Formalising Flash Memory: First Steps

Andrew Butterfield
School of Computer Science and Statistics
Trinity College Dublin
Dublin 2, Ireland
Andrew.Butterfield@cs.tcd.ie

Jim Woodcock
Department of Computer Science
University of York
Heslington, York, YO10 5DD, UK
Jim.Woodcock@cs.york.ac.uk

Abstract

We present first steps in the construction of formal models of NAND Flash memory, based on a recently emerged open standard for such devices. The model is at a level of abstraction that captures the internal architecture of such a device, as well as the commands that are used to operate it. The model is intended as a key step in a plan to develop a verified filestore system, by providing a description of the hardware devices that would be used in it implementation.

1. Introduction

The "Grand Challenge in Computing" [4] on Verified Software [17], has a stream focussing on mission-critical filestores, such as may be required for space-probe missions [7]. Of particular interest are filestores based on the relatively recent NAND Flash Memory technology, now very popular in portable datastorage devices such a MP3 players and datakeys. Flash memory is seen as ideal for these kinds of missions as it has good physical handling properties, being non-volatile, shock-resistant and capable of operating under a wide range of pressures and temperatures. It also has the vary valuable property, for space-borne vehicles, of having no moving, and in particular, no rotating parts.

There are two types of Flash Memory: (i) NOR Flash Memory, which can be programmed (written) at byte level, but must be erased at block level, is relatively slow, but suits random access; and (ii) NAND Flash Memory with higher speed, but where programming must be done at the page level, making it a sequential access device. The former suits non-volatile core-memory, whilst the latter is suited to implementing data-stores and file-systems.

This paper describes an initial formal model of NAND Flash Memory, based on the recently released specification of same from the "Open NAND Flash Interface (ONFI)" consortium [5]. This specification covers such details as the device packaging pin-outs, electrical and timing characteristics of the device pins, timing diagrams for the operations, that are all beyond the scope of this paper. This paper focusses on the structural aspects of the devices, *i.e*, their internal organisation, and an abstract view of the behaviour of those operations deemed mandatory by the standard.

It should also be noted that the model presented in this paper is not an abstract specification of a system to be developed, as is the "traditional" role of a formal specification, with a view to avoiding "implementation bias", in order to allow a developer freedom to seek the best solution. Instead we are modelling an existing artefact (the ONFI specification) and the real devices that already exist or that are likely to come into existence in the near future. As a consequence there will be clear examples of implementation bias in this model.

We briefly describe related work in §2, and then in §3 we describe the internal organisation of NAND Flash devices, while in §4 we describe the operations that are mandatory according to [5]. We finish (§5) with a discission of the future progression of this work. We use the Z notation [15], [18] in this paper.

2. Related Work

There is no work on the formal modelling of NAND Flash devices known to the authors, but there has been a considerable body of work done on formal models of file systems, and the technical, usage and reliability aspects of NAND Flash devices.

Formal aspects of file systems have covered specifications [13, 12, 3] and approaches towards their verification [2]. Some recent work has also looked at applying model-checking techniques to entire file systems [20], with considerable success.

There has been a wide range of material published regarding the implementation of file systems on NAND Flash memory, most of which utilise some form of log-structuring [6, 19, 10, 9]. Of interest to a potential space-borne application are techniques that use NAND flash to implement low-power file caches for mobile devices [11, 8]. A key feature of these schemes is the need to cope with the accumulation of errors over time, the mechanism of which is very well understood [1, 14]. Of particular interest is a recent patent application filed by Microsoft for a controller that hides the faults behind an interface that looks like an ideal perfect NAND flash device [16].

3. Flash Memory Structure

3.1. Memory Organisation

The basic data unit in a Flash memory is either a Byte (8 bits), or a Word (16 bits), depending on the type of device.

$$\begin{array}{ll}
Bit & == \{0,1\} \\
Byte & == \{\, s : \mathrm{seq}\, Bit \mid \#s = 8 \,\} \\
Word & == \{\, s : \mathrm{seq}\, Bit \mid \#s = 16 \,\}
\end{array}$$

We are going to abstract away from this little detail, and assume a given type called *Data* that denotes the basic information unit:

$$[Data]$$

A page is an array of data items, consisting of a main page, plus some “spare” locations. This is the basic unit for writing, or programming. The spare locations are designed to assist with error detection and correction. In Z we model this as a schema as there is a link between the page count and the column address type used to access items on such a page. The page count is not a global constant, but is in fact part of the (fixed) state of an ONFI device that characterises it (see §4.3 for more details). The page size must be a power of two, and the column address bits must be sufficient to address both the mainpage as well as the spare area.

