NAND 驱动编写与适配与实现

为了能够适配虚拟机并在未来进一步适配 2k500 开发板, 本研究同时进行了驱动的编写。 具体来说, 2k500 的开发板配备了一个 NAND 控制器作为其配置的外设。 龙芯提供了一个参考的 C 语言实现(在 u-boot 引导器中), 其中包括大量的寄存器读写操作, 同时, 还提供了详细的手册以备参考(注意, 这个手册的地址是错的, 地址实际上是 0x1ff58000)。 另外, 龙芯提供的 QEMU 虚拟机本身也能模拟 2k500 的外设, 这也大大方便了驱动的编写和调试。

1.1.1 移植思路

在开始驱动的编写之前, 我们预先进行的路线规划如下：

一、驱动交互方式分析

分析应当并行地从下列方向入手:

1. 分析官方参考驱动的框架, 识别出其内部的结构和 NPUcore 的驱动架构的公共部分, 并了解大体的运行顺序和交互机理。

   一方面手册中的数据甚至可能有错(比如本例), 毕竟一个正确运行的驱动其中包含的内容至少是经过实践验证的。

   另一方面, 手册往往是数据的罗列, 而驱动是有机的整体, 对后者的学习往往能得到超出数据本身的动态整体的认识。 甚至对参考驱动进行编译和跟踪调试(这是一个在当时没有想到, 但事后想起来其实应当增加的步骤), 可以省下很多理解, 检查和调试的开销。

   具体到本实例, 就是读写和擦除部分的代码。注意其中必然会包含大量的常数和寄存器读写, 这时候对不了解的常数, 可以参考第二步。
2. 在分析驱动的同时, 参考官方手册了解所需常数的功能和意义, 从而对驱动结构作进一步了解。
3. 根据这里获得的大体的知识精读手册, 了解其中的具体细节, 尤其对手册中涉及到其他章节的部分进行更详细的分析

二、驱动编写

跨语言的驱动编写和移植有两种思路, 一种是大体借鉴其原程序的架构, 另一种是按照目标语言的语法和思想进行重构。 原驱动是 C, 而 NPUcore 的代码是 Rust, 其安全性检查和面向方面的特性可以对驱动的编写提供部分细节上的辅助。 考虑到系统编写和调试的简单化与本次毕设对笔者的实验与学习目的, 因此这一驱动编写采用所谓"地道的 Rust"(idiomatic Rust), 而非直接将 C 语言的操作翻译为裸指针操作作为驱动的一部分。

时候发现, 常数应当尽量从驱动而非手册复制, 并同手册进行交叉核验, 因为手册可能有错, 但正确运行的驱动往往问题不大。

三、驱动调试

硬件相对于软件而言黑箱特性较重，由于其内部状态和调试接口并不对程序员开放，我们只能通过其状态寄存器的完成位和最终读写的结果进行状态的推测。

由于有参考驱动，我们可以编译带有符号的 bootloader，然后对驱动进行 gdb 的单步调试，尤其可以将断点设置在读写 MMIO 寄存器的部分，从而检查对驱动的交互，流程和常数的正确性。

具体来说，可以同时在两个模拟器或者两个开发板上运行两个驱动，检查其读写同一个任务时的寄存器变化，并利用 gdb 的内存观察能力检查内存映射的控制寄存器中的数值变化。

不难看出，当没有参考驱动的时候，手册往往就是唯一的知识来源，这时候如果不配合其他的调试工具(如 jtag 调试器或某些功能完整的硬件模拟器)，那么驱动的编写会十分困难。

1.1.2 龙芯 NAND 驱动架构分析

一、驱动分析

龙芯的 NAND 驱动控制器必须通过 DMA 才能进行读写，因此驱动事实上分为控制器和中断部分。 通过阅读参考驱动的代码，我们可以得出如下读写流程:

如图 3-11，首先需要对 NAND 控制器进行设置，设置其读取的方式，然后设置 DMA 后等待 DMA 命令完成并清 0，最后等待命令完成标志位置 1(如果不开启中断的话)。

