Broadcom 平台 Flash 模块

SWD 聂勇

版 本 历 史

版本/状态	责任人	起止日期	备注
V1.0/草稿	聂勇	25Feb2011	创建文档
V1.1/草稿	聂勇	11Mar2011	补充 flashDrvLib 的 buffer 策略

目　录

1. 说明

本文档是对 Broadcom 平台下 Flash 模块的概括和总结。下面将按照硬件到软件，上层应用到底层驱动的的顺序组织整个文档。章节包括：

硬件部分：

1. 交换机机型与 Flash 型号
2. Flash 接口电路设计

软件部分：

3. Flash 驱动与 VxWorks
4. Flash Basic Operation
5. Flash Wiki

如果你想快速为新型号 Flash 编写驱动代码，你可以看第三部分“Flash 驱动与 VxWorks”;如果你想了解并口 NOR Flash 芯片的操作,你可以重点看第四部分“Flash Basic Operation”；如果你想了解整个 Flash 存储器的相关知识，请参考第五部分“Flash Wiki”。

2. 交换机机型与 Flash 型号

下表是 Broadcom 的 Harrier 平台或者 Hawkeye 平台的各机型使用的 Flash 芯片的总结。注意，这里都是使用并行 NOR Flash。如无特别说明，下文中的 Flash 都是指的并行 NOR Flash。

方案平台和机型	Flash 芯片	驱动文件	Flash Datasheet	备注
harrier 5428/3428	JS28F640	flash28f640DrvLib.c	Intel-J3-Flash.pdf	
Hawkeye 3424/3216	M29W640GH	flashm29w640DrvLib.c	M29W640G.pdf	硬件原因实现不了锁，现已不用
Hawkeye 3424/3216/ 3824F1.0/	M29W128GH	flash29gl128DrvLib.c	M29W128GH.pdf	

3210(UN)1.0				

2- 1 交换机机型与 Flash 型号

3. Flash 接口电路设计

NOR Flash 接口电路，是 MCU 和 NOR Flash 之间的电路连接。通过了解实际电路，能够深入了解很多 Flash 相关的概念，有助于底层驱动的编写。可以深入了解的内容有：NOR Flash 的随机存储，NOR Flash 的芯片内执行（XIP，eXecute In Place），NOR Flash 何时不需要驱动，NOR Flash 的驱动需要怎么设计，双字节/4 字节地址对齐等。

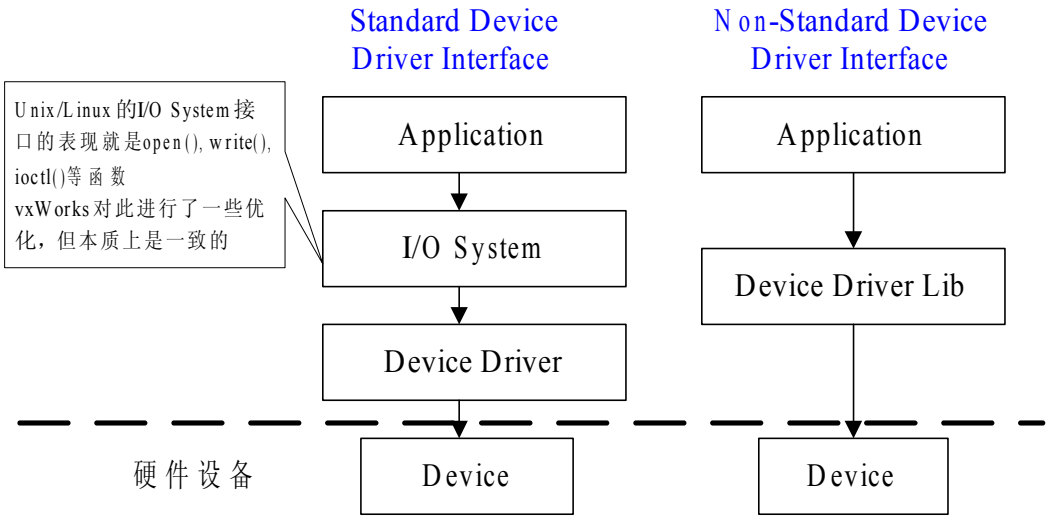
具体的 Flash 接口电路参考硬件部门的文档《TP-Link 硬件工程师培训教材 2010 版.pdf》。

