《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8章

突破短名，搞定长名：突破8·3短名限制，全面地实现长文件名

起初znFAT是不支持长文件名的，其主要原因是因为振南觉得它并没有多么重要，而认为文件名只是一个象征性的代号，文件的重点应该在于数据。但后来发现，很多人都对长文件名很感兴趣，一方面想知道FAT32的长文件名是如何实现的，另一方面也希望在自己的产品中加入对长文件名的支持。于是振南继续努力，最终实现了znFAT的长文件名功能。说实话，振南在本书的写作过程中，长文件名这一章的去留曾经是一个问题。因为FAT32中的长文件名微软是有专利的。所以，本章的内容在技术上将有所保留，并用一些被开源界承认的巧妙方法来对一些敏感的技术点进行避让。到底是如何做的？请看正文。

8.1 FAT32的长文件名

长文件名(LFN)是与短文件名(SFN)相对的，它允许文件名所包含的字符更多，表达的意义更为丰富，命名的方法也更为灵活。但是在FAT32中对长文件名的实现是比较繁琐的，同时也会牵扯到很多新的知识和技术。

8.1.1 何为长文件名

什么样的文件名才算是长文件名呢？有人说："这还不简单，比短文件名长的文件名就是长文件名呗。"没错，但不全面。长度超过8·3格式的文件名，必定是长文件名，但是长文件名却并不一定都长于8·3格式。举一个例子：

abcd.txt	短文件名	abcdefghi.txt	长文件名	abcD.TXT	长文件名
a[cd.txt	长文件名	你好再见.TXT	短文件名	abcd.Txt	长文件名

可以看到，有一些文件名长度符合8·3格式，但是也被认为是长文件名。这到底是为什么呢？长文件名具体而确切的定义到底是怎样的？请看下面这几条：

- 文件名长于8·3格式，即主文件名长于8字节或扩展名长于3字节如abcdefghi.txt；

- 文件名的主文件名或扩展名中含有大小写混排，如 abcD. TXT 或 abcd. Txt；
- 文件名中包含特殊字符“+[],;=”与空格（空格在末尾例外），如 a[cd. txt 或 a　b. txt；
- 文件名中包含两个或两个以上的“.”（“.”在末尾例外），如 a. b. c. txt。

这 4 条基本涵盖了长文件名的所有情况，归结起来就一句话：“一切不能由 8·3 格式短名来表达的合法文件名都是长名。”因此，我们不要被长文件名这个名称所迷惑，长文件名不一定“长”。

8.1.2　长文件名的存储机理

如同短文件名是由文件目录项来记录一样，长名也必然以某种形式存储于某个功能单元的一些字段之中。那它到底是存在哪里？具体又是如何存储的呢？为了回答这一问题，我们来做下面的实验。

首先将 SD 卡格式化，然后在首目录下创建一个长名文件，如图 8.1 所示。接下来使用 WinHex 软件来看一下这张 SD 卡的首目录扇区，如图 8.2 所示。

