V-SPELLS Use Case 2 User Manual

“Flaw Fix via DSL”

TA4 Provatek, LLC / August 2022

Jake Decker / jake.decker@provatek.com

Matt Evenson / matt.evenson@provatek.com

Tim Fraser / tim.fraser@provatek.com

Provatek Technical Report #117

Approved for Public Release, Distribution Unlimited. (DISTAR Case #36830.)

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4026. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or NIWC Pacific.

In the event permission is required, DARPA is authorized to reproduce the copyrighted material for use as an exhibit or handout at DARPA-sponsored events and/or to post the material on the DARPA website.

1. Introduction

This is the user manual for Provatek, LLC's second "flaw fix via Domain-Specific Language (DSL)" V-SPELLS use case distribution. This use case invites the V-SPELLS research teams to:

1. examine a corpus of example legacy C-language device drivers for an emulated raw NAND flash storage device that correctly implement composition and loop structure patterns commonly found in real Linux device drivers,
2. examine a corpus of example legacy device drivers that demonstrate common implementation flaws,
3. invent one or more DSLs for implementing such device drivers,
4. reimplement the legacy drivers in these new DSLs,
5. argue that the new reimplementations are compatible with the bug-free legacy drivers, and
6. demonstrate that the DSLs discourage or rule out the kinds of flaws found in the buggy legacy drivers.

This use case distribution contains a test rig to support these activities. The test rig is a user-mode C program for GNU/Linux operating systems on 64-bit Intel instruction set CPUs. As shown in the diagram in Figure 1, it consists of four components:

Device emulator: an emulator for a simple raw NAND storage device that presents device drivers with an interface that resembles the one that the Linux kernel presents to real device drivers. This interface implements a message-passing composition pattern based on Input/Output registers and General Purpose Input/Output pins. Section 3 describes this interface in detail.

Framework emulator: an emulator that presents device drivers with a collection of interfaces that resemble the ones the Linux kernel's NAND framework presents to real device drivers. Each interface implements a different composition pattern: function call through a jump table, instruction streams through a command interpreter, and modifications to a shared data structure. Section 4 describes these interfaces in detail.

Device driver: researchers can use the test rig's makefile to build an executable that includes one device driver at a time, choosing either one of the examples included in the distribution or (with some modification) one of their own DSL-based reimplementations. The corpus of correct device drivers demonstrates examples of several loop structure patterns common to real device drivers; researchers interested in automated program analysis and reasoning may find these to be an interesting challenge. Section 5 describes these loop structure patterns. It also provides a table that indicates which loop structure and framework interface composition patterns each example driver demonstrates.

Test suite: the makefile builds an executable that includes a suite of functional tests appropriate for its configured device driver. Researchers can use these tests to demonstrate at least partial compatibility between the legacy device drivers and their DSL-based replacements.

Section 2 presents the low-level details of how the device emulator works, how the test rig uses 64-bit Intel CPU hardware watchpoints and software breakpoints to mimic the semantics of real IO registers and GPIO pins, and how the test rig must consequently run as a pair of processes with one using the Ptrace debugging library to trace the other. Section 6 concludes this user manual with detailed instructions on building and executing the test rig.

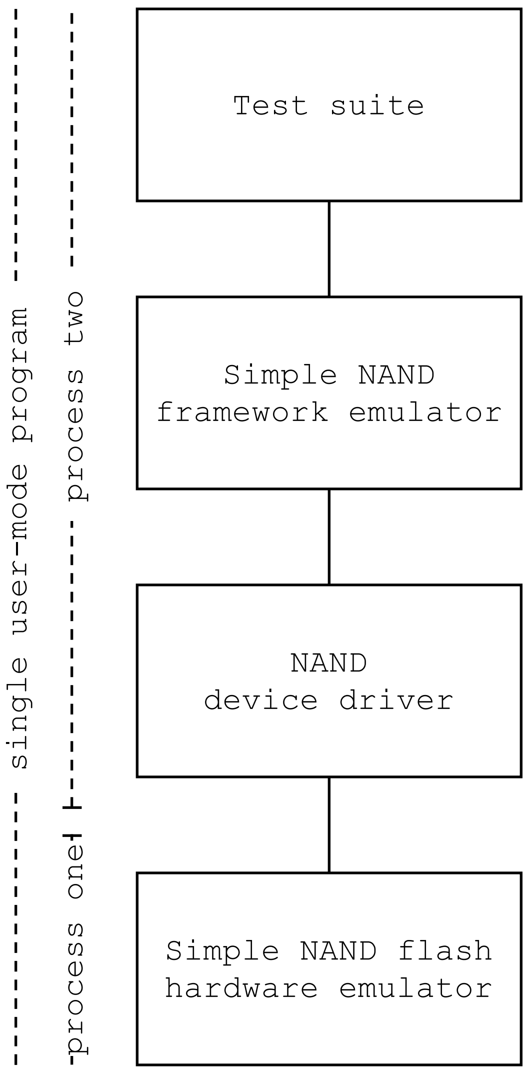

Figure 1 - Test rig architecture.

2. The test rig debugs itself

The most unusual aspect of the test rig's design is its use of the Ptrace process trace library to emulate the semantics of IO registers and GPIO pins. The test rig's source code includes several details specific to 64-bit Intel instruction set CPUs to support this usage; the test rig will build only on GNU/Linux platforms with this kind of CPU. This section explains the need for Ptrace and its impact on the test rig's design.

During test rig runtime, the device driver interacts with the device emulator according to a message-passing protocol described in Section 3. This protocol is synchronous; the device driver makes a request of the device emulator by writing to an IO register and blocks awaiting its reply. The device emulator services the request by writing to an IO register and then blocks awaiting the next request. Logically, control passes back and forth between device driver and device emulator with only one of the two operating at any given moment. Subsection 2.1 describes how the test rig uses hardware watchpoints to implement this message passing through IO registers.