4. Flash 驱动与 VxWorks

4.1 Flash 驱动与文件系统

首先，简要的介绍一下嵌入式系统中设备驱动接口的概念。我们将从整体上来把握，就像从飞机上来俯瞰一样，这样能够让我们更加明白自己所在的位置和角色。

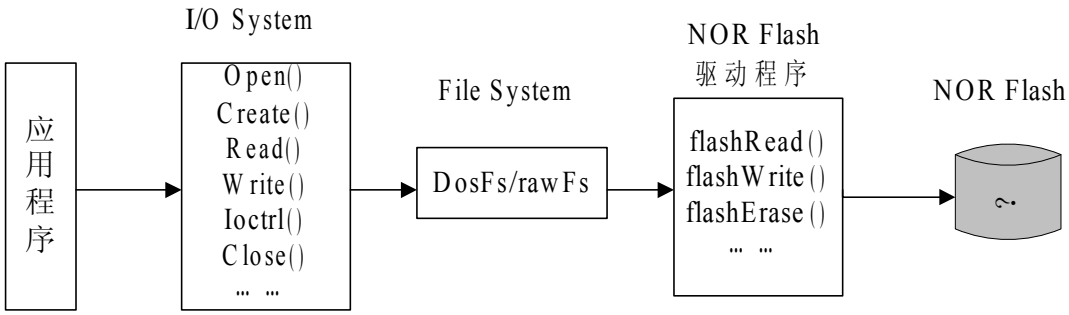
如 4- 1 所示，设备驱动接口分为两种，一种是标准的设备驱动接口，另外一种为非标准的，如下图所示。区别在于，标准的设备驱动接口符合操作系统中对设备进行操作的相关流程，而非标准接口由于应用程序直接和驱动程序联系，绕过 OS 中的 I/O 系统，在效率，实时性等方面更强。



4- 1 设备驱动接口

以上是一个总体的设备驱动接口的框图。本文档中需要分析的是 Device Driver 或者 Device Driver Lib 针对 NOR Flash 设备是如何实现的。

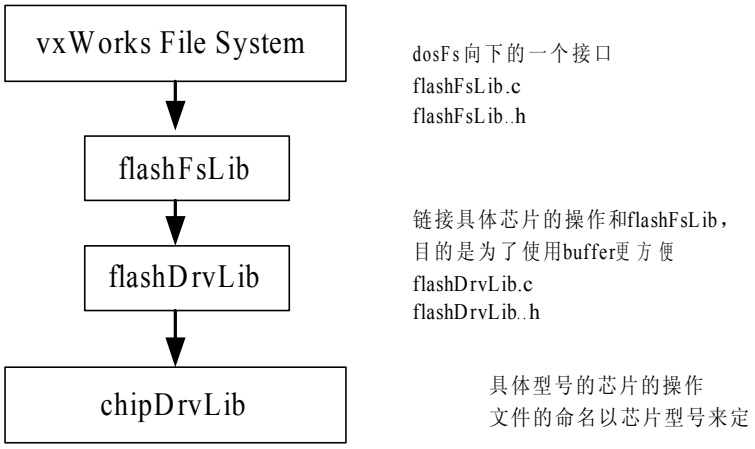
对于 NOR Flash 这样的设备，由于其中还加入了文件系统一层，其总体的框图又有所变化。



4- 2 VxWorks 与 Flash 驱动

通过可以看出，文件系统就是接在了 I/O 系统和 NOR Flash 驱动程序之间。那么，如果使用了文件系统，那么 NOR Flash 中就是存放的文件，目录等。如果没文件系统，设计的 NOR Flash 驱动程序也能够对 NOR Flash 中的任何地址，数据进行操作。例如在 bootload 阶段没有使用文件系统，也需要读取 NOR Flash 中的配置信息等等。

在 MIPS BSP 中，Flash 驱动采用模块化设计，各模块的关系如 4- 3 所示。



4- 3 Flash 驱动模块框图

4.2 flashFsLib

在这里，需要指出的一点就是，对 FlashFsLib 和 FLashDrvLib 的一些区别的理解：FlashFsLib 是作为文件系统和 NOR FLASH 驱动的接口。文件系统做为块设备，具备块设备以块为读写单位的特征。读写的最小单位是一个 block，在 flashFsLib 中定义为：