$$\begin{array}{|l}
\hline
\textit{PageDim} \\
pageaddrsize, coladdrsize : \mathbb{N}_1 \\
pagecount, spare : \mathbb{N}_1 \\
ColAddr : \mathbb{F}\,\mathbb{N} \\
Page : \mathbb{P}(\mathbb{F}\,\mathbb{N} \times Data) \\
\hline
pagecount = 2^{pageaddrsize} \\
pagecount + spare \leq 2^{coladdrsize} \\
ColAddr = 0 \,..\, pagecount + spare - 1 \\
Page = ColAddr \rightarrow Data \\
\hline
\end{array}$$

A block is a collection of pages, and is the smallest unit to which an erase operation can be applied. The number of pages per block is constrained to a multiple of 32.

$$\begin{array}{|l}
\hline
\textit{BlockDim} \\
pagesperblock : \mathbb{N}_1 \\
PageAddr : \mathbb{F}\,\mathbb{N} \\
Block : \mathbb{P}(\mathbb{F}\,\mathbb{N} \times Page) \\
\hline
PageAddr == 0 \,..\, pagesperblock - 1 \\
\exists\, n : \mathbb{N}_1 \bullet pagesperblock = 32 * n \\
Block = PageAddr \rightarrow Page \\
\hline
\end{array}$$

A logical unit (LUN) is the smallest sub-entity within a device that is capable of operating independently. It comprises a collection of blocks, along with at least one page-register and a status register. The page-registers are used as temporary locations while data is being transferred to and from the LUN. For present purposes we assume a single page-register. We define a LUN schema as follows (leaving the details of the status register to follow):

$$\begin{array}{|l}
\hline
\textit{LUN} \\
PageDim \\
BlockDim \\
blocksperLUN : \mathbb{N}_1 \\
SR : Status \\
PR : Page \\
BlkAddr : \mathbb{F}\,\mathbb{N} \\
blks : BlkAddr \rightarrow Block \\
\hline
BlkAddr = 0 \,..\, blocksperLUN - 1 \\
\hline
\end{array}$$

The status register has 8 bits, of which 5 bits currently have defined meanings. Two of these (FAILC, ARDY) are out of the scope of the current model leaving the following three to be considered:

FAIL Set if a program or erase operation failed.

RDY Set when the LUN is ready to perform a command

WP Write-protect flag.

Note that the FAIL flag is only valid when RDY is being asserted, while WP is always valid.

$$\begin{array}{l}
\mathit{Flag} ::= \mathit{fail} \mid \mathit{rdy} \mid \mathit{wp} \\
\mathit{Status} == \mathbb{P}\,\mathit{Flag} \\
\mathit{validStatus}\_ == \{ s : \mathbb{P}\,\mathit{Flag} \mid \mathit{fail} \in s \Rightarrow \mathit{rdy} \in s \}
\end{array}$$

The power-up status is that the device is not ready and is write-protected. Once reset (power-up or command) is complete, then the status becomes ready and write protected.

A target, within a device, is the smallest unit that can communicate independently off-chip. It is made of of one or more logical units;

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{LUNDim}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\mathit{LUNspertarget} : \mathbb{N}_1 \\
\mathit{LUNAddr} : \mathbb{F}\,\mathbb{N} \\
\mathit{Target} : \mathbb{P}(\mathbb{F}\,\mathbb{N} \times \mathit{LUN}) \\
\hline
\mathit{LUNAddr} = 0\,..\,\mathit{LUNspertarget} - 1 \\
\mathit{Target} == \mathit{LUNAddr} \rightarrow \mathit{LUN} \\
\hline
\end{array}$$

A device (single NAND Flash chip) encapsulates a number of Targets, numbered from 1 upwards:

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{TargetDim}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\mathit{targetsperdevice} : \mathbb{N}_1 \mathit{TgtId} : \mathbb{F}\,\mathbb{N}_1 \\
\mathit{Device} : \mathbb{P}(\mathbb{F}\,\mathbb{N}_1 \times \mathit{Target}) \\
\hline
\mathit{TgtId} = 1\,..\,\mathit{targetsperdevice} \\
\mathit{Device} = \mathit{TgtId} \rightarrow \mathit{Target} \\
\hline
\end{array}$$

We can then define the state of an ONFI-compliant NAND Flash Memory device as the aggregation of the definitions and constraints introduced to date:

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{NANDFlash}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\mathit{PageDim} \\
\mathit{BlockDim} \\
\mathit{LUNDim} \\
\mathit{TargetDim} \\
\mathit{device} : \mathit{Device} \\
\mathit{maxbad} : \mathbb{N} \\
\hline
\end{array}$$

The manufacturer provides a guarantee regarding how many blocks will be good in a shipped device, as captured by variable *maxbad*.

3.2. Memory Addressing

The address data sent into a device conceptually splits into two parts, the row and column addresses. The column address corresponds to an index into a page, while the row address identifies which page is currently being accessed. The row address is itself obtained by concatenating the LUN, Block and Page addresses in that order:

$$\begin{array}{ll}
\mathit{RowAddr} & == (\mathit{LUNAddr} \times \mathit{BlkAddr}) \times \mathit{PageAddr} \\
\mathit{Address} & == \mathit{RowAddr} \times \mathit{ColAddr}
\end{array}$$

The target under consideration is identified by specific pins on the device, rather than by any address bits.