The device emulator does not present any asynchronous or interrupt-driven IO features to the driver. When the relative slowness of device operations forces the driver to sleep until operations complete, the driver polls the device by getting the state of a status GPIO pin rather than registering for an interrupt. A driver can also reset the device to its initial state by setting the state of a reset GPIO pin. Subsection 2.2 describes how the test rig uses software breakpoints to implement the getting and setting of GPIO pin state.

2.1. Emulating IO registers with hardware watchpoints

The device emulator presents an interface to device drivers that consists of three IO registers named command, address, and data. Logically, each IO register is an unsigned one-byte integer. All three are stored in the least significant three bytes of a single eight-byte static volatile unsigned integer word in the initialized data segment of the device driver's address space. The four bytes listed most- to least-significant are unused, command, address, and data.

Device drivers send commands and addresses to the emulated device by writing values to the command and address registers, respectively. For example, in C:

```
#define COMMAND 0x00FF0000

#define ADDRESS 0x0000FF00

static volatile unsigned int ioregisters = 0;

ioregisters = COMMAND;

ioregisters = ADDRESS;
```

Similarly, they can read data values from the emulated device by reading values from the data register.. For example:

```
#define MASK_DATA 0x000000FF

unsigned int data = ioregisters & MASK_DATA;
```

In the real world, the NAND flash storage device would react to each of the device driver's reads and writes to its registers by performing some operation. These operations might change the device's internal state, change the values of the IO registers, or both. The test rig uses the Ptrace process trace library to enable the device emulator to implement these operations.

On startup, the test rig fork()s into two nearly identical processes: the parent tracer process and the child tracee process. Neither the parent tracer process nor the child tracee process call execve(); they both continue to run different parts of the same test rig program. During subsequent runtime, the parent tracer process executes the test rig's device emulator logic. The child tracee executes the remainder of the test rig's logic, including the device driver. The diagram in Figure 1 shows the parent tracer as "process one" and the child tracee as "process two".

The parent tracee process traces (that is, debugs) the child tracee process. Although fork() provides the parent tracer and child tracee with separate address spaces, they each have an ioregister variable at the same location in their initialized data segments. The parent tracer uses the knowledge of their common address layout to set a read/write hardware watchpoint on the IO register variable in the child tracee's address space. Whenever the child tracee reads or writes to its IO register variable, this watchpoint causes it to block and control passes to the parent tracer.

Upon activation, the parent tracer knows the child tracee's device driver logic has either read or written some part of its IO register variable. The device driver logic presumes the device driver is following the driver-device message passing protocol correctly and decides the activation was due to a read or write of a particular logical register based on its present state. In cases where the driver wrote to an IO register, the emulator uses the ptrace() function to retrieve the value it wrote from the child tracee's IO register variable. In cases where the driver read from an IO register, the emulator uses the ptrace() function twice, once to write the value the driver ought to have seen to the child tracee's IO register variable, and once to write the same value to the whichever of the child tracee's general-purpose CPU registers was involved in the read.

Once done, the parent tracer reactivates the child tracee and blocks awaiting the next watchpoint or breakpoint activation.

2.2. Emulating GPIO pins with software breakpoints

The NAND framework emulator component provides device drivers with two GPIO pin support functions that, in its source, appear to do little more than invoke the breakpoint assembly instruction. Both take an argument to specify a particular GPIO pin. The gpio_set() function takes an additional value argument that must be 0 or 1. When the device driver in the child tracee calls either of these functions, the breakpoints pass control to the device emulator in the parent tracer. The device emulator then uses the ptrace library to emulate the effects of interacting with the device. For the gpio_set() breakpoint, it reads the pin and value parameters and sets the state of the indicated emulated GPIO pin accordingly. For the gpio_get() breakpoint, it reads the pin parameter, looks up the emulated state of that pin, and then sets the value gpio_get() will return accordingly. The parent tracer then reactivates the child tracee and blocks itself until the next breakpoint or watchpoint activation.

3. Device emulator

The device emulator component of the test rig emulates a simple NAND flash storage device. It emphasizes features that impact the design of Domain-Specific Languages (DSLs) for device drivers and the proofs that drivers reimplemented in those DSLs remain compatible with the legacy drivers they intend to replace. These features include:

- message passing between the driver and device via IO registers,
- a protocol where commands, parameters, and data must appear in a particular order and with a particular timing, and
- device cursor and cache state variables that are not visible in the driver source yet nonetheless impact the results of its computations.

The device presents drivers with an interface that resembles the one the Linux kernel presents to its device drivers, as opposed to the lower-level abstractions described in the data sheets of typical real-world storage devices:

- the device's GPIO pins appear as function calls, and
- the device's Input/Output (IO) registers appear as addresses in the device driver's address space that have special properties.

The device emulator also exposes some further features that are typical of real-world NAND flash devices:

- its storage is logically divided into regions called erase blocks. These erase blocks contain logical programmable pages, which in turn contain storage bytes,
- drivers may read bytes from storage, and
- drivers may write data to unused pages by programming them and return used pages to an unused state by erasing entire blocks.

The device emulator makes no attempt to emulate further details of real-world NAND flash devices. In particular, repeated program operations do not wear out the storage media.

Subsection 3.1 describes the device in terms of a finite state machine that transitions between states when prompted by the driver via its GPIO and IO register interfaces. Subsection 3.2 describes the portion of the message passing protocol the driver uses to read data from the device. Subsection 3.3 describes the portion for programming (that is, writing data to) the device. Subsection 3.4 describes the portion for erasing data from the device. Subsection 3.5 describes how the driver determines if the device is ready or busy and how it resets the device to its initial state. Subsection 3.6 concludes this section with by explaining the presence of a special c_dummy command and how the test rig uses it to clarify messages that might otherwise be ambiguous.

3.1. Device state machine

The device emulator implements a finite state machine that tracks its primary state in several private state variables that are not directly visible to other test rig components.

Storage: The device emulator's notional storage device provides 16MB of storage. This storage is backed by RAM; data stored does not persist across separate runs of the test rig. This storage is logically divided into 256 erase blocks. These erase blocks are further divided into 256 programmable pages. Each programmable page can store 256 bytes of data.

