).replace(".", ",") for f in vals])
    data.append([str(f).replace(".", ",") for f in times])
    writer.writerows(zip(*data))

    tmp = plt.figure()
    plt.plot(times, vals)
    plt.savefig(filename + ".png")   # ".pdf"
    plt.close(tmp)

  return


def diff(t1, t2, showPlot=False):
  data=[t1,t2]
  lens = [len(tmp[0]) for tmp in data]
  lensmin=min(lens)
  for tmp in data:
    tmp[0]=tmp[0][:lensmin]
    tmp[1]=tmp[1][:lensmin]

  tmp=[]
  for i in range(len(t1[1])):
    tmp.append(t1[1][i]-t2[1][i])

  diff=[t2[0],tmp]

  if showPlot:
    plt.subplot(211)
    plt.plot(*t1, color="blue", label="0x00")
    plt.plot(*t2, color="red", label="0xff")

    plt.subplot(212)
    plt.plot(*diff, color="black", label="diff")

    plt.show()

  return diff



def main():
```

```
  if len(sys.argv) != 4:
    print("Usage: python3.4 "+ sys.argv[0] + " <infile 1> <infile 2> <
        ↪ outfile>")
    return

  x = readData(sys.argv[1])
  y = readData(sys.argv[2])

  z = diff(x,y, True)

  out = sys.argv[3]
  if out.endswith(".png"):
    out=out[:-4]

  writeData(out, *z)