3.3. Device Initial State

When shipped from the factory, a device will be completely erased (all logic '1's). The only exception to this is for those blocks identified as bad at manufacture time. These blocks will have zeros programmed somewhere into the spare parts of either their first or last pages.

$$\begin{array}{|l}
\mathit{erased}, \mathit{zeroed} : \mathit{Data} \\
\mathit{erasedBlk} : \mathit{Block} \\
\hline
\mathit{erased} \neq \mathit{zeroed} \\
\forall p : \mathit{PageAddr};\ c : \mathit{ColAddr} \bullet \\
\qquad \mathit{erasedBlk}(p)(c) = \mathit{erased}
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{ShipFlash}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta \mathit{NANDFlash} \\
\mathit{quality}? : \mathbb{N} \\
\mathit{badblocks}? : \mathbb{F}(\mathit{TgtId} \times \mathit{LUNAddr} \times \mathit{BlkAddr}) \\
\hline
\#\mathit{badblocks}? \leq \mathit{maxbad}' \\
\mathit{maxbad}' = \mathit{quality}? \\
\forall t : \mathit{TgtId};\ \ell : \mathit{LUNAddr};\ b : \mathit{BlkAddr} \bullet \\
\qquad \mathbf{if}\ (t, \ell, b) \in \mathit{badblocks}? \\
\qquad \mathbf{then}\ \mathit{defectMarked}(\mathit{device}', t, \ell, b) \\
\qquad \mathbf{else}\ (\mathit{device}'(t)(\ell)).\mathit{blks}(b) = \mathit{erasedBlk} \\
\hline
\end{array}$$

A defective page is indicated by having a zeroed data item somewhere in the spare area of its first or last page:

$$\begin{array}{l}
\mathit{defectMarked}(\mathit{dev}, t, l, b) = \\
\qquad \exists p : \mathit{PageAddr};\ c : \mathit{ColAddr} \bullet \\
\qquad\qquad (p = 0 \vee \\
\qquad\qquad\qquad p = \mathit{pagesperblock} - 1) \\
\qquad\qquad \wedge\ c \geq \mathit{pagecount} \\
\qquad\qquad \wedge\ ((\mathit{dev}(t)(l)).\mathit{blks}(b)(p)(c)) = \mathit{zeroed}
\end{array}$$

The state of the device on-power up is captured in this model by the Reset Operation (see §4.1).

3.4. Structure Summary

The model as presented matches very closely the levels of hierarchy described in the ONFI specification,

Read	Change Read Column
Page Program	Change Write Column
Read Status	Block Erase
Read ID	Read Parameter Page
Reset	Write Protect

Figure 1. ONFI Mandatory Operations

and it may appear: that (i) this is hierarchy for hierarchy's sake; and (ii) this is at too low a level for formal modelling. Nevertheless, each step of the hierarchy captures a distinct change in how the device is accessed and operated, and awareness of these distinctions is important when developing systems where performance is important. The ONFI specification also gives descriptions of finite state machines (FSMs) that capture the behaviour of targets and LUNs, viewing these as separate machines communicating with one another. By capturing the target/LUN distinct at this level, we facilitate future work in showing that the FSM view is a refinement of this one. At the level of this model, the only real complexity is the nesting of the various maps. This is one example of the implementation bias alluded to in the introduction.

4. Flash Memory Operations

Adopting the terminology of [5], we use the term *device* to refer to the NAND Flash memory under consideration, and the term *host* to refer to the system in which it is deployed. We also only consider those operations that are deemed mandatory by the specification (Fig. 1). It should be noted that the ONFI specification does not consider Write Protect to be an "operation", insofar as it is not listed in the "Command Set" table [5, Table 14, p39]. However, it can be invoked/revoked by a signal transition on a target-specific device pin when no other command is executing on the device. We shall treat this as a (mandatory) operation.

We shall conceptually divide the operations into two groups: simple, that involve the transfer of a small fixed amount of information; and complex, that require the transfer of an arbitrary amount of data. At this level of abstraction the difference is moot, but the distinction captures the fact that the complex operations can in principle be interleaved with each other and with certain simple operations. The complex operations are Read, Page Program and Read Parameter Page.

The various operations work at different levels, device, target, LUN, block or page. In this paper we describe the operations at the appropriate level of granularity, and then rely on the concept of promotion [18, Chp. 13] to lift these to the device level. We shall not give details of the promotion.