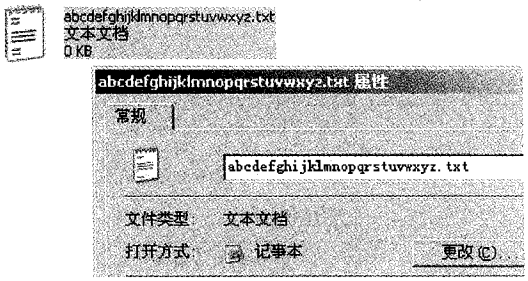

图 8.1　在 SD 卡首目录下创建长名文件

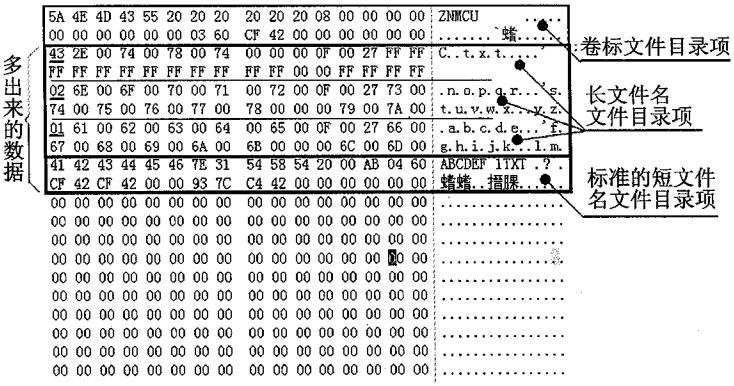

图 8.2　SD 卡首目录扇区中的数据

因为 SD 卡是刚被格式化的，所以首目录扇区中除了原本卷标文件目录项以外的数据应该全部是 0。图 8.2 中那些多出来的数据应该都是属于新创建的这个长名

文件的。从这些数据的内容上来看，确实好像与 abcdefghijklmnopqrstuvwxyz. txt 这个文件名有些关联。其实，这些数据就是长名文件的长文件名文件目录项及其相应的短文件名文件目录项。为什么一个文件会有这么多的文件目录项？其中怎么又涉及了短名文件目录项？这些问题的答案就将揭示长文件名实现的核心机理。

为了便于理解，振南先介绍一下长名文件目录项的具体定义。如图 8.2 所示，长名文件目录项同样也是 32 字节，但是它的字段和功能却有新的划分和定义，具体如表 8.1 所列。

表 8.1　长名文件的文件目录项结构定义

字节偏移	字节数	定　义		
0X00	1	属性字节位定义	7	保留未用
			6	1 表示长名文件最后一个文件目录项
			5	保留未用
			4	文件目录项序号
			3	
			2	
			1	
			0	
0X01～0X0A	10	长文件名 unicode 码 ①		
0X0B	1	长名文件目录项标志，取值为 0X0F		
0X0C	1	系统保留		
0X0D	1	与短名文件目录项的绑定检验值		
0X0E～0X19	12	长文件名 unicode 码 ②		
0X1A～0X1B	2	0X00		
0X1C～0X1F	4	长文件名 unicode ③		

我们发现其中有一项叫“文件目录项序号”，这说明长文件名的文件目录项不只一个，而是多个。还可以发现每一个长名文件目录项中都以 unicode 编码形式记录了 13 个字符(unicode 编码是双字节编码)，如表 8.1 中的①②③。长文件名就是由这多个文件目录项中记录的 unicode 码拼接而成的，这使它可以包含更多的字符。对图 8.2 中的长名文件目录项进行解析，如图 8.3 所示。

图 8.3 已经很清楚地阐明了长名的存储方式。但是在这个过程中有一个概念是我们比较陌生的，那就是“unicode 编码”。FAT32 中的长文件名采用 unicode 编码来记录，这一点是与短文件名的主要区别之一。有人说：“FAT32 在长文件名中引入了 unicode 编码，是一个突破性的进步和创举！”振南觉得这话并不为过，为什么？我们还是先来详细介绍一下 unicode 编码吧。

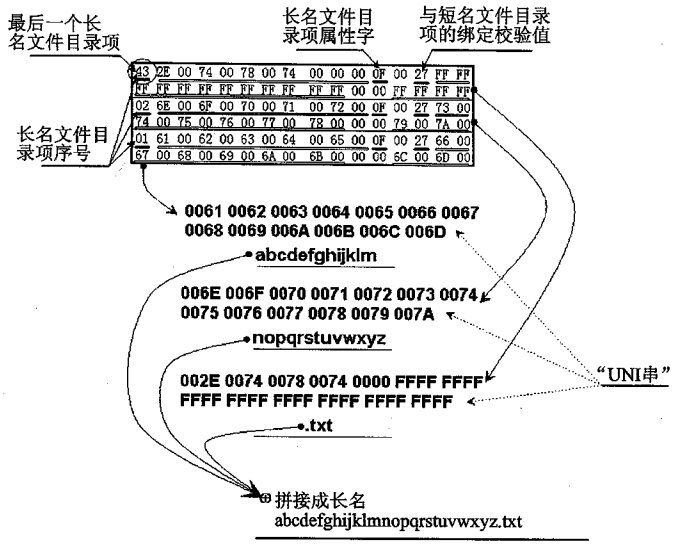

图 8.3　对长名文件目录项的解析与长名的拼接

8.2　UNICODE 编码

振南称 UNICODE 编码为“全世界文字符号大团结”，是现在唯一一个最通用、涵盖文字符号最广、最全面的编码方案（据说包含了迄今为止人类使用的所有文字，并且具有极强的可扩展性，比如哪天火星人入驻地球，可以很方便地把火星文纳入其中）。可以说，FAT32 的长名中使用了 unicode 编码，象征着 FAT32 向通用化、标准化迈出了重要的一步。

