应用笔记

APM32F4xx Flash 模拟 EEPROM 应用笔记

文档编号：AN1094

版本：V 1.0

1 引言

本应用笔记用于介绍在 APM32F4xx 系列上实现应用内部的 Flash 模拟 EEPROM 的功能，包括硬件设计和软件设计方法。关于芯片和寄存器等信息请参考官方网站的用户手册和数据手册。

本应用笔记 APM32F4xx 系列中的 APM32F407ZE 型号进行举例说明。

目录

2 APM32F4xx FLASH 存储器简介

FLASH 存储器又称为闪存，通常用于存储程序代码、数据等信息。在 APM32F4xx 系列中，可以通过配置 BOOT0 接 GND 引脚时，让系统从 Flash 存储器中启动。当主闪存存储器被映射到启动空间时，其仍然能够在它原有的地址访问，即闪存存储器的内容可以在两个地址区域访问。

2.1 FLASH 存储器结构框图

APM32F4xx 的 Flash 存储器结构主要由主存储块和信息块等组成。在信息块中，又有系统存储块、OTP 区域和选项字节等组成，具体情况如下图 2-1：

表格 8Flash 存储器结构

块	名称	地址范围	大小（字节）	扇区
主存储块		0x0800 0000 - 0x0800 3FFF	16K	扇 0
		0x0800 4000 - 0x0800 7FFF	16K	扇 1
		0x0800 8000 - 0x0800 BFFF	16K	扇 2
		0x0800 C000 - 0x0800 FFFF	16K	扇 3
		0x0801 0000 - 0x0801 FFFF	64K	扇 4
		0x0802 0000 - 0x0803 FFFF	128K	扇 5
		…	…	…
		0x080E 0000 - 0x080F FFFF	128K	扇 11
信息块	系统存储	0x1FFF 0000 - 0x1FFF 77FF	30K	-
	OTP 区域	0x1FFF 7800 - 0x1FFF 7A0F	528	-
	选项字节	0x1FFF C000 - 0x1FFF C00F	16	-

图 2-1 FLASH 存储器结构框图

2.2 FLASH 存储器结构框图介绍简述

主存储块：这个部分主要是用于存储应用程序代码，引导程序代码和数据常数等。其中，它主要分为 12 个扇区，前 4 个扇区是 16k，扇区 4 是 64k，扇区 5-11 是 128k。主存储块的起始地址为 0x08000000，即代码开始运行的地方。

系统存储块：这个部分主要用于存放 BootLoader 代码，它用于给系统引导初始化阶段，在载入操作系统前进行硬件初始化，载入操作系统。

OTP 区域：一次性可编程(存储区)，即数据一旦被写入，便无法修改。这个区域共有 528 字节，且被分为 2 个区域，一个被分为 512 字节，可以用来存储用户数据，一个被分为 16 字节，可以用来锁定对应块。

选项字节：用于配置 FLASH 的读写保护、电源管理中的 BOR 级别、软件/硬件看门狗等功能。

3 APM32F4xx FLASH 模拟 EEPROM 原理

由于 APM32F4xx 系列中不带有 EEPROM 存储器，但 APM32F4xx 系列中内置了 Flash 存储器和 RAM 存储器等，可以利用 Flash 存储器结合 RAM 存储器，来实现模拟 EEPROM，在使用体验上达到和读写 EEPROM 存储器一样的效果。

3.1 EEPROM 简介

EEPROM，全称：Electrically Erasable Programmable Read Only Memory，即电可擦除可编程只读存储器，是一种非易失性存储区。EEPROM 可以按字节读写数据，但写入之前不需要先擦除数据。

3.2 APM32F4xx Flash 与 EEPROM 对比

Flash 与 EEPROM 都属于非易失性存储器，他们都可以掉电保存数据。但他们也有很多的区别，如图 3-1 列出了 APM32F4xx 芯片内部的 Flash 与 EEPROM 的主要区别：

对比项	Flash	EEPROM
接口	无	I2C、SPI
擦除操作	写入数据前必需先擦除	无需擦除
读写单元	可按字节读写	可按字节读写
容量	容量大	小，一般不超过1M bit
数据存储时间	10~20年	大于100年
读写次数	10万次	100万次
读写速率	快	慢