Cache: When the driver asks the device to read data from its storage, the device slowly reads the data into a cache. Once the data is cached, it rapidly serves the data from the cache to the driver. Similarly, when the driver asks the device to program (that is, write) data to its storage, the device rapidly receives the data from the driver into the cache. Once the data is cached, the driver slowly programs it to its storage. The cache is one programmable page in size.

Cursor: The device uses a single cursor to index both its storage and cache during read and program operations. This cursor is a 3-byte-wide unsigned integer. Logically, the most significant of these three bytes indicates the erase block, the middle byte indicates the programmable page, and the least significant byte indicates the byte in storage. The cursor's least significant byte also indicates the byte in the cache.

Deadline: In real-world NAND flash devices, operations that reading data from storage into cache and programming storage from cache take time to complete. The emulator provides this timing feature; the feature uses a deadline state variable remember the system clock time at which the most recent read or program operation will complete.

Machine state: The device emulator is a state machine that, once initialized at test rig startup, changes its state only in response to prompts by the device driver. The device emulator uses this variable to keep track of its present machine state.

On startup, the test rig initializes the device with all storage bytes, the cache, and deadline state variables cleared to 0, and machine state set to ms_initial_state. The device driver component prompts the device to move from machine state to machine state with one of the following actions:

- a read or write to its ioregisters variable,
- a call to its gpio_get() function, or
- a call to its gpio_set() function.

When the driver prompts the device with one of these actions, the device decides what processing to do based on its current machine state, does that processing, and moves to a new machine state.

Table 1 describes the initial ms_initial_state machine state and how the device responds to driver prompts while in that machine state. It also describes an ms_bug state to aid in driver debugging. The device enters its ms_bug machine state when a buggy driver provides incorrect prompts. It remains in that machine state until the driver resets it to its initial state.

3.2. Reading data from the device

Drivers can read data from the device with the following sequence of actions:

1. Set the command IO register to c_read_setup. This command informs the device that the driver will specify the address of the first byte to read in the following three steps.
2. Set the address IO register to the number of the erase block to read. This command causes the device to store the block number in its private cursor.
3. Set the address IO register to the number of the page in that block to read. This command causes the device to store the page number in its private cursor.
4. Set the address IO register to the number of the first byte in that page to read. This command causes the device to store the byte number in its private cursor. After this third address command is complete, the device's cursor contains the entire three-byte address of the first byte to read.
5. Set the command IO register to c_read_execute. This command causes the device to begin reading the entire page specified by its private cursor into its private cache. Note that it reads the entire page into cache with the page's first byte at the cache's first byte, regardless of the value of the cursor's byte number byte.
6. It takes time for the device to read the entire page into its cache. The device will be busy during the read and return to ready once it is complete. The driver will poll the device's status until it becomes ready.
7. Once ready, the driver will read the value of the data IO register one or more times. Upon each read, the device will set the data IO register to the value of the cache byte corresponding to the byte number portion of its cursor, incrementing the cursor as it goes to index through the cache.

   The portion of the cache the driver will read depends on how it set the byte number and how many times it reads from the data IO register. If the driver sets the byte number to zero, it will begin by reading the first byte in the cache. If it sets the byte number to some other value, it will begin reading somewhere else in the cache. It reads a number of cache bytes equal to the number of times it reads the data IO register.

The device passes through the machine states described in Table 2 to support this series of actions. Once the driver has read the last byte it wishes to read, it has several options:

- The driver can read from the next consecutive page by returning to step #5 and writing c_read_execute to the command IO register. When the driver reads the last byte from one page, step #7 leaves the cursor pointing to the first byte in the next consecutive page. Returning to step #5 causes the device to load that next page into its cache. The device will wrap its cursor back to zero if a read increments it beyond the last page storage.
- The driver can begin programming pages by writing the c_program_setup command to the command IO register.

- The driver can begin erasing blocks by writing the c_erase_setup command to the command IO register.

3.3. Programming data to the device

The driver can program individual full pages. The driver may use the cursor to specify values for only some contiguous run of bytes in the page. Bytes not explicitly specified by the driver will be implicitly zero (0x00). To program some or all data on a particular page, the driver makes the following actions:

1. Set the command IO register to c_program_setup. This command informs the device that the driver will specify the block and page number of the page it wishes to program and the number of the first byte in that page for which it intends to specify a value.
2. Set the address IO register to the number of the erase block to program. This command causes the device to store the block number in its private cursor.
3. Set the address IO register to the number of the page in that block to program. This command causes the device to store the page number in its private cursor.
4. Set the address IO register to the number of the first byte in that page for which the driver wishes to specify a value. This command causes the device to store the byte number in its private cursor. After this third address command is complete, the device's cursor contains the entire three-byte address of the first byte to specify.
5. Once ready, the driver will write the value of the data IO register one or more times. Upon each write, the device will set the cache byte indicated by the cursor to the value of the data IO register, incrementing the cursor as it goes to index through the cache.
6. Set the command IO register to c_program_execute. This command causes the device to begin programming the entire page specified by its private cursor to contain the present contents of the cache. Note that it programs the entire page into storage with the page's first byte at the cache's first byte, regardless of the value of the cursor's byte number byte.
7. It takes time for the device to program the entire page into its cache. The device will be busy during programming and return to ready once it is complete. The driver will poll the device's status until it becomes ready before attempting another operation.

The device passes through the machine states described in Table 3 to support this series of actions. Once the driver has patiently waited for the device to return to ready status, it has several options:

- After the device finishes programming one page in step #7, it leaves its cursor pointing to the start of the next consecutive page. The driver can specify values to program to this page by returning to step #5 and writing values to the data IO register. The device will wrap its cursor back to zero if a program increments it beyond the last page of storage.

- The driver can begin reading bytes from the device by writing the c_read_setup command to the command IO register.
- The driver can begin erasing blocks by writing the c_erase_setup command to the command IO register.

