

AN02003

使用 MC998/MC2002 数据库功能时应注意的问题

作者：技术支持部 胡永健

一 介绍：

本文主要介绍如何在 MC998 及 MC2002 手持机中编程实现数据库功能以及在编程中的注意事项。

MC998 与 MC2002 都拥有 1Mbyte 甚至更多的 FLASH ROM，在两者的 FLASH ROM 中除去 BIOS、字库及用户程序占用部分空间外，还有大量空间可供用户存储数据。在早期 MC998 及 MC2002 API 中只提供了两个直接操作 FLASH ROM 的底层函数，使用这种方法进行用户数据的存储及读取操作需要编程人员对 FLASH ROM 特性有足够的了解，并且在操作时需要自己管理数据空间的分配，使得编写应用程序难度大且易出错。现在，MC998 及 MC2002 API 中已加入了数据库管理功能，用户不必再像以前一样进行繁琐的地址计算以及考虑由于 FLASH 特性所引起的缺陷。通过这些数据库 API 用户可以方便地管理数据，如创建或删除数据库、添加或删除记录、查找记录等等。

本文先介绍 FLASH ROM 特性，然后分别就 MC998 及 MC2002 中两种不同的数据管理方法进行分析，并提供部分实例程序供大家参考。

二 FLASH ROM 特性

用直接读写 FLASH 的方法管理数据需要编程人员对 FLASH ROM 特性有一定的了解；即使使用数据库 API 管理，了解 FLASH ROM 特性对理解 API 及自己编写程序都会有很大的帮助。下面就介绍 FLASH ROM 特性及 MC998 与 MC2002 中 FLASH ROM 的空间分配。

1 FLASH ROM 特性

FLASH ROM 是一种新型的电可擦除存储器(EEPROM)器件，具有擦除速度快、可重复擦写次数多等优点，但也有其缺点：只能成块(在此为 64K 字节/块)擦除 FLASH ROM 中数据，而不能单独对某一字节进行擦除(在 FLASH ROM 中擦除是将所有位设置为 1)。擦除后的 FLASH

ROM 可按字写入，但写入只能将某一位保持原状态或将其从 1 写为 0，而不能从 0 改写为 1。若想在同一区域重新写入新数据，数据中哪怕是一个字节的一个位需要从 0 改写为 1，都必须将整块擦除后才能正确写入。FLASH ROM 特性决定了它适合存储一些不经常改变的数据。

2 MC998 及 MC2002 中使用 FLASH 大小及空间分配

MC998 有大小为 128K~512K Byte 的 SRAM 以及 1M~4M Byte 的 FLASH ROM;而 MC2002 中则有大小为 8MByte 的 DRAM 以及 1M~8MByte 的 FLASHROM。其中 FLASHROM 中，标准型 BIOS 及中文简体 GB2312 字库占去大约 5 个 FLASH 块 (64KX5=320K Byte)，剩下的为应用程序及数据库所用。另外，应用程序中不能直接访问 BIOS 和字库区域。有关 FLASH 地址分配请参阅《开发者手册》4.9 章节。

用户程序始于 0x0050000，占用块数=(用户程序 BIN 文件字节数+16 字节 BIOS 添加信息)/FLASH 块尺寸(本例中：64K 字节)，不足一块的也占用一块(例如：某 38K 程序即占用 0x0050000-0x005FFFF 计一块；某 70K 程序则占用 0x0050000-0x006FFFF 计两块)。数据库使用时占用从用户程序后第一个可用 FLASH 块开始直至所有剩余可用 FLASH 空间(上例中数据库分别开始于 0x0060000 及 0x0070000 地址)。用户不使用数据库 API 而直接读写 FLASH 时也应使用该段空间。

特别指明的是第一代 MC998 手持机 1M 字节型产品(仅限于此型号)，其 FLASH 最后一个 64K 字节块由于 FLASH 硬件设计的原因被分为了 8 个 8K 字节小块，每一小块只能单独擦除。由于此特性，该 8 个 8K 字节小块不能归入数据库统一管理(当然仍可由用户采用直接读写 FLASH 的方式使用)。