#define FLASH_FS_BLOCK_SIZE 512

区别于 NOR FLASH 的块擦除单位，其实在代码中将其写成 sector，而文件系统的块操作中的块是 block，只是翻译为中文都把它叫做“块”。

所以，FlashFsLib 中所有函数的输入参数一般为：开始的 block，block 的个数等。FlashDrvLib 作为具体芯片操作的一个接口汇聚，输入的参数是开始的 sector，在 sector 中的 offset。

这也就是为什么，在 FlashFsLib 中，需要将 block 转化为 sector 的函数

flashFsGetPhys(int blkNum, int *sectorNum, int *offset)

下面在对 sector 和 block 的概念做一个总结：

概念	描述	大小	备注
sector	Flash 的擦除单位	128KByte（根据芯片而定）	在很多 Flash 芯片资料中，也就 sector 写成 block
block	块设备的读写操作单位	512Byte（根据块设备而定）	

4.3 flashDrvLib

4.3.1 flashDrvLib 的驱动加载

如何为新 Flash 芯片开发驱动程序呢？我们的 Flash 驱动又是如何来识别该 Flash 芯片的型号，如何加载对应的驱动函数的呢？

首先，我们来看如何为新 Flash 芯片开发驱动程序。由我们的 Flash 驱动的模块划分可以看到，我们只需要针对特定芯片，实现 chipDrvLib 层的函数。对应的文件的命名规则是：flash28[29]xx320[640,256]DrvLib.c。其中，28,29 表示的是两大 NOR Flash 系类，请参考 Flash Wiki。那么在 chipDrvLib 层需要实现哪些函数呢？我们可以看到，flashDrvLib 层是通过一个函数结构体来和 chipDrvLib 层关联的。

flash28f640DrvLib.c

```
……
struct flash_drv_funcs_s flash28f640 = {
    FLASH_2F640, INTEL,
    flashAutoSelect,
    flashEraseSector,
    flashRead,
    flashWrite,
```

```
    flashLockSector,
    flashUnlock
};
```

参考这个上面的例子，我们很容易就可以看出，在 chipDrvLib 层我们需要实现上面这几个基本的 Flash 操作的函数。

下面我们解决第二个问题，Flash 驱动又是如何来识别该 Flash 芯片的型号，如何加载对应的驱动函数的呢？请看 flashDrvLibInit()函数。

稍作分析，我们可以看到，不同驱动的 Flash 芯片对应一个函数结构体，如果该函数结构体中的 flashAutoSelect()能够成功返回，则 flashDrvLib 层的 flashDrvFuncs 结构体指针就会被赋值为对应芯片的函数结构体的地址。

4.3.2 flashDrvLib 的 buffer 策略

在 flashDrvLib 层使用了 3 个 sector 大小的 buffer。主要是为了减小对 Flash 的反复写入，提高 flash 操作的速度。为什么会这样呢？因为 Flash 的写入必须首先擦除整个 sector（一般为 128Kbyte），然后将修改的数据写入，当然也包括这个 sector 中未被修改的数据，因为这些已经被擦除了。下面就分析如何实现这个 buffer 策略。

首先请看 3 个 buffer 的数据结构：

flashDrvLib.c

```
……
#define TOTAL_LOADED_SECS   3
……
LOCAL struct flashLoadedSectorInfo {
    SEM_ID fsSemID;
    int    sector;
    int    dirty;
    char *buffer;
} flashLoadedSectors[TOTAL_LOADED_SECS];
```

可以看到，每个 buffer 结构体有一个信号量，该 buffer 所存放的 sector 号，该 buffer 是否被修改（dirty），然后一个指向数据块的指针 buffer。这个数据块在 Flash 初始化函数中动态申请，大小为一个 sector 大小。

下面，我们来看看这 3 个 buffer 在 Flash 的读写中起的作用。我们约定，一次对 Flash 的读写操作不能够超过一个 sector 的大小，也就是不能够一次操作的数据，跨越两个 sector 中。一般在代码中使用下面的判断限制。

flashDrvLib.c

```
……
    if (offset < 0 || offset >= FLASH_SECTOR_SIZE) {
        printf("flashBlkRead(): Offset 0x%x invalid\n", offset);
        return (ERROR);
    }

    if (count < 0 || count > FLASH_SECTOR_SIZE - offset) {
        printf("flashBlkRead(): Count 0x%x invalid\n", count);
        return (ERROR);
    }
……
```

