NAND Flash Interface Specification

Bruno Simões Dias (35194)
<brsimoes@gmail.com>

Miguel Alexandre Ferreira (33194)
<a33194@alunos.uminho.pt>

Informatics and Systems Engineering
Dep. Informatics, University of Minho *

July 2007

Abstract. *The present project was born to answer the "mini challenge" proposed by Rajeev Joshi and Gerard J. Holzmann, for building a filesystem that can be automatically verified. We believe that, in order to reach a reliable solution, one first needs to abstract the hardware layer where the filesystem will reside in and then gradually introduce hardware details in the model. So, the purpose of this paper is to present a specification of a NAND type flash memory device.*

*Campus de Gualtar, 4700 Braga, Portugal.

Preamble. *Sections 2 and 3 of this report where verbatim extracted from the Open NAND Flash Interface Specification [ONFIS06]. Along with the text the reader will find annotations in gray boxes containing* VDM++ *code and yellow boxes containing informal descriptions and notes on the code.*

Contents

1 Introduction 4

2 Memory Organization 5

2.1 Addressing . . . 7

2.2 Factory Defect Mapping . . . 9

2.2.1 Device Requirements . . . 10

2.2.2 Host Requirements . . . 10

2.3 Discovery and Initialization . . . 12

2.3.1 CE# Discovery . . . 12

2.3.2 Target Initialization . . . 13

3 Command Definition 14

3.1 Command Set . . . 14

3.1.1 Read Definition . . . 16

3.1.2 Block Erase Definition . . . 20

3.1.3 Page Program Definition . . . 23

4 Behavior Simulation 27

4.1 Buffer Model . . . 27

4.2 Simulation Environment . . . 29

4.2.1 Type Definitions . . . 29

4.2.2 Host Device Communication . . . 29

4.2.3 Functionality Definition . . . 30

4.2.4 Get Arguments Operations . . . 32

5 Final considerations 34

Function/Method Cross-Reference Index

Bit, **7**
Block, **6**
BlockErase, **20**
Buffer, **27**
BufferModDef, 27
BufferType, **27**
Byte, **7**, **27**

ceDiscovery, **12**
checkBlock, **12**
checkColAddress, **19**
checkInterLeavedOp, **19**
checkPage, **12**
checkRowAddress, **19**
Clear, **27**
ColAddress, 7
ColAddressType, **7**
Column, **7**
CreateBadBlocksTable, **11, 12**

Data, **27**
Device, 10–13, 16, 19, 20, 23
DeviceAlg, 5
DeviceInit, **13**
DeviceType, **5**
Drop, **28**
DropN, **29**

ElementAt, **28**
ErrorMessage, **29**

Flash, **6**

Get, **28**
GetAll, **33**
GetCode, **32**
GetColAddr, **32**
GetN, **28**
GetRowAddr, **32**

Input, **29**
is00, **12**
isDefective, **10**

LUN, **6**
Lun, 6, 12, 16, 20, 24

Output, **29**

Page, **7**
PageProgram, **23, 24**
PageRegister, **6**
Put, **28**
PutN, **28**

Read, **16**
readPartialPage, **19**
Result, **29**
RowAddress, 8
RowAddressType, **8**, *9*

Signal, **5**
Simulate, **30**
SimulateBlockErase, **31**
SimulateError, **32**
SimulatePageProgram, **31**
SimulateRead, **30**
Size, **27**

Target, **5**

Word, **7**

1 Introduction

After some research on flash types, we decided to work on NAND type flash, because they present a better life span and major similarities with common block devices (e.g. Hard Drives) than the NOR type flash memories. The developed model is based on the *Open NAND Flash Interface Specification revision 1.0* [ONFIS06] by the *Open NAND Flash Workgroup* (ONFI) which aims to define a standardized NAND Flash device interface.

The goal of the project would be to specify all aspects of the device behavior and host device communications, but that would require more time than some hours per week during a college semester. So, the focus here is on developing a working model that would offer the basic functionalities for deploying a file system. To achieve this, the model includes memory organization, addressing, initialization routines and some commands of such a device.

Sections 2 and 3 where extracted from [ONFIS06], and some key aspects where coloured. It will be easy to notice that red corresponds to key nouns, green for key verbs and orange for key properties. There will be also, gray boxes with the *VDM++* code corresponding to the colored text and yellow boxes adding comments and informal descriptions of the code.

In Section 4 we define a first approach to model the communication between the host and the device and to a simulation environment.

Rather than adding a ”never ending” appendix to the end of this report with all the classes of the model, the code will be supplied has a stand alone reference.

2 Memory Organization

Figure 1 describes the basic memory and device organization. In some implementations, additional page registers may be present within each logical unit.

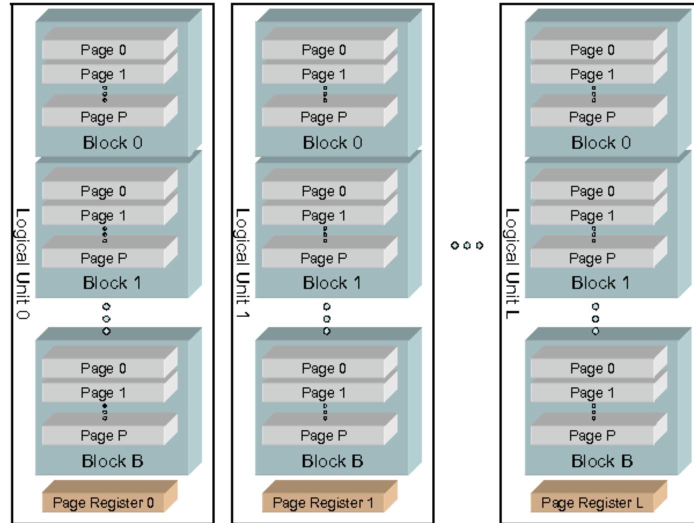

Figure 1: Target memory organization

A **device contains** one or more **targets**. A **target** is **controlled** by one **CE# signal**. A **target** is **organized** into one or more **logical units** (LUNs).