图 3-1 Flash 与 EEPROM 对比图

3.2.1 APM32F4xx Flash 模拟 EEPROM 的优势

1. 成本低，使用内部 Flash 模拟 EEPROM，可减少 EEPROM 存储芯片的使用，节约成本。

2. 读写速率快，MCU 内部的 Flash 读写速率高于 EEPROM，可以在短时间内读取大量数据。

3. 容量大，Flash 存储器具有比 EEPROM 存储器更大的容量，可以存储更多的数据和程序。

4. 抗干扰能力强，MCU 内部的 Flash 没有使用 I2C、SPI 等外部通讯总线接口，不存在外部通信总线被干扰的问题。

3.3 实现 APM32F4xx FLASH 模拟 EEPROM

3.3.1 实现步骤

写操作：

1. 在 RAM 中开辟一个与扇区大小相同的缓冲区。
2. 把要写的扇区数据读取到缓存中。
3. 在缓存中改写数据。
4. 擦除扇区（在进行擦除扇区操作前，会进行判断，如果已经是擦除状态，则无需再次进行擦除）。
5. 在把数据写到扇区。

读操作：

1. 类似读取内存操作一样。例如要读取 0x8000000 数据：*(uint32_t*)0x80000000。

3.3.2 实现的难点和解决方法

1. 在 APM32F4xx 系列中，Flash 的最小扇区是 16KB，最大扇区是 128KB，而能给到用户使用最大的 RAM 大小是 128KB。在 Flash 模拟 EEPROM 过程中，为了在擦除 Flash 时不改变已有的数据，需要开辟一块和 Flash 扇区相同大小的 RAM 空间，作为写或者擦除 Flash 时的缓存。如果选取 128KB 的扇区大小作为 Flash 模拟 EEPROM 的空间，那么没有足够的 RAM 大小开辟一块 Flash 模拟 EEPROM 的缓存。

2. 在 APM32F4xx 系列中，前 4 个扇区是 16KB，所以适合用于进行 Flash 模拟 EEPROM。但在这 4 个扇区中，第一个扇区是 MCU 上电时就会去该区域取指令运行程序，如果这个区域没有存储合法的 CPU 指令，会导致程序运行异常。

3. 综合 1、2 两点分析，适合用于 Flash 模拟 EEPROM 的扇区只有扇区 1~3。

4 APM32F4xx FLASH 模拟 EEPROM 例程初始化参数说明

通过 APM32F4 Flash 模拟 EEPROM 时，需要配置 Flash 的扇区大小，Flash 的总大小，Flash 的起始地址，测试数组大小等。下面对这部分的初始化参数宏定义或者变量定义进行解释及配置说明。

4.1 指定 Flash 起始地址与 RAM 空间

代码如下：

```
static const uint8_t Flash_Para_Area[FLASH_EE_TOTAL_SIZE] __attribute__((section(".ARM.__at_0x08004000")));
static uint8_t Flash_EE_Ram_Buffer[FLASH_SECTOR_SIZE];
```

1. 第一句代码中，定义一个 const 数组，使用编译器属性指定该数组在 Flash 的起始地址，这样当我们在使用第 1~3 个扇区的时候，可以不被代码覆盖该 Flash 空间。该数组的大小就是 Flash 模拟 EEPROM 的大小，用户可以自行定义大小空间，但必须是扇区大小的整数倍。

2. 在 RAM 中定义一个与 Flash 扇区相同大小的缓冲区，用以解决擦除 Flash 时不能改变原有数据的问题。

4.2 Flash 扇区配置初始化

代码如下：

```
/* flash sector satrt address */
#define ADDR_FLASH_SECTOR_1     ((uint32_t)0x08004000)   /* 16 Kbytes */
#define ADDR_FLASH_SECTOR_2     ((uint32_t)0x08008000)   /* 16 Kbytes */
#define ADDR_FLASH_SECTOR_3     ((uint32_t)0x0800C000)   /* 16 Kbytes */

/* flash sector size */
#define FLASH_SECTOR_SIZE               ((uint32_t)(1024 * 16))

/* flash emulation eeprom total size. This value must be a multiple of 16KB */
#define FLASH_EE_TOTAL_SIZE             ((uint32_t)(1024 * 16 * 2))

/* flash emulation eeprom sector start address, it's must be sector aligned */
#define FLASH_EE_START_ADDR             ADDR_FLASH_SECTOR_1

/* flash emulation eeprom sector end address */
#define FLASH_EE_END_ADDR               (ADDR_FLASH_SECTOR_1 + FLASH_EE_TOTAL_SIZE)

/* test buffer size */
#define BUFFER_SIZE  64

/* flash emulation eeprom test address */
#define FLASH_EE_TEST_ADDR  ((uint32_t)0x08007FF0)
```

