RFID实验三总结

原创

龙云尧 于 2017-04-23 03:18:31 发布

分类专栏： rfid 文章标签： RFID Java-Card

本文链接：https://blog.csdn.net/Michael753951/article/details/70481546

> 一次debug到哭泣的经历。

龙云尧个人博客，转载请注明出处。

CSDN地址：http://blog.csdn.net/Michael753951/article/details/70481546

个人blog地址：http://yaoyl.cn/rfidshi-yan-san-zong-jie/

在实验过程中，需要不断翻阅实验课PPT之**《04 电子钱包的功能》**，word之**《实验3文档》**，**CSDN大佬吕浪的课程总代码以及他的的Java card开发系列文章**。

本次实验和前两次实验相比，代码量多**很多**，并且**实验思路稍有区别**。实验之前可以不太懂实验流程（主要是因为流程本身就太复杂了），但是一定要一**遍又一遍阅读源代码**，只有在读源码的过程中，才能体会整个验证过程，对项目中涉及到的函数方法的使用才能有一个更加深入的了解。接着自己不断重写代码，理解整个实现过程，才能对这个课程实验有较为深入的了解。

代码在未征得本人同意之前，请勿直接Ctrl+C，Ctrl+V，谢谢。

正式实验

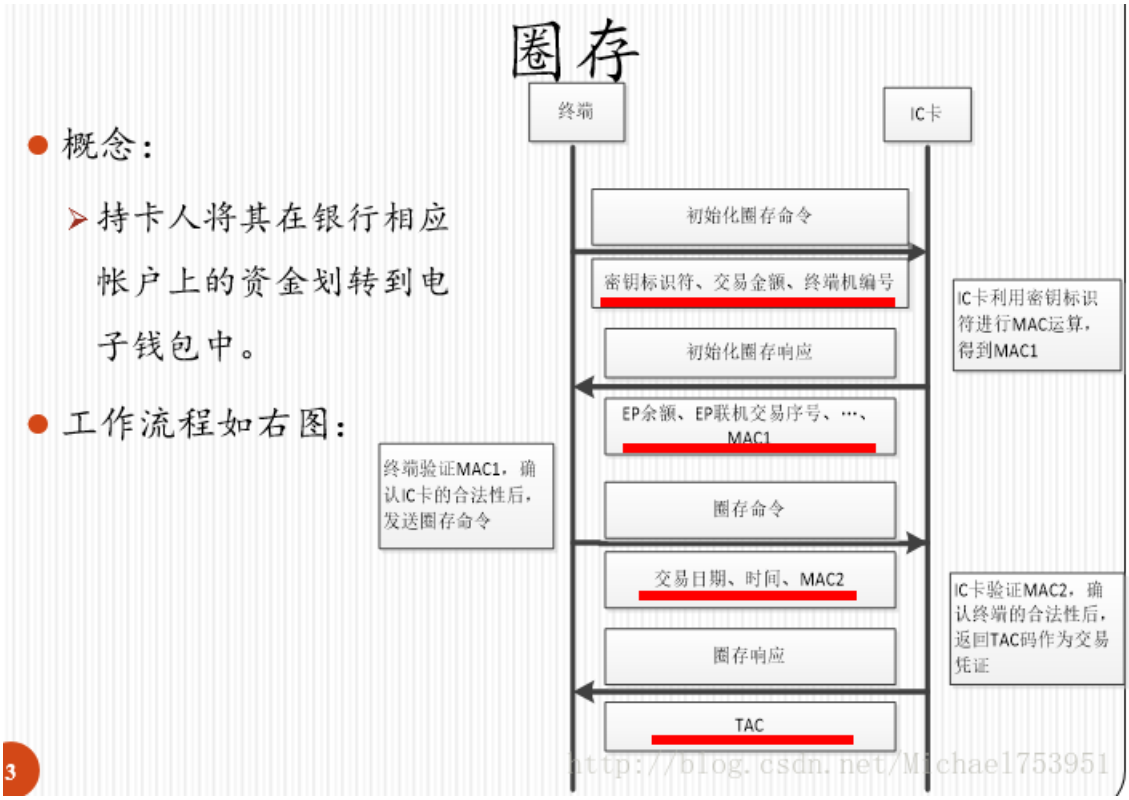
实验分析

首先我们在PPT中知道本次实验的主要需要实现的功能是：

- 圈存
- 消费
- 余额查询

接下来我们开始看ppt《04 电子钱包的功能》和《实验3文档》。

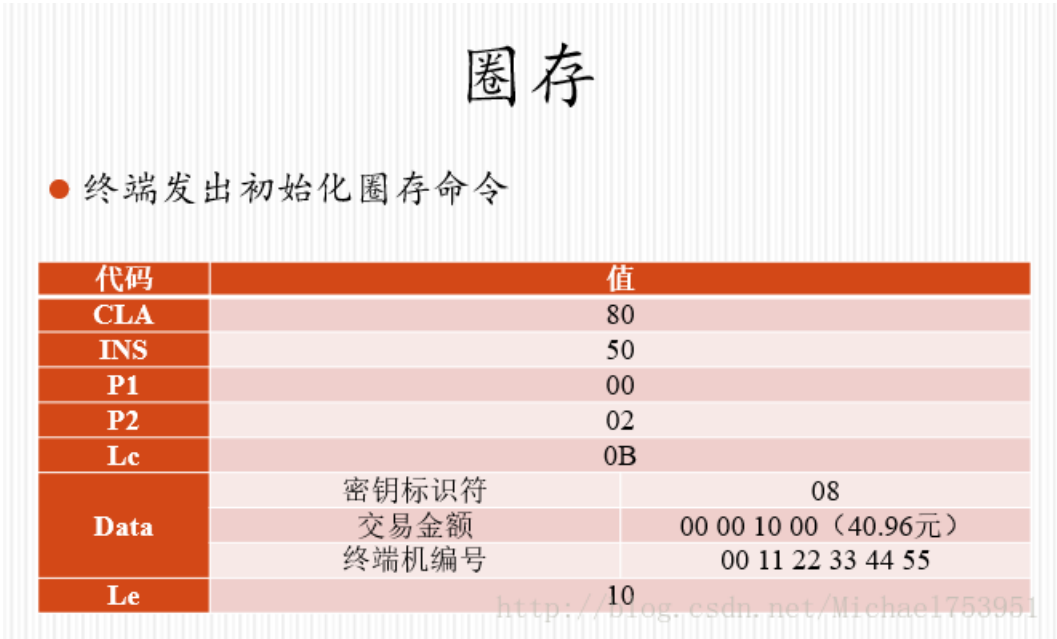
首先是圈存功能的流程图。



流程图中我们可以分析出圈存一共有4个步骤：

- **终端发送消息初始化**
- **IC响应初始化，并且发送MAC1验证**
- **终端验证MAC1，确认IC卡是否合法，然后发送包含MAC2的圈存命令**