首先，我们看 buffer 在 flash 的读操作中的作用。

flashDrvLib.c

```
……
    /*
     * If the sector is loaded, read from it.   Else, read from the
     * flash itself (slower).
     */
    for (i = 0; i < TOTAL_LOADED_SECS; i++) {
        if (flashLoadedSectors[i].sector == sectorNum) {
            if (flashVerbose)
                printf("flashBlkRead(): from loaded sector %d\n", sectorNum);
            bcopy(&flashLoadedSectors[i].buffer[offset], buff, count);
            return (OK);
        }
    }

    flashDrvFuncs->flashRead(sectorNum, buff, offset, count);

    return (OK);
……
```

上表中灰色背景为对数据的操作。可以看到，首先会查找要读的 sector 是否在这三个 buffer 中，如果在，那么就直接从这个 buffer 中读取，然后成功返回。如果不在，那么就直接从 Flash 芯片读取，然后成功返回。

下面，我们重点来看一下写的操作。由于代码太多，下面就不列举出来了。下面将其流程描述如下：

第一步：会查找要写的 sector 是否在这三个 buffer 中，如果在，那么就将修改的数据直接写到这个 buffer 中的相应位置，标志该 buffer 已经被修改，dirty=1。然后成功返回。（你可能会怀疑，这时候数据还是在内存的 buffer 中，并没有真正写入 Flash 中？恩，确实是这样的。那么什么时候会写入呢？我们最后再解决这个问题）

第二步：若要写的 sector 不在这三个 buffer 中，则测试能否将数据直接写入 Flash 中。能够将数据直接写入 Flash 的条件是，Flash 中需要写入的位置处的数据都为 0xFF。如果可以直接写入，则调用 chipDrvLib 的函数直接写入，然后成功返回。

第三步：将第 1 个 buffer 中的数据写入 Flash 中。然后将 buffer 依次前移，空出最后一个 buffer。

第四步：将要写的 sector 读入最后一个 buffer 中，然后将要修改的数据直接写入这个 buffer 中，标志该 buffer 已经修改，dirty=1。然后成功返回。

如果是由第一步或者第四步成功返回，那么数据其实还是在内存的 buffer 中，尚未写入到 Flash 中，那么什么时候会写入呢？这个功能，就是由函数 flashSyncFilesystem()实现的。

flashDrvLib.c

```
int flashSyncFilesystem(void)
{
    int i;
    for (i = 0; i < TOTAL_LOADED_SECS; i++) {
        FS_CACHE_LOCK(i);
        if (flashFlushLoadedSector(i, 0) != OK) {
            FS_CACHE_UNLOCK(i);
            return (ERROR);
        }
        FS_CACHE_UNLOCK(i);
    }
    return (OK);
}
```

4.4 chipDrvLib

chipDrvLib 层的函数是实际操作 NOR Flash 的寄存器。在这部分的实现过程中，需要反复参考 Flash Basic Operation 中的内容。你暂时可以将 I/O 系统，文件系统的知识都抛掷脑后，它们和我们现在要做的一点关系都没有。