1. 前三句代码中，配置了每个扇区的起始地址，用于实验操作。

2. 第 4 句代码中，配置了实验扇区的大小。

3. 第 5 句代码中，配置了实验扇区的总大小，且是实验扇区大小的 2 倍。原因是，在 Flash 模拟 EEPROM 实验中，每次写入数据时，都会进行一个擦除操作，以便给新数据留出空间。且 Flash 只能在整个扇区内进行擦除，所以新数据如果只覆盖某个扇区的部分数据的话，将会导致新旧数据混合，从而出现数据部分丢失或错误的情况。

4. 第 6 句代码中，配置了 Flash 模拟 EEPROM 实验中的开始地址，且与扇区对齐。

5. 第 7 句代码中，配置了 Flash 模拟 EEPROM 的结束地址。

6. 第 8 句代码中，配置了测试缓存大小。

7. 第 9 句代码中，选取第一个扇区的边界地址作为测试地址，用来测试能否跨扇区测试，来观察整个 Flash 芯片内的读写稳定性。

5 APM32F4xx FLASH 模拟 EEPROM 例程设计介绍

例程功能简介：下载程序后，进行串口的初始化操作，然后通过循环，填充测试数组数据，之后 Flash 写操作接口函数，进行写操作。写操作结束后，调用 Flash 读操作接口函数，进行读操作。最后，通过循环，判断写入与读取是否一致，如若一致，则打印”Test Successful!”,否则，则打印”Test Error!”。

5.1 硬件设计

本次例程实验中所需要的硬件设计：

1. APM32F4 内部 Flash
2. USB 转 TTL 线

本次实验用到的串口接线如图 5-1 中的 PA9，PA10，利用 USB 转 TTL 线进行串口接收和发送数据。

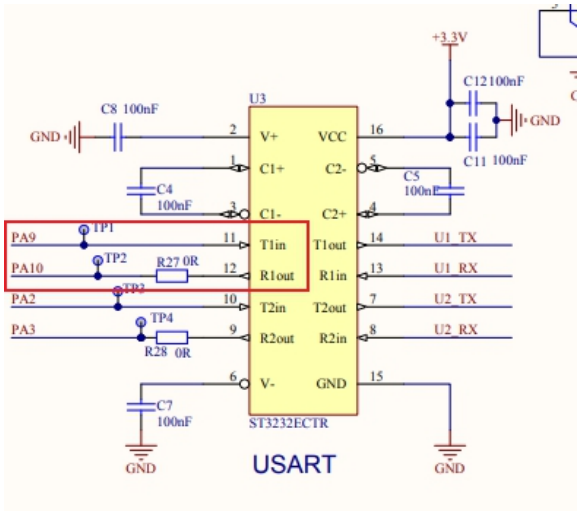

图 5-1 串口接线原理图

5.2 软件设计

要实现 APM32F4xx Flash 模拟 EEPROM 实验，要规划定义好扇区的大小，且要实现 Flash 的读写操作函数。主程序代码主要是将定义好的缓存大小写入测试数组，然后进行 Flash 的读写操作，最后，操作完成后，对操作好的写入数组和读取数组进行对比验证。下面将会对主程序代码，读写操作流程代码进行流程图及代码内容介绍，如下：