3.4. Erasing device storage blocks

The driver can erase all the pages in erase blocks. This erasure clears all of the bytes in those pages to all-zeroes (0x00). To erase some or all of the erase blocks in storage, the driver makes the following actions:

1. Set the command IO register to c_erase_setup. This command informs the device that the driver will specify the block it wishes to erase.
2. Set the address IO register to the number of the erase block to erase. This command causes the device to store the block number in its private cursor.
3. Set the command IO register to c_erase_execute. This command causes the device to begin erasing the entire erase block specified by its private cursor.
4. It takes time for the device to erase the entire block. The device will be busy during erasure and return to ready once it is complete. The driver will poll the device's status until it becomes ready before attempting another operation.

The device passes through the machine states described in Table 4 to support this series of actions. Once the driver has patiently waited for the device to return to ready status, it has several options:

- After the device finishes erasing one block in step #4, it leaves its cursor pointing to the start of the next consecutive block. The driver can erase the next block by to step #3. The device will wrap its cursor back to zero if an erase increments it beyond the last block of storage.
- The driver can begin reading bytes from the device by writing the c_read_setup command to the command IO register.
- The driver can begin programming pages by writing the c_program_setup command to the command IO register.

3.5. Distinguishing busy and ready, resetting the device, selecting chips

After the driver uses one of the _execute commands to instruct the device to begin a read, program, or erase operation, the device will be busy for a considerable amount of time. As noted in the previous subsections, the driver must poll the device until it once again becomes ready before interacting with the device further. Drivers take the following steps to poll:

1. Sleep for some driver-specific constant amount of time that seems likely to cover most or all of the device's busy period.
2. Awaken and use get_gpio(pn_status) to query the status of the device. If the device is still busy, the driver will sleep for a short period of time and repeat this step. If ready, it will complete the polling procedure and move on.

Buggy drivers may cause the device to enter a confused state from which it cannot make useful progress. In these cases, drivers may use set_gpio(pn_reset, 1) to cause the device to reset itself to its initial state.

Table 5 describes how the device reacts to gpio_get and gpio_set calls while it is in any machine state.

3.6. Note on command IO register

This note explains the lines in several state machine tables that set the command IO register to c_dummy. Ideally, the driver would be the only component that writes to the command IO register; the device would read from it, but never write. Although our IO register watchpoint activates the device emulator whenever the driver reads or writes an IO register, it does not inform the device emulator whether the driver read or wrote an IO register or identify the IO register involved. In most cases the device emulator can deduce those details from its present state and from noting what register values have changed since it last examined them. However, there are some cases where the device emulator must be able to determine that the driver has written the same command to the command IO register twice in a row. The device emulator writes c_dummy to the command IO register after the device writes the command for the first time. The device emulator can subsequently deduce that the driver has written that command a second time when it observes that the command IO register contains that command again rather than c_dummy. This use of c_dummy is an artifact of our debugger-based implementation; it is not meant to represent a feature of real-world NAND flash storage devices.

Table 1 – Device initial and idle machine state. Common actions.

```
Machine state ms_initial_state:
  On ioregisters read/write:
    Case c_read_setup:
      Set machine state to ms_read_awaiting_block_address.
    Case c_program_setup:
      Set machine state to ms_program_awaiting_block_address.
    Case c_erase_setup:
      Set machine state to ms_erase_awaiting_block_address.
    Default:
      Set state to ms_bug.

State ms_bug:
  On ioregisters read/write:
    Remain in state ms_bug.


Additional cases for the Else switch on command IO register clause of
ms_read_providing_data, ms_program_accepting_data, and
ms_erase_awaiting_execute:

      Case c_read_setup:
        Clear cursor, deadline, cache.
        Set machine state to ms_read_awaiting_block_address.
      Case c_program_setup:
        Clear cursor, deadline, cache.
        Set machine state to ms_program_awaiting_block_address.
      Case c_erase_setup:
        Clear cursor, deadline, cache.
        Set machine state to ms_erase_awaiting_block_address.
```

Table 2 - Device machine states for reading.

```
State ms_read_awaiting_block_address:
  On ioregisters read/write:
    If system clock < deadline variable
       Or command IO register is not c_read_setup
    Then set machine state to ms_bug.
    Else
      Set block byte of cursor to address IO register.
      Set machine state to ms_read_awaiting_page address.

State ms_read_awaiting_page_address:
  On ioregisters read/write:
    If system clock < deadline variable
       Or command IO register is not c_read_setup
    Then set machine state to ms_bug.
    Else
      Set page byte of cursor to address IO register.
      Set machine state to ms_read_awaiting_byte_address.

State ms_read_awaiting_byte_address:
  On ioregisters read/write:
    If system clock < deadline variable
       Or command IO register is not c_read_setup
    Then set machine state to ms_bug.
    Else
      Set byte number byte of cursor to address IO
      register.  Set machine state to ms_read_awaiting_execute.

State ms_read_awaiting_execute:
  On ioregisters read/write:
    If system clock < deadline variable
       Or command IO register is not c_read_execute
    Then set machine state to ms_bug.
    Else
      Set deadline to current system clock time plus
      READ_PAGE_DURATION.
      Set machine state to ms_read_providing_data.
      Set command IO register to c_dummy.  (See Note 3.6.)

State ms_read_providing_data:
  On ioregisters read/write:
    If system clock < deadline variable
    Then set machine state to ms_bug.
    Else switch on command IO register
      Case c_dummy:
        Set data register to cache byte indicated by
        cursor.  Increment cursor.  Wrap cursor to remain
        in storage.
        Keep machine state set to ms_read_providing_data.
      Case c_read_execute:
        Set deadline to current system clock time plus
        READ_PAGE_DURATION.
        Set machine state to ms_read_providing_data.
        Set command IO register to c_dummy.  (See Note 3.6.)
        Keep machine state set to ms_read_providing_data.
```