了解了 MC998 与 MC2002 中 FLASH 特性及空间分配后就可以选择合适的方法编写程序。

三 用直接读写 FLASH 的方法管理数据

直接读写 FLASH 来管理数据是较为底层的实现方法，对编程人员要求相对较高，但更为灵活。对于要实现一般数据库管理功能的应用可使用数据库 API 来实现，也可跳过此节直接阅读下一节的内容。如果使用，请先仔细阅读《API 手册》2.13 章节。

在本文中所列出的 API 及实例程序，若无特别说明都适用于 MC998 及 MC2002。

1 相关 API

(1) short FLASH_erase_block(void *blockaddress);

功用：将 FLASH 中地址为 blockaddress 的一块(64K)擦除。注意：只要给出的地址位于某一块地址空间内，则该块将整个被擦除。

(2) short FLASH_write_record(FLASH_wr_param *pfpw);

功用：将用户数据写入 FLASH 中指定址开始的空间中。注意：本机中 FLASH 使用的是 16 位设备方式，故写入是按字(WORD=16 位)进行的，参数结构中长度也是指以字为单位的长度。

注：更详细的内容请参阅相关资料。

2 编程步骤

为了便于理解，在这里用一工程实例来介绍如何使用上述 API 实现数据管理功能。

工程需求：利用 MC998 或 MC2002 手持实现公交车乘车计费系统。要求手持机能读取乘客 IC 卡信息并存入 FLASH ROM 中，在存储多笔乘客信息后可将所有记录上

传给 PC 机进行处理。

分析：依据需求，编程人员必要要做到数据的写入、读取以及擦除操作。
在实现这些功能之前我们先定义一些全局变量。

```
extern char FLASH_DB_START;
#define StartAddr (&FLASH_DB_START) //数据库起始地址
#define INFLLEN 42 //用户数据长度。typ_user_inf 类型长度
long recno; //当前记录数

typedef union{
    struct inf{
        unsigned char user_id[8];
        unsigned char user_name[8];
        unsigned char company[8];
        unsigned short n;
        unsigned char exact_time[16];
    }inf;
    char inf_buf[INFLLEN];
}typ_user_inf;
typ_user_inf *userinf;
// typ_user_inf 为共用体，记录了乘客信息，如：卡号、姓名、公司、乘车次数以及乘车时等。
//Userinf 为指向 typ_user_inf 共用体的指针。
```

注：本文所举实例具有一定的代表性，有利于理解此中编程方法，但不可生搬硬套。

(1) 存储数据

手持机读取 IC 卡数据后便要存入 FLASH ROM 中，使用 FLASH_write_record 可以实现。其原形为：

```
short FLASH_write_record(FLASH_wr_param *pfpw);
```

参数为 FLASH_wr_param 指针类型。FLASH_wr_param 为 API 中定义结构体类型：

```
typedef{
    void *ptr_buffer;
    void *ptr_FLASH_addr;
    unsigned long data_length;
}Flash_wr_param;
```

其中:ptr_buffer 是指向用户数据的指针

ptr_FLASH_addr 指向 FLASH 中的某一存储区域，在此存储由 ptr_buffer 指向的用户数据。其实，这里的指针即为存储器地址。

Data_length 为需要写入 FLASH 中的数据长度。这里的长度是以字 (WORD) 为单位。

数据信息是由读卡 API 从乘客 IC 卡中读入，此部分内容已超出本文所讨论的范围。在这里我们假定数据以读入全局数组 tempbuff 中，就可通过编写一子程序 stored_into_db 实现数据存储功能。

```
short stored_into_db(void){
    FLASH_wr_param *pfpw,param;//定义 FLASH_wr_param 类型指针(pfpw)和变量(param);
```

```

pftp = &param;                // 使 pftp 指向 param.
userinf = (typ_user_inf *)tempbuff;

pftp->ptr_buffer = userinf->inf_buf; //参数设定
pftp->ptr_FLASH_addr = StartAddr+recno*INFLLEN;
pftp->data_length = INFLLEN/2;
if((FLASH_write_record(pftp))==0x80){ // FLASH_write_record 成功返回 0x80
    recno++;
    return 1;
}
else
    return 0;
}