- **IC卡验证终端机的合法性，执行完成以后返回TAC响应操作完成**

接下来我们将一步一步仔细分析圈存是如何实现的。



step1：圈存机发送的初始信息如下所示。消息中包含了**秘钥标识符，交易金额，终端机编号**。

圈存

- IC卡对初始化圈存命令进行处理

1. IC卡根据密钥标识符查找圈存密钥
2. IC卡生成随机数，利用圈存密钥产生过程密钥
3. IC卡利用所生成的过程密钥产生MAC1
4. IC卡将返回相应的数据

说明	值
EP余额	00 00 00 00
EP联机交易序列号	00 00
密钥版本号DPK	01
算法标识DPK	00
伪随机数（IC卡）	B1 EE 18 0C
MAC1	F2 0B 5E 52

1、IC 卡对初始化圈存命令进行处理。

1、IC 卡根据密钥标识符，在密钥文件中查找该密钥标识符对应的圈存密钥，如果找不到，就返回状态字“9403”，表示不存在相对应的密钥。如果找到的话，就进行以下的处理。

2、IC 卡生成随机数，利用所查找到的密钥产生过程密钥。过程密钥的生成方式，我将在之后的密钥管理中进行说明。其输入的数据为伪随机数||电子钱包联机交易序号||8000，密钥为所查找到的圈存密钥。

3、IC 卡利用所生成的过程密钥产生 MAC1。其 MAC 1 的生成方式，我也将在之后的密钥管理中进行说明。其输入的数据为电子钱包余额（交易前）||交易金额||交易类型标识||终端机编号，密钥为过程密钥，

4、在进行这些操作后，IC 卡将返回相应的数据。

step2：

- **IC卡根据秘钥标识符寻找圈存秘钥**
- **生成过程秘钥。输入数据为[伪随机数||电子钱包联机交易序号||8000]，秘钥为圈存秘钥，使用3DES加密算法。**
- **生成MAC1。输入数据为[电子钱包余额（交易前）||交易金额||交易类型标识||终端机编号]，秘钥为过程秘钥，使用我们在上一次实现的MAC生成函数gmac4，计算出MAC1用来表明身份。**
- **IC卡返回[余额||联机交易序列号||秘钥版本号||算法标识||伪随机数||MAC1]。**