4.1. Reset Operation

The reset operation simply puts the device into the state it is in immediately after power-up. As NAND Flash is non-volatile memory, this does not mean that any data is altered or lost. At this level of abstraction the only change is the setting of status flags to indicate readiness and write-protection for all targets.

$$\begin{array}{l}
\textit{Reset} \\
\hline
\Delta \textit{NANDFlash} \\
\hline
\forall t \in \textit{TgtId};\ \ell \in \textit{LUNAddr} \bullet \\
\quad (\textit{device}'(t)(\ell)).\textit{SR} = \{\textit{RDY}, \textit{WP}\} \\
\wedge\ (\textit{device}'(t)(\ell)).\textit{PR} = (\textit{device}(t)(\ell)).\textit{PR} \\
\wedge\ (\textit{device}'(t)(\ell)).\textit{blks} = (\textit{device}(t)(\ell)).\textit{blks} \\
\hline
\end{array}$$

4.2. Read ID Operation

There are two forms of identifier that can be read from an ONFI-compliant device. The first is the manufacturers own JEDEC[1]and device identifiers, which we model as given types:

$$[\textit{JEDEC}] \qquad [\textit{DevId}]$$

The second is the ONFI signature that confirms the device is compliant.

$$\textit{ONFI} = \langle 79, 78, 70, 73 \rangle$$

The operation has an argument that specifies which identifier form is required:

$$\textit{IdType} ::= \textit{onfisig} \mid \textit{jedecid}$$

We model the operation as having as input a boolean that if true requests the ONFI signature:

$$\begin{array}{l}
\textit{ReadId} \\
\hline
\Xi \textit{NANDFlash} \\
\textit{idtype}? : \textit{IdType} \\
\textit{onfi}! : \mathbb{N}^4 \\
\textit{jedec}! : \textit{JEDEC} \\
\textit{devid}! : \textit{DevId} \\
\hline
\textit{idtype}? = \textit{onfisig} \Rightarrow \textit{onfi}! = \textit{ONFI} \\
\hline
\end{array}$$

We leave this very underspecified, simply asserting what the outcome is if the ONFI signature is requested.

The ONFI specification defines a broad family or NAND Flash Devices, with a common packaging and

[1]Joint Electron Device Engineering Council

pin-out for all. A key feature of this standard is that it allows for the swapping of one device for another in a host, even if the replacement device has different parameters such as page, block-size, or even number of targets. The specification describes a number of procedures, both at the hardware and operating level for a host upon power-up to interrogate the device in order to see exactly what its parameters are.

The first part of this process simply involves a Reset followed by successive Read Id operations, starting with target 1 and proceeding through increasing target-ids until the operation fails to return the valid ONFI signature. This establishes the number of targets present.

4.3. Read Parameter Page Operation

Once the targets in a device are known, then the host needs to establish the parameters associated with each target. This is done by performing a read of the parameter page [5, pp42–52], which is always 256 bytes long, with a format common to all ONFI devices. The last two bytes are a cyclic redundancy checksum (CRC), which are used to ensure no error occurred when reading the page.

$$CRC ::= crcok \mid crcfail$$

Will will only model those parts of the parameter page that have relevance at the level of abstraction we have been using for this model. Essentially this comprises those parameters to do with page and block sizes and related dimensions.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ParameterPage}} \\
\hline
bytesPerPage, sparePerPage : \mathbb{N}_1 \\
pagesPerBlock : \mathbb{N}_1 \\
blocksPerLUN, maxBadPerLUN : \mathbb{N}_1 \\
\hline
\end{array}$$

There are multiple redundant copies of the parameter page, and in the event of CRC errors, these are read in succession until it is possible to reconstruct the page. The input to the operation is the number of the parameter page, starting from zero. The output is either an error indication (CRC failed) or a valid parameter page. We first model the case of a CRC failure:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ReadParameterPageFail}} \\
\hline
\Delta NANDFlash \\
pno? : \mathbb{N} \\
crc! : CRC \\
parpage! : ParameterPage \\
\hline
crc! = crcfail \\
\hline
\end{array}$$

Next we describe a successful read, in which case the page will correctly describe the parameters of the device:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ReadParameterPageOK}} \\
\hline
\Delta NANDFlash \\
pno? : \mathbb{N} \\
crc! : CRC \\
parpage! : ParameterPage \\
\hline
crc! = crcok \\
parpage!.bytesPerPage = pagecount \\
parpage!.sparePerPage = spare \\
parpage!.pagesPerBlock = pagesperblock \\
parpage!.blocksPerLUN = blocksperLUN \\
parpage!.maxBadPerLUN = maxbad \\
\hline
\end{array}$$

We have implicitly assumed in this example that *Data* is the same as *Byte*. We then define the whole operation as the non-deterministic choice between success or falure:

$$\begin{array}{l}
ReadParameterPage \mathrel{\widehat=} \\
\qquad ReadParameterPageFail \\
\qquad \lor ReadParameterPageOK
\end{array}$$