```

(2) 读取数据

当上传或查询手持机内数据时都需要读取数据操作。方法比较简单，就是将 FLASH 中相应地址的数据取出，难点在于地址的计算。在这里我们完成简单的上传功能，将手持机中所有记录逐条上传，并在每笔记录后加 "\r" 与 "\n" 字符。

```

void UART_send(char *lb, short len){ //将 lb 所指向数据中 len 个字符通过串口发送出去
    short i;
    for(i=0;i<len;i++){
        while(UART_send_char(*lb));
        lb++;
    }
}

```

```

void upload_data(void) { //上传数据子程序
    userinf = StartAddr;
    while((userinf->inf.user_id[1]!=0xff)&&(userinf->inf.user_name[1]!=0xff))
    {
        //判断次区域是否有数据，若无则上传完毕。注：判断条件可根据实际情况进行修改。
        UART_send(userinf->inf_buf, INFLLEN);
        while(UART_send_char('\r'));
        while(UART_send_char('\n'));
        userinf = (long)userinf+INFLLEN;
        recno--;
    }
}

```

(3) 擦除所有数据

当数据块满或其他原因（如已上传完成）需要擦除数据时应用程序需执行下面的操作：
FLASH_erase_block(void *blockaddress);

其中参数 blockaddress 为将要擦除的数据块的起始地址，在此实例中只用到一块，可将 blockaddress 设为 StartAddr。如占用多于一块则应逐块分别擦除。

3 注意事项

前面通过一实例简要介绍了直接读写 FLASHROM 的方式进行数据存储、读取以及擦除，用户可以灵活运用上述两个 API 实现更为复杂的逻辑功能，但有以下几点需要提醒用户注意：(1)在编写程序前一定要清楚 FLASH ROM 的特性，要知道它与 RAM 不同，在擦除一整块(64K)后，写入任何数据时都只能将某一数据位保持或从 1 写为 0，不能从 0 写为 1，数据位从 0 变为 1 只能通过整块擦除。

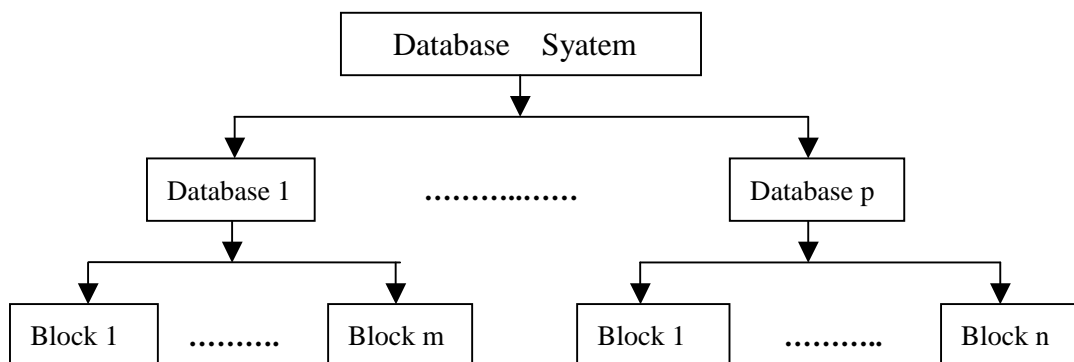
(2)要对指针有一定的了解，理解 FLASH ROM 地址即是指针，在程序中灵活运用指针。另外，要精确计算地址。在程序中引用错误的指针会出现意想不到的后果。

