计算机测量与控制.2015.23(1)
Computer Measurement & Control

文章编号:1671-4598(2015)01-0201-04　　中图分类号:TP303　　文献标识码:A

基于 STM32 的 NAND flash 的块分配框架设计

吴　磊，翟云飞
（北方工业大学 信息工程学院，北京　100144）

摘要：STM32 对大容量数据文件存储与管理问题可通过 NAND flash 来解决；而 NAND flash 的高效管理需要文件系统参与；NAND flash 有特殊的块读写及擦除机制，一般的嵌入式文件系统组织结构并不完全兼容 NAND flash；针对 NAND flash 的特点兼顾 STM32 的资源承受力要提出新的 NAND flash 块分配框架，框架通过块分配槽这种数据结构，在不使用块分配表与垃圾表的情况下，实现了 NAND flash 均衡负载与垃圾块的回收；同时通过节点分配栈与文件节点表的结合来提高 STM32 对文件的读写速度，空间利用率和系统性能；仿真实验和计算结果表明该块分配框架可有效提高 NAND flash 块的均衡负载与节省 RAM 空间。

关键词：块分配槽；节点分配栈；文件节点表；STM32；NAND flash

A Block Allocation Frame Design Based on NAND flash of STM32

Wu Lei，Zhai Yunfei
（Institution of Information Engineering，North China University of Technology，Beijing　100144，China）

Abstract：The problem that store and management of STM32 to large volume data files can be solved through large capacity NAND flash. The high efficient management of NAND flash needs the participant of file system. NAND flash has special read，/write and erasure mechanism，and general embedded file system organization structure can not be completely compatible with NAND flash. A new NAND flash blocks allocation frame is proposed aiming at the characteristics of NAND flash and the capability of STM32 resources，which realizes the balanced load and recycle of invalid blocks of NAND flash through block allocation box without using block allocation table and invalid block，and enhances the read/write speed to files，the utility rate of space and system performance through node allocation stack and file node table at the same time. Simulation experiment and calculation result indicate that the block allocation frame can efficiently realizes the balanced load and the saving of RAM space.

Keywords：digital watermark；node allocation stack；file node table；STM32；NAND flash

0　引言

STM32 微控制器被广泛应用于工业领域。NAND flash 具有高密度存储能力，因此 STM32 结合 NAND flash 可有效解决近年来嵌入式系统面临的大容量数据存储的问题。但 NAND flash 存在空间利用率低以及块负载不均衡等问题。

传统的嵌入式文件系统主要缺点有与 NAND flash 块的分配与回收方式不兼容；以链接来组织 flash 分配表项，不利于 STM32 高速性能的发挥；进行 flash 的均衡负载时，借助块空闲表与垃圾块表等静态表，降低系统性能。

因此基于 STM32 的架构结合 NAND flash 结构特点。以节省 RAM 空间，块的负载均衡为目的提出一种改进的块分配框架。该框架以块分配槽和节点分配栈为核心，省去了分配表与垃圾表等静态表，优化了块分配结构，对进一步提高使用 NAND flash 进行大数据存储的 STM32 架构的效率将是十分有意义。

1　硬件信息

1.1　STM32 简介

STM32 是 32 位微控制器。STM32 片内配有 512 kB flash、64 kB RAM。STM32 基于 Cortex-M3 内核并采用 FSMC（可变静态存储控制器），可直接与 Nor flash、NAND flash、SRAM 等存储器的引脚相连，使得 STM32 可以更方便地管理多个不同种类的存储器。

1.2　NAND flash 结构

NAND flash 强调位存储，以 K9F1G08X0A 系列为例，一个基本存储单元只存储一个 bit，8 个存储单元组成存储行，存储行组成页，页是访问 NAND flash 的基本单位。64 页组成块，1024 个块组成 NAND flash。每个页中有 64 Byte 的 spare 区。NAND flash 容量为（2K＋64）B×64（Page）×1024（Block）＝132MB。spare 区不存储数据，因此 K9F1G08X0A 的实际存储容量为 128MB。NAND flash 以页为单位读写，以块为单位擦除，块在写入前要先进行擦除。