class $DeviceAlg$
  $DeviceType = Signal \xrightarrow{m} Target$
  inv $D \triangleq$ dom $D \neq \{\}$;

  $Target :: LUNs : Lun^{+}$
    $DBT : \mathbb{N}_1 \xrightarrow{m} (\mathbb{N}_1\text{-set})$
    $parameters : ParameterPage$
  inv $t \triangleq \forall i \in$ inds $t.LUNs \cdot$
    $\forall j \in$ inds $t.LUNs \cdot$
      $t.LUNs(i).flashSameSize$
        (
        $t.LUNs(i).GetFlash(),$
        $t.LUNs(j).GetFlash());$

  $Signal = \mathbb{N}_1$
  inv $s \triangleq s \leq 4$;
...
end $DeviceAlg$

A **logical unit** (LUN) **is the minimum unit that can independently execute commands** and **report status**. For example, it is permissible to start a Page Program operation on LUN 0 and then prior to the operations completion to start a Read command on LUN 1. See multiple LUN operation restrictions in section 3.1.3. A **LUN contains** at least one **page register** and a **Flash array**. The **number of page registers** is dependent on the number of **interleaved operations supported** for that LUN. The **Flash array contains** a number of **blocks**.

$$
\begin{array}{l}
\textsf{class } Lun \\
\quad LUN :: pr : PageRegister^{+} \\
\quad\quad\quad\quad flash : Flash \\
\quad \textsf{inv } l \triangleq \forall i \in \textsf{inds } l.pr \cdot \\
\quad\quad\quad\quad \forall j \in \textsf{inds } l.pr \cdot \\
\quad\quad\quad\quad\quad pageSameSize\,(l.pr\,(i), l.pr\,(j));
\end{array}
$$

$$
PageRegister = Page;
$$

$$
\begin{array}{l}
\quad Flash = Block^{+} \\
\quad \textsf{inv } f \triangleq \forall i \in \textsf{inds } f \cdot \\
\quad\quad\quad\quad \forall j \in \textsf{inds } f \cdot \\
\quad\quad\quad\quad\quad blockSameSize\,(f\,(i), f\,(j));
\end{array}
$$

A **block is the smallest erasable unit of data within the Flash array of a LUN**. There is no restriction on the number of blocks within the LUN. A **block contains** a number of **pages**.

$$
\begin{array}{l}
\quad Block = Page^{+} \\
\quad \textsf{inv } b \triangleq isMultiple\,(\textsf{len } b, 32) \wedge \\
\quad\quad \forall i \in \textsf{inds } b \cdot \\
\quad\quad\quad\quad \forall j \in \textsf{inds } b \cdot \\
\quad\quad\quad\quad\quad pageSameSize\,(b\,(i), b\,(j));
\end{array}
$$

Although there is no reference in the text to the homogeneity the Lun's components that is enforced by Figure 1. So there are invariants stating that all pages in a block have the same size, all blocks in a flash have the same size and all page registers in a Lun have the same size.

A **page is the smallest addressable unit for read and program operations**. For targets that support partial page programming with constraints, the smallest addressable unit for program operations is a partial page. A **page consists** of a number of **bytes or words**. **The number of user data bytes per page, not including the spare data area, shall be a power of two**. **The number of pages per block shall be a multiple of 32**.

$Page :: data : Column^{+}$
$\quad defect : Column^{+}$
inv $p \triangleq isP\,(\text{len } p.data, 2)$;

Each **LUN shall have at least one page register**. A **page register is used for the temporary storage of data** before it is **moved** to a **page** within the **Flash array** or after it is **moved** from a **page** within the **Flash array**.

The **byte or word location** within the **page register** is **referred to** as the **column**.

$Column = Byte$;

$Byte = Bit^{+}$
inv $b \triangleq \text{len } b = 8$;

$Word = Byte^{+}$
inv $w \triangleq \text{len } w = 2$;

$Bit = \mathbb{B}$
...
end Lun

2.1 Addressing

There are two **address** types **used**: the **column address** and the **row address**. The **column address** is **used** to access **bytes or words** within a page, i.e. the column address is the byte/word offset into the page. The **row address** is **used** to address **pages**, **blocks**, and **LUNs**.

class $ColAddress$
  private $ColAddressType = \mathbb{N}_1$
instance variables
  private $address : ColAddressType$;

...
end $ColAddress$

When both the column and row addresses are required to be issued, the column address is always issued first in one or more 8-bit address cycles. The row addresses follow in one or more 8-bit address cycles. There are some functions that may require only row addresses, like Block Erase. In this case the column addresses are not issued.

For both column and row addresses the first address cycle always contains the least significant address bits and the last address cycle always contains the most significant address bits. If there are bits in the most significant cycles of the column and row addresses that are not used then they are required to be cleared to zero.

The row address structure is shown in Figure 2 with the **least significant** row address bit **to the right** and the **most significant** row address bit **to the left**.

Figure 2: Row Address Layout

The number of blocks and number of pages per block is not required to be a power of two. In the case where one of these values is not a power of two, the **corresponding address** shall be **rounded** to an integral number of bits **such that it addresses a range up to the subsequent power of two value**. The **host shall not access upper addresses** in that range that are shown as not supported.

class $RowAddress$
private
$RowAddressType :: Laddr : Bit^*$
$Baddr : Bit^*$
$Paddr : Bit^*$
$n\text{-}blocks : \mathbb{N}_1$
$n\text{-}pages : \mathbb{N}_1$
inv $rw \triangleq$
$2 \uparrow (\text{len } rw.Baddr) \geq rw.n\text{-}blocks \wedge$
$2 \uparrow (\text{len } rw.Paddr) \geq rw.n\text{-}pages \wedge$
$bin\text{-}to\text{-}nat\,(rw.Baddr) \leq rw.n\text{-}blocks \wedge$
$bin\text{-}to\text{-}nat\,(rw.Paddr) \leq rw.n\text{-}pages$
...
end $RowAddress$

For example, if the number of pages per block is 96, then the page address shall be rounded to 7 bits such that it can address pages in the range of 0 to 127. In this case, the host shall not access pages in the range from 96 to 127 as these pages are not supported.