(3)要注意数据库起始地址。在本例中使用了数据库 API 库(database.h)中已声明的外部变量 FLASH_DB_START 的地址作为用户数据库起始地址。在系统中此变量的地址即为用户程序后第一个可用 FLASH 块起始地址。用户也可自己计算数据库起始的地址，如 0x80000。建议使用前一种方法。

(4) 在使用 FLASH_write_record 时要注意其参数传递。此 API 函数原形中参数为 FLASH_wr_param 类型的指针，我们在应用中必须要同时定义 FLASH_wr_param 类型的变量(param)和指针(pfw)，使指针指向 FLASH_wr_param 类型的变量，在“填充”好变量成员后，用指针作为 FLASH_wr_param 的参数。另外，FLASH_wr_param 类型成员 data_length 为写入数据长度，它以字(WORD)为单位。

四 运用数据库 API 管理数据

对于实现一般数据库管理功能的应用，建议使用数据库 API 来实现。首先让我们了解一下此数据库系统的组织形式：



由此结构可以看出 Database System 可以管理一个或多个 Database，每个 Database 又可管理一个或多个 Block。在这里要说明的是每个 Database 只可存储同一类型的记录，如果应用中需要存储其它类型记录，则需要在 Database System 中建立新的 Database；Block 为 FLASH 中的块(64K/块)。

1 相关 API 及全局变量

针对上述数据库组织结构，可以简单的将数据库 API 分为 Database System 级与 Database 级。

(1) Database System 级 API 及全局变量

```
void get_db_sys_param(typ_db_sys_param *pdsp);
```

功用：通过 typ_db_sys_param 类型参数返回当前 Database System 信息。

```
short DB_check_db(typ_db_description *fs);
```

功用：返回当前 Database System 中可用的 Database 数。

```
void erase_db_sys(void);
```

功用：低级格式化整个 Database System，相当与擦除 Database System 占用的所有 Block 内容。

```
typ_db_sys_param dsp;
```

功用：全局变量。记录 Database System 信息，如：块数、起始地址以及块大小。

```
typ_db_description filesys[MAX_FS_NUMBER];
```

功用：全局变量。记录 Database System 中每个 Database 的信息，如 Database 标识、所含 Block 数以及起始地址（按 API 手册规范编写数据库程序后，BIOS 系统会自动确定数据库在 FLASH 中的起始地址，用户不需要直接管理地址问题了）。

```
short file_sys_cnt;
```

功用：全局变量。记录 Database System 中 Database 的个数。

```
unsigned long current_rec_num[MAX_FS_NUMBER];
```

功用：全局变量。记录 Database System 中每个 Database 当前所使用的记录号。

(2) Database 级 API

```
short DB_format_db(unsigned short fsid, unsigned char  
blocknum, unsigned short recsize);
```

功用：格式化一新的 Database，其中 Database 标识为 fsid，块数为 blocknum，记录数据类型长度为 recsize。

```
short DB_erase_filesys(unsigned short fs_num);
```

功用：根据 Database System 中记录的 Database 号 (fs_num) 擦除此 Database。注意：这里的 fs_num 与 fsid 不同。

```
unsigned long DB_capacity(unsigned short fs_num);
```

功用：根据 Database System 中记录的 Database 号 (fs_num) 返回此 Database 所能容纳记录的数量。

```
unsigned long DB_count_records(unsigned short fs_num);
```

功用：根据 Database System 中记录的 Database 号 (fs_num) 返回此 Database 中已加入记录数（包括已被删除记录）。

```
Void * DB_jump_to_record(unsigned short fs_num, unsigned long  
rec_num, char *flag);
```

功用：根据参数 fs_num、rec_num 和 flag 返回指定的记录指针。

```
unsigned long DB_add_record(unsigned short fs_num, void *pRec);
```

功用：将 pRec 所指向的记录加入由 fs_num 所指定的 Database 中。

```
void DB_delete_record(unsigned short fs_num, unsigned long r_num);
```