Table 3 - Device machine states for programming

```
State ms_program_awaiting_block_address:
  On ioregisters read/write:
    If system clock < deadline variable
       Or command IO register is not c_program_setup
    Then set machine state to ms_bug.
    Else
      Set block byte of cursor to address IO register.
      Set machine state to ms_program_awaiting_page address.

State ms_program_awaiting_page_address:
  On ioregisters read/write:
    If system clock < deadline variable
       Or command IO register is not c_program_setup
    Then set machine state to ms_bug.
    Else
      Set page byte of cursor to address IO register.
      Set machine state to ms_program_awaiting_byte_address.

State ms_program_awaiting_byte_address:
  On ioregisters read/write:
    If system clock < deadline variable
       Or command IO register is not c_program_setup
    Then set machine state to ms_bug.
    Else
      Set byte number byte of cursor to address IO
      register.  Set machine state to ms_program_accepting_data.
      Set command IO register to c_dummy.  (See note 3.6.)

State ms_program_accepting_data:
  On ioregisters read/write:
    If system clock < deadline variable
    Then set machine state to ms_bug.
    Else switch on command IO register
      Case c_dummy:
        Set byte of cache to value of data IO register.
        Increment cursor.  Wrap cursor to remain in page.
      Case c_program_execute:
        Set deadline to current system clock time plus
        WRITE_PAGE_DURATION. Set storage page indicated by
        cursor to the values in cache.  Set cursor to the
        start of the next consecutive page, wrapping to 0
        as needed to stay within storage.
        Clear cache to all-zeroes.
        Set command IO register to c_dummy.  (See note 3.6.)
        Set machine state to ms_program_accepting_data.
```

Table 4 - Device machine states for erasing

```
State ms_erase_awaiting_block_address:
  On ioregisters read/write:
    If system clock < deadline variable
       Or command IO register is not c_erase_setup
    Then set machine state to ms_bug.
    Else
      Set block byte of cursor to address IO register.
      Set machine state to ms_erase_awaiting_execute.

State ms_erase_awaiting_execute:
  On ioregisters read/write:
    If system clock < deadline variable
    Then set machine state to ms_bug.
    Else switch on command IO register
      Case c_erase_execute:
        Clear the storage block indicated by the cursor
        to all zeroes (0x00).  Set deadline to system clock
        time plus ERASE_BLOCK_DURATION.  Set the cursor to
        the start of the next consecutive block, wrapping
        to 0 as needed to stay within storage.
        Set machine state to ms_erase_awaiting_execute.
        Set command IO register to c_dummy.  (See note 3.6.)
```

Table 5 - GPIO actions common to all machine states.

```
All machine states have the following additional common GPIO actions:

  On gpio_get(pin number) call:
    Case pn_status:
      If system clock < deadline then cause gpio_get to return 1 to
      Indicate busy, else cause it to return 0 to indicate ready.
    Case pn_reset:
      Getting the pn_reset pin’s value is a meaningless operation.
      Cause gpio_get to return 0.

  On gpio_set(pin number, value) call:
    Case pn_status:
      Setting the pn_status pin is a meaningless operation.
    Case pn_reset:
      Clear cursor, deadline, cache.
      Set machine state to ms_initial_state.
      Delay for RESET_DURATION.
```

4. NAND framework emulator

The NAND framework emulator provides device drivers with a collection of interfaces that resemble those the Linux NAND framework and other parts of the Linux kernel provide to real device drivers. There are four interfaces:

GPIO functions: the framework provides drivers with the gpio_set() and gpio_get() functions described in Subsection 2.

Jump table: upon initialization, the simplest device drivers provide the framework with a jump table of driver operations. The framework then commands the device driver to operate the device by invoking these functions through that jump table. Subsection 4.1 describes this interface further.

Command interpreter: instead of a jump table, more complex device drivers provide the framework with a pointer to a command interpreter function. The framework then commands the device driver to operate the device by providing this command interpreter function with a sequence of instructions. Subsection 4.2 describes this interface further.

Device Information Base: the most complex device drivers add a subgraph of nodes describing their device and its capabilities to a complex graph structure called the Device Information Base (DIB). These device drivers share the DIB with the framework; both components are responsible for leaving the DIB in a well-formed state after each update. Subsection 4.3 describes this interface further.

4.1. Function call through jump table

Table 6 describes the jump table functions. This function call through jump table composition pattern represents a legacy interface supported by the real-world Linux kernel's NAND framework; its jump table can be found in mtd/rawnand.h:struct nand_legacy.

To operate the emulated NAND storage device, test suites must cause the framework to call the functions in this jump table in a manner that follows the driver-device protocol described in Section 3. For example, to read data from a pair of two contiguous device pages, the test suite would make this series of calls through the jump table:

1. One call to set_register() to set the device's command register to c_read_setup.
2. A sequence of three calls to set_register() to write each byte of the three-byte location to begin reading from to the device's address register.
3. One call to set_register() to set the device's command register to c_read_execute, prompting the device to read the first page of data from its storage to its cache.
4. One call to wait_ready() to wait for the device to finish reading from storage.
5. One call to read_buffer() to read the first page's data from the device's cache into the test suite's buffer.
6. One call to set_register() to set the device to read the second page of data from its storage to its cache.
7. One call to wait_ready() to wait for the device to finish reading from storage.

8. One call to read_buffer() to read the second page's data from the device's cache into the test suite's buffer.

Test suites must ask the framework to make a similar series of calls through the driver's jump table to properly implement write and erase tasks.

4.2. Command interpreter

More complex device drivers to provide the framework with a function that implements an interpreter for operations (that is, programs) consisting of a series of instructions drawn from a simple command language and their arguments. Table 7 contains the grammar for that command language. Once registered, the framework can operate the device by calling this parser function and providing it with an array of commands and arguments in that language. This composition pattern represents the exec_op() functions that many real-world Linux raw NAND device drivers provide to the real-world Linux NAND framework.