The first 3 examples are valid values of RowAddressType.

values

$rw1 : RowAddressType =$ mk-$RowAddressType$
(
[false, false],
[false, false, false, false, false, false, false, false],
[false, false, false, false, false, false, false],
256, 96);

$rw2 : RowAddressType =$ mk-$RowAddressType$
(
[false, false],
[false, false, false, false, false, false, false, false],
[false, false, true, false, false, true, false],
256, 96);

$rw3 : RowAddressType =$ mk-$RowAddressType$
(
[false, false],
[false, false, false, false, false, false, false, false],
[true, true, false, false, false, false, false],
256, 96);

This last example is an invalid value of RowAddressType, where the host tries to address a page that is out of the established range.

$rw4 : RowAddressType =$ mk-$RowAddressType$
(
[false, false],
[false, false, false, false, false, false, false, false],
[true, true, false, false, false, false, true],
256, 96)

The page address always uses the least significant row address bits. The block address uses the middle row address bits and the LUN address uses the most significant row address bit(s).

2.2 Factory Defect Mapping

The Flash array is not presumed to be pristine, and a number of defects may be present that renders some blocks unusable. Block granularity is used for mapping factory defects since those defects may compromise the block erase capability.

2.2.1 Device Requirements

If a block is defective and 8-bit data access is used, the **manufacturer** shall **mark** the block **as defective** by **setting** at least **one byte** in the **defect area**, as shown in Figure 3, **of the first or last page of the defective block to** a **value of 00h**. If a block is defective and 16-bit data access is used, the manufacturer shall mark the block as defective by setting at least one word in the defect area of the first or last page of the defective block to a value of 0000h.

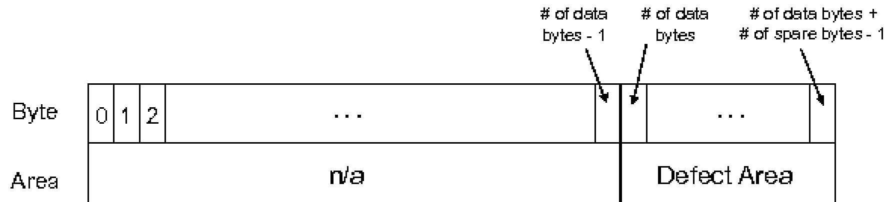

Figure 3: Area marked in factory defect mapping

2.2.2 Host Requirements

The **host** shall not **erase** or **program** **blocks** **marked as defective** by the manufacturer, and any attempt to do so yields indeterminate results.

class $Device$ is subclass of $DeviceAlg$
...

$isDefective : Target \times RowAddress \rightarrow \mathbb{B}$
$isDefective\,(t, r) \triangleq$
  let $lindex = r.GetLun\,()$,
    $bindex = r.GetBlock\,()$ in
  $bindex \in t.DBT\,(lindex)$;
...
end $Device$

Figure 4 outlines the algorithm to scan for factory mapped defects. This algorithm should be performed by the host to create the initial bad block table prior to performing any erase or programming operations on the target. The initial state of all pages in non-defective blocks is FFh (or FFFFh for 16-bit access) for all page addresses, although some bit errors may be present if they are correctable via the required ECC reported to the host. An defective block is indicated by abyte value equal to 00h for 8-bit access or a word value equal to 0000h for 16-bit access being present at any page byte/word location in the defect area of either the first page or last page of the block. The host shall check the defect area of both the first and last past page of each block to verify the block is valid prior to any erase or program operations on that block.

Note: Over the lifetime use of a NAND device, the defect area of defective blocks may encounter read disturbs that cause values to change. The manufacturer defect markings may change value over the lifetime of the device, and are expected to be read by the host and used to create a bad block table during initial use of the part.

```
for (i=0; i<NumLUNs; i++)
{
    for (j=0; j<BlocksPerLUN; j++)
    {
        Defective=FALSE;

        ReadPage(lun=i; block=j; page=0; DestBuff=Buff);
        for (column=PageSize; column<PageSize+SpareBytes; column++)
        {
            if (Buff[column] == 00h)        // Value checked for is 0000h for 16-bit access
                Defective=TRUE;
        }

        ReadPage(lun=i; block=j; page=PagesPerBlock-1; DestBuff=Buff);
        for (column=PageSize; column<PageSize+SpareBytes; column++)
        {
            if (Buff[column] == 00h)        // Value checked for is 0000h for 16-bit access
                Defective=TRUE;
        }

        if (Defective)
            MarkBlockDefective(lun=i; block=j);
    }
}
```

Figure 4: Factory defect scanning algorithm

class $Device$ **...**
private
$\quad CreateBadBlocksTable : Signal \xrightarrow{o} ()$
$\quad CreateBadBlocksTable\ (signal) \triangleq$
$\quad\quad$ for all $i \in$ inds $device\ (signal).LUNs$
$\quad\quad$ do $device(signal).DBT := device\ (signal).DBT \uplus$
$\quad\quad\quad \{i \mapsto device\ (signal).LUNs\ (i).CreateBadBlocksTable\ ()\}$
...
end $Device$

```
class Lun
...

    CreateBadBlocksTable : () -o-> ℕ1-set
    CreateBadBlocksTable () ≜
      return {block | block ∈ inds lun.flash ·
                      checkBlock (block, lun)}
    checkBlock : ℕ1 × LUN → 𝔹
    checkBlock (b, lun) ≜
      let fstpage = lun.flash (b) (1),
          lstpage = lun.flash (b) (len lun.flash) in
      checkPage (fstpage) ∨ checkPage (lstpage);
    checkPage : Page → 𝔹
    checkPage (p) ≜
      let S = {is00 (p.defect (i)) | i ∈ inds p.defect} in
      true ∈ S;
    is00 : Byte → 𝔹
    is00 (b) ≜
      Converter‘byte-to-hex (b) = "00";
end Lun
```

2.3 Discovery and Initialization

2.3.1 CE# Discovery

There may **be up to four chip enable** (CE#) signals on a package, **one for each separately addressable target**. To **determine the targets** that are connected, the **procedure** outlined in this section **shall be followed** for **each distinct CE# signal**. CE# signals shall be used sequentially on the device; **CE1#** is always **connected** and **CE#** signals shall be **connected** in a **numerically increasing order**. The host shall attempt to enumerate targets connected to all host CE# signals.

```
class Device ...
    ceDiscovery : DeviceType → Signal+
    ceDiscovery (d) ≜
      let s = [i | i ∈ dom d] in
      insertionSort[Signal] (λ a : Signal · λ b : Signal · a < b) (s);
...
end Device
```

The discovery process for a package that supports independent dual data buses includes additional steps to determine which data bus the target is connected to. The LGA package with 8- bit data access is the only defined package within ONFI that has a dual data bus option.