8.2.1　“各自为战”的 DBCS

计算机里表示字符通常使用 ASCII 编码，它定义每个字符占用一个字节，因此在程序里通常把字符类型变量定义为 char。对于西方国家，它们的文字均由字母和有限的一些符号组成，因此 ASCII 编码对他们已经够用了。但是，对于亚洲、非洲乃至所有使用象形文字的地区来说，这是远远不够用的。那怎么办？最直接的办法就是增加用于表达文字编码的字节长度，因此双字节字符集 DBCS（Double Byte Character Set）应运而生，后来还产生了 MBCS，即多字节字符集，这里主要还是针对前者来进行讲解。

起初，关于 DBCS 编码并没有一个标准，通常都是由某个语言区域自行定义的。其实这些各自定义的编码体系都是一些映射表，确定了当地使用的每个文字与其双字节编码之间的对应关系。而且这些映射表通过固化到相应版本的操作系统中，从而实现对此区域文字的兼容。被本地化之后的文字编码通常称为 OEM 编码（OEM

一般是指通过生产商专门订制过的产品)。正是因为有了各种各样的 OEM 编码,才使得操作系统依语言不同产生了不同的版本,如 Windows 就有简/繁体中文版、韩文版、俄文版等多个版本。在中国大陆普遍使用 GB2312、GBK 等编码体系(平时编程的时候可能会定义一些含有中文的字符串,其实它就是用 GB2312 来进行编码的)。其实我们对它们并不陌生,还记得当年填写报名单或是图像采集单上的姓名时的情形吗(如图 8.4 所示)?不光要写上汉字,还要为每个字配上一个区位码,这个区位码其实就是由 GB2312 编码通过简单计算而得到的,为的是能够方便、快速地把我们的名字录入到计算机中。

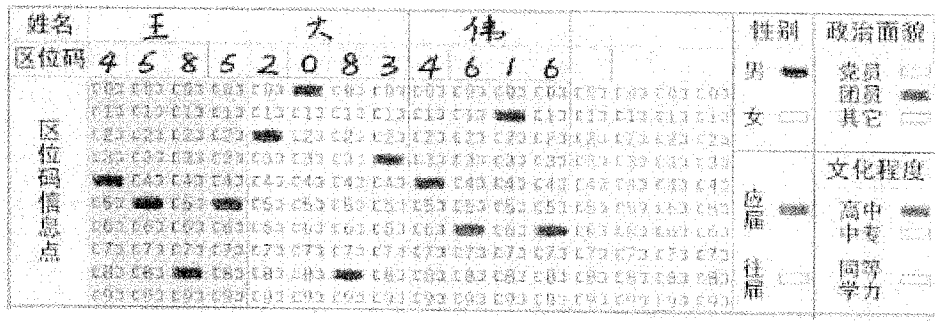

图 8.4　某报名单上对姓名及其区位码的填写

图 8.4 中“王”字的区位码是 4585,前两位为区码,后两位为位码。把 45 和 85 转为十六进制分别为 2D 和 55,然后加上 A0,就得到了 CDF5,它就是汉字“王”的 GB2312 编码(其实这个计算过程在上册的汉字电子书实验中就碰到过),可以到编码表中找到其所在,如图 8.5 所示。

code	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
CCA0		獭	挞	踢	踏	胎	苔	抬	台	泰	酞	太	态	汰	坍	摊
CCB0	贪	瘫	滩	摊	坛	檀	痰	潭	谭	谈	坦	毯	袒	碳	探	叹
CCC0	汤	塘	搪	堂	棠	膛	唐	糖	倘	躺	淌	趟	烫	掏	涛	滔
CCD0	绦	萄	桃	逃	淘	陶	讨	套	特	藤	腾	疼	誊	梯	剔	踢
CCE0	锑	提	题	蹄	啼	体	替	嚏	惕	涕	剃	屉	天	添	填	田
CCF0	甜	恬	舔	腆	挑	条	迢	眺	跳	贴	铁	帖	厅	听	烃	