下面为对 Flash 地址的操作宏。在 chipDrvLib 中反复使用到该宏。

flashDrvLib.c

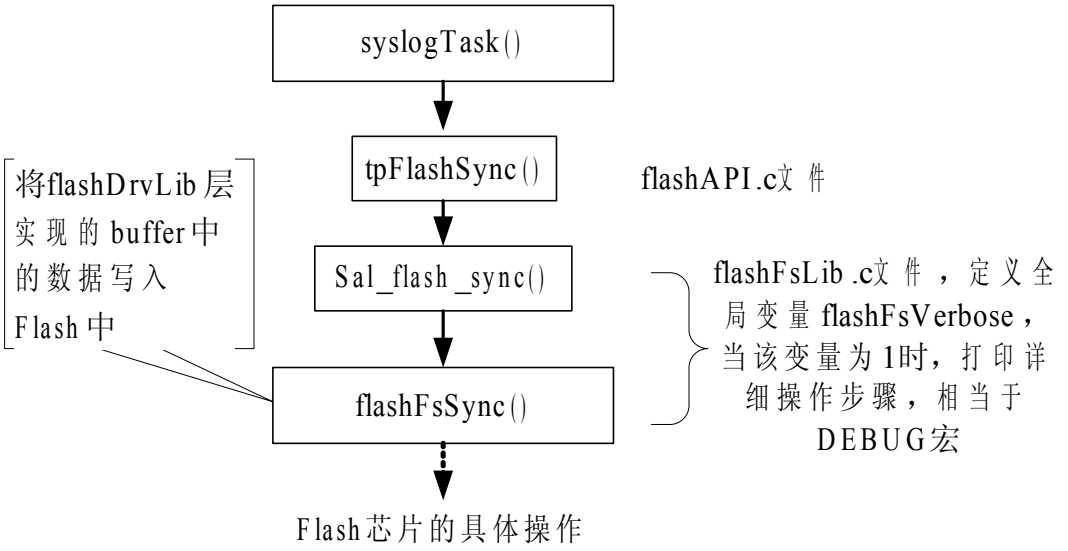
```
#define FLASH_ADDR(dev, addr) \
    ((volatile UINT8 *) (((int)(dev) + (int)(addr)) ^ xor_val))

#define FLASH_WRITE(dev, addr, value) \
    (*FLASH_ADDR(dev, addr) = (value))

#define FLASH_READ(dev, addr) \
    (*FLASH_ADDR(dev, addr))
```

4.5 Flash 驱动应用实例

xCat 平台 SDK：CPSS-3.4P1

问题描述：打开交换机的网页之后，在终端（串口）反复打印信息：flashFsSync()…… done。

问题原因：syslogTask 在反复的写 Flash，其函数的调用关系如下：



出现打印信息的原因是在 flashFsLib 中的调试用全局变量被置位 1。当然，syslogTask 任务反复擦写 Flash 也是不正常的，但是不在我们这里讨论的范围之内。

5. Flash Basic Operation

MCU 对 Flash 的基本操作有两个，一是从 Flash 读取数据，另一个是将数据写入 Flash 中。这分别对应读操作（read）和写操作（program/write）。由于 Flash 特性的原因，在写操作前需要将被写的区域全部擦除，所以还需要一个擦除操作（erase）。

在开始编写具体的操作代码之前，需要了解的就是一张 Flash Command 表。这张表规定了在开始某一种操作之前，首先要发给 Flash 芯片的命令。例如，要读取 Flash 芯片中某一个地址的一个字节的数据，首先得告诉 Flash 芯片，下面要进行 Flash 的读取操作。那怎么做到呢，那么就向 Flash 芯片的基地址处写入一个规定的字节，例如规定向 Flash 基地址写入数据 0XFF 表示读操作。只有发送了这样一个命令之后，Flash 芯片才会切换到读的操作。然后你再发送你想要获取的数据的地址到地址线上，这样 Flash 芯片就会把该地址处的值读取出来，传到数据线上。

下面这个表格就是 Intel® Embedded Flash Memory (J3 v. D) device 的命令表。各个不同的芯片型号对应的命令表可能会有区别。

Table 31. Command Bus Operations for

Command		Setup Write Cycle		Confirm Write Cycle	
		Address Bus	Data Bus	Address Bus	Data Bus
Registers	Program Enhanced Configuration Register	Register Data	0060h	Register Data	0004h
	Program OTP Register	Device Address	00C0h	Register Offset	Register Data
	Clear Status Register	Device Address	0050h	---	---
	Program STS Configuration Register	Device Address	00B8h	Device Address	Register Data
Read Modes	Read Array	Device Address	00FFh	---	---
	Read Status Register	Device Address	0070h	---	---
	Read Identifier Codes (Read Device Information)	Device Address	0090h	---	---
	CFI Query	Device Address	0098h	---	---
Program and Erase	Word/Byte Program	Device Address	0040h/ 0010h	Device Address	Array Data
	Buffered Program	Word Address	00E8h	Device Address	00D0h
	Block Erase	Block Address	0020h	Block Address	00D0h
	Program/Erase Suspend	Device Address	00B0h	---	---
	Program/Erase Resume	Device Address	00D0h	---	---
Security	Lock Block	Block Address	0060h	Block Address	0001h
	Unlock Block	Device Address	0060h	Device Address	00D0h