  return


if __name__ == "__main__":
  main()
```

Bibliography

[1] R. Micheloni, L. Crippa, and A. Marelli, *Inside NAND Flash Memories*. Springer Netherlands, 2010, ISBN: 9789048194315. [Online]. Available: https://books.google.at/books?id=vaq11vKwo%5C_kC.

[2] P. Cappelletti, C. Golla, P. Olivo, and E. Zanoni, *Flash Memories*. Springer US, 1999, ISBN: 9780792384878. [Online]. Available: https://books.google.at/books?id=WES87yafag8C.

[3] D. Samyde, S. Skorobogatov, R. Anderson, and J.-J. Quisquater, „On a new way to read data from memory", in *Security in Storage Workshop, 2002. Proceedings. First International IEEE*, IEEE, 2002, pp. 65–69.

[4] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, „Introduction to differential power analysis", English, *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, 2011, ISSN: 2190-8508. DOI: 10.1007/s13389-011-0006-y. [Online]. Available: http://dx.doi.org/10.1007/s13389-011-0006-y.

[5] E. Brier, C. Clavier, and F. Olivier, „Correlation power analysis with a leakage model", English, in *Cryptographic Hardware and Embedded Systems - CHES 2004*, ser. Lecture Notes in Computer Science, M. Joye and J.-J. Quisquater, Eds., vol. 3156, Springer Berlin Heidelberg, 2004, pp. 16–29, ISBN: 978-3-540-22666-6. DOI: 10.1007/978-3-540-28632-5_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-28632-5_2.

[6] P. Pavan, R. Bez, P. Olivo, and E. Zanoni, „Flash memory cells-an overview", *Proceedings of the IEEE*, vol. 85, no. 8, pp. 1248–1271, Aug. 1997, ISSN: 0018-9219. DOI: 10.1109/5.622505.

[7] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, „Introduction to flash memory", *Proceedings Of The Ieee, 2003 Apr, Vol.91(4), pp.489-502*, ISSN: 0018-9219.

[8] S. Boboila and P. Desnoyers, „Write endurance in flash drives: measurements and analysis", in *FAST*, vol. 10, 2010, pp. 9–9.

[9] P. Desnoyers, „Empirical evaluation of nand flash memory performance", *ACM SIGOPS Operating Systems Review*, vol. 44, no. 1, pp. 50–54, 2010.

[10] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, „Leveraging value locality in optimizing nand flash-based ssds", in *FAST*, 2011, pp. 91–103.

[11] F. Chen, T. Luo, and X. Zhang, „Caftl: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives“, in *FAST*, vol. 11, 2011.

[12] M. Murugan and D. Du, „Rejuvenator: a static wear leveling algorithm for nand flash memory with minimized overhead“, in *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, May 2011, pp. 1–12. DOI: 10.1109/MSST.2011.5937225.

[13] A. J. Hong, E. B. Song, H. S. Yu, M. J. Allen, J. Kim, J. D. Fowler, J. K. Wassei, Y. Park, Y. Wang, J. Zou, R. B. Kaner, B. H. Weiller, and K. L. Wang, „Graphene flash memory“, *ACS Nano*, vol. 5, no. 10, pp. 7812–7817, 2011, PMID: 21854056. DOI: 10.1021/nn201809k. eprint: http://dx.doi.org/10.1021/nn201809k. [Online]. Available: http://dx.doi.org/10.1021/nn201809k.

[14] T.-C. Chang, F.-Y. Jian, S.-C. Chen, and Y.-T. Tsai, „Developments in nanocrystal memory“, *Materials Today*, vol. 14, no. 12, pp. 608–615, 2011, ISSN: 1369-7021. DOI: http://dx.doi.org/10.1016/S1369-7021(11)70302-9. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1369702111703029.

[15] K.-J. Baeg, D. Khim, J. Kim, B.-D. Yang, M. Kang, S.-W. Jung, I.-K. You, D.-Y. Kim, and Y.-Y. Noh, „High-performance top-gated organic field-effect transistor memory using electrets for monolithic printed flexible nand flash memory“, *Advanced Functional Materials*, vol. 22, no. 14, pp. 2915–2926, 2012.

[16] L. M. Grupp, J. D. Davis, and S. Swanson, „The bleak future of nand flash memory“, in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12, San Jose, CA: USENIX Association, 2012, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2208461.2208463.

[17] M. Breeuwsma, M. De Jongh, C. Klaver, R. Van Der Knijff, and M. Roeloffs, „Forensic data recovery from flash memory“, *Small Scale Digital Device Forensics Journal*, vol. 1, no. 1, pp. 1–17, 2007.

[18] S. Willassen, „Forensic analysis of mobile phone internal memory“, in *Advances in Digital Forensics*, Springer, 2005, pp. 191–204.

[19] S. Skorobogatov, „Data remanence in flash memory devices“, in *Cryptographic Hardware and Embedded Systems–CHES 2005*, Springer, 2005, pp. 339–353.

[20] T. Roche, V. Lomné, and K. Khalfallah, „Combined fault and side-channel attack on protected implementations of aes“, in *Smart Card Research and Advanced Applications*, Springer, 2011, pp. 65–83.

[21] A. Moradi, M. Kasper, and C. Paar, „Black-box side-channel attacks highlight the importance of countermeasures“, in *Topics in Cryptology–CT-RSA 2012*, Springer, 2012, pp. 1–18.

[22] T. Güneysu and A. Moradi, „Generic side-channel countermeasures for reconfigurable devices“, in *Cryptographic Hardware and Embedded Systems–CHES 2011*, Springer, 2011, pp. 33–48.

[23] T. Kasper, D. Oswald, and C. Paar, „Side-channel analysis of cryptographic rfids with analog demodulation“, in *RFID. Security and Privacy*, Springer, 2012, pp. 61–77.

[24] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, „Acoustic side-channel attacks on printers“, in *USENIX Security Symposium*, 2010, pp. 307–322.

[25] S. Skorobogatov, „Optical fault masking attacks“, in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*, IEEE, 2010, pp. 23–29.

[26] P. Olivier, J. Boukhobza, and E. Senn, „Toward a unified performance and power consumption nand flash memory model of embedded and solid state secondary storage systems“, *arXiv preprint arXiv:1307.1217*, 2013.

[27] V. Mohan, T. Bunker, L. Grupp, S. Gurumurthi, M. R. Stan, and S. Swanson, „Modeling power consumption of nand flash memories using flashpower“, *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, no. 7, pp. 1031–1044, 2013.

Acronyms

μC Micro Controller. 1, 10, 17, 19, 20, 29, 39

CG Control Gate. 5

CLI Command Line Interface. 36

CPA Correlation Power Analysis. 4, 5

CSV Comma Separated Value. 27, 28, 31

DPA Differential Power Analysis. 4, 13, 16, 28, 33, 39

EMR Electromagnetic Radiation. 3, 14

FG Floating Gate. 5, 6, 9

FMC Flash Memory Chip. ix, 1–3, 7, 9–11, 13, 14, 19–21, 26, 34, 37, 39, 40

I/O Input/Output. 1, 3, 7, 14, 17, 19, 21, 23–25, 39

MC Memory Controller. ix, 1, 2, 9, 10, 39, 40

MLC Multi Level Cell. 7, 8

MOS Metal Oxide Semiconductor. 5

OS Operating System. 2, 19

PCB Printed Circuit Board. 19

SC Side Channel. 2, 3, 5, 10, 11, 13, 14, 20, 23, 26, 31, 32, 36, 37, 39

SCA Side Channel Attack. ix, 10, 11, 13, 14, 37, 39

SLC Single Level Cell. 7, 8

SPA Simple Power Analysis. 3, 13

SSD Solid State Disk. ix, 1, 40

UI User Interface. 15, 19, 24, 36, 37, 39

VISA Virtual Instrument Software Architecture. 23, 25