4.4. Write Protect Operation

We model the Write Protect operation as a command that sets the target WP flags according to a boolean input:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{WriteProtect}} \\
\hline
tgt, tgt' : Target \\
wp? : \mathbb{B} \\
\hline
\forall \ell : LUNAddr \bullet \\
\qquad (tgt'(\ell)).SR \\
\qquad\qquad = setWP(wp?, (tgt(\ell)).SR) \\
\land (tgt'(\ell)).PR = (tgt(\ell)).PR \\
\land (tgt'(\ell)).blks = (tgt(\ell)).blks \\
\hline
\end{array}$$

Here the *setWp* function sets the Status flag according to the boolean input:

$$\begin{array}{l}
setWP(wp?, status) \\
\qquad = \mathbf{if}\ wp?\ \mathbf{then}\ status \cup \{wp?\} \\
\qquad\qquad\quad \mathbf{else}\ status \setminus \{wp?\}
\end{array}$$

4.5. Page Program Operation

The page is declared as the basic unit of programming (writing), but we will describe the operation at the LUN level, because the page-register plays an important role. This is because programming has two main phases, *upload* and *store*[2]. During the upload phase, the command to program is issued to the device, along with the data, which is loaded into the page register associated with the LUN. In the store phase, the data is transferred into the page identified by the row address. This two-phase operation allows the interleaving of operations between different LUNs, as one can be uploading, whilst the other is storing. At the level of abstraction of this model, we view all operations as atomic, effectively defining their overall effect, so we cannot model interleaving directly. These aspects of NAND Flash behaviour, important for performance reasons, are left for more detailed future modelling work.

Given that the page is "the smallest addressable unit for read and program operations." [5, §1.3.1.9, p2], it may a matter of puzzlement as to why any column addresses exist, as these identify data units *within* a page. The resolution of this conflict lies in the fact that these devices allow the programming of part of a page, by supplying a starting column address, and just the data that is to be changed. This obviates the need to send page data consisting largely of blanks if we are only programming a few data items. Clearly the amount of data supplied must fit into a single page, plus the spare data area, given the starting address. Other preconditions for this operation is that the LUN's status indicated ready, and the LUN is not write protected.

First, we describe a successful page program operation:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PageProgramOK}} \\
\hline
\Delta LUN \\
b? : BlkAddr \\
p? : PageAddr \\
c? : ColAddr \\
data? : \mathrm{seq}\, Data \\
\hline
rdy \in SR \\
wp \notin SR \\
c? + \#data? < pagecount + spare \\
SR' = \{rdy\} \\
PR' = overwrite(PR, c?, data?) \\
blks' = blks \oplus \{b? \mapsto (blks(b?) \oplus \{p? \mapsto PR'\})\} \\
\hline
\end{array}
$$

We have introduced a function *overwrite* that captures the changes in a page where data is overlaid at a given starting address:

$$
\begin{array}{|l}
overwrite : Page \times ColAddr \times \mathrm{seq}\, Data \rightarrow Page \\
\hline
overwrite(p, c, d) \\
\quad = (\{0 \mathinner{\ldotp\ldotp} c-1\} \upharpoonleft p) \frown d \\
\qquad\quad \frown (\{c + \#d \mathinner{\ldotp\ldotp} pagecount + spare - 1\} \upharpoonleft p)
\end{array}
$$

[2]Our terminology, not ONFI's

However, a program operation may also fail, in a manner detectable by the device, in which case the status records this fact, and we assume the contents of both the page register and target page are now undefined (and most likely also different):

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PageProgramFail}} \\
\hline
\Delta LUN \\
b? : BlkAddr \\
p? : PageAddr \\
c? : ColAddr \\
data? : \mathrm{seq}\, Data \\
\hline
rdy \in SR \\
wp \notin SR \\
c? + \#data? < pagecount + spare \\
SR' = \{rdy, fail\} \\
\exists\, garbage : Page \bullet PR' = garbage \\
\exists\, junk : Page \bullet blks' \\
\qquad = blks(?b) \oplus \\
\qquad\qquad (\{b? \mapsto blks(b?) \oplus \{p? \mapsto junk\}\}) \\
\hline
\end{array}
$$

The page program operation is then modelled as the non-deterministic choice between these two:

$$
\begin{array}{l}
PageProgram \\
\qquad = PageProgramOK \lor PageProgramFail
\end{array}
$$

4.6. Read Status Operation

The Page Program operation does not return a success/fail indicator as an output, but rather signals it by setting a status register bit. This requires the host to interrogate the device to determine the contents of this register in order to establish if the programming was successful. The Read Status operation however does not specify a LUN, but only identifies the target in question. What is returned is an amalgamation of the individual LUN status registers within the designated target:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ReadStatus}} \\
\hline
\Xi NANDFlash \\
t? : TgtId \\
status! : Status \\
\hline
(\mathbf{let}\ ss == \{\ell : LUNAddr \bullet device(t?)(\ell).SR\} \\
\bullet\ status! = statusmerge(ss)) \\
\hline
\end{array}
$$