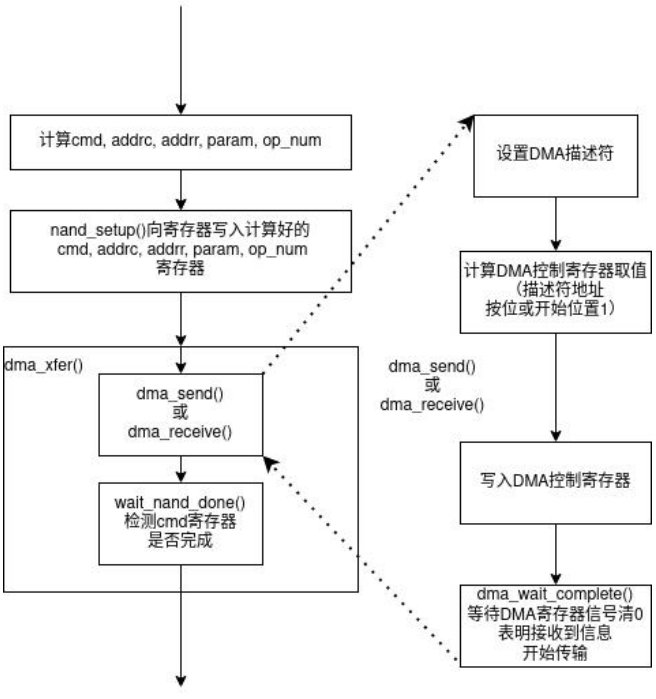

图 3-11 NAND 驱动总体流程

NAND 控制器设置相对简单, 只需要其各个参数寄存器设置完成, 最后设置命令寄存器即可。

DMA 设置, 使用的是手册中描述的一个内存结构：DMA 描述符。

DMA 描述符是一个内存中的数据结构, 可以放置在任何符合对齐要求（32 字节对齐）的物理地址上, 按照需要填写 DMA 的数据长度等内容， 最后再写入 DMA 控制器。

另外, 虽然 NAND 驱动理论上支持硬件的 ECC 纠错， 但由于参考驱动采用了纯软件 ECC 校验的方式, 其实际上使用上述同一套逻辑来读写数据区和校验码。因此如果不实现 ECC 校验/纠错, 其读写纠错码的行为可以在实际的驱动中去掉, 只留下读写主区域的部分。

二、龙芯 NAND 控制器

NAND 存储的块大小是由芯片本身决定的, 不同于 SD 卡, NAND 存储的读取最小单位块的大小依芯片本身而不同。每块内部有用于存储数据的主区域和用于存储元数据(包括坏块标记和校验码, 纠错码)的空闲区域(也称 OOB)。

龙芯的 NAND 控制器在 NAND 芯片的基础上进一步进行了封装, 简化了 NAND 驱动的开发。 具体来说, NAND 驱动器通过下列寄存器控制（注意， 官方手册这里地址高 16 位误为 0xBFF5）[15]

表 3-4 龙芯 NAND 控制器寄存器列表

地址	名称	用途或存储内容
0x1FF5_8000	NAND_CMD	命令
0x1FF5_8004	ADDR_C	列地址
0x1FF5_8008	ADDR_R	行地址
0x1FF5_800C	NAND_TIMING	保持时间与等待时间
0x1FF5_8010	ID_L	型号ID低位
0x1FF5_8014	STATUS & ID_H	状态与型号ID高位
0x1FF5_8018	NAND_PARAMETER	NAND类型与操作范围
0x1FF5_801C	NAND_OP_NUM	操作块/字节数量
0x1FF5_8020	CS_RDY_MAP	片选
0x1FF5_8040	DMA access address	DMA数据地址

其中，最后一个用于 DMA 读写的数据寄存器，其他是控制寄存器。ADDR_C 是列地址，相当于块内偏移量，ADDR_R 是行地址，相当于块号，PARAM 决定操作的参数(包括范围和芯片型号)，OP_NUM 决定操作的字节数。

大多数的位定义可以参考龙芯的 2k0500 手册，但是这里单独需要强调的是 CMD 的最低位为 VALID 位：VALID 位不为 1 时,命令只是写入寄存器， 但不会触发硬件执行; 当命令需要开始执行的时候， 用户向该寄存器存命令按位或 1，也就是将该位置 1， 然后命令会开始执行，直到完成后清 0。

在操作时， 首先应当设置好除了 CMD 之外的寄存器， 然后单独设置 CMD 寄存器， 可以先写入 VALID 为 0 的命令，再向其写入为 VALID 为 1 的命令。

三、驱动编写

了解了 NAND 驱动的原理后，可以确定模仿参考驱动进行实现。 对此，我们应当遵循以下原则:

1. 使用 Rust 原生的读写函数风格，用类似面向对象的 getter 和 setter 进行寄存器值的配置而非直接使用原代码，增强驱动代码的可读性和可维护性。
2. 使用宏定义和 Rust 已有的 bit_field 包生成寄存器的读写函数，减小编写的工作量和难度并确保其位宽和偏移的正确性。

例如，用下列宏进行各个寄存器中域的声明，然后定义了保持时间和等待时间两个域:

```
macro_rules! impl_get_set_field {
    ($csr_ident:ident,$csr_set_ident:ident,$range:expr) => {
        #[inline(always)]
        pub fn $csr_ident(&self) -> usize {
            self.get_bits($range).bits as usize
        }

        #[inline(always)]
        pub fn $csr_set_ident(&mut self, status: usize) -> &mut Self {
            self.set_bits(
                $range,
                Self {
                    bits: status as u32,
                },
            );
            self
        }
    };
}