2　块分配框架的基本数据结构

2.1　块分配槽

块分配槽结构如图 1 所示。块分配槽中每一项映射 NAND flash 的一块，块分配槽中的项数是 NAND flash 中有效块的数目加 1，多出来的一项记录 NAND flash 中有效块数量。

块分配槽是文件管理算法中块分配的关键数据结构。块分配槽项中有以下内容：①块物理地址②块的当前状态。程序中块分配槽项可由结构体 struct Block _ info 来描述：

```
{
  short int block_add;//块物理地址
unsigned char block_status;//块状态
};
```

要建立块分配槽，首先要遍历整个 NAND flash 块，将出厂坏块跳过，统计有效块数量为 nand _ block _ num，同时建立块分配槽并将有效块物理地址加入到块分配槽中。

收稿日期：2014-05-08；　**修回日期**：2014-06-23。
基金项目：北京市自然科学基金资助项目(4131001)。
作者简介：吴　磊(1963-)，男，北京市石景山区人，副教授，硕士研究生导师，主要从事嵌入式方向的研究。

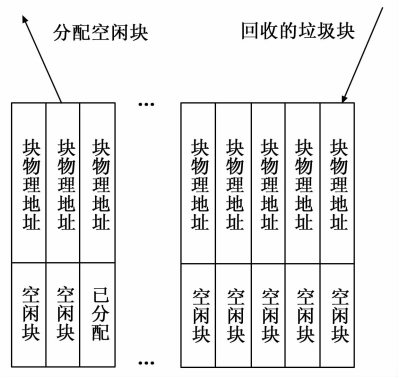

图 1 块分配槽示意图

块的状态有 4 种：空闲、已分配、垃圾块和已坏。虽然块分配槽中的一项与一块映射，但映射并不是固定的，随着文件的增加删除而动态变化。

2.2 文件节点与文件节点表

块分配表是管理 NAND flash 块的数据结构，而对文件信息的存储管理则需要另外 3 个数据结构，文件目录表 FDT (File Directory Table)，文件节点 fnode（file node）和文件节点表 FNT（File Node Table)。FDT 是系统索引文件必须的数据结构。

文件索引可分为分散式索引与集中式索引。分散式索引方式扩展性强，空间利用率较高，但保存的文件信息十分琐碎，文件不能快速定位；集中式索引能快速定位读取文件，但扩展性差。

该框架采用分散式与集中式相结合的存储机制，将文件所占块的指针使用连续的 RAM 空间进行存放。集中存放块指针的数据结构称为文件节点（fnode)。各个文件的 fnode 的最后一项存入该文件下一个 fnode 的编号或存入 0xFFFF 表示文件结束。fnode 的大小固定，一个文件可能有一个或多个 fnode。单个 fnode 集中记录了文件块指针，可以提高读取文件的速度，同时同一文件 fnode 的之间使用链接相连，保留了 FAT 表扩展性强的特点。fnode 可用结构体 struct Fnode _ info 来进行表示。

```
struct Fnode _ info
{
  short int * block _ address [node _ num]; //指向 NAND _ flash 块的指针
  short int next _ node; //文件下一节点编号
};
```

其中 node _ num 表示节点中指针的数量，这个根据 STM32 的实际情况由开发者指定。以达到速度、扩展性和空间利用率之间的平衡。

FDT 表是 NAND flash 中所存储文件的总索引目录。FDT 的表项结构分为两部分，即文件名和该文件首个 fnode 的编号。FDT 表的数据结构可以由结构体 struct File _ Dir 表示如下：

```
struct File _ Dir
{
  char file _ name [filename _ len]; //文件名
short int fnode _ num; //文件节点编号
}
```

