《振南 **znFAT--**嵌入式 **FAT32** 文件系统设计与实现》一书

【上下册】已正式出版发行

全国各渠道全面发售

（在当当、京东、亚马逊、淘宝等网络平台上搜索

关键字"**znFAT**"即可购买，各地实体书店也有售）

此书是市面上 唯一 一套详细全面而深入讲解嵌入式存储技术、**FAT32** 文件系统、**SD** 卡驱动与应用方面的专著。全套书一共 **25** 章，近 **70** 万字。从基础、提高、实践、剖析、创新、应用等很多方面进行阐述，力求通俗，振南用十年磨一剑的精神编著此书，希望对广大工程师与爱好者产生参考与积极意义。

此书在各大电子技术论坛均有**长期的**「抢楼送书活动」，如 21IC、elecfans 等等。

振南的【ZN-X 开发板】是市面上唯一全模块化、多元化的开发板，可支持 51、AVR、STM32(M0/M3/M4)

详情请关注 **www. znmcu. cn** （振南个人主页!!）

第5章

模式变换，百花争艳：znFAT与其他FAT的全面PK

通过第4章的努力，我们极大地提升了数据的写入效率。但是最后振南提到了“非实时模式”这一概念，还说它可以进一步提升数据写入效率。这是怎么回事？本章将给出解答。随后，znFAT的数据写入效率将达到极限，振南一直盼望的时刻终于到来了—“数据写速大比拼”，znFAT将与现有优秀方案(FATFS与EFSL)进行“较量”，看看谁的数据写入速度更快！接下来，振南还将继续介绍znFAT的工作模式，包括其配置方法以及各种工作模式的特点。不同工作模式在资源占用量、数据读写效率等方面均有不同，从而使znFAT可以适用于不同的硬件平台，满足不同的应用需求。最后，振南将对不同工作模式下的数据写入效率进行对比，让大家明确各种工作模式的性能水平。

5.1 登顶效率之峰

1. 非实时模式

在向文件写入一次数据之后，有两个操作是我们必须要做的，即更新FSINFO与文件大小。但是，如果是进行频繁数据写入的话，它将会产生大量的扇区读/写操作。而且它们每次都是对固定某个扇区进行操作，那么对同一扇区频繁读/写将会越来越慢，对整体的数据写入效率造成不小的影响，如图5.1所示。

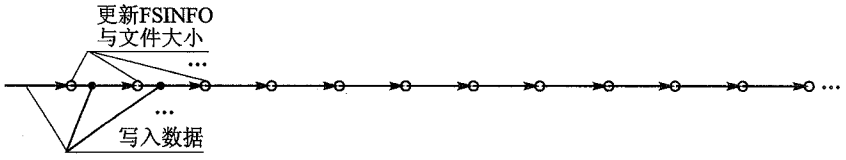

图5.1 向文件频繁写入数据时对FSINFO与文件大小实时更新

实际上，完全没有必要这样做。我们可以省去中间的若干次更新操作，而只在最后一次数据写完之后更新一次即可，如图5.2所示。这就是“非实时模式”，与之相对的便是“实时模式”，即前面的那种做法。

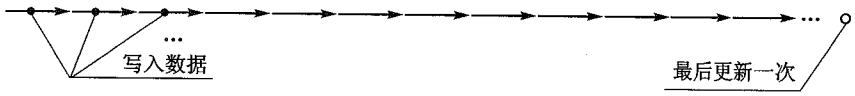

图 5.2　以非实时模式向文件频繁写入数据

这样，数据的写入效率就进一步得到了提升，从而达到极限。但是此时，有一个很重要的问题一定要意识到：在使用非实时模式向文件写入数据时，如果出现中途断电或 CPU 死机等异常情况，就可能会因为来不及更新 FSINFO 与文件大小（主要是文件大小）而造成数据的丢失。其实这一问题也存在于 CCCB 与 EXB 这些缓冲机制中，因为它们同样也来不及进行回写操作。但是，实时模式却可以在很大程度上避免这种情况的发生，它可以保证从一开始直到异常发生前最后一次写入的数据的完整性。当然，牺牲的是数据的写入效率与扇区的使用寿命。