code	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
CDA0		汀	廷	停	亭	庭	挺	艇	通	桐	酮	瞳	同	铜	彤	童
CDB0	桶	捅	筒	统	痛	偷	投	头	透	凸	秃	突	图	徒	途	涂
CDC0	屠	土	吐	兔	湍	团	推	颓	腿	蜕	褪	退	吞	屯	臀	拖
CDD0	托	脱	鸵	陀	驮	驼	椭	妥	拓	唾	挖	哇	蛙	洼	娃	瓦
CDE0	袜	歪	外	豌	弯	湾	玩	顽	丸	烷	完	碗	挽	晚	皖	惋
CDF0	宛	婉	万	腕	汪	王	亡	枉	网	往	旺	望	忘	妄	威	

code	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
CEA0		巍	微	危	韦	违	桅	围	唯	惟	为	潍	维	苇	萎	委
CEB0	伟	伪	尾	纬	未	蔚	味	畏	胃	喂	魏	位	渭	谓	尉	慰
CEC0	卫	瘟	温	蚊	文	闻	纹	吻	稳	紊	问	嗡	翁	瓮	挝	蜗
CED0	涡	窝	我	斡	卧	握	沃	巫	呜	钨	乌	污	诬	屋	无	芜
CEE0	梧	吾	吴	毋	武	五	捂	午	舞	伍	侮	坞	戊	雾	晤	物
CEF0	勿	务	悟	误	昔	熙	析	西	硒	矽	晰	嘻	吸	锡	牺	

图 8.5　根据 GB2312 编码找到编码表中对应的汉字

一些较为生僻的字查不到相应的区位码,可见 GB2312 的编码表中并非涵盖了

所有的中文字符。因此，后来出现了 GBK 编码，是对 GB2312 的扩展，包含的字符数量从 7 000 多个增加到了 20 000 多个或者更多。

当然，GB2312 和 GBK 都是仅限在中国大陆使用的。如果把一份在中国大陆编写的文档发送到其他国家，那可能看到的就是一堆乱码，这就是因为它们使用的是另一套编码体系，比如日本的 JIS、韩国的 ks_c_5601 等。

8.2.2　UNICODE 带来的问题

各种编码体系“各自为战”的局面促使人们发明了 UNICODE，它集所有文字于一身，通过它可以实现文字编码的全球统一。但是这一过程必然是任重而道远的，因为现有的编码体系不可能短时间内被废除。所以，UNICODE 与现有的各编码体系之间的相互转换常常成为我们应用中的一道难题，这里讲的 FAT32 的长文件名就是一个最典型的例子。

前面解析长名文件目录项时最终拼接得到了一个双字节数据序列，其实它就是长文件名的 UNICODE 编码串，振南称之为 UNI 串。UNICODE 是对现有 ASCII 编码的扩展，对于原 ASCII 字符编码只是扩展为了双字节，即高字节补 0。想一想，如果要用一个长文件名来打开文件，比如“abcdefghijklnmopqrstuvwxyz. txt”，该如何实现呢？首先要从长名文件目录项中提取并拼接得到长名的 UNI 串，然后再将它与长文件名匹配。但是我们在程序中所写的长文件名字符串其实并不是由 UNICODE 来编码的，因此这就将涉及字符编码转换的问题，如图 8.6 所示。

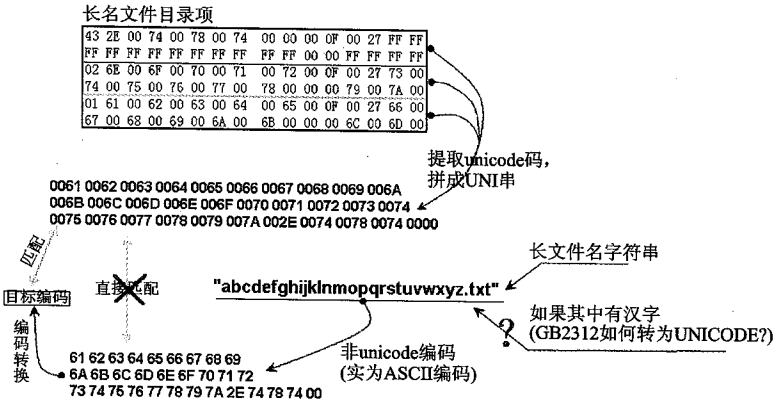

图 8.6　FAT32 的长名匹配产生的编码转换问题

有人说：“把长名字符串中的每个字符编码补了‘00’就可以。”对于 ASCII 字符确实如此，但如果长文件名中有汉字该如何转呢？比如当年财务报表. txt 或设备图纸 final. doc 等。这些字符串中的汉字是以 GB2312 的编码方式来存储的，如何将它们转为 unicode 编码的 UNI 串，进而进行文件名的匹配呢？有人会想：“这两种编码体系之间可能有一种换算关系，可以相互转换。”但它们确实毫无关系，这也是振南在