The merged status records *rdy* if all the individual ones do, and reports fail in the event that any LUN has failed. However the ONFI document [5] is unclear at this point, as it states (p56) that the value of the FAIL bit is only valid when RDY is asserted, but we are also told (p54) that the reported value of FAIL is the logical-or of all the individual FAIL bits. In order to maintain the invariant for status registers, we only assert fail in all LUNs report ready an at least one reports fail:

$$
\begin{array}{|l}
statusmerge : \mathbb{P}\,Status \rightarrow Status \\
\hline
rdy \in statusmerge(ss) \Leftrightarrow rdy \in \bigcap ss \\
fail \in statusmerge(ss) \Leftrightarrow fail \in \bigcup ss
\end{array}
$$

4.7. Change Write Column Operation

Just as the page program operation facilitates writing a small part of a page, as described above, the ONFI specification also defines a related operation that assist when we want to write a number of small fragments to a page. The operation called Change Write Column is a way of supplying a fresh column address and data to an existing page program operation, without incurring the full overhead of issuing a fresh Page Program command, plus all the row address information.

We cannot capture this distinction at the level of abstraction in this model, where it is simply modelled as another Page Program operation to the same page that has just been programmed.

4.8. Read Operation

A Read operation is similar in many respects to a program operation, except for the direction of information flow, with a *retrieve* phase where data is fetched from a block page into the page-register, and a *download* phase where the data is transferred off-chip.

$$
\begin{array}{|l}
\multicolumn{1}{l}{ReadSig} \\
\hline
\Delta LUN \\
b? : BlkAddr \\
p? : PageAddr \\
c? : ColAddr \\
data! : \mathrm{seq}\,Data \\
blks' = blks \\
ok! : \mathbb{B} \\
\hline
rdy \in SR \\
c? + \#data! < pagecount + spare \\
SR' = rdy \\
\hline
\end{array}
$$

One key difference is that no status regarding success or failure is reported. It is up to the host to use techniques like CRC in order to ensure the integrity of the data read out.

$$
\begin{array}{|l}
\multicolumn{1}{l}{ReadOK} \\
\hline
ReadSig \\
\hline
PR' = blks(b?)(p?) \\
data! = \{c \,..\, c + \#data! - 1\} \upharpoonright PR' \\
ok! \\
\hline
\end{array}
$$

We cannot simply model this as the disjunction of a successful operation schema and one denoting failure as the distinction between success and failure is not observable, and we would end up with a model that allowed any data values to be returned by a read. We could assume that read always succeeds, and that any errors that occur do so as a result of some state-changing operation such as page programming or block erasure. This is not totally accurate as it effectively makes read failure observable indirectly, in that we can predict it will occur if we read a page after a failed program of that same page.

Ultimately there is no guarantee we can observe a read failure, only techniques for reducing the probability of un-observed failure down to an acceptable low level. As an appropriate separation of concerns, we shall therefore model the read operation as having an output that reports success or failure, and leave the issue of ensuring that it is very unlikely to be in error to the design of the error-detection and correction mechanisms used in an implementation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{ReadFail} \\
\hline
ReadSig \\
\hline
\exists garbage : Page \bullet PR' = garbage \\
\exists junk : Page \bullet data! = \{c \,..\, c + \#data! - 1\} \upharpoonright junk \\
\not{o}k! \\
\hline
\end{array}
$$

Read can succeed or fail:

$$Read \mathrel{\widehat{=}} ReadOK \lor ReadFail$$

4.9. Change Read Column Operation

Just as for page program, we can read different fragments of a page out simply by supplying a fresh column address. As before, we simply model this as another Read operation, at this level of abstraction.

4.10. Block Erase Operation

Block erasure simply sets every item of every page of the designated block to the erase values (all bits set to '1'). It sets the LUN status register appropriately, so we consider this a LUN-level operation:

$$\begin{array}{|l}
\hline \textit{BlockEraseOK} \\
\Delta LUN \\
b? : BlkAddr \\
\hline
rdy \in SR \\
wp \notin SR \\
SR' = \{rdy\} \\
PR' = PR \\
blks' = blks \oplus \{b? \mapsto erasedBlk\} \\
\hline
\end{array}$$

The status register does record failure here if it occurs, in which case the block contents are undefined:

$$\begin{array}{|l}
\hline \textit{BlockEraseFail} \\
\Delta LUN \\
b? : BlkAddr \\
\hline
rdy \in SR \\
wp \notin SR \\
SR' = \{fail, rdy\} \\
PR' = PR \\
\exists junk : Block \bullet blks' = blks \oplus \{b? \mapsto junk\} \\
\hline
\end{array}$$

$$BlockErase = BlockEraseOK \lor BlockEraseFail$$

5. Future Work

These are only first steps and there is a lot of key work to be done. Formal models will be needed to capture the fact that individual targets within a device can be operating concurrently, with interleaving of data-transfers. Also, the behaviour of these devices is described in the specification document using two finite-state machines, one for target behaviour, the other for LUN activity. A model of these needs to be shown as a refinement of the abstract operation model presented in this paper. This requires that the existing operators need to be expressed in terms of basic host/device communication actions, which transfer a single item of information, such as a command, address or data byte/word. A sketch of the structure of such a model is given as an appendix (Appendix **??**).