2.3.2 Target Initialization

To initialize a **discovered target**, the **following steps shall be taken**. The **initialization process** shall be followed for each **connected CE#** signal, **including performing** the **Read Parameter Page (ECh)** command **for** each **target**. Each **chip enable** corresponds to a **unique target** with its own **independent properties** that the host shall observe and subsequently use.

The **host** shall **issue** the **Read Parameter Page (ECh) command**. This command returns information that includes the capabilities, features, and operating parameters of the device.

After successfully retrieving the parameter page, the host has all information necessary to successfully communicate with that target. If the **host has not previously mapped defective block** information for this target, the **host shall** next **map out all defective blocks** in the target. The host may then proceed to utilize the target, including erase and program operations.

```
class Device ...

    DeviceInit : () -o-> ()
    DeviceInit () ≜
      let dis = ceDiscovery (device) in
      for all i ∈ inds dis
      do CreateBadBlocksTable(dis (i)) ;
...
end Device
```

3 Command Definition

3.1 Command Set

Table 14 outlines the ONFI command set.

The value specified in the first command cycle identifies the command to be performed. Some commands have a second command cycle as specified in Table 14. Typically, commands that have a second command cycle include an address.

Command	O/M	1st Cycle	2nd Cycle	Acceptable while Accessed LUN is Busy	Acceptable while Others LUNs are Busy	Target level commands
Read	M	00h	30h		Y	
Block Erase	M	60h	D0h		Y	
Page Program	M	80h	10h		Y	

Table 14: Command Set

The following definition is a representation of a device created in the *VDM++ Toolbox*, with the commands (where data1 is defined in the class Tests2):

```
> create device := new Device(1,1,1,1,32,4)
> p device.Input(data1)
> p device.Simulate(1,1)
```

The representation of the device bytes in hexadecimal can be obtained by doing:

```
> p device.Echo()
```

This device will be used to demonstrate each of the commands in the implemented Command Set.

Device

Target 1

Lun 1

PageRegister 1		AA	FF	AA	FF
Block 1					
Page 1	::	00	00	00	00
Page 2	::	00	00	00	00
Page 3	::	00	00	00	00
Page 4	::	00	00	00	00
Page 5	::	00	00	00	00
Page 6	::	00	00	00	00
Page 7	::	00	00	00	00
Page 8	::	00	00	00	00
Page 9	::	00	00	00	00
Page 10	::	00	00	00	00
Page 11	::	00	00	00	00
Page 12	::	00	00	00	00
Page 13	::	00	00	00	00
Page 14	::	00	00	00	00
Page 15	::	00	00	00	00
Page 16	::	00	00	00	00
Page 17	::	00	00	00	00
Page 18	::	00	00	00	00
Page 19	::	00	00	00	00
Page 20	::	00	00	00	00
Page 21	::	00	00	00	00
Page 22	::	00	00	00	00
Page 23	::	00	00	00	00
Page 24	::	00	00	00	00
Page 25	::	00	00	00	00
Page 26	::	00	00	00	00
Page 27	::	00	00	00	00
Page 28	::	00	00	00	00
Page 29	::	00	00	00	00
Page 30	::	00	00	00	00
Page 31	::	00	00	00	00
Page 32	::	AA	FF	AA	FF

BadBlocks

1 |-> {}

3.1.1 Read Definition

The Read **function** **reads** a **page** of data **identified** by a **row address** for the LUN specified. The **page** of data is **made available** to be read from the **page register** starting at the column address specified. Figure 30 defines the Read behavior and timings. **Reading beyond the end of a page results in indeterminate values being returned to the host**.

```
class Device is subclass of DeviceAlg
...
private
    Read : ℕ1 × Signal × ColAddress × RowAddress -o-> Column*
    Read (i, s, c, r) ≜
      let lindex = r.GetLun (),
          bindex = r.GetBlock (),
          pindex = r.GetPage (),
          t = device (s),
          lun = t.LUNs (lindex) in
      (   lun.Read(i, bindex, pindex) ;
          return readPartialPage (lun.GetPageRegister (i), c)
      )
    pre  checkRowAddress (device (s), r) ∧
         checkColAddress (device (s), c, r) ∧
         checkInterLeavedOp (i, device (s), r)
      ;
...
end Device
```

```
class Lun
...

    Read : ℕ1 × ℕ1 × ℕ1 -o-> ()
    Read (i, b, p) ≜
      let block = lun.flash (b),
          page = block (p) in
      lun.pr := lun.pr † {i ↦ page}
    pre  b ∈ inds lun.flash ∧
         p ∈ inds lun.flash (b) ∧
         i ∈ inds lun.pr
      ;
...
end Lun
```

While monitoring the read status to determine when the tR (transfer from Flash array to page register) is complete, the host shall re-issue a command value of 00h

to start reading data. Issuing a command value of 00h will cause data to be returned starting at the selected column address.