文件名需要额外消耗 RAM 空间，因此要权衡 STM32 中 RAM 的容量及实际应用对文件名长度的要求合理制定 filename _ len 的大小。

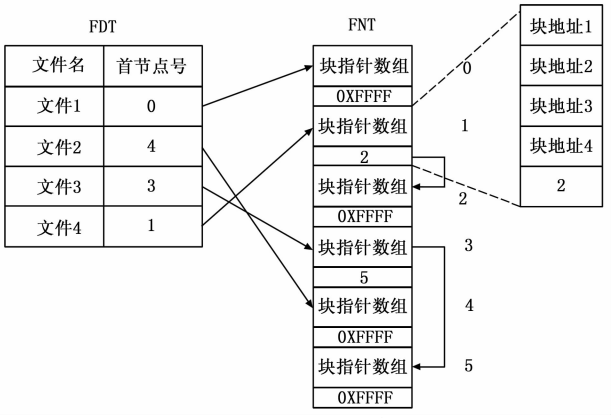

图 2 FDT 与 FNT 映射关系

如图 2，FDT 表建立了文件名与文件节点编号之间的映射关系，fnode 是固定大小，因此通过文件首节点编号即可快速在 FNT 中索引。

2.3 节点分配栈

节点分配栈中记录的是空闲节点的节点号。在已分配的节点释放时，并不真正释放 RAM 空间，而只是将文件占用节点中的块指针清除，将 next _ node 字段修改为 0XFFFF。并将节点号重新放回节点分配栈之中。

节点分配栈数据结构可由一个数组表示，通过指针实现节点分配与回收。

```
short int node _ malloc _ stack [nand _ block _ num+1];
short int * sp;
```

指针 sp 永远指向第一个可分配的节点号。起始状态，节点分配栈中的个单元的节点号依次为 0-nand _ block _ num-1，与 FNT 中的节点编号一一对应。多出来的空间记录 FNT 表的长度。

3 文件管理算法描述

定义变量如下：

```
Block _ info block _ queue [nand _ block _ num]; //块分配槽定义
File _ dir file _ dir _ table [nand _ block _ num]; //文件目录表定义
short int fdt _ num=0; //文件目录表编号
short int fnt _ num; //文件节点编号，从节点分配栈中获得
struct Block _ info * p1, * p2, * p3; //定义 Block _ info 类型指针
struct Fnode _ info * p, * q; //定义文件节点的分配指针
```

3.1 文件操作过程

文件存储过程如下：

1）在 FDT 表中登记；

从节点分配栈中取一个节点号作为该文件的首节点号；

```
fnt _ num= * sp;
file _ dir _ table [fdt _ num]. fnode _ num=fnt _ num;
```