The model needs to be extended to cover the non-mandatory operations of the standard, many of which provide improved performance, via various forms of caching and interleaving. It is to be anticipated that any filestore will make extensive use of these in order to meet mission performance targets. In many cases a useful model of these will require that the operations are broken down to a smaller granularity.

NAND Flash devices are prone to the unrecoverable failure of blocks over time, through what basically amounts to an ageing process, that is strongly workload related. This requires so-called "wear-levelling" algorithms to minimise the failure rate, as well as some form of fault tolerance to cope with the failures that do occur. This requires us to model failure properly, with a particular emphasis on the fact that such failures have a persistent and lasting effect.

We also need to look upwards (in an abstract sense) from the NAND devices to model how they are used to give an illusion of ideal behaviour. Whilst the spare area associated with each data page is there to assist with error detection and recovery, the Flash devices themselves have no fault-tolerant mechanisms built-in. Instead the devices have to be interfaced to a controller that manages the faults, and presents a fault-free model of data storage to the level above.

References

[1] Seiichi Aritome et al. Reliability issues of flash memory cells (invited paper). *Proc. of the IEEE*, 81(5):776–788, May 1993.

[2] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin C. Rinard. Verifying a file system implementation. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 373–390. Springer, 2004.

[3] Maritta Heisel. Specification of the unix file system: A comparative case study. In *Algebraic Methodology and Software Technology*, pages 475–488, 1995.

[4] Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[5] Hynix Semiconductor et al. Open NAND Flash Interface Specification. Technical Report Revision 1.0, ONFI, www.onfi.org, 28th December 2006.

[6] Han joon Kim and Sang goo Lee. A new flash memory management for flash storage system. In *COMPSAC*, page 284. IEEE Computer Society, 1999.

[7] Rajeev Joshi and Gerard J. Holzmann. A mini challenge: Build a verifiable filesystem. In *Proc. Verified Software: Theories, Tools, Experiments (VSTTE), Zürich*, 2005.

[8] Taeho Kgil and Trevor Mudge. Flashcache: a nand flash memory file cache for low power web servers. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 103–112, New York, NY, USA, 2006. ACM Press.

[9] Seung-Ho Lim and Kyu-Ho Park. An efficient NAND flash file system for flash memory storage. *IEEE Transactions on Computers*, 55(7):906–912, July 2006.

[10] Charles Manning. Introducing YAFFS, the first NAND-specific flash file system. *LinuxDevices.com*, Sep 2002.

[11] B. Marsh, F. Douglis, and P. Krishnan. Flash memory file caching for mobile computers. In Trevor N. Mudge and Bruce D. Shriver, editors, *Proceedings of the 27th Annual Hawaii International Conference on*

System Sciences, Vol. I: Architecture, HICSS'94 (Maui, Hawaii, January 4-7, 1994), volume 1, pages 451–460, Los Alamitos-Washington-Brussels-Tokyo, 1994. IEEE Computer Society Press.

[12] Silvio Lemos Meira, Ana Lúcia C. Cavalcanti, and Cassio Souza Santos. The Unix filing system: A MooZ specification. In Kevin Lano and Howard Haughton, editors, *Object-Oriented Specification Case Studies*, The Object-Oriented Series, chapter 4, pages 80–109. Prentice-Hall, New York, NY, 1994.

[13] Carrol Morgan and Bernard Sufrin. *Specification case studies*, chapter Specification of the UNIX filing system, pages 91–140. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1987.

[14] Axel Sikora, Frank-Peter Pesl, Walter Unger, and Uwe Paschen. Technologies and reliability of modern embedded flash cells. *Microelectronics Reliability*, 46(12):1980–2005, 2006.

[15] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, 2nd edition, 1992.

[16] Gregory G. Williams et al. NAND flash memory management. US Patent Application Publication, US2006/0239075 A1, Oct 2006.

[17] Jim Woodcock. First steps in the verified software grand challenge. *IEEE Computer*, 39(10):57–64, 2006.

[18] Jim Woodcock and Jim Davies. *Using Z*. Intl. Series in Computer Science. Prentice Hall, 1996.

[19] David Woodhouse. JFFS: The Journalling Flash File System. *Ottawa Linux Symposium 2001*, Oct 2001.

[20] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst*, 24(4):393–423, 2006.

A. Hardware Interface Model

Here we simply sketch out the shape of the low-level model that will be required to capture more detail of how a host actually interacts with a NAND Flash device. In particular this model would allow us to address impoertant performance and reliability issues, such a wear-levelling, or operation interleaving.