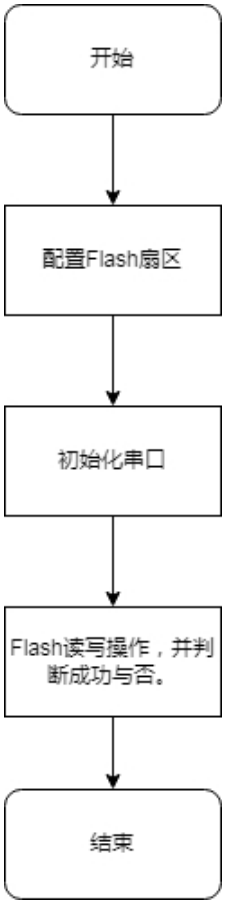

图 5-2 Flash 模拟 EEPROM 实验主程序代码流程图

```
printf("\r\nThis is an example of flash emulation eeprom.\r\n");

/* fill Test_Write_buffer */
for (int i=0; i<BUFFER_SIZE; i++)
{
    Test_Write_buffer[i] = i;
}

/* write the specified sector address data */
Flash_EE_Write(FLASH_EE_TEST_ADDR, Test_Write_buffer, BUFFER_SIZE);

/* read the specified sector address data */
Flash_EE_Read(FLASH_EE_TEST_ADDR, Test_Read_buffer, BUFFER_SIZE);

/* compare the values of two buffers for equality */
for (int i=0; i<BUFFER_SIZE; i++)
{
    if (Test_Write_buffer[i] != Test_Read_buffer[i])
    {
        printf("Test Error!\r\n");
        break;
    }
}

printf("Test Successful!\r\n");
```

图 5-3 Flash 模拟 EEPROM 实验主程序代码图

5.2.1 USART GPIO 引脚配置

对 USART 所使用到的 GPIO 引脚，进行基本的结构体配置，配置波特率和输出

模式等，参考代码如下：

```
USART_Config_T usartConfigStruct;

/* USART configuration */
USART_ConfigStructInit(&usartConfigStruct);
usartConfigStruct.baudRate = 115200;
usartConfigStruct.mode = USART_MODE_TX_RX;
usartConfigStruct.parity = USART_PARITY_NONE;
usartConfigStruct.stopBits = USART_STOP_BIT_1;
usartConfigStruct.wordLength = USART_WORD_LEN_8B;
usartConfigStruct.hardwareFlow = USART_HARDWARE_FLOW_NONE;

/* COM1 init*/
APM_MINI_COMInit(COM1, &usartConfigStruct);
```

5.2.2 Flash 写操作配置

在 APM32F4xx Flash 模拟 EEPROM 实验中，写操作的实现十分关键，下面将对写操作实现的代码框图及代码内容进行介绍。首先将对 Flash_EE_Write 函数中进行变量初始化定义。具体代码如下：

```
uint32_t NumOfSector = 0, NumOfByte = 0, OfsetAddr = 0;

uint32_t count = 0, temp = 0;

/* offerset address in the sector */
OfsetAddr = WriteAddr % FLASH_SECTOR_SIZE;

/* The size of the remaining space inthe sector from WriteAddr */
count = FLASH_SECTOR_SIZE - OfsetAddr;

/* Calculate how many sectors to write */
NumOfSector =  len / FLASH_SECTOR_SIZE;

/* Calculate how many bytes are left less than one sector */
NumOfByte = len % FLASH_SECTOR_SIZE;
```

Flash_EE_Write 函数初始化定义中：

OfsetAddr：计算传入参数在扇区中的偏移地址。此变量用来判断传入的偏移地址是否与扇区起始地址对齐。

count：用来统计操作扇区还有多少剩余空间大小可以用来写入。

NumOfSector：计算有多少扇区去进行写入操作。

NumOfByte:计算扇区少于一个扇区的剩余字节数。当扇区中有已写入的字节时，此变量用于计算扇区中剩余的字节数，判断还有多少地方可以进行写入操作。

介绍完初始化定义后，将对 Flash_EE_Write 函数中的具体操作进行介绍，具体代码如下：

Flash_EE_Write：

if 语句判断是否偏移地址是否与扇区起始地址对齐操作。

```
/* OfsetAddr = 0, WriteAddr is sector aligned */
if (OfsetAddr == 0)
{
    /* len < FLASH_SECTOR_SIZE */
    if (NumOfSector == 0)
    {
        Flash_EE_WriteOneSector(WriteAddr, pData, len);
    }
    /* len > FLASH_SECTOR_SIZE */
    else
    {
        /* write NumOfSector sector */
        while (NumOfSector--)
        {
            Flash_EE_WriteOneSector(WriteAddr, pData, FLASH_SECTOR_SIZE);
            WriteAddr +=  FLASH_SECTOR_SIZE;
            pData += FLASH_SECTOR_SIZE;
        }

        /* write remaining data */
        Flash_EE_WriteOneSector(WriteAddr, pData, NumOfByte);
    }
}
```