sp－－；//文件节点栈指针下移

file _ dir _ table［fdt _ num］. file _ name＝“filename”；

2）分配文件节点；

计算所要分配的节点数 fnode _ amount。

fnode _ amount＝（file _ size/block _ size＋1）/node _ num；

循环 fnode _ amount 次进行（3）的操作。

3）分配节点；

p＝（struct Fnode _ info ＊）realloc（p，sizeof（struct Fnode _ info））；

q＝q＋ file _ dir _ table［fdt _ num］. fnode _ num；//找到所分配空间

从块分配槽中得到 node _ num 个空闲块，将物理地址加入到 fnode 中的块指针数组，循环 node _ num 次执行。

q－＞block _ address［node _ num］＝block _ add；//将块物理地址加入块指针数组

如果 fnode _ amount＞1，那么从节点分配栈中取节点号 fnt _ num。

fnt _ num＝＊sp；

sp－－；//文件节点栈指针下移

将该文件下一个节点号放入节点中；

q－＞next _ node＝fnt _ num；

q＝p；//将 q 移至 FNT 表头之后跳转至（3）再执行，否则加入结束码；

q－＞next＝0XFFFF；//加入结束码

（4）启动 STM32 向 NAND flash 发送地址和命令码，将 fnode 写入 NAND flash。

3.2　文件删除过程

1）根据要删除文件的文件名查找 FDT，找到所映射首节点编号 fnode _ num。

while（file _ dir _ table［i＝0－nand _ block _ num］. file _ name! ＝“filename”）；

2）定位要删除的首个节点

q＝ q＋file _ dir _ table［fdt _ num］. fnode _ num；

3）将节点号放回节点分配栈中

sp＋＋；//文件节点栈指针上移；

＊sp＝file _ dir _ table［fdt _ num］. fnode _ num；

将块物理地址放回块分配槽，将块分配槽抹除 fnode 中的块指针数组中的指针，循环 node _ num 次执行。

＊p3＝q－＞block _ address［node _ num］；

p3＋＋；//将快分配队列中的 p3 指针移到下一位置

q－＞block _ address［node _ num］＝0XFFFF；//抹除块指针数组中的指针信息

4）查看节点中的 next _ node 字段，如果为 0XFFFF 则删除结束。否则定位下一个节点。

q＝p＋q－＞next _ node；

之后循环执行（3）、（4）直到 next _ node 字段内容为 0XFFFF；

3.3　块分配槽管理算法

块分配槽有 nand _ block _ num＋1 个分配单元，所多出的一个分配单元存储有效块的数量 effect _ block _ num，随着坏块的逐渐增加，effect _ block _ num 也再不断递减。块分配槽由 3 个 struct Block _ info 类型的指针 sp1，sp2 与 sp3 来实现块的分配、回收以及 NAND flash 块的均衡负载。

sp1 指向第一个空闲块，sp2 指向槽队列入口，sp3 永远指向最近的一个待回收块。当文件需要物理块时，块分配进程 process _ alloc 会把 sp1 所指的当前空闲块状态设置为分配，将其中的块物理地址填入到节点中的块指针数组中，sp1 加 1 并与 effect _ block _ num 取模，如果分配多块那么就循环执行 proces _ alloc 进程；当文件删除，物理块需要回收时，回收进程 process _ recycle 会将待回收块的物理地址写入到 sp3 所指的当前块分配槽项中，并将块状态设置为垃圾块，之后 sp3 加 1 并与 effect _ block _ num 取模。回收进程擦除 sp2 指针当前所指的物理块进行回收，擦除完毕后，sp2 将所指队列单元状态设置为空闲，并加 1 与 effect _ block _ num 取模；当出现坏块时，坏块处理进程 process _ badblock 会将坏块数量记录下来，当 sp1 指针走完一个周期后，将块分配槽中存储有效块数量单元的内容减去坏块数目即改变 effect _ block _ num 为当前值；当 sp1 等于 sp2 时，表示没有物理块可供分配。当 sp2 等于 sp3 时，表示所有的待回收块已经回收完毕。算法的动态过程如图 3 所示，图 3（a）是某时刻的状态，图 3（b）是 sp1 分配一个物理块后的状态，图 3（c）是 sp3 接收一个待回收块后的状态，图 3（d）是通过 sp2 回收一个物理块后的状态。

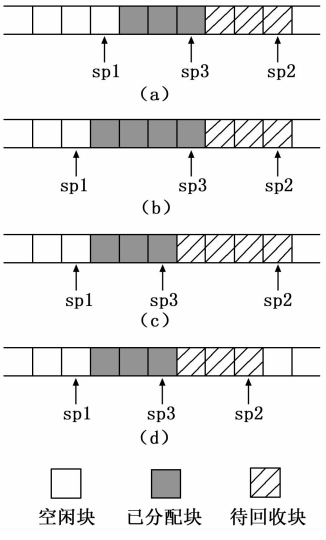

图 3　块分配槽操作过程示意图

通过使用块分配槽，避免了对空闲块表与垃圾块表的使用，节省了 RAM 空间；而且只需移动指针就可完成对块的查找。

4　框架性能分析

4.1　空间资源占用分析

STM32 的 RAM 空间仅有 64 kB，NAND flash 共有 1 024 块物理块，nand _ block _ num 为 1 024，每块 64 字节。文件管理框架中需要占用 RAM 空间的数据结构有块分配槽、节点分配栈、FDT 表和 FNT 表。