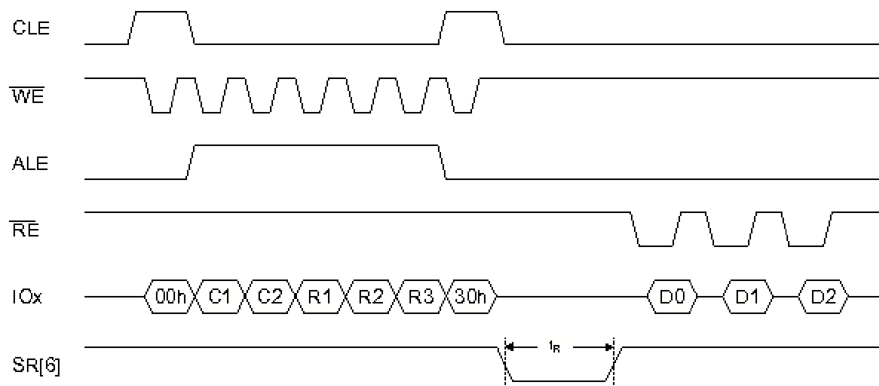

Figure 30: Read timing

C1 - C2	Column address of the page to retrieve. C1 is the least significant byte.
R1 - R3	Row address of the page to retrieve. R1 is the least significant byte.
Dn	Data bytes read from the addressed page.

Example of a Read execution (where data3 is defined in class Tests2):

```
> p device.Input(data3)
> p device.Simulate(1,1)
```

```
Device
    Target 1
        Lun 1
            PageRegister 1                  | 00  00  00  00 |
            Block 1
                        Page 1   ::  | 00  00  00  00 |
                        Page 2   ::  | 00  00  00  00 |
                        Page 3   ::  | 00  00  00  00 |
                        Page 4   ::  | 00  00  00  00 |
                        Page 5   ::  | 00  00  00  00 |
                        Page 6   ::  | 00  00  00  00 |
                        Page 7   ::  | 00  00  00  00 |
                        Page 8   ::  | 00  00  00  00 |
                        Page 9   ::  | 00  00  00  00 |
                        Page 10  ::  | 00  00  00  00 |
                        Page 11  ::  | 00  00  00  00 |
                        Page 12  ::  | 00  00  00  00 |
                        Page 13  ::  | 00  00  00  00 |
                        Page 14  ::  | 00  00  00  00 |
                        Page 15  ::  | 00  00  00  00 |
                        Page 16  ::  | 00  00  00  00 |
                        Page 17  ::  | 00  00  00  00 |
                        Page 18  ::  | 00  00  00  00 |
                        Page 19  ::  | 00  00  00  00 |
                        Page 20  ::  | 00  00  00  00 |
                        Page 21  ::  | 00  00  00  00 |
                        Page 22  ::  | 00  00  00  00 |
                        Page 23  ::  | 00  00  00  00 |
                        Page 24  ::  | 00  00  00  00 |
                        Page 25  ::  | 00  00  00  00 |
                        Page 26  ::  | 00  00  00  00 |
                        Page 27  ::  | 00  00  00  00 |
                        Page 28  ::  | 00  00  00  00 |
                        Page 29  ::  | 00  00  00  00 |
                        Page 30  ::  | 00  00  00  00 |
                        Page 31  ::  | 00  00  00  00 |
                        Page 32  ::  | AA  FF  AA  FF |
        BadBlocks
            1 | - > {}
```

The value is read in to the page register and dumped in the buffer.

```
private
    readPartialPage : Page × ℕ₁ → Column*
    readPartialPage (p, c) ≜
      [newColumn (false) | i ∈ inds p.data · i < c] ↷
      [p.data (i) | i ∈ inds p.data · i ≥ c]
    post isP (len RESULT, 2) ;
...
end Lun
```

```
class Device is subclass of DeviceAlg
...

    checkRowAddress : Target × RowAddress → 𝔹
    checkRowAddress (t, r) ≜
      let lindex = r.GetLun (),
          bindex = r.GetBlock (),
          pindex = r.GetPage () in
      ¬ isDefective (t, r) ∧
      lindex ∈ inds t.LUNs ∧
      let flash = t.LUNs (lindex).GetFlash () in
      bindex ∈ inds flash ∧
      pindex ∈ inds flash (bindex);
```

```
    checkColAddress : Target × ColAddress × RowAddress → 𝔹
    checkColAddress (t, c, r) ≜
      let lindex = r.GetLun (),
          bindex = r.GetBlock (),
          pindex = r.GetPage (),
          flash = t.LUNs (lindex).GetFlash (),
          page = flash (bindex) (pindex) in
      c.GetColumn () ∈ inds page.data;
```

```
    checkInterLeavedOp : ℕ × Target × RowAddress → 𝔹
    checkInterLeavedOp (i, t, r) ≜
      let lindex = r.GetLun (),
          pr = t.LUNs (lindex).GetAllPageRegisters () in
      i ∈ inds pr;
...
end Device
```

3.1.2 Block Erase Definition

The Block Erase **function** **erases** the **block** of data **identified** by the block **address** parameter on the LUN specified. **After a successful Block Erase, all bits shall be set to one in the block**. SR[0] is valid for this command after SR[6] transitions from zero to one until the next transition of SR[6] to zero. Figure 31 defines the Block Erase behavior and timings.

```
class Device is subclass of DeviceAlg
...
private
    BlockErase : Signal × RowAddress -o-> ()
    BlockErase (s, r) ≜
      let lindex = r.GetLun (),
          bindex = r.GetBlock (),
          t = device (s),
          lun = t.LUNs (lindex) in
      lun.BlockErase(bindex)
    pre  checkRowAddress (device (s), r)
      ;
...
end Device
```

```
class Lun
...

    BlockErase : ℕ1 -o-> ()
    BlockErase (b) ≜
      let block = lun.flash (b),
          block' = [emptyPage (block (i)) | i ∈ inds block] in
      lun.flash := lun.flash † {b ↦ block'}
    pre  b ∈ inds lun.flash
      ;
...
end Lun
```

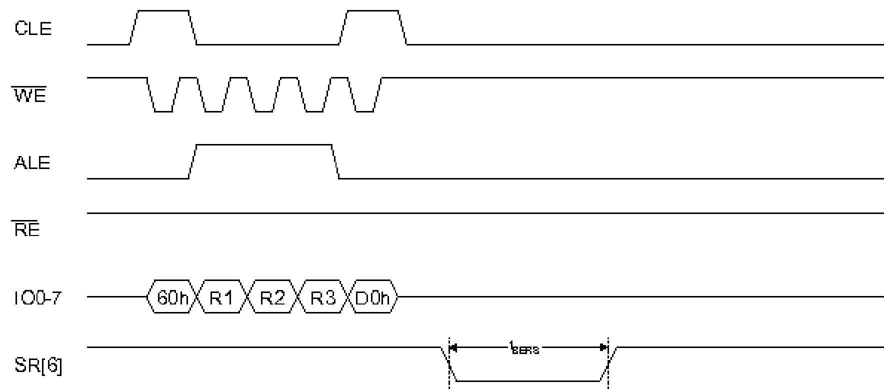

Figure 31: Block Erase timing

R1 - R3	The row address of the block to be erased. R1 is the least significant byte in the row address.