```

图 3-12 宏定义实现寄存器域存取

```
#[allow(unused)]
impl NandTiming {
    impl_get_set_field!(get_hold_time, set_hold_time, 8..=15);
    impl_get_set_field!(get_wait_cycle, set_wait_cycle, 0..=7);
}
```

图 3-13 通过宏定义实现 NAND 时序控制寄存器各个域声明

这样就能只用一行代码，实现对寄存器的单个域的设置函数的定义，减少了大量的重复代码编写。该宏可以用于所有的寄存器域的定义以提升开发效率。

3. 同样模仿参考驱动将寄存器的读写分为三个层次，即数值编辑，寄存器设置(setup)，寄存器写入，并使用 read/write_volatile 函数保证写入的易失性

具体来说，沿用之前的例子，一个寄存器的写入应当有:

```
src > arch > la64 > ls_nand > nand.rs > {} impl_Nand
339     fn setup(
340         &self,
341         cmd: &mut NandCmd,
342         addr_c: &mut AddrC,
343         addr_r: &mut AddrR,
344         param: &mut NandParam,
345         op_num: &mut NandOpNum,
346     ) {
347         let base = self.get_base();
348         param.write(base);
349         op_num.write(base);
350         addr_c.write(base);
351         addr_r.set_cs(self.chip, *param).write(base);
352         NandCmd::empty().write(base);
353         cmd.set_cmd_valid(false).write(base);
354         cmd.set_cmd_valid(true).write(base);
355     }
```

图 3-14 NAND 寄存器设置函数

其中, write()方法就是封装好的 write_volatile()函数。

最后， DMA 使用的内存数据结构 DMA_DESC 要求按 32 字节对齐， 为了方便对齐(编译器有时候无法严格对齐栈上数据结构)且减小由于临时通过伙伴

系统进行堆分配失败导致的读取失败，我们将 DMA 描述符作为静态数据(避免在栈上或者堆上申请的时间开销)。

我们可以将 C 版本与 Rust 版本的驱动的对比如下：

表 3-5 C 版驱动与 Rust 版驱动的对比

对比项目	参考驱动	重写版驱动
语言	C	Rust
模块划分	NAND，DMA两个结构	NAND，DMA， 以及各个寄存器均为对象
ECC	软ECC	无
数值合法性检查	无	有

1.1.3 驱动调试

在测试过程中，由于硬件内部的黑箱性质，部分的状态在的时候无法轻易报错，因此我们可以使用编译出的正确版本进行对比。

例如，在调试初期，我们注意到对错误的地址写入命令后，该地址的完成标志位显然永远不会被置 1。经过对参考驱动的写入部分进行跟踪调试发现，其使用的地址并不是手册上的 0xBFF58000，而是以 0x1ff58000 地址为基址的一系列地址，在修改了地址后，驱动的读写也得以顺利进行。如图 3-15 中所示， r13 的值为 0x800000001ff58000， 即不经过缓存访问 0x1ff58000 的地址， 并对该地址写入寄存器值。另外，用类似的方法还解决了部分常数的位移错误，由此可见对参考驱动的并行调试是一种比较合适的驱动移植调试思路。

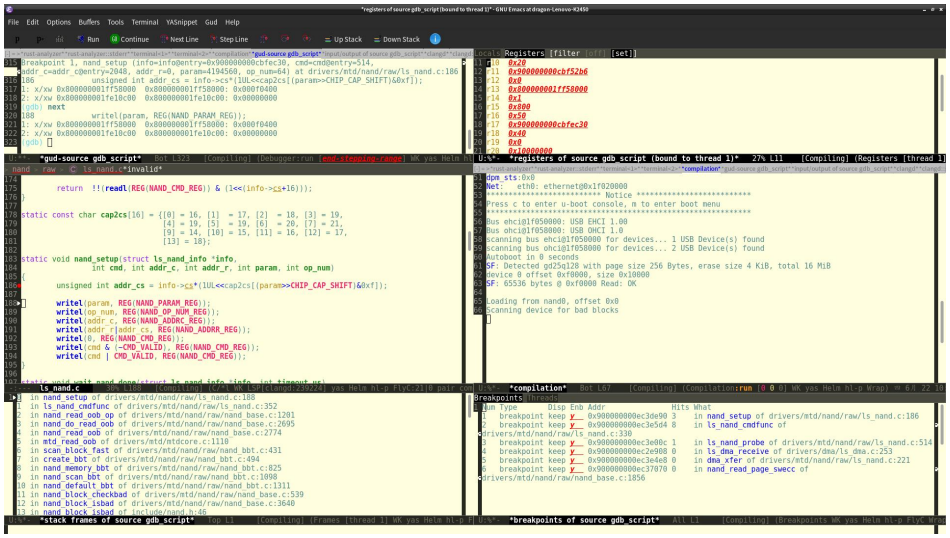

图 3-15 GDB 调试 u-boot 中 2k500 的 NAND 驱动