As with the function call through jump table composition pattern, test suites must submit operations to the interpreter that follow the device-driver protocol described in Section 3. For example, to read data from a pair of two contiguous device pages, the test suite would submit an operation consisting of this series of instructions:

1. IN_CMD c_read_setup to write the c_read_setup command to the device's command register.
2. IN_ADDR 3, [block, page, byte] to write the block, page and byte parts of the location to begin reading to the device's address register.
3. IN_CMD c_read_execute to write the c_read_execute command to the device's command register, prompting it to read the first page from storage into its cache.
4. IN_WAIT_READY 100 to cause the driver to sleep for the 100 microseconds it takes for the device to read the first page and then poll until the device is ready.
5. IN_DATA_OUT length bufferaddress to cause the driver to read length bytes from the device's data register into the test suite buffer starting at bufferaddress.
6. IN_CMD c_read_execute to write the c_read_execute command to the device's command register, prompting it to read the second page from storage into its cache.
7. IN_WAIT_READY 100 to cause the driver to sleep for the 100 microseconds it takes for the device to read the second page and then poll until the device is ready.
8. IN_DATA_OUT length bufferaddress to cause the driver to read length bytes from the device's data register into the test suite buffer starting at bufferaddress.

Note that the above grammar permits operations composed of any instructions in any order. However, the emulated NAND storage device expects only certain instructions in certain orders, as described in Section 3. Operations that violate those expectations will drive the device into a bug state. Like this permissive grammar, real-world Linux device driver exec_op() functions will accept operations with incorrectly ordered instructions. They rely on the NAND framework and higher-level components to emit operations that will not confuse the device.

4.3. Shared device information base data structure

Only the most complex device drivers will describe their device and its capabilities by modifying a Driver Information Base (DIB) data structure that is shared between the device driver and framework emulator. Like the Linux kernel's Device Tree (DT), the DIB is a graph of nodes in links that, to be well-formed, must obey rules concerning the arrangement of nodes of particular types, the links between those nodes, and the values of node reference count fields. Both the device driver and the framework emulator must ensure they leave the DIB in a well-formed state after modifying it.

Figure 2 contains a diagram of an example DIB. The DIB is a linked list of Device nodes, each describing a NAND storage device. Each NAND storage device consists of exactly one controller chip and one or more storage chips. The diagram shows examples of devices with three, two, and one storage chip. Each Device node points to one and only one Controller Chip node. Each Controller Chip node has a linked list of Storage Chip nodes. There are double links from the Controller Chip node to the first and last nodes in the Storage Chip list.

The nodes themselves have the following fields:

Device: a pointer to a string containing a make and model name for the device.

Controller chip: the number of storage chips and a pointer to the driver's command interpreter similar to the foxtrot driver.

Storage chip: the number of blocks, the number of pages per block, and the number of bytes per page.

In addition to the fields above and the links shown in the diagram, each node has a reference count field.

Table 6 - Device driver to framework jump table interface.

set_register(register, value)
This function enables the framework to set the emulated NAND storage device’s registers to particular values; the register parameter indicates which register, the value parameter indicates the value to set. It has no return value.

read_buffer(buffer, length)
This function enables the framework to read data from the emulated NAND storage device into a buffer. The buffer parameter indicates the buffer to receive the data; the length parameter indicates the number of bytes to read. The test suite should take care to ask read_buffer() to read only within the bounds of a single emulated NAND storage device page; asking read_buffer() to cross a page boundary represents an error that will result in undefined behavior. This function returns -1 on error, otherwise the number of bytes read.

write_buffer(buffer, length)
This function is similar to read_buffer() except that it writes data to the device rather than reading data from the device.

wait_ready(timeout)
This function causes the device driver to poll a busy emulated NAND storage device until the device becomes ready. It takes a timeout interval in microseconds. This function returns 0 to indicate the device is ready or -1 to indicate the device did not become ready within the timeout interval.

Table 7 - Device driver to framework command grammar interface.

```
; An “operation” is a sequence of “instructions.”  In the real
; world, Linux represents this abstraction with a struct
; nand_operation.
OPERATION       ->   COUNT          ; count of instructions
                     INSTRUCTION+   ; array of instructions

; An “instruction” is a an “instruction type” – essentially an
; opcode followed by whatever arguments are needed by that type.
; In the real world, Linux represents this abstraction with a
; struct nand_op_instr.
INSTRUCTION     ->   IN_CMD        OPCODE
                     ; IN_CMD writes OPCODE to the device’s
                     ; command register.
                |    IN_ADDR       NUM ADDRESSBYTES
                     ; IN_ADDDR loops over a NUM-long ADDRESSBYTES
                     ; array of bytes and writes each one to the
                     ; device’s address register.
                |    IN_DATA_IN    LENGTH BUFFERADDRESS
                     ; IN_DATA_IN writes LENGTH bytes from the
                     ; buffer at BUFFERADDRESS to the data
                     ; register.  (Corresponds to “program”.)
                |    IN_DATA_OUT   LENGTH BUFFERADDRESS
                     ; IN_DATA_OUT reads LENGTH bytes from the
                     ; device’s data register and stores them to
                     ; the buffer at BUFFERADDRESS. (Corresponds
                     ; to “read”.)
                |    IN_WAIT_READY TIMEOUT
                     ; IN_WAIT_READY causes the driver to wait
                     ; until (a) the device becomes ready, or
                     ;(b) timeout microseconds have elapsed.

OPCODE          ->   c_read_setup    | c_read_execute
                |    c_program_setup | c_program_execute
                |    c_erase_setup.  | c_erase_execute

NUM             ->   ; unsigned integer value, always 3.
ADDRESSBYTES    ->   ; array of block, page, byte address values
LENGTH          ->   ; unsigned integer value <= device page size.
BUFFERADDRESS   ->   ; address of buffer to provide/receive bytes.
TIMEOUT         ->   ; unsigned integer delay in microseconds.
```