块分配槽每项中的块物理地址占用 2 Byte 空间，而块状态占用 1 Byte 空间，则块分配槽大小为 $L=$（2 B＋1 B）×1 024＝3 kB；节点分配栈中每项需要 2 Byte 空间来记录节点编号，因此，节点分配栈初始占用空间为 $S=2\mathrm{B}\times(1\ 024+1)\approx 2$ kB；FDT 表有文件名和文件首节点号，系统支持 8 字符文件名占 8 Byte 空间，而另一部分为节点的映射部分，占用 2 Byte，那么 FDT 中的每一项占用 10 Byte 空间，FDT 表大小

为 S_d = (8 B+2 B)×1 024=10 kB。

三者所占 RAM 空间为：$S+L+S_d=15$ (kB) 约为 RAM 的 23.5%。

FNT 随着文件数量的增加而不断加长。设所有文件共占有物理块数为 X，文件平均所占物理块数为 N_a，每个节点中块指针数组维数为 N_p。则平均每个文件所占节点数为 $N_n=\frac{N_a}{N_p}+1$，系统中的文件数为 $N_f=\frac{X}{N_a}$，每个文件节点的大小为 $S_n=2\cdot(N_p+1)$ 字节。

由此可得 FNT 中大小为：

$$S_f=N_n\cdot N_f\cdot S_n=$$

$$2X\cdot(\frac{1}{N_p}+\frac{1}{N_a})(N_p+1)=2X\cdot(\frac{1}{N_p}+\frac{N_p}{N_a}+\frac{1}{N_a}+1)(\text{Byte})$$

在 X 一定的情况下，设文件平均所占物理块数为定值，则可令 $f(N_p)=\frac{1}{N_p}+\frac{N_p}{N_a}$，对变量 N_p 对函数 $f(N_p)$ 求一阶导数得：

$f'(N_p)=(-\frac{1}{N_p^2}+\frac{1}{N_a})$，令 $f'(N_p)=0$ 计算得 $N_p=\sqrt{N_a}$。

由此可得当文件平均所占物理块数一定时，节点中块指针数组维数为 $\sqrt{N_a}$，可使 FNT 占用最少的 RAM 资源。从 S_f 的表达式可以看出，N_a 越大，FNT 越小，因此综合这两个条件就可以合理安排节点中块指针数组的维数。以较小的 RAM 代价换取较高的空间利用率。

4.2 块负载均衡与垃圾块回收策略

所谓块负载均衡指对物理块的磨损平均到各个物理块上，其平均度越强，块负载越均衡，NAND flash 的寿命也越长。在该块分配框架通过块分配槽实现块的负载均衡，框架中通过物理地址实现对物理块的索引。其原理是在初始时刻将所有块的物理地址组成一个集合 $A=\{x1,x2,x3\cdots,xn\}$，n =nand _ block _ num；将 FNT 中占用的物理块组成集合 $B=\{\Phi\}$；待回收块集合 $C=\{\Phi\}$。物理块地址信息在 FNT 与块分配槽中不断流动，实质上是在这 3 个集合中相互流动。集合 A 可分为两个子集 $A1=\{x1,x2,x3\cdots,xn\}$ 以及 $A2=\{\Phi\}$。初始时刻 $A1$ 表示待分配空闲块集，$A2$ 表示已回收空闲块集。

当需要分配物理块时，将 $A1$ 中的元素移至 B 中：$A1\xrightarrow{X1,X2,\ldots,Xm}B$；

当需要回收物理块时，将 B 中的元素移至 C 中：$B\xrightarrow{X1,X2,\ldots,Xp}C$；

当垃圾块回收完毕后，将 C 中元素移至 $A2$ 中：$C\xrightarrow{X1,X2,\ldots,Xq}A2$；

当需要 $A1$ 中的物理块分配完毕时，将 $A2$ 中的元素移至 $A1$ 中：$A2\xrightarrow{X1,X2,\ldots,Xr}A1$。

可以看到集合 $A1\cap A2\cap B\cap C=\{\Phi\}$，同时有 $A\cup B\cup C=\{x1,x2,x3,\cdots,xn\}$，且满足条件 $1\leqslant r\leqslant q\leqslant p\leqslant m\leqslant n$。