Example of a Block Erase execution (where data4 is defined in class Tests2):

```
> p device.Output() – clean the buffer
> p device.Input(data4)
> p device.Simulate(1,1)
```

```
Device
    Target 1
        Lun 1
            PageRegister 1          | 00 00 00 00 |
            Block 1
                Page 1   :: | FF FF FF FF |
                Page 2   :: | FF FF FF FF |
                Page 3   :: | FF FF FF FF |
                Page 4   :: | FF FF FF FF |
                Page 5   :: | FF FF FF FF |
                Page 6   :: | FF FF FF FF |
                Page 7   :: | FF FF FF FF |
                Page 8   :: | FF FF FF FF |
                Page 9   :: | FF FF FF FF |
                Page 10  :: | FF FF FF FF |
                Page 11  :: | FF FF FF FF |
                Page 12  :: | FF FF FF FF |
                Page 13  :: | FF FF FF FF |
                Page 14  :: | FF FF FF FF |
                Page 15  :: | FF FF FF FF |
                Page 16  :: | FF FF FF FF |
                Page 17  :: | FF FF FF FF |
                Page 18  :: | FF FF FF FF |
                Page 19  :: | FF FF FF FF |
                Page 20  :: | FF FF FF FF |
                Page 21  :: | FF FF FF FF |
                Page 22  :: | FF FF FF FF |
                Page 23  :: | FF FF FF FF |
                Page 24  :: | FF FF FF FF |
                Page 25  :: | FF FF FF FF |
                Page 26  :: | FF FF FF FF |
                Page 27  :: | FF FF FF FF |
                Page 28  :: | FF FF FF FF |
                Page 29  :: | FF FF FF FF |
                Page 30  :: | FF FF FF FF |
                Page 31  :: | FF FF FF FF |
                Page 32  :: | FF FF FF FF |
        BadBlocks
            1 | - > {}
```

3.1.3 Page Program Definition

The Page Program **command** **transfers** a **page** or **portion** of a page of data **identified** by a **column address** to the **page register**. The **contents** of the page register are then **programmed** into the **Flash array** at the **row address** indicated. SR[0] is valid for this command after SR[6] transitions from zero to one until the next transition of SR[6] to zero. Figure 32 defines the Page Program behavior and timings. **Writing beyond the end of the page register is undefined.**

```
class Device is subclass of DeviceAlg
...
private
    PageProgram : ℕ1 × Signal × ColAddress × RowAddress × Column+ -o-> ()
    PageProgram (i, s, c, r, data) △
      let lindex = r.GetLun (),
          bindex = r.GetBlock (),
          pindex = r.GetPage (),
          t = device (s),
          lun = t.LUNs (lindex),
          col = c.GetColumn () in
      lun.PageProgram(i, col, bindex, pindex, data)
    pre  isP (len data, 2) ∧
        checkRowAddress (device (s), r) ∧
        checkColAddress (device (s), c, r) ∧
        checkPageOverflow (device (s), c, r, data) ∧
        checkInterLeavedOp (i, device (s), r)
      ;
...
end Device
```

```
class Lun
...

  PageProgram : ℕ1 × ℕ1 × ℕ1 × ℕ1 × Byte+ ⟶o ()
  PageProgram (i, c, b, p, data) ≜
    let block = lun.flash (b),
        page = block (p),
        prs = lun.pr † {i ↦ page},
        pager = prs (i),
        pager' = μ (pager,

                        data ↦ [pager.data (j) | j ∈ inds pager.data ·
                                    j < c] ⁀
                        [data (j) | j ∈ inds data ·
                                    j ≥ c ∧ j < c + (len data)] ⁀
                        [pager.data (j) | j ∈ inds pager.data ·
                                    j ≥ c + (len data)]),
        prs' = prs † {i ↦ pager'},
        block' = block † {p ↦ pager'},
        flash' = lun.flash † {b ↦ block'} in
    lun := mk-LUN (prs', flash')
  pre  isP (len data, 2) ∧
       b ∈ inds lun.flash ∧
       p ∈ inds lun.flash (b) ∧
       c ∈ inds lun.flash (b) (p).data ∧
       i ∈ inds lun.pr ∧
       c + len data − 1 ≤ len lun.flash (b) (p).data
    ;
...
end Lun
```

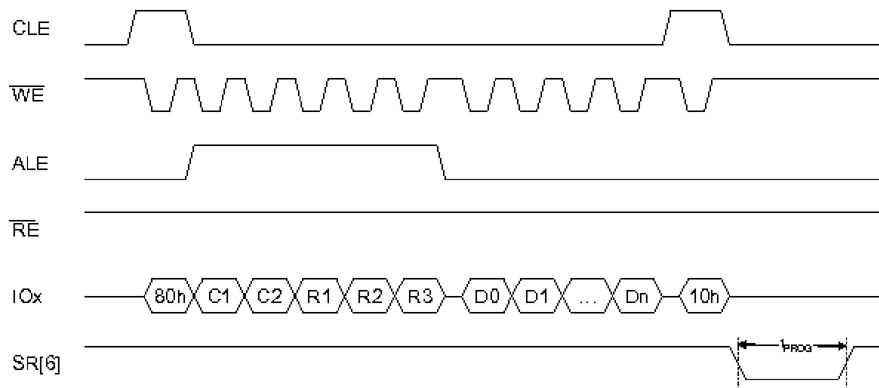

Figure 32: Page Program timing

C1 - C2	Column address of the starting buffer location to write data to. C1 is the least significant byte.
R1 - R3	Row address of the page being programmed. R1 is the least significant byte.
D0 - Dn	Data bytes/words to be written to the addressed page.