2. 模式宏配置

实际使用哪种模式，“实时”还是“非实时”，这要由使用者来决定。到底是追求更高的数据写入速度，还是更看重数据的安全性。为了方便使用者根据自身需要在这两种模式之间进行选择，我们使用条件编译来进行处理，代码如下（znFAT.c）：

```
#ifdef RT_UPDATE_FILESIZE
Update_File_Size(pfi); //更新文件大小
#endif
#ifdef RT_UPDATE_FSINFO
Update_FSINFO(); //更新 FSINFO 扇区
#endif
```

还记得第 4 章中的硬件多扇区与软件多扇区吗？为了便于对其进行选择，我们也用的是条件编译的方法。根据特定宏是否被定义，从而决定了某个代码段是否参与编译，比如前面的 USE_MULTISEC_W、USE_MULTISEC_R 以及这里的 RT_UPDATE_FILESIZE、RT_UPDATE_FSINFO。振南将这些宏统一归类到一个头文件之中，它就是 znFAT 中的“配置文件”（config.h）。

5.2　与强者竞速

既然 znFAT 的数据写入速度号称已经“飙到极限”，那就有必要把它跟国际上现有的主流优秀方案做个“较量”了，看看相差多少或者超越多少。

现在比较有名的优秀方案有 FATFS、EFSL、UCFS、TFFS、DOSFS、ZLG/FS、沁恒 FAT 等。其实有很多人都对 FAT 很感兴趣，这些兴趣可能来自于对 FAT 及其相关算法、思想、编程技巧的好奇，或者是项目和产品的需求。其中一些高手也写出了自己的 FAT 方案，比如在 amoBBS（原 OurAVR）上用 AVR 单片机 DIY MP3 的

BoZai,再如许乐达同学作的xldFAT,还有号称中国第一的cnFAT等。对于这些方案,我们首先进行一个简介。

5.2.1 国内外优秀FAT方案简介

1. FATFS

这里直接引用chaN(FATFS的作者,身在日本)在其发布网站上对FATFS的简介:FATFS是一个通用的文件系统模块,用于在小型嵌入式系统中实现FAT文件系统。FATFS的编写遵循ANSI C,因此不依赖于硬件平台。它可以嵌入到各种价格低廉的微控制器中,如8051、PIC、AVR、SH、Z80、H8、ARM等(这里面有很多是日系CPU),而不需要做任何修改。

对于FATFS,振南要说:它确实很牛!无论在功能的完善程度上,还是在代码的运行效率以及可移植性上都可称得上是众多现有优秀方案中的佼佼者(到底有多优秀,读者到后面就能看到了)。FATFS一度是振南研发和推广znFAT道路上的最大劲敌(其实现在也是)。

起初,振南认为FATFS的作者是个绝顶聪明的人。但是,在看了他代码中的研发编年史之后改变了这种想法:他在把FAT当作毕生事业来做,FATFS的研发始于2006年,一直到现在还在不断更新和改进。真可谓十年磨一剑了!但是,经过对FATFS的深入研究之后发现,它并非像想像中的那么完美,仍然存在着很多问题,其中有一些可谓"硬伤":

① 它最低需要1 300字节左右的内存,所以在一些低端处理器上无法使用(其根源在于其数据缓冲的实现策略);

② 它没有实时模式,始终会有数据暂存于内存中,如果突然断电或CPU死机,必然造成数据丢失;

③ 物理层接口比较复杂,而且必须由使用者提供多扇区读写驱动的实现;

④ 代码可读性不强,使用者很难了解其内部实现,所以一旦出现Bug,很难立即解决;

⑤ 纯开源软件,因此缺乏原作者的相关技术支持与指导,只能靠使用者自行领悟。

2. EFSL

EFSL的全称是Embeded File System Library,即嵌入式文件系统库。它是来自sourceforge的、由比利时的一个研究小组发起的开源项目,此项目正在持续更新,源码中也有很多注释,研读起来比较容易,潜力不错。EFSL兼容FAT32,支持多设备及多文件操作。使用每个设备的驱动程序时,只需要提供扇区写和扇区读两个函数即可。

据说,EFSL的效率是非常高的。但是它的物理层接口只支持单扇区读/写,并

没有多扇区的接口。同时，还要看它所占用的内存量。就算它的效率确实很高，但是若以较高的内存消耗为代价，也不足为取。不过，一切还是要以实测结果为准。

3. UCFS

对于 UCFS 可能有些人并不熟悉，但是提起 UCOS 一定有所耳闻，它们都来自于 Micrium 公司。出身“名门”，其代码质量、稳定性及可移植性自然无可挑剔。不过它并非开源项目，而是商用软件。从性能和执行效率上来说，振南并不认为 UCFS 会有多好，因为它的物理层驱动接口也只支持单扇区读/写，而无多扇区。