功用：将 fs_num 所指定的 Database 中记录号为 r_num 的记录删除。

```
short DB_find_1st_match(unsigned short fs_num, short (*
```

```
match_routine) (void *rec_buffer);
```

功用：找出由 fs_num 所指定的 Database 中符合条件的第一条记录号，并将记录号存入全局变量 current_rec_num[fs_num]中。

```
short DB_find_next_match(unsigned short fs_num,short (*match_routine) (void *rec_buffer);
```

功用：找出由 fs_num 所指定的 Database 中符合条件的下一条记录号，并将记录号存入全局变量 current_rec_num[fs_num]中。

```
short DB_find_pre_match(unsigned short fs_num,short (*match_routine) (void *rec_buffer);
```

功用：找出由 fs_num 所指定的 Database 中符合条件的前一条记录号，并将记录号存入全局变量 current_rec_num[fs_num]中。

注：这里只是将数据库 API 进行简单罗列，更详细的内容请参阅相关资料。

2 编程步骤

为了便于理解，我们仍然使用上一节所列举的实例进行讲解。在这里我们同样需要定义应用程序中的全局变量：

```
#define INFLEN 42 //用户数据长度。typ_user_inf 类型长度
long recno; //当前记录号
typedef union{
    struct inf{
        unsigned char user_id[8];
        unsigned char user_name[8];
        unsigned char company[8];
        unsigned short n;
        unsigned char exact_time[16];
    }inf;
    char inf_buf[INFLEN];
}typ_user_inf;
typ_user_inf *userinf;
// typ_user_inf 为共用体，记录了乘客信息，如：卡号、姓名、公司、乘车次数以及乘车时等。
//Userinf 为指向 typ_user_inf 共用体的指针。
```

注：本文所举实例具有一定的代表性，有利于理解此中编程方法，切不可生搬硬套。

(1) 数据库初始化

应用程序应当在手持机每次上电初始化后进行 Database System 初始化，并能根据 Database System 中不同情况做出相应操作。下面给出一段 Database System 初始化代码，它所完成操作是初始化时若在 FLASH 中没有发现 Database System，则格式化出一 Database；即使发现 Database System，也要检查其 Database 是否为本应用程序所使用的 Database，否则也要进行格式化。代码如下：

```
#define DB_ID 0x3F00 //定义数据库标识
#define BLOCK_CNT 4 //定义块数
void init_DB_system(void){
```

```

get_db_sys_param(&dsp); //获取数据库信息
if ((file_sys_cnt=DB_check_db(filesys))==0){
    //若无数据库系统,格式化建立
    erase_db_sys();
    DB_format_db(DB_ID,BLOCK_CNT,sizeof(typ_user_inf));
    //格式化
}
else {
    if((filesys[0].db_id!=DB_ID)|| (filesys[0].block_c
nt!=BLOCK_CNT)){ //数据库系统不符,需进行重新格式化
        erase_db_sys();
        DB_format_db(DB_ID, BLOCK_CNT, sizeof(typ_user_inf));
    }
    else{
        // 成功初始化数据库
    }
}
file_sys_cnt = DB_check_db(filesys);
}

```

(2) 存储数据

将用户数据存入数据库中使用 DB_add_record。假设从 IC 卡中读入的数据信息已存入全局数组 tempbuff 中, 就可通过编写一子程序 stored_into_db 实现数据存储功能。

```

short stored_into_db(void){
    userinf = (typ_user_inf *)tempbuff;
    if(DB_add_record(0,userinf))
        return 1; //成功
    else
        return 0; //失败
}

```

(3) 读取数据

利用数据库 API 可以方便的查询数据库, 将需要需要的记录读出。在这里我们使用使用较为直观的方法: 通过记录号读取记录。首先要能得到记录的记录号, 如下面一组函数:

```

#define FAILURE -1

int find_first_record(unsigned short fs_num){ //找到 Database 中第一条记录的记录号
    if(DB_find_1st_match(fs_num,NULL))
        return current_rec_num[fs_num];
    else
        return FAILURE;
}

int find_next_record(unsigned short fs_num,){ //找到 Database 中下一条记录的记录号