Example of a Page Program execution (where data2 is defined in class Tests2):

```
> p device.Input(data2)
> p device.Simulate(1,1)
```

```
Device
        Target 1
                Lun 1
                        PageRegister 1        | 55 00 55 00 |
                        Block 1
                                Page 1   :: | FF FF FF FF |
                                Page 2   :: | FF FF FF FF |
                                Page 3   :: | FF FF FF FF |
                                Page 4   :: | FF FF FF FF |
                                Page 5   :: | FF FF FF FF |
                                Page 6   :: | FF FF FF FF |
                                Page 7   :: | FF FF FF FF |
                                Page 8   :: | FF FF FF FF |
                                Page 9   :: | FF FF FF FF |
                                Page 10  :: | FF FF FF FF |
                                Page 11  :: | FF FF FF FF |
                                Page 12  :: | FF FF FF FF |
                                Page 13  :: | FF FF FF FF |
                                Page 14  :: | FF FF FF FF |
                                Page 15  :: | FF FF FF FF |
                                Page 16  :: | FF FF FF FF |
                                Page 17  :: | FF FF FF FF |
                                Page 18  :: | FF FF FF FF |
                                Page 19  :: | FF FF FF FF |
                                Page 20  :: | FF FF FF FF |
                                Page 21  :: | FF FF FF FF |
                                Page 22  :: | FF FF FF FF |
                                Page 23  :: | FF FF FF FF |
                                Page 24  :: | FF FF FF FF |
                                Page 25  :: | FF FF FF FF |
                                Page 26  :: | FF FF FF FF |
                                Page 27  :: | FF FF FF FF |
                                Page 28  :: | FF FF FF FF |
                                Page 29  :: | FF FF FF FF |
                                Page 30  :: | FF FF FF FF |
                                Page 31  :: | 55 00 55 00 |
                                Page 32  :: | FF FF FF FF |
                BadBlocks
                        1 |-> {}
```

4 Behavior Simulation

To model the hardware behaviour when a command is issued by the host, one has to consider mutiple aspects.

For starters, there must be a way for the two to comunicate, e.g., a data path. In the present model this comunication is done by meen of a buffer, where both host and device can read and write. Also, it will be necessary to take in account that commands are issued in cycles.

Timmings are another key aspect, because hardware devices take their time to complete some operations and have multiple delays.

At last the model shall be refined to simulate the variation of the main signals involved in a command execution.

4.1 Buffer Model

```
class Buffer
    Byte = Lun`Byte;
    BufferType = Data*;
    Data = Byte
```

```
instance variables
    private buffer : BufferType := clear ();
```

```
Buffer : () o→ Buffer
Buffer () △
    buffer := [];
```

```
Clear : () o→ ()
Clear () △
    buffer := clear ();
```

```
Size : () o→ ℕ
Size () △
    return size (buffer);
```

$Put : Data \xrightarrow{o} ()$
$Put\,(d) \triangleq$
  $buffer := put\,(d, buffer);$

$Get : () \xrightarrow{o} Data$
$Get\,() \triangleq$
  let mk- $(b', d) = get\,(buffer)$ in
  $($ $buffer := b';$
    return d
  $)$
pre $size\,(buffer) > 0$
  ;

$Drop : () \xrightarrow{o} ()$
$Drop\,() \triangleq$
  $buffer := drop\,(buffer)$
pre $size\,(buffer) > 0$
  ;

$ElementAt : \mathbb{N}_1 \xrightarrow{o} Data$
$ElementAt\,(n) \triangleq$
  return $elementAt\,(n, buffer);$

$PutN : Data^* \xrightarrow{o} ()$
$PutN\,(d) \triangleq$
  $buffer := putN\,(d, buffer)$
post $size\,(buffer) = size\,(d) + size\,(\overleftarrow{buffer})$;

$GetN : \mathbb{N} \xrightarrow{o} Data^*$
$GetN\,(n) \triangleq$
  let mk- $(b', d) = getN\,(n, buffer)$ in
  $($ $buffer := b';$
    return d
  $)$
post $size\,(buffer) = size\,(\overleftarrow{buffer}) - n \wedge$
  $size\,(RESULT) = n$;

$DropN : \mathbb{N} \xrightarrow{o} ()$
$DropN\ (n) \triangleq$
  $buffer := dropN\ (n, buffer)$
pre $size\ (buffer) \geq n$

post $size\ (buffer) = size\ (\overleftarrow{buffer}) - n$
end $Buffer$

4.2 Simulation Environment

4.2.1 Type Definitions

The output of simulating a command can be the resulting state of the device in which the command is executed and the resulting buffer or, in case something goes wrong, an error message.

$Result = [ErrorMessage];$
$ErrorMessage = \text{char}^*$

To select which command to execute, the host shall send the appropriate code to the device. The codes are identified in hexadecimal notation, so an hexadecimal representation of a data byte is useful.