实现 znFAT 的长文件名功能之初所遇到的第一个难题。最终的解决方案是：建立一张 UNICODE - GB2312 的映射表，通过查表来完成转换。振南通过专门的工具生成了这张映射表，如图 8.7 所示。

图 8.7 是映射表的 C 语言二维数组格式的具体实现，每个单元的第 2 个数为 GB2312 码值，第 1 个数就是其对应的 UNICODE 码值。它可以完成 6 768 个汉字的转换，大多数情况下是够用的了。（这个映射表通常以数组的形式固化在 ROM 中，因此要使用长文件名功能，所使用的 CPU 芯片一定是要有足够的 ROM 资源才行。当然，也可以把它放在外部的存储器上，比如 FlashROM 或 EEPROM 上，不过这就需要开发者自行实现数据访问接口及查表操作了。）

```
#define MAX_UNI_INDEX  6768

ROM_TYPE_UINT16 oem_uni[MAX_UNI_INDEX][2]= {
{0x554A,0xB0A1},//GB2312:啊
{0x963F,0xB0A2},//GB2312:阿
{0x57C3,0xB0A3},//GB2312:埃
{0x6328,0xB0A4},//GB2312:挨
{0x54CE,0xB0A5},//GB2312:哎
{0x5509,0xB0A6},//GB2312:唉
{0x54C0,0xB0A7},//GB2312:哀
{0x7691,0xB0A8},//GB2312:皑
{0x764C,0xB0A9},//GB2312:癌
{0x853C,0xB0AA},//GB2312:蔼
{0x77EE,0xB0AB},//GB2312:矮
{0x827E,0xB0AC},//GB2312:艾
{0x788D,0xB0AD},//GB2312:碍
{0x7231,0xB0AE},//GB2312:爱
        ......
{0x9EE2,0xF7F1},//GB2312:黢
{0x9EE9,0xF7F2},//GB2312:黩
{0x9EE7,0xF7F3},//GB2312:黧
{0x9EE5,0xF7F4},//GB2312:黥
{0x9EEA,0xF7F5},//GB2312:黪
{0x9EEF,0xF7F6},//GB2312:黯
{0x9F22,0xF7F7},//GB2312:鼢
{0x9F2C,0xF7F8},//GB2312:鼬
{0x9F2F,0xF7F9},//GB2312:鼯
{0x9F39,0xF7FA},//GB2312:鼹
{0x9F37,0xF7FB},//GB2312:鼷
{0x9F3D,0xF7FC},//GB2312:鼽
{0x9F3E,0xF7FD},//GB2312:鼾
{0x9F44,0xF7FE} //GB2312:齄
};
```

图 8.7　GB2312 编码到 UNICODE 编码的映射表

8.2.3　编码转换的实现

有了 GB2312 到 UNICODE 的转换表，接下来要做的就是一个查表的工作了。可以顺序遍历查找，代码如下：

```
UINT8 OEM2UNI(UINT16 oem_code,UINT16 * uni_code)
                        //本地字符编码(汉字就是 GB2312)转换为 UNICODE 编码
{
 UINT32 i = 0;
 for(i = 0;i<MAX_UNI_INDEX;i ++ ) //遍历编码转换表中的所有单元
 {
  if(oem_code == oem_uni[i][1]) //如果 GB2312 编码一致
  {
   * uni_code = oem_uni[i][0]; //取其对应的 UNICODE 编码
   return 0; //返回成功
  }
 }
 return 1; //返回失败
}
```