4. TFFS

对于 TFFS 可能很多人都没听说过。但是如果提起 Vxworks，大家就会耳熟能详。TFFS 就是专门服务于 Vxworks 的文件系统，全称为 True Flash File System。TFFS 可以在 Flash 存储设备上构建一个基于 DOS 的文件系统(即 FAT)，用于存放操作系统镜像以及应用程序，以便于程序的更新和升级。因为振南本人长期从事 Vxworks 的相关研发工作，因此对于 TFFS 所带来的便捷深有感触。不过，TFFS 基本上是与 Vxworks 绑定的，想要把它从中提取出来为我们所用，难度较大。而且，它也不是免费软件，不能私自使用。

5. DOSFS

DOSFS 是由美国一个叫 Lewin A. R. W. Edwards 的人研发的(这个人好像还出了一本书叫《嵌入式工程师必知必会》，有兴趣的读者可以看看)。从它的名字上可以看出来，Lewin 是想在嵌入式微处理器上实现一个类似 DOS 的系统，其实质就是 FAT 文件系统。从它的代码来看，也只是一个雏形，功能还比较少，配套的文档资料也不够齐全。关于 DOSFS，振南没有实际用过，不过曾见过有人把它用在了产品里，似乎还比较稳定。

上面介绍的几种比较流行而知名的嵌入式 FAT 文件系统方案均来自国外，下面就是国内的方案了。

6. ZLG/FS

ZLG/FS，顾名思义，就是周立功公司研发的文件系统方案，说得更准确一些应该是周立功公司的 ARM 研发小组的成果。它是以 UCOS 嵌入式操作系统的一个中间件方式出现的，也就是说，可以与 μC/OS 很好地进行协同工作。它也是一个开源的软件，在国内嵌入式平台上，尤其在 ARM 平台上得到了较为广泛的应用。但是，ZLG/FS 的数据读/写速度实在让人堪忧。仔细研读它的源代码就会发现，它在实现上使用的一些策略导致了它的效率低下。

7. 沁恒 FAT

南京沁恒公司的 FAT 方案做得就很不错。提起沁恒，似乎有点耳闻。那振南再提醒一下：CH375 芯片。对，它是专门用于读写 U 盘等 USB 存储设备的控制器芯

片，沁恒FAT文件系统就是与这个芯片配套绑定的，用于实现U盘上的文件操作。CH375已经算是一个经典芯片，凡是有U盘读/写需求的中低端项目估计有一半以上都在用这个芯片。可以说，沁恒FAT是嵌入式FAT文件系统商业化的一个典范。不过遗憾的是，沁恒FAT是纯商业软件，我们是看不到半点源代码的。振南感觉，FAT文件系统业已成为沁恒公司的一大产品和技术支柱，这也揭示了嵌入式FAT文件系统在功能需求以及市场价值上的巨大潜力。

总体来说，国内在嵌入式文件系统方面的研究仍然起步较晚，而且在原创开源与创新意识上远远落后于国外。国内的很多开发者一直秉承着“拿来主义”，但是这样我们不会有任何发展。让我们真正动起来，做出属于我们自己的东西。

列举了这么多的方案，是不是感觉znFAT其实很渺小。尽管如此，znFAT还是要向它们发起挑战。也许，它可以像“兵临城下”中的瓦西里一样将敌人个个秒杀，你看到硝烟了吗？

5.2.2 速度的“较量”

下面就从诸多方案中选取两个最具代表性的方案（FATFS与EFSL）来与znFAT进行较量，同时还要兼顾它们在空间方面的占用情况。在这场较量之中，我们会让各个方案均运行在最高速的极限状态下。它们所占用的内存资源量同样也是一个很重要的指标。谁能做到既省内存，速度又快，谁才是真正的胜者。这也算是一种“时空平衡”吧。