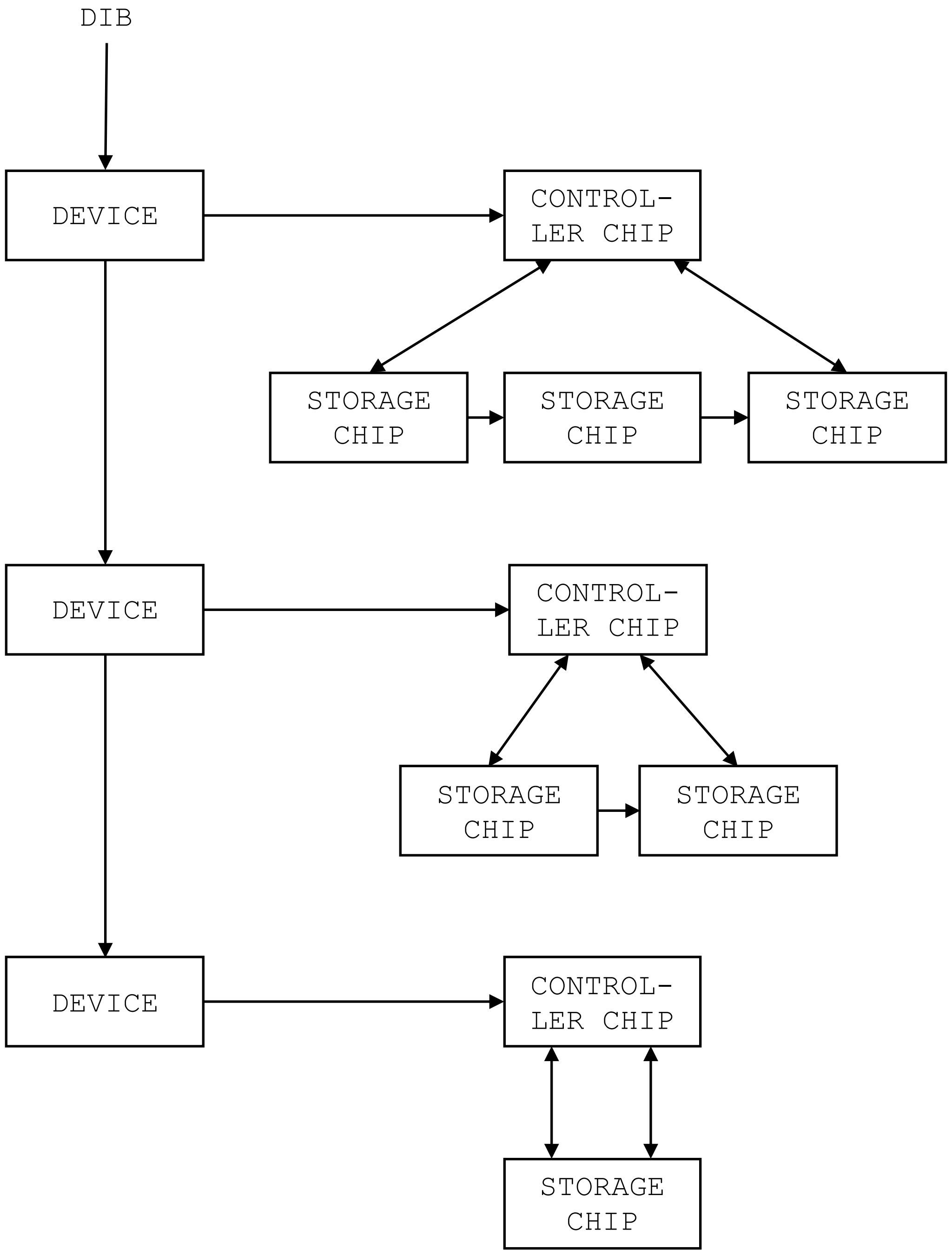

Figure 2 - Example Device Information Base (DIB) with entries for three devices, one with three storage chips, one with two, and one like the device emulator with only one.

5. Device drivers

The corpus of example drivers contains a series of drivers ordered alphabetically by name: Alpha, Bravo, and so on. The series of names skips some letters to permit the potential insertion of additional driver examples in the future. The "_0" variant of each driver correctly implements a particular combination of composition and loop patterns found in real Linux NAND device drivers. Those with higher numbers implement flaws. The following subsections describe the composition patterns and loop patterns of each _0 driver, the properties a developer would likely want to prove regarding those patterns to support an argument that a DSL-based replacement was compatible with the legacy C version, and describe the bugs in the accompanying flawed examples that invalidate those properties.

5.1. The Alpha driver

The Alpha_0 driver correctly implements the jump table composition pattern with the framework. It transfers data to and from the device using loop structures that conform to the pattern found most often in real Linux NAND drivers: a simple loop that decrements a count of bytes remaining to be transferred and increments a pointer to the driver-side buffer, similar to the readsb() and writesb() macros from asm-generic/io.h used by many real-world drivers. The data transfer loop structures have these properties:

- The data transfer loop always terminates.
- It transfers the requested number of bytes.
- It transfers them in the proper order.

The Alpha_0 driver waits for the device to become ready with a simple loop structure similar to the one found in nand_base.c:nand_gpio_waitrdy(). The loop repeatedly polls the device's ready status and terminates when the device becomes ready or the kernel's clock reaches the timeout value. The wait-ready loop structure has these properties:

- The wait-ready loop always terminates.
- On termination, either (a) the device is ready, or (b) the clock has ticked through the timeout interval.
- The driver spends most of the timeout interval sleeping.

The flawed versions of the Alpha driver demonstrate the bugs described below. Each one invalidates one or more of the above properties.

Alpha_1: Wait-ready loop polls the device with the proper timeout but does not sleep.

Alpha_2: Wait-ready loop polls the device and returns ready if the device returns ready, but it does not compute a timeout time or compare the current time to that timeout value. If the device never returns ready it will never time out.

Alpha_3: Wait-ready loop incorrectly computes the timeout time as simply the timeout interval and not the current time plus the timeout interval, thus making the timeout time a moment in the distant past. Wait-ready loop always sleeps for a bit and does at least one poll of the device. Thus in test so long as the device rapidly returns ready all will seem well. However, the loop will declare timeout on its first clock comparison and not wait the proper timeout value.

Alpha_4: The copy is done by an unnecessarily complicated nested loop. The inner loop copies eight bytes at a time. The outer loop runs the inner one the proper number of times if the total number of bytes to copy is a multiple of 8. However, it has a bug: if the total number of bytes is not a multiple of 8, it fails to copy the last 1-7 bytes.