分析可得 $A1$ 中的物理块是当前损耗程度最低的，而 $A2$ 中的物理块是最近一次分配出去的回收块。文件不关心物理块的逻辑编号，只要知道物理地址就行，因此块分配槽机制可省去物理块的逻辑编号，做到物理块的最佳负载均衡。

4.3 仿真实验分析

设时间周期 T 后所有块的平均擦写次数为 $\bar{E}$，第 i 块的擦写次数为 E_i，通过计算方差 D，可以得到块的负载情况：

$$D=\frac{1}{n}\sum_{i=1}^{n}(E_i-\bar{E})^2$$

图 4 显示了对模拟块损耗的仿真结果。存储文件大小均值的模拟量从 1 到 10，统计在各个文件均值的情况下模拟块的损耗均值与最大值。

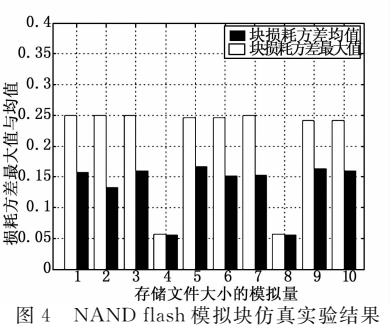

图 4 NAND flash 模拟块仿真实验结果

可以看到，通过本文中块分配框架对 NAND flash 块进行分配，其损耗方差最大不会超过 0.25，其负载均衡的程度是比较理想的。

5 结论

该 NAND flash 块框架所付出的静态开销占 RAM 资源约为 23.5%，这对 STM32 微控制器来说是完全可接受的；通过动态文件节点表与节点分配栈机制提高了文件的存取速度与 RAM 利用率，有利于发挥 STM32 性能，同时节俭了 RAM 开销；块分配槽使系统省去了空闲表，垃圾块表等静态数据结构，节省了 RAM，同时提高了 NAND flash 块的均衡负载，延长了存储器的使用寿命，很适用于需要处理大容量数据的嵌入式系统应用领域。

参考文献：

[1] 贾源泉，肖 依，赖明澈，欧 洋. 基于 NAND flash 的多路并行存储系统中坏块策略的研究 [J]. 计算机研究与发展，2012，49 (z1)：68－72

[2] 赵桦云，张敬帅. 基于 NAND flash 的数据存储系统设计 [J]. 单片机与嵌入式系统应用，2012，12 (1)：71－73

[3] 匡 伟，吕霞付，陈 勇，郑思远. 一种支持 FAT 文件系统的 Flash 转换层设计 [J]. 重庆邮电大学 (自然科学版)，2012，24 (2)：259－263

[4] 张亚辉，马胜前. 基于 STM32 的 Flash 存储器坏块自动检测 [J]. 计算机系统应用，2013 (6)：209－211.

[5] 熊安萍，刘进进，邹 洋. 基于对象存储的负载均衡存储策略 [J]. 计算机工程与设计，2012 (7)：2678－2682.

[6] 董 萍. NAND 存储器均衡算法实现 [J]. 微电子学与计算机，2013，30 (2)：53－56

[7] 张小进，罗海波. 一种基于 NAND flash 的 FAT 文件系统的研究与实现 [J]. 海南大学学报 (自然科学版)，2012，30 (4)：334－337.

[8] 王 标，周新志，罗志平. 嵌入式系统中 Nand flash 写平衡的研究 [J]. 微计算机信息，2008，24 (5)：8－9，26

[9] 杨 博，李 波. 基于 NAND flash 的嵌入式系统启动速度的研究 [J]. 计算机测量与控制，2010，18 (8)：1869－1871.