测试方法很简单：用各个方案向文件中写入相同数据量的数据，看看它们分别会花费多少时间。但是在具体测试方法的细节上，我们分为以下4种情况：

ⓐ 向文件写入10 000次数据，每次数据量512字节；

ⓑ 向文件写入10 000次数据，每次数据量578字节；

ⓒ 向文件写入1 000次数据，每次数据量5 678字节（不使用硬件多扇区）；

ⓓ 向文件写入1 000次数据，每次数据量5 678字节（使用硬件多扇区）。

有人可能会问：“为什么要分这4种情况？它们各有什么意义呢？”前两项可以测出小数据量频繁写入时的效率表现（ⓑ比ⓐ多出了55个字节的不足扇区数据，看看znFAT中的EXB能发挥多大作用）；后两项则可以用于测试在频繁大数据量写入时，尤其是使用软硬两种多扇区实现方式的情况下，文件系统方案的效率表现（看看znFAT中的CCCB以及基于连续簇链段的硬件多扇区优化能将数据写入效率提升多少）。表5.1列出了在ZN-X开发板（51平台）上测出的上述4种情况下各方案的实际结果。

从表5.1可以看到，在以整扇区（即512字节，不涉及不足扇区数据的处理）或者以较大数据量（使用软件多扇区，CCCB机制发挥作用）进行数据写入时，数据的平均写入速度已经很接近物理层直接写单扇区的速度。我们要明白一点，文件系统层面上的数据写入速度再快也不可能比物理层快，最多与之持平。所以，振南说znFAT

的数据写入效率已经达到极限。另一方面，在使用硬件多扇区的情况下，znFAT 把数据写入速度从 126 KB/s 提升到了 162 KB/s，而 FATFS 从 123 KB/s 提升到了 150 KB/s，分别提升了 36 和 27 个单位。很显然，znFAT 对硬件多扇区优势的利用更加充分。说白了就是，znFAT 比 FATFS 找到了更多的连续扇区。也许我们基于连续簇链段思想的硬件多扇区优化比 FATFS 中使用的多扇区策略更加优越、更加强劲。最后，不要忽略更重要的一点：znFAT 比 FATFS 还少用了 500 多字节的内存资源。

表 5.1　FATFS、EFSL 与 znFAT 在 51 平台的数据写入效率比较

内核	文件系统方案	RAM 使用量/字节	ROM 使用量/KB	每次数据量/B	写入次数	总数据量/KB	用时/s	数据写入速度/(KB/s)	多扇区
51 单片机（主频 22 MHz）、物理单扇区写入速度 144 KB/s、硬件多扇区写入速度 168 KB/s	znFAT（最大模式）	1 348（不计数据缓冲）	35	512	10 000	5 000	37	135	
				578	10 000	5 645	86	66	
				5 678	1 000	5 545	44	126	软多
				5 678	1 000	5 545	34	162	硬多
	FATFS（非 Tiny 模式）	1 856（不计数据缓冲）	31	512	10 000	5 000	37	135	
				578	10 000	5 645	85	66	
				5 678	1 000	5 545	45	123	软多
				5 678	1 000	5 545	38	150	硬多
	EFSL	3 286	40	未在 51 上移植成功					

曾经有人指着上面的测试结果向振南质疑："我觉得你这个测试实验还不太具有代表性，也许 FATFS 在 51 平台上确实表现不给力，但这并不能说明它在其他 CPU 平台上也不敌 znFAT！"确实是这么回事，不过振南要说：水涨船高，随着 CPU 性能的提升，znFAT 的效率表现也会更加出色。为了证明这一点，振南已经把 znFAT 移植到了很多其他的 CPU 上来进行测试，包括 Cortex - M3、ColdFire、AVR、MSP430 等，实际测试结果如表 5.2～表 5.4 所列。

相信上面的这些测试数据已经足够说明问题了。说实话，为了制作上面的这些表格及其相关测试数据，振南花了近 1 个月的时间，主要原因在于：

① 测试中涉及的 CPU 平台比较多，很多芯片振南也是第一次使用，所以基本都是现学现用。当然，这里面也有一些网友和爱好者的协助（限于篇幅很多 CPU 上的测试结果并没有列举出来）；

② 将 znFAT 移植到这些 CPU 上也要花费大量的时间和精力。"对哦？znFAT 具体该如何移植呢？主要步骤有哪些？都要注意些什么？能不能详细全面地介绍一下。"这确实很有必要，如果我们要在某个 CPU 平台上使用 znFAT，那就必须先让它在这个平台上正确地运行起来，具体的移植方法请参见上册的《znFAT 移植与应用》。