$Hex = \text{char}^+$
inv $hex \triangleq \forall\, i \in \text{inds } hex \cdot$
  $isHexDigit\ (hex\ (i))$

4.2.2 Host Device Communication

$Input : Data^+ \xrightarrow{o} ()$
$Input\ (data) \triangleq buffer.$
  $PutN(data)$;

$Output : () \xrightarrow{o} Data^*$
$Output\ () \triangleq$
  let $n = buffer.Size\ ()$ in
  return $buffer.GetN\ (n);$

4.2.3 Functionality Definition

Caring out the simulation is done by the following functions.

```
Simulate : ℕ1 × Signal →o Result
Simulate (i, s) ≜
   let code = GetCode (),
       size = buffer.Size (),
       hex = byte-to-hex (code) in
    cases hex:
      "00" → if size ≠ 6
                  then SimulateError(commandArgs ("read"))
                  else SimulateRead(i, s) ,
      "60" → if size ≠ 4
                  then SimulateError(commandArgs ("block erase"))
                  else SimulateBlockErase(s) ,
      "80" → if size < 5
                  then SimulateError(commandArgs ("page program"))
                  else SimulatePageProgram(i, s) ,
      others → SimulateError(unsupportedCode (hex))
   end
pre buffer.Size () > 0
   ;
```

```
private
   SimulateRead : ℕ1 × Signal →o Result
   SimulateRead (i, s) ≜
      let nblocks = GetTargetBlockNumber (device (s)),
          npages = GetTargetPageNumber (device (s)),
          caddr = GetColAddr (),
          raddr = GetRowAddr (nblocks, npages),
          code = GetCode (),
          hex = byte-to-hex (code) in
      if hex ≠ "30"
      then SimulateError(wrongCode (hex, "30"))
      else let data = Read (i, s, caddr, raddr) in
             (    buffer.PutN(data) ;
                  return nil
             )
   pre buffer.Size () = 6
      ;
```

```
private
    SimulateBlockErase : Signal ◦→ Result
    SimulateBlockErase (s) △
       let nblocks = GetTargetBlockNumber (device (s)),
           npages = GetTargetPageNumber (device (s)),
           raddr = GetRowAddr (nblocks, npages),
           code = GetCode (),
           hex = byte-to-hex (code) in
       if hex ≠ "D0"
       then SimulateError(wrongCode (hex, "D0"))
       else (    BlockErase(s, raddr) ;
                 return nil
            )
    pre buffer.Size () ≥ 4
       ;
```

```
private
    SimulatePageProgram : ℕ1 × Signal ◦→ Result
    SimulatePageProgram (i, s) △
       let nblocks = GetTargetBlockNumber (device (s)),
           npages = GetTargetPageNumber (device (s)),
           caddr = GetColAddr (),
           raddr = GetRowAddr (nblocks, npages),
           res = GetAll ("10") in
        cases res:
          [] → SimulateError(noData ("PageProgram")) ,
          nil → SimulateError("PageProgram :  Wrong  OPcode") ,
          data → if ¬ isP (len data, 2)
                    then SimulateError(dataNotP2 ("PageProgram"))
                    else (    PageProgram(i, s, caddr, raddr, data) ;
                              return nil
                         )
       end
    pre buffer.Size () > 5
       ;
```

```
private
    SimulateError : char* -o-> ErrorMessage
    SimulateError (e) ≜
        return "ERROR : :   " ⌒ e;
private
    GetColAddr : () -o-> ColAddress
    GetColAddr () ≜
        (    dcl bytes : Data* := buffer.GetN (2);
             return new ColAddress() (bin-to-nat ((bytes (1) ⌒ bytes (2))))
        )
    pre  buffer.Size () ≥ 2
        ;
```

4.2.4 Get Arguments Operations

```
private
    GetRowAddr : ℕ1 × ℕ1 -o-> RowAddress
    GetRowAddr (nblocks, npages) ≜
        (    dcl bytes : Data* := buffer.GetN (3);
             return new RowAddress()
                        (
                         bytes (1),
                         bytes (2),
                         bytes (3),
                         nblocks,
                         npages)
        )
    pre  buffer.Size () ≥ 3
        ;
```

```
private
    GetCode : () -o-> Byte
    GetCode () ≜
        return buffer.Get ()
    pre  buffer.Size () ≥ 1
        ;
```

private
    $GetAll : Hex \xrightarrow{o} [Byte^*]$
    $GetAll\,(hex) \triangleq$
      (dcl $data : Data^* := buffer.GetN\,(buffer.Size\,() - 1)$,
            $code : Hex := byte\text{-}to\text{-}hex\,(buffer.Get\,())$;
        if $code \neq hex$
        then return nil
        else return $data$
)
    pre $buffer.Size\,() > 0$
      ;

5 Final considerations

At this point, the developed model abstracts the behavior of a flash memory device in a functional way, so that it's usable has a storage device. It simulates a data path between the device and the host and includes bad block checking.

To reach this stage it took approximately 48 hours of work per person involved. That sums up to 96 hours, since the team has two elements.

For future development, we would like to point some directions. One of which is a refinement step, that would benefit greatly (really much time!!) it's performance, that step would aim to introduce some counters in the target's definition to store information on, how many LUNs, how many blocks per flash and how many pages per block the target has. These counters would save much time in invariant verification on Block, Flash and LUN type definitions.

There are many aspects to deal with if one wants bring the model closer to the hardware behavior, according to the specification [ONFIS06] that guided this project. The first aspect we would like to point out refers to the signals that control the device operations. Both, device and host, watch or stimulate signals when want to output, or respectively input data from/to the other. The signals variations have well defined timings and durations (has described in the figures included with every command definition on Section 3) which could be included in the model has is explained in [RealTimeVDM++]. But, to realistically model this aspect, it would be necessary to thoroughly investigate Sections 1 and 2 of [ONFIS06], which we invite the interest reader to take a look at.

Secondly, interleaved operations could also be addressed and it would lead to concurrency introduction in the model. Has far has the *VDM++ Toolbox* goes there won't be any obstacles to concurrency.

References

[LangManPP] CSK SYSTEMS CORPORATION,
The VDM++ Language Manual 1.1,
CSK Group.

[ONFIS06] Hynix Semiconductor, Intel Corporation, MicronTechnology, Inc., Phison Electronics Corp., Sony Corporation, STMicroelectronics,
Open NAND Flash Interface Specification revision 1.0,
ONFI Workgroup

[ValDes] Fitzgerald, Jonh, Larsen, Peter Gorm,
Validated Designs for Object-oriented Systems,
Springer, 2005.

[RealTimeVDM++] Verhoef, Marcel, Larsen, Peter, Hooman, Jozef
Modeling and Validating Distributed Embedded Real-Time Systems with VDM++,
Springer Berlin, 2006.