```

```

        if(DB_find_next_match(fs_num,NULL))
            return current_rec_num[fs_num];
        else
            return FAILURE;
    }

int find_prev_record(unsigned short fs_num){ //找到 Database 中前一条记录的记录号
    if(DB_find_prev_match(fs_num,NULL))
        return current_rec_num[fs_num];
    else
        return FAILURE;
}

```

然后，根据记录号就可得到记录内容(注意：返回为指向 FLASH 中记录内容的地址指针，该地址处内容只能读取，不能写入，否则运行时会出现 BUS ERROR 系统异常!)，如需修改内容须使用结构赋值将 FLASH 记录内容复制到 RAM 中用户定义的变量中，修改后另行写入数据库)。

```

void *get_recorder_by_recno(unsigned short fs_num,short recno){
    char temp;
    return DB_jump_to_record(fs_num,recno,&temp);
}

```

我们利用上面子函数完成实例中的数据上传功能：

```

void UART_send(char *lb,short len){//将 lb 所指向数据中 len 个字符通过串口发送出去
    short i;
    for(i=0;i<len;i++){
        while(UART_send_char(*lb));
        lb++;
    }
}

```

```

void upload_data(void) { //上传数据子程序
    recno = find_first_record(); //得到第一条记录的记录号
    while(recno!= FAILURE){
        userinf = get_recorder_by_recno(recno); //根据记录号得到记录
        UART_send(userinf->inf_buf, INFLLEN);
        while(UART_send_char('\r'));
        while(UART_send_char('\n'));
        recno = find_next_record(); //得到下一条记录的记录号
    }
}

```

(4)删除记录及擦除 FLASH ROM

删除记录 API 为 DB_delete_record；擦除 fs_num 所制定的 Database 使用 DB_erase_filesys；擦除整个 Database System 使用 erase_db_sys。这些 API 使用较为简单，在这里就不多作介绍。

3 注意事项

- (1) 在使用数据库 API 时, dsp、filesys 等全局变量只能用作 API 的参数, 不要在调用相关 API 前修改这些全局变量。
- (2) 由于 FLASH ROM 特性的限制, 执行 DB_delete_record 删除一条记录时, 在 FLASH ROM 中并未真正意义上删除, 而是将此记录标记为删除。如使用 DB_count_records 及 DB_jump_to_record 仍可找到这些已删除的记录。
- (3) 要注意 Database 标识(fs_id)与 Database 号(fs_num)是不同的。fs_id 是编程者给 Database 所定义的标识(类似于以数字方式为数据库所起的名字)只在 DB_format_db 中用到, 每个数据库必须 FORMAT 为不同的标识(fs_id), 相同的标识(fs_id)再次进行 FORMAT 时将不会成功。; fs_num 则是 Database 在 Database System 中的标识。在查找、删除记录等操作中使用 fs_num。数据库系统建立后第一个 FORMAT 的数据库其 fs_num 为 0, 第二个数据库为 1, 以此类推。
- (4) 使用 DB_format_db 格式化一 Database 后不能再次对此 Database 进行格式化, 否则 DB_format_db 不能正确执行。若要对擦除已格式化过的 Database 中所有内容, 应执行 DB_erase_filesys()。若要对已格式化过的 Database 再次格式化(如分配不同的记录尺寸), 则只能将所有数据库系统擦除(使用 erase_db_sys())后重新规划。

五 参考资料

1. <<MC998 DEVELOPER'S MANUAL V3.2>>
2. <<MC998 API REFERENCE DOCUMENT V3.2>>
3. <<MC2002 DEVELOPER'S MANUAL V1.0>>
4. <<MC2002 API REFERENCE DOCUMENT V1.0>>