表5.2 FATFS、EFSL与znFAT在Cortex-M3平台上的数据写入速率比较

内核	方案	RAM用量/字节	ROM用量/KB	数据量/字节	写入次数	总数据量/KB	用时/s	数据写入速率/(KB/s)	多扇区
Cortex-M3（主频70 MHz）、物理单扇区写入速度360 KB/s、硬件多扇区写入速度426 KB/s	znFAT（最大模式）	1 760（不计数据缓冲）	12	512	10 000	5 000	15	336	
				578	10 000	5 645	17	328	
				5678	1 000	5 545	16	334	软多
				5678	1 000	5 545	13	412	硬多
	FATFS（非Tiny模式）	1 740（不计数据缓冲）	12	512	10 000	5 000	15	336	
				578	10 000	5 645	19	298	
				5678	1 000	5 545	17	329	软多
				5678	1 000	5 545	14	398	硬多
	EFSL	1 514（不计数据缓冲）	14	512	10 000	5 000	19	266	
				578	10 000	5 645	36	156	
				5 678	1 000	5 545	27	209	软多

表5.3 FATFS、EFSL与znFAT在AVR平台上的数据写入速率比较

内核	方案	RAM用量/字节	ROM用量/KB	数据量/字节	写入次数	总数据量/KB	用时/s	数据写入速率/(KB/s)	多扇区
AVR单片机（主频16 MHz）、物理单扇区写入速度243 KB/s、硬件多扇区写入速度267 KB/s	znFAT（最大模式）	1 630（不计数据缓冲）	26	512	10 000	5 000	22	223	
				578	10 000	5 645	26	212	
				5 678	1 000	5 545	24	229	软多
				5 678	1 000	5 545	21	261	硬多
	FATFS（非Tiny模式）	1 710（不计数据缓冲）	25	512	10 000	5 000	22	223	
				578	10 000	5 645	28	198	
				5678	1 000	5 545	26	213	软多
				5678	1 000	5 545	22	256	硬多
	EFSL	2 320（不计数据缓冲）	31	512	10 000	5 000	30	162	
				578	10 000	5 645	46	121	
				5 678	1 000	5 545	38	144	软多

表 5.4 FATFS、EFSL 与 znFAT 在 ColdFile V2 平台上的数据写入速率比较

内核	方案	RAM 用量/字节	ROM 用量/KB	数据量/字节	写入次数	总数据量/KB	用时/s	数据写入速率/(KB/s)	多扇区
CF V2(主频 80 MHz)、物理单扇区写入速度 418 KB/s、硬件多扇区写入速度 467 KB/s	znFAT(最大模式)	1 787(不计数据缓冲)	12	512	10 000	5 000	12	402	
				578	10 000	5 645	14	392	
				5 678	1 000	5 545	14	398	软多
				5 678	1 000	5 545	12	446	硬多
	FATFS(非 Tiny 模式)	1 710(不计数据缓冲)	12	512	10 000	5 000	12	403	
				578	10 000	5 645	14	400	
				5 678	1 000	5 545	14	401	软多
				5 678	1 000	5 545	13	418	硬多
	EFSL	1 623(不计数缓冲)	14	512	10 000	5 000	14	356	
				578	10 000	5 645	29	195	
				5 678	1 000	5 545	21	266	软多