The hardware interface model will require more details about the control-state of a NAND Flash device, in addition to the memory data that makes up the *NANDFlash* model just presented. This is because access to a NAND Flash device is mediated by a sequence of low-level access operations that transfer a single unit of data between the host and the device. Operations such as Page Program or Read are a choreographed sequence of lower level operations, whose behaviour is described by two types of finite-state machine (FSM), described the ONFI specification. One FSM type describes the control behaviour associated with a given LUN [5, §7.1, pp77–89]:

LUN_FSM
$\ldots$

The other type of FSM describes how a target is controlled [5, §7.2, pp90–100]:

$TargetFSM$
$\ldots$

Conceptually the control of a NAND Flash device is described as a hierarchy of communicating FSMs. The host communicates with a target FSM through the low-level access operations. Internally the target FSM communicates with each of its constituent LUN FSMs.

$Hardware$
$fms : TgtId \rightarrow (TargetFSM \times (LUNAddr \rightarrow LUN\_FSM))$
$\ldots$

At this level our operations model a single communication event between host and device in which a single information item (byte or word) is transferred. There are three distinct types of information that the host can send to the device: Command, Address and Data. We model this using three different *Write*... operations.

Commands are modelled as an enumerated (free) type:

$$Cmd ::= reset \mid rdstatus \mid \ldots$$

We send a command to a device by supplying it and the relevant target identifier:

$WriteCommand$
$\Delta Hardware$
$t? : TgtId$
$cmd? : Cmd$
$\ldots$

Addresses model byte-sized fragments of both row and column addresses, so we generally need send an address in pieces, least significant byte first, also identifying the target:

$WriteAddress$
$\Delta Hardware$
$t? : TgtId$
$addr? : Byte$
$\ldots$

The operation *WriteData* is described analogously.

There are two types of information that can be read from a device, and transferred back to the host. Firstly, we can get a data item transferred:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ReadData}}\\
\Delta \mathit{Hardware}\\
t? : \mathit{TgtId}\\
\mathit{data}! : \mathit{Byte}\\
\hline
\ldots
\end{array}$$

Although this operation looks like a query function returning a value, it does in fact change the state. The data item returned comes from the currently active LUN, from the current Read Column address. This address is then incremented to point to the next data item. A similar use of a Write Column address occurs also in the *WriteData* operation. The second way to return information from device to host is to check the ready/busy status of the device:

$$\mathit{Readiness} ::= \mathit{ready} \mid \mathit{busy}$$

This operation is required to ensure that the device has completed operations such as Read or Page Program, where time is required internally by the device to transfer data to/from the relevant page register. We could define a polling operation that returns this status, but in practice, given that it is signalled by a single target-specific device pin, it is more likely to generate an interrupt. Given this, we model it as an operation that waits for it to take the value ready before completing:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{WaitForReady}}\\
\Delta \mathit{Hardware}\\
t? : \mathit{TgtId}\\
\hline
\ldots
\end{array}$$

The post-condition here describes the device state that would be consistent with a completed operation.

A point worth noting is that all these hardware-level operators are total, with the trivial pre-condition *true*.

An operation from the *NANDFlash* model is then modelled as the sequential composition of appropriate hardware operations, as defined in the ONFI specification [5, §5, pp39–70]. For example, the Read Status operation at the hardware level consists of writing the *readstatus* command to the device, and then reading back the status as a data item:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{HardwareReadStatus}}\\
\Delta \mathit{Hardware}\\
t? : \mathit{TgtId}\\
\mathit{status}! : \mathit{Byte}\\
\hline
\exists \mathit{Hardware}'' \bullet\\
\quad \exists \mathit{Hardware}'; \mathit{cmd}? : \mathit{Cmd} \bullet\\
\quad\quad [\mathit{WriteCommand}; \mathit{Hardware}''\\
\quad\quad \mid \mathit{cmd}? = \mathit{readstatus}\\
\quad\quad \theta \mathit{Hardware}' = \theta \mathit{Hardware}'']\\
\quad \wedge\\
\quad \exists \mathit{Hardware} \bullet\\
\quad\quad [\mathit{ReadData}; \mathit{Hardware}''\\
\quad\quad \mid \theta \mathit{Hardware} = \theta \mathit{Hardware}'']
\end{array}$$

The model to be developed here will allow us to formally verify two key and important aspects of the ONFI specification:

FSM Validation We will be able to show that the finite state machines described in the document are a correct implementation of the abstract behaviour of the NAND Flash commands as described in the *NANDFlash* model.

Efficient Interleaving Efficient use of NAND Flash devices involves interleaving of operations, by allowing external host-device transfers to interleave with internal page/page-register data movement. Also, among the optional ONFI operations are cached versions of the data-transfer operations that allow an even higher degree of data transfer speedup. The *Hardware* model can support the verification of the correctness of filestore algorithms that exploit these features. It can also be used to verify the correctness of certain protocols described in the ONFI document regarding the use of the interleaved features. These protocols allow the interleaving to be safely invoked at a higher level of abstraction, such as at the level of the *NANDFlash* model, which makes the verification task somewhat simpler.