5.1 Read Operation

读操作在很多文档中也叫做读模式（read mode）。一般来说，读操作获得的数据有 4 种。分别是 Read Array，Read Status Register，Read ID，CFI Query。其中，Read Array 就是获得存储在 Flash 中的数据。

下面以 Read Array 操作，来说明对 Flash 中数据的读取过程。代码如下：

flash28f640DrvLib.c

```
LOCAL int flashRead(int sectorNum, char *buff, unsigned int offset, unsigned int count)
{
    flashReadReset();
    if (sectorNum < 0 || sectorNum >= flashSectorCount)
    {
        printf("flashRead(): Illegal sector %d\n", sectorNum);
        return (ERROR);
    }

    bcopy((char *)(FLASH_SECTOR_ADDRESS(sectorNum) + offset), buff, count);

    return (0);
}
```

flashReadReset()函数：Read Array Command，向 Flash 基地址（0x00）发送 0xFF，告诉 Flash 芯片，下面进入了对存储数据的读操作。

bcopy(……)函数：真正的对 Flash 中数据的读操作。可以看出，其操作方式和对内存中（SDRAM）数据的操作时一样的，都是使用的 bcopy 函数。(char *)(FLASH_SECTOR_ADDRESS(sectorNum) + offset)参数对应的就是数据在 Flash 中的位置，或者说是地址。

5.2 Write Operation

这是因为 Flash 的存储单元中，将比特 1 变为比特 0 和将比特 0 变为比特 1 需要两种不同的操作机制。其中，Erase 操作只能将存储单元的比特 0 变为比特 1，并且，一次只能够以一个 block 来操作(一个 block 为 128K 或者 64K，器件不同会有差别)。而 Program/Write 操作只能将存储单元的比特 1 变成比特 0，反之就不行。例如，可以将 FF 变成任意的 8bit 值，也可以将 0xA3 变成 0xA2，再变成 0xA0，但不能变成 0xA1。

所以，将数据写入 Flash 这个过程，其实对应了 Flash 操作中的两个操作，首先将该地址处的 block 擦除（使用 Erase 操作），然后再将数据写入到该地址处（使用 Program/Write 操作）操作。

5.3 Flash 安全和保护

对 Flash 中数据的安全和保护，主要有软件和硬件两种方式，每种芯片支持的可能有些不同，或者甚至不支持某一些操作。下面以 Intel® Embedded Flash Memory (J3 v. D) device 为例说明其保护的机制。

- block lock&unlock

Flash 支持对每一个 block 上锁，被上锁的 block 不能够进行擦除（Erase）操作和编程（Program/Write）操作。并且，在 Flash 芯片复位或者断电重启之后，上锁状态不会丢失。

Flash 支持对整片芯片进行 unlock 解锁。

这个功能有什么作用呢？这个可以在防止程序修改某些 block 的内容。例如，存放代码的 block，在每一次烧写成功之后，要进行上锁操作。这样就可以防止这些代码被修改。又例如，某些只读的配置文件，可以对其所在的 block 进行上锁，这样就可以防止程序修改其中的内容。

存在的疑惑：解锁时整片芯片，上锁是针对一个 block。如果一个需要上锁的 block 更新，必须先解锁（整片 Flash 芯片），在更新该 block 之后再上锁（该 block 上锁）。这样其他的原来上锁的 block 不就处在 unlock 状态了吗？

- OTP Protection Registers

该寄存器为 128 比特，其中前 64 比特为芯片唯一序列号，只读，后 64 比特为用户可读写。可以利用此来进行 Flash ID 加密。有关 Flash 加密的相关内容，请参考详细的资料。

- VPP/VPEN 引脚

可以在硬件设计时利用这些引脚电平对整片 Flash 芯片来进行保护。

6. Flash Wiki

关于 Flash 的基本概念和知识，请参考 Flash wiki。

下面就几个用到的比较重要的概念作出解释。

block（在驱动代码中用 sector 表示）：块，在 NOR Flash 中进行擦除的单位就是 block，也就是 Erase 操作一次必须擦除一个 block，而不能够是一个字节，几个字节等。block 的大小根据芯片的不同会有变化，常见的例如 128Kbytes，64Kbytes。