很显然，这种方法的查询效率不高。这里介绍一种简单、高效的查询算法——“二分搜索”。其实它是一种很常用的算法，用来在一个具有线性、有序的数据序列中对某一元素进行快速查找。我们先抛开字符编码转换的问题，专门来说说“二分搜索”算法。举一个例子：在递增序列{2,5,8,10,16,19,23,28,34,56,78,88,89,101,123,134,165,178,199,201}中如何查找某一元素的位置？使用“二分搜索”来解决这一问题是最恰当不过的，请看以下程序：

```
unsigned char BinarySearch(unsigned char * pdat,unsigned char n,unsigned char m)
                                    //pdat 指向序列，n 是序列元素总个数，m 是要查找的目标元素
{
 unsigned char low = 0,high = n - 1,mid;//设定查找区间上下限的初值
 while(low<= high) //当前查找区间[low..high]非空
 {
  mid = low + (high - low)/2;
  if(m == pdat[mid])
  {
   return mid; //查找成功返回
  }
  if(pdat[mid]>m)
  {
   high = mid - 1;  //继续在[low..mid - 1]中查找
  }
  else
  {
   low = mid + 1; //继续在[mid + 1..high]中查找
  }
 }
 return -1; //当 low>high 时表示查找区间为空，查找失败
}
```

实际验证一下：查找89在序列中的位置，如图8.8所示。

验证结果很显然是正确的。那二分搜索算法的原理和实现过程到底是怎么样的呢？其实非常简单，可以通过图8.9来说明。图中描述的实现过程其实就是一个不断“折叠”的过程，折叠之后判断选择哪一半，然后再继续折叠，直到找到目标元素为止。所以，二分搜索又形象地称为“折半查找”。它的查找速度是呈现指数级规律的，因此执行效率比较高。

GB2312到UNICODE的转换就可以使用这种算法。不过，我们要注意到二分搜索应用的前提是数据序列必须是线性有序的，也就是要么递增，要么递减，而不能是杂乱无章的。仔细观察图8.7中的字符编码映射表可以发现，表中GB2312编码其实已经是以递增的方式进行排序了（其实这个映射表是振南为了使用二分搜索算

```
void main()
{
 unsigned char array[20]=
          {2,5,8,10,16,19,23,28,34,56,78,88,89,101,123,134,165,178,199,201};
 unsigned char i=0;

 printf("index:%d\n",BinarySearch(array,20,89));
 getchar();
}
```

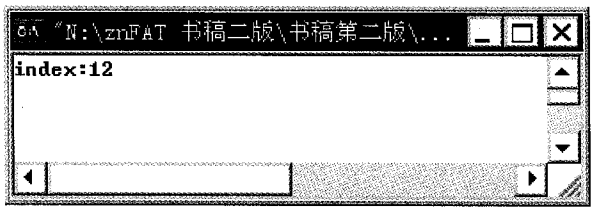

图 8.8　对二分搜索算法的验证

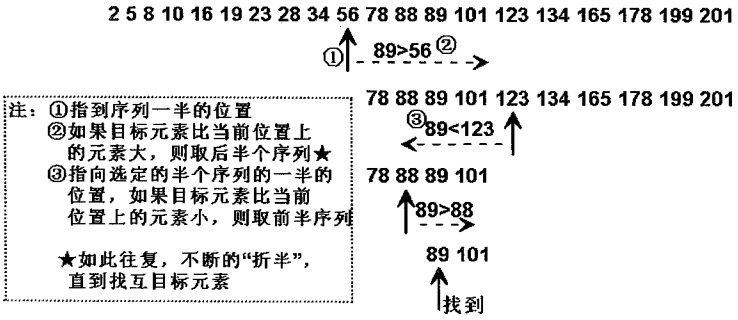

图 8.9　二分搜索算法的具体实现过程

法而专门设计的)。接下来就来对函数 OEM2UNI 进行实现,代码如下(znFAT.c):

```
UINT8 OEM2UNI(UINT16 oem_code,UINT16 * uni_code)
                                    //获取 OEM 字符编码(汉字即为 GB2312)所对应的 unicode 编码
{
 UINT32 low = 0,high = MAX_UNI_INDEX - 1,mid; //设定查找区间上下限的初值
 if(oem_code<oem_uni[0][1]) return 1;
 if(oem_code>oem_uni[MAX_UNI_INDEX - 1][1]) return 1;
                                    //如果输入的 oem_code 不是表中,则直接返回
 while(low<=high) //当前查找区间[low..high]非空
 {
  mid = low + (high - low)/2;
  if(oem_code == oem_uni[mid][1]) //如果找到目标 OEM 编码(GB2312)
  {
   * uni_code = oem_uni[mid][0]; //将其对应的 unicode 编码赋给 uni_code 指向的变量
   return 0; //查找成功
  }
  if(oem_uni[mid][1]>oem_code)
```

```
    {
     high = mid - 1; //继续在[low..mid - 1]中查找
    }
    else
    {
     low = mid + 1; //继续在[mid + 1..high]中查找
    }
   }
   return 1; //当 low>high 时表示查找区间为空，查找失败
  }
```