else 语句判读为不对齐。

```
   else
   {
       /* len < FLASH_SECTOR_SIZE, the data length is less than one sector */
       if (NumOfSector == 0)
       {
           /* NumOfByte > count,  need to write across the sector */
           if (NumOfByte > count)
           {
               temp = NumOfByte - count;
               /* fill the current sector */
               Flash_EE_WriteOneSector(WriteAddr, pData, count);

               WriteAddr +=  count;
               pData += count;
               /* write remaining data */
               Flash_EE_WriteOneSector(WriteAddr, pData, temp);
           }
           else
           {
               Flash_EE_WriteOneSector(WriteAddr, pData, len);
           }
       }
       /* len > FLASH_SECTOR_SIZE */
       else
       {
           len -= count;
           NumOfSector =  len / FLASH_SECTOR_SIZE;
           NumOfByte = len % FLASH_SECTOR_SIZE;

           /* write count data */
           Flash_EE_WriteOneSector(WriteAddr, pData, count);

           WriteAddr +=  count;
           pData += count;
```

```
            /* write NumOfSector sector */
            while (NumOfSector--)
            {
                Flash_EE_WriteOneSector(WriteAddr, pData, FLASH_SECTOR_SIZE);
                WriteAddr +=  FLASH_SECTOR_SIZE;
                pData += FLASH_SECTOR_SIZE;
            }

            if (NumOfByte != 0)
            {
                Flash_EE_WriteOneSector(WriteAddr, pData, NumOfByte);
            }
        }
    }
```

具体操作代码有以下几步：

1. 当传入地址与扇区起始地址对齐，首先判断写入的空间跟扇区的空间大小是否一致，其次判断扇区剩余字节数跟剩余空间的大小能否一次写入。判断完后，便是对变量初始化定义进行计算和写入操作。

2. 当传入地址与扇区起始地址不对齐，首先判断写入的空间跟扇区的空间大小是否一致，其次判断扇区剩余字节数跟剩余空间的大小能否一次写入。判断完后，便是对变量初始化定义进行计算和写入操作。

接下来，将对 Flash_EE_WriteOneSector 中的变量初始化定义进行介绍，具体介绍如下：

```
startAddr = WriteAddr / FLASH_SECTOR_SIZE * FLASH_SECTOR_SIZE;
offesetAddr = WriteAddr % FLASH_SECTOR_SIZE;
```

Flash_EE_Write 函数初始化定义中：

startAddr：用于统计写入扇区数据时的起始地址。此变量用于擦除操作前，将数据读取到 RAM 中，也用于写入数据到 RAM 中。

offesetAddr：用于将数据写入到该偏移地址上。

Flash_EE_WriteOneSector：

```
/* unlock flash for erase or write*/
FMC_Unlock();

/* check whether the sector need to be erased */
for (i=0; i<len; i++)
{
    if (Flash_EE_ReadByte(WriteAddr + i) != 0xFF)
    {
        isErase = 1;
        break;
    }
}
/* the sector needs to be erase */
if (isErase == 1)
{
    /* read the entire sector data to the buffer before write or erase */
    Flash_EE_Read(startAddr, Flash_EE_Ram_Buffer, FLASH_SECTOR_SIZE);

    /* copy the data to the buffer */
    for (i=0; i<len; i++)
    {
        Flash_EE_Ram_Buffer[offesetAddr + i] = pData[i];
    }

    /* erase the sector where the address is located */
    FMC_EraseSector(Get_Flash_Sector_Num(WriteAddr), FMC_VOLTAGE_3);

    /* write the entire sector data */
    for (i=0; i<FLASH_SECTOR_SIZE; i++)
    {
        FMC_ProgramByte(startAddr, Flash_EE_Ram_Buffer[i]);
        startAddr += 1;
    }
}
```

```
/* the sector don't need to be erase */
else
{
    /* write n bytes of data to the sector */
    for (i=0; i<len; i++)
    {
        FMC_ProgramByte(WriteAddr, pData[i]);
        WriteAddr += 1;
    }
}

/* lock flash */
FMC_Lock();
```

具体操作代码有以下几步：

1. 进行解锁操作
2. 检查扇区是否需要进行擦除操作。
3. 如果需要进行擦除操作，先将数据读取和复制到 RAM 中，进行擦除操作后，写入整个扇区的数据到 RAM 中。
4. 如果不需要进行擦除操作，则可直接写入数据到传入的地址中。