Alpha_5: The copy is done by a similar nested loop that works if the total number of bytes to copy is a multiple of 8. The outer loop fails to terminate if the count is not a multiple of 8.

Alpha_6: The copy loop stores the bytes read into the destination buffer in reverse order.

5.2. The Bravo driver

The Bravo_0 driver correctly implements the same jump table composition pattern with the framework as the Alpha driver. However, it transfers data to and from the device using a less common loop structure pattern: a simple loop that increments a count of the bytes transferred and uses the C array index operator to index through the driver-side buffer, similar to amd-delta.c:gpio_nand_read_buf() and gpio_nand_write_buf().

It waits for the device to become ready with a doubly-nested loop structure similar to the one found in hisi504_nand.c:wait_controller_finished(). This curious loop structure waits for erase operations differently from other operations: it sleeps while polling only for erase operations; it has a timeout bound only for non-erase operations.

The Bravo_0 driver's data transfer and wait-ready loop structures have the same properties as those in Alpha_0. The Bravo_1 through 6 drivers demonstrate the same bugs.

5.2. The Foxtrot driver

The Foxtrot driver adds support for processing sequences of instructions to the basic functionality found in the Alpha driver. Many real-world drivers provide this functionality in an exec_op() function. The Foxtrot driver implements a typical command sequence processing loop that iterates through an operation represented by an array of instructions (ideally) conforming to the grammar in Table 7. The loop contains a large switch that dispatches control to a handler for each kind of instruction. The gpio.c:gpio_nand_exec_op() function provides an example of this pattern.

Foxtrot_0's command interpreter loop structure has the following properties:

- The loop to process all instructions in an operation always terminates.
- The loop sends all instructions to the device.
- It sends the instructions in the proper order.

The flawed versions of the Foxtrot driver demonstrate the bugs described below. Each one invalidates one or more of the above properties.

Foxtrot_1: The driver copies the operation into a buffer before processing them. The driver's buffer is large enough to accommodate reads of a single page, but longer operations get truncated and some instructions don't make it to the device.

Foxtrot_2: For no good reason, the index for the instruction loop gets modded to some N, if there are N or fewer instructions in an operation, all will be well. If there are more than N, it will dispatch the first N instructions to the device forever.

5.3. The Kilo driver

The Kilo driver updates the DIB with a description of its device and its capabilities. A well-formed DIB has these properties:

- Its nodes are linked correctly.
- Each node's reference count is set as follows:

  storage chip nodes: reference count is 1.

  controller chip nodes: reference count is equal to its number of storage chip nodes.

  device nodes: reference count is its controller chip node plus 1.
- The linked list of Device nodes contains no more than 64 nodes.
- Each linked list of Storage Chip nodes contains no more than 8 nodes.

The flawed versions of the Kilo driver demonstrate the bugs described below. Each one invalidates one or more of the above properties.

kilo_1: The framework provides the driver with a well-formed initial DIB that already contains some devices. The driver has a bug that de-links a device already in the DIB upon registration.

kilo_2: The driver links a new Device node, but no Controller or Storage nodes.

kilo_3: The driver links a new Device and Controller node, but no Storage nodes.

kilo_4: The driver registers itself properly in all respects except all reference counts are zero.

kilo_5: The driver registers itself properly in all respects except that it incorrectly sets two of the Controller-Storage chip node links to NULL.

6. Building and running the test rig

The test rig is a user-mode C program for GNU/Linux operating systems on 64-bit Intel instruction set CPUs. The test rig's source code includes several details specific to 64-bit Intel instruction set CPUs; the test rig will build only on GNU/Linux platforms with this kind of CPU.

Test rig makefile's default target will build multiple executables, one for each example driver. To build this default target, simply run:

```
make
```

The makefile has specific targets for each example driver. To build an executable for a specific driver, run make with the target named for that driver, for example:

```
make test_alpha_0
```

Researchers can extend the makefile to include similar specific targets for their own DSL-based driver reimplementations.

Running a test rig executable runs the test suite for the configured driver. For example:

```
./test_alpha_0
```

Note that you will need to terminate the tests for drivers with deliberate hanging bugs with ctrl-C.

6.1. A warning about debugging

The test rig's parent tracer process runs the device emulator and debugs the child tracee process using the Ptrace library. The parent tracer receives a signal when the child tracee hits a hardware watchpoint on its emulated IO register and tries to puzzle out how to emulate an IO register read or write based mainly on the state of its simulated NAND storage device. The child tracee process runs the driver. If your driver has a bug that causes the child tracee to segmentation fault, the child tracee will not dump core. Instead, the parent tracer, acting as a debugger, will get a signal indicating the child tracee has faulted. The current version of the parent tracer will misinterpret this signal as watchpoint activation and do something unhelpful with it.

In the best case, it will drive its emulated storage device into a bug state, report an error, and halt the test rig. In the worst case, it will do *something* to the emulated storage device and wait patiently for the dead child tracee to proceed, causing the test rig to hang.

6.2. A warning about instruction support

When the child tracee uses a mov instruction to read from the emulated IO register, the instruction moves the incorrect value present at that moment in the emulated IO register variable in its memory into one its general-purpose CPU registers. It then passes control to the parent tracer device emulator via the hardware watchpoint.

At that point, the parent tracer device emulator can correct the contents of the IO register in the child tracee's memory to reflect the value that would have been there on a real device. However, because the CPU has already executed the child tracee's mov instruction to move the previous incorrect value into one of the child tracee's general-purpose CPU registers, the parent tracer device emulator must update the value in that register, as well.

Identifying which register the mov instruction used requires the parent tracer device emulator to examine and decode the child tracee’s mov instruction. There are many forms of mov in the IA64 instruction set. The current version of the device emulator supports only four forms: two that GCC used in our tests and two more that seemed to be likely alternates. If your compiler uses a different form of mov, the device emulator will report an error and halt. The error output will include the bytes of the unsupported instruction; including those bytes in a bug report will aid in adding support.