这个函数就可以完成单个汉字的GB2312编码向unicode编码的转换。有了它，我们进一步实现长名字符串向UNI串的转换就变得简单了。请看下面这个函数(znFAT.c)：

```
UINT8 oemstr2unistr(INT8 * oemstr,UINT16 * unistr)
{
 UINT32 len = StringLen(oemstr),i = 0,pos = 0;
 UINT8 res = 0;
 for(i = 0;i<len;i++)
 {
  if(IS_ASC(oemstr[i])) //检查是否是 ASCII 码(ASCII 码值范围 0X00～0X7F)
  {
   unistr[pos] = (UINT16)oemstr[i]; //ASCII 字符编码直接扩展为双字节，即高字节补 0
   pos++;
  }
  else //不是 ASCII 码(OEM 编码，其值通常大于 0X80，对于汉字来说就是 GB2312)
  {
   res = OEM2UNI((((UINT16)oemstr[i])<<8)|((UINT16)oemstr[i+1]),unistr + pos);
                                                    //将 OEM 编码转为 unicode 编码
   if(res) //编码转换出现错误
   {
    unistr[0] = 0;
    return 1; //返回错误
   }
   pos++;i++;
  }
  if(pos>MAX_LFN_LEN) //如果超过了 znFAT 中定义的长名缓冲区最大长度
  {
   unistr[0] = 0;
   return 1;  //返回错误
  }
 }
```

```
 unistr[pos] = 0;
 return 0;
}
```

这个函数中对 ASCII 与 OEM 编码分别进行了处理，这主要是考虑到长文件名有可能出现中英文混排的情况，比如“重要文档 abc. txt”。

8.2.4 长名的提取与匹配

oemstr2unistr 函数实现了长名字符串向 UNI 串的转换，再将长名文件目录项中的 UNI 串提取出来即可对它们进行匹配。下面就来编程实现长名的提取。根据表 8.1 中对长名文件目录项结构的定义，有下面这个结构体(znFAT.c)：

```
struct LFN_FDI      //长名文件目录项结构定义
{
 UINT8 AttrByte[1];//属性字节，序号
 UINT8 Name1[10];  //第一部分长名(unicode 编码)
 UINT8 LFNSign[1]; //长名项标志
 UINT8 Resv[1];    //保留
 UINT8 ChkVal[1];  //检验值，与 SFN 的绑定校验
 UINT8 Name2[12];  //第二部分长名
 UINT8 StartClu[2];//取 0
 UINT8 Name3[4];   //第三部分长名
};
```

长名 UNI 子串提取函数的具体定义如下：

```
UINT8 Get_Part_Name(UINT16 * lfn_buf,struct LFN_FDI * plfndi,UINT8 n)
```

形参中的 lfn_buf 是指向用于存储长名 UNI 串的缓冲区的指针；plfndi 是指向长名文件目录项的指针；n 是当前长名文件目录项中的 UNI 子串要放到 lfn_buf 中的起始偏移量(函数的详细代码参见 znFAT 源代码)。此函数的详细描述如图 8.10 所示。

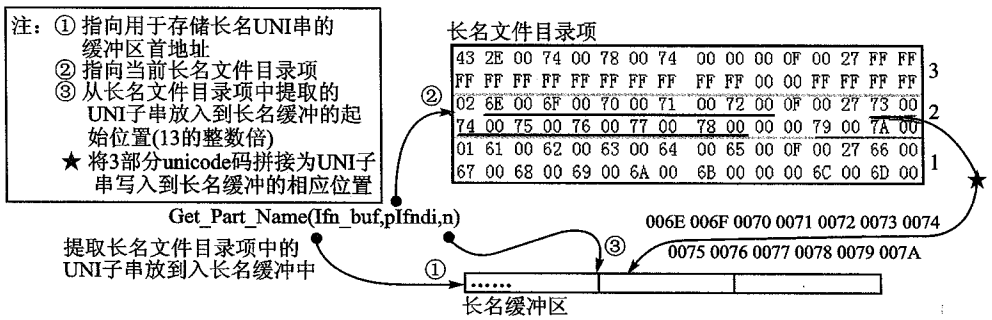

图 8.10 Get_Part_Name 函数的功能示意图