5.2.3 Flash 读操作配置

APM32F4xx Flash 模拟 EEPROM 实验中，读操作是只需要从 Flash 中读取一个八位数据即可，操作的步骤是用解引用操作。下面将对涉及读操作的函数 Flash_EE_Read 和 Flash_EE_ReadByte 代码内容进行介绍，如下：

Flash_EE_Read：

```
/* read data */
for (int i = 0; i < len; i++)
{
    pData[i] = Flash_EE_ReadByte(ReadAddr);
    ReadAddr += 1;
}
```

具体操作代码有以下几步：

1. 在循环中依次读取传入地址中的值。

Flash_EE_ReadByte：

```
static uint8_t Flash_EE_ReadByte(uint32_t Addr)
{
    return (*(__IO uint8_t*)Addr);
}
```

具体操作代码有以下几步：

1. 通过解引用读取一个八位数据。

6 版本历史

表 1 文档版本历史记录

日期	版本	变更历史
2023.06.16	1.0	初稿

声明

本手册由珠海极海半导体有限公司（以下简称“极海”）制订并发布，所列内容均受商标、著作权、软件著作权相关法律法规保护，极海保留随时更正、修改本手册的权利。使用极海产品前请仔细阅读本手册，一旦使用产品则表明您（以下称“用户”）已知悉并接受本手册的所有内容。用户必须按照相关法律法规和本手册的要求使用极海产品。

1、权利所有

本手册仅应当被用于与极海所提供的对应型号的芯片产品、软件产品搭配使用，未经极海许可，任何单位或个人均不得以任何理由或方式对本手册的全部或部分内容进行复制、抄录、修改、编辑或传播。

本手册中所列带有“®”或“TM”的“极海”或“Geehy”字样或图形均为极海的商标，其他在极海产品上显示的产品或服务名称均为其各自所有者的财产。

2、无知识产权许可

极海拥有本手册所涉及的全部权利、所有权及知识产权。

极海不应因销售、分发极海产品及本手册而被视为将任何知识产权的许可或权利明示或默示地授予用户。

如果本手册中涉及任何第三方的产品、服务或知识产权，不应被视为极海授权用户使用前述第三方产品、服务或知识产权，除非在极海销售订单或销售合同中另有约定。

3、版本更新

用户在下单购买极海产品时可获取相应产品的最新版的手册。

如果本手册中所述的内容与极海产品不一致的，应以极海销售订单或销售合同中的约定为准。

4、信息可靠性

本手册相关数据经极海实验室或合作的第三方测试机构批量测试获得，但本手册相关数据难免会出现校正笔误或因测试环境差异所导致的误差，因此用户应当理解，极海对本手册中可能出现的该等错误无需承担任何责任。本手册相关数据仅用于指导用户作为性能参数参照，不构成极海对任何产品性能方面的保证。

用户应根据自身需求选择合适的极海产品，并对极海产品的应用适用性进行有效验证和测试，以确认极海产品满足用户自身的需求、相应标准、安全或其它可靠性要求；若因用户未充分对极海产品进行有效验证和测试而致使用户损失的，极海不承担任何责任。

5、合规要求

用户在使用本手册及所搭配的极海产品时，应遵守当地所适用的所有法律法规。用户应了解产品可能受到产品供应商、极海、极海经销商及用户所在地等各国有关出口、再出口或其它法律的限制，用户（代表其本身、子公司及关联企业）应同意并保证遵守所有关于取得极海产品及／或技术与直接产品的出口和再出口适用法律与法规。

6、免责声明

本手册由极海“按原样”（as is）提供，在适用法律所允许的范围内，极海不提供任何形式的明示或暗示担保，包括但不限于对产品适销性和特定用途适用性的担保。

对于用户后续在针对极海产品进行设计、使用的过程中所引起的任何纠纷，极海概不承担责任。

7、责任限制

在任何情况下，除非适用法律要求或书面同意，否则极海和/或以“按原样”形式提供本手册的任何第三方均不承担损害赔偿责任，包括任何一般、特殊因使用或无法使用本手册相关信息而产生的直接、间接或附带损害（包括但不限于数据丢失或数据不准确，或用户或第三方遭受的损失）。

8、适用范围

本手册的信息用以取代本手册所有早期版本所提供的信息。