5.3 znFAT 的工作模式

前面介绍了实时与非实时模式,它们各有特点及其适用的场合。其实 znFAT 中还有更多的工作模式。

5.3.1 缓冲工作模式

CCCB 与 EXB 的提出使得数据的写入效率得到了大幅度的提升,但是也必须为此付出代价:它们在本质上都是缓冲机制,所以必然会消耗更多的内存资源。其实 CCCB 还好,主要是 EXB,一个扇区缓冲就要占用 512 字节的内存,更不用说独立方式了(第 4 章讲过独立方式会给每个文件都分配一个专属的 CCCB 与 EXB 缓冲区,这种情况下对内存的需求将是巨大的)。在实际应用中,目标平台的硬件资源也许并不足以让我们能够使用这些缓冲机制,或者实际项目指标根本就不要求多高的数据写入速度。所以,应该让开发者自己来决定是否使用这些缓冲机制,而不能把它们固定化。因此,我们引入了下面的这些宏,代码如下(config.h):

```
//CCCB 是 znFAT 中所使用的独特的簇链缓冲算法,可以极大地提升数据的写入速度
#define RT_UPDATE_CLUSTER_CHAIN    //是否实时更新物理 FAT 簇链
                                   //若不实时更新,则使用 CCCB 簇链缓冲
#define USE_ALONE_CCCB  //是否使用独立 CCCB,这种方式下 znFAT 将给每个文件分配一个
                        //独立的专属簇链缓冲,与之相对的是共享 CCCB,多个文件共享
```

```
                                    //一个簇链缓冲，这会涉及簇链缓冲的争抢问题，效率较前者低
#define CCCB_LEN (8)        //簇链缓冲的长度，必须为偶数，且不小于4
//EXB，即扇区交换缓冲，是znFAT中针对不足整扇区数据的专用缓冲区
//可较大程度上改善因数据拼接而导致的效率低下
#define USE_EXCHANGE_BUFFER //是否使用EXB缓冲机制
#define USE_ALONE_EXB    //是否使用独立EXB，这种方式下每个文件都有
                         //它单独的扇区交换缓冲，否则使用共享EXB
```

在znFAT的代码中，我们通过大量的#ifdef...#else...#endif或#ifndef...#else...#endif条件预编译语句来对代码进行选择性的编译，从而实现多种工作模式的切换。示例如下（具体实现请参见znFAT源代码）（znFAT.c）：

```
#ifndef  RT_UPDATE_CLUSTER_CHAIN //不实时更新物理FAT，即使用CCCB机制
 //将簇链暂存入CCCB中
#else //实时更新物理FAT，即不使用CCCB
 //将簇链直接写入物理FAT
#endif
#ifdef  USE_EXCHANGE_BUFFER //使用EXB缓冲机制
 //将不足扇区数据暂存入EXB中
#else //不使用EXB缓冲机制
 //将不足扇区数据直接写入物理扇区
#endif
```

5.3.2 自身模式较量

工作模式的提出自然会让人们产生这样的疑问："在各种工作模式下，znFAT分别会占用多少内存？数据的写入速度都能达到什么样的水平？"明确了这一点将有助于实际开发过程中在硬件资源与速度需求这两个方面寻求一个最佳平衡点。下面就针对znFAT的各种工作模式来进行一个比较（使用数据写入的ⓑ方案），如表5.6～表5.8所列。

如果说znFAT与FATFS、EFSL的较量是"横向较量"的话，那上面我们所做的就是针对于znFAT自身各种工作模式之间的"纵向较量"。可以看到，在各种CPU平台上，实时＋无缓冲模式（即最原始的全实时模式，没有任何优化与加速机制）所占用的内存资源是最少的，但是它的数据写入速度也是最低的（下降到了全速模式下速度的10%～30%）。这再一次印证了"时空平衡"的基本原理。其他模式对资源占用量与数据写入速度也都有不同程度的影响，希望可以为读者的实际应用提供参考。

另外，我们还留意到：znFAT的内存使用量可以最低降到819字节这个水平。即使是把CCCB用上，也只不过是867字节而已（仅仅多占用了几十个字节，但是数

据的写入速度却提升了 20%～30%，基本已达到全速的一半，这也充分说明了 CCCB 确实是一种巧妙而实用的策略)。这也许是一项突破，象征着 znFAT 可以应用到像 51、AVR、PIC 这种内存资源相对较少的 CPU 上，而且速度也不会有太多损失(FATFS 最少需要 1 300 字节左右的内存资源，虽然它有精简的 Tiny 版，但在功能和速度上有较大程度的裁减和损失)。

表 5.5　znFAT 在 51 平台上各种工作模式下的数据写入速度表现

内核	工作模式	RAM 用量/字节	数据量/字节	写入次数	用时/s	数据写入速度/(KB/s)	与全速的比率
8051 (22 MHz) 物理单扇区写速率 144 KB/s 全速模式下的数据写入速度 66 KB/s	实时更新文件大小与 FSINFO	1 386 (不计数据缓冲)	578	10 000	165	34.3	52%
	不使用 CCCB 缓冲机制	1 386 (不计数据缓冲)	578	10 000	110	51.4	78%
	不使用 EXB 缓冲机制	867 (不计数据缓冲)	578	10 000	182	31	47%
	实时模式＋不使用任何缓冲机制(全实时)	819 (不计数据缓冲)	578	10 000	285	19.8	30%

表 5.6　znFAT 在 Cortex－M3 平台上各种工作模式下的数据写入速度表现

内核	工作模式	RAM 用量/字节	数据量/B 字节	写入次数	用时/s	数据写入速度/(KB/s)	与全速的比率
Cortex－M3 (70 MHz) 物理单扇区写入速度 360 KB/s 全速模式下的数据写入速度 328 KB/s	实时更新文件大小与 FSINFO	1 774 (不计数据缓冲)	578	10 000	57	98.4	30%
	不使用 CCCB 缓冲机制	1 734 (不计数据缓冲)	578	10 000	31	180.4	55%
	不使用 EXB 缓冲机制	1 258 (不计数据缓冲)	578	10 000	48	118	36%
	实时模式＋不使用任何缓冲机制(全实时)	1 218 (不计数据缓冲)	578	10 000	123	45.9	14%

表 5.7 znFAT 在 AVR 平台上各种工作模式下的数据写入速度表现

内核	工作模式	RAM用量/字节	数据量/字节	写入次数	用时/s	数据写入速度/(KB/s)	与全速的比率
AVR (16 MHz) 物理单扇区写入速度 243 KB/s 全速模式下的数据写入速度 212 KB/s	实时更新文件大小与FSINFO	1 623 (不计数据缓冲)	578	10 000	57	81.2	33%
	不使用 CCCB 缓冲机制	1 595 (不计数据缓冲)	578	10 000	31	123.9	51%
	不使用 EXB 缓冲机制	1 051 (不计数据缓冲)	578	10 000	48	92.3	38%
	实时模式+不使用任何缓冲机制(全实时)	1 021 (不计数据缓冲)	578	10 000	123	38.8	16%

表 5.8 znFAT 在 ColdFile V2 平台上各种工作模式下的数据写入速度表现

内核	工作模式	RAM用量/字节	数据量/字节	写入次数	用时/s	数据写入速度/(KB/s)	与全速的比率
CF V2 (80 MHz) 物理单扇区写入速度 418 KB/s 全速模式下的数据写入速度 392 KB/s	实时更新文件大小与FSINFO	1 787 (不计数据缓冲)	578	10 000	40	141.1	36%
	不使用 CCCB 缓冲机制	1 747 (不计数据缓冲)	578	10 000	24	239.1	61%
	不使用 EXB 缓冲机制	1 230 (不计数据缓冲)	578	10 000	35	160.7	41%
	实时模式+不使用任何缓冲机制(全实时)	1 190 (不计数据缓冲)	578	10 000	90	62.7	16%

也许，有个问题一直萦绕在读者心中："为什么只比较数据写入速度，而对数据读取速度丝毫不提呢？难道数据读取速度就不重要吗？"不是，数据读取速度依然极为重要，但它并不是最主要的矛盾！扇区读操作比扇区写操作花费的时间要短得多(这主要是因为写操作会涉及存储设备内部介质的烧录或编程，而这一过程通常是比较长的)，这使得数据读取的速度并不会太多地牵制于物理操作。我们按照常规方法去实现，那么数据的读取速度也慢不到哪去。如果在数据读取上也使用诸如 CCCB 或

EXB 之类的缓冲机制，整体速度可能会有所上升，但并不会像数据写入那样明显，最多提高 5%～10%。我们引入一套复杂的机制或算法，但实际上却收效甚微，真是很不值得。振南可以告诉读者，使用 znFAT 中的 znFAT_ReadData 函数进行数据读取，其速度基本与 FATFS 持平。一般来说，数据读取速度是写入速度的 3 倍左右，希望这能给读者提供一个定性的参考。

5.4 znFAT 的功能裁减

到现在为止我们介绍的所有功能函数如下：znFAT_Init、znFAT_Open_File、znFAT_ReadData、znFAT_Create_File、znFAT_WriteData、znFAT_Create_Dir、znFAT_Close_File、znFAT_Flush_FS，但是在实际的开发过程中通常并不会用到所有函数，那没有用到的函数该如何处置呢？有人可能会说："不用管它，编译器会自动将没用到的函数剔除的。"确实，但是不能完全依赖编译器，因为不是所有编译器都那么友好而聪明的。比如 Keil 中的 C51 编译器：如果我们定义了一个函数，并且对其进行了实现，但实际上并没有对它进行调用，那么，它仍会占用很多的内存。反之，如果我们调用了它，内存占用量又会骤减。另一方面，就算这些闲置的代码不占用内存，它们也可能会占用 CPU 芯片的 ROM 资源。这似乎是一个棘手的问题，唯一的方法只有对这些没有用到的函数进行彻底剔除，让编译器根本不去编译它们。这如何实现？请看下文。

5.4.1 功能裁减宏

无一例外，我们还是要用条件编译的方法来进行实现。引入更多的宏，代码如下(config.h)：

```
#define ZNFAT_OPEN_FILE
//#define ZNFAT_READDATA     //如果将宏注释掉，则相应函数不参与编译
#define ZNFAT_CREATE_FILE
#define ZNFAT_CREATE_DIR
#define ZNFAT_WRITEDATA
#define ZNFAT_CLOSE_FILE
#define ZNFAT_FLUSH_FS
```

在功能函数的代码前后加上条件编译语句，具体实现如下(znFAT.c)：

```
#ifdef ZNFAT_READDATA
UINT32 znFAT_ReadData(struct FileInfo *pfi,UINT32 offset,
                                          UINT32 len,UINT8 *app_Buffer)
{
 //函数的实现代码
}
```

```
#endif
```

这样，我们只要把宏注释掉，相应的函数便不会再参与编译了。所以，振南称这些宏为“功能裁减宏”。

5.4.2 裁减宏的嵌套

功能裁减宏在实现上其实并不像上面所说的这么简单，还有更为深层的问题——“宏的嵌套”。我们在功能函数开头与结束加上了“#ifdef... #endif”语句，从而可以控制其是否参与编译。但是不要忘了，功能函数中还会去调用一些中间函数，而这些中间函数可能又会去调用更低一级的函数。这种调用关系就像是一棵树，枝叶蔓延，生生不息。如果我们要裁掉某一个功能函数，那么这就不单单是一个函数的问题，而会波及这棵“函数调用关系树”上的所有函数。拿znFAT_Open_File函数来举例说明，如图5.3所示。

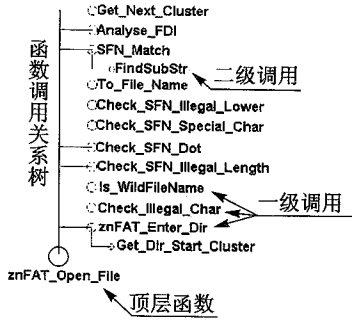

图5.3 znFAT_Open_File函数调用关系树示意图

我们首先给znFAT中的所有函数都加上条件编译宏控制语句，然后再根据调用关系编写下面的“嵌套宏”(ccmacro.h)：

```
#ifdef  ZNFAT_OPEN_FILE //如果此宏有定义，那么下面所有的宏定义均有效
#define ZNFAT_ENTER_DIR
#define GET_DIR_START_CLUSTER
#define CHECK_ILLEGAL_CHAR
#define IS_WILDFILENAME
#define CHECK_SFN_ILLEGAL_LENGTH
#define CHECK_SFN_DOT
#define CHECK_SFN_SPECIAL_CHAR
#define CHECK_SFN_ILLEGAL_LOWER
#define TO_FILE_NAME
#define SFN_MATCH
#define FINDSUBSTR
#define ANALYSE_FDI
#define GET_NEXT_CLUSTER
#endif
```

这样一来，只要功能函数对应的裁减宏(比如上例中的ZNFAT_OPEN_FILE)未被定义，那么此函数及其相关的中间函数就都不会参与编译了。不过，要理清每一个功能函数的“函数调用关系树”也不是一件轻松的事，可以借助于一些代码分析软件来帮我们完成这项工作，比如SVN等。

当然，znFAT 中的功能裁减机制主要是针对那些内存资源比较“贫瘠”、对代码体积比较敏感的硬件平台，振南希望通过它尽量降低 znFAT 对硬件资源的需求，从而为项目和产品节省成本。

好，通过上面的内容相信读者已经对 znFAT 的数据读/写效率、工作模式以及相关的宏配置方法有了一个更加明确的认识。对于本章中所出现的较量、对比和评价，振南绝无褒贬之意，只是想通过一些客观的数据说明事实。

感谢对振南及 znFAT 的关注与支持，希望振南在嵌入式 FAT32 文件系统方面的研究对您有所帮助！

更多内容请关注 振南电子网站

www.znmcu.cn