圈存

- 终端验证MAC1。若验证成功，将同意交易进行。
- 终端向IC卡发送圈存命令

代码	值	
CLA	80	
INS	52	
P1	00	
P2	00	
Lc	0B	
Data	交易日期（主机）	20 11 12 21（2011年12月11日）
	交易时间（主机）	21 48 22（21点48分22秒）
	MAC2	3A 84 5B F0
Le	04	

4、IC 卡收到圈存命令后，利用过程密钥生成 MAC2。其输入数据为交易金额||交易类型标识||终端机编号||交易日期（主机）||交易时间（主机）。密钥为过程密钥。与圈存命令传送的 MAC2 进行比较，如果相同，则 MAC2 有效。

5、IC 卡将电子钱包联机交易序号加 1，并且把交易金额加在电子钱包的余额上。

6、IC 卡生成 TAC 码。TAC 码的生成方式和 MAC 码的生成方式一致。其输入的数据：电子钱包余额（交易后）||电子钱包联机交易序号（加 1 前）||交易金额||交易类型标识||终端机编号||交易日期（主机）||交易时间（主机）。密钥为 TAC 密码最左 8 个字节与 TAC 密码最右 8 个字节异或的结果。

7、IC 卡将 TAC 码返回给终端。至此，IC 卡端的圈存交易已经完成了。

step3：

- **圈存机对IC卡发挥的MAC1信息进行校验，如果正确就说明IC卡信息合法。**
- **计算MAC2。输入信息为[交易金额||交易类型标识||终端机编号||交易日期（主机）||交易时间（主机）]，秘钥为过程秘钥，加密算法为依然为gmac4。用来表明自己的身份。**
- **发送圈存指令。消息中包含[交易日期||交易时间||MAC2]。**

IC 卡生成 TAC 码。TAC 码的生成方式和 MAC 码的生成方式一致。其输入的数据：电子钱包余额（交易后）||电子钱包联机交易序号（加 1 前）||交易金额||交易类型标识||终端机编号||交易日期（主机）||交易时间（主机）。密钥为 TAC 密码最左 8 个字节与 TAC 密码最右 8 个字节异或的结果。

step4：**IC使用同样的算法计算MAC2，如果计算结果和终端返回的MAC2一致，就说明终端的身份合法。IC卡就会执行圈存命令。同时返回TAC。其中TAC计算时，输入数据为[电子钱包余额（交易后）||电子钱包联机交易序号（加 1 前）||交易金额||交易类型标识||终端机编号||交易日期（主机）||交易时间（主机）]，密钥为TAC密码最左8个字节与TAC密码最右8个字节异或的结果。**

到这里整个圈存过程就结束了。消费以及查询和圈存的实现原理一致，这里就不赘述了。

读代码

前面的分析中，我们已经对本次实验有了大致的了解，接下来就是开始**读源码**的过程了。不过本次实验中，因为我们只需要对IC卡的系统进行编程实现，而对终端机需要靠人脑完成，所以我们重心就会放在圈存的初始化和圈存的执行上面了。

圈存初始化

还是老样子，我们**先读项目给的源码中的`Purse`部分**，里面有圈存初始化和圈存确认信息的处理函数。

在TA给的源代码中，`init_load`和`load`方法是已经给好了的，我们先读这两部分的源代码，理解整个设计思路。

首先我们需要**修改Purse类**，让其`process`方法里面增执行**圈存初始化，圈存，消费初始化，消费，以及查询的入口。**

```
/*
 * 功能：对命令的分析和处理
 * 参数：无
 * 返回：是否成功处理了命令
 * 《01 Java智能卡之概述》P30
 */
private boolean handleEvent(){
    switch(papdu.ins){
        case condef.INS_CREATE_FILE:return create_file();              // 0xE0 文件创建
        //todo：完成写二进制命令，读二进制命令，写密钥命令
        case condef.INS_WRITE_KEY:  return write_key();                // 0xD4 写秘钥
        case condef.INS_WRITE_BIN:  return write_read_bin((byte)'U');  // 0xD6 写二进制文件
        case condef.INS_READ_BIN:   return write_read_bin((byte)'R');  // 0xB0读二进制文件
        case condef.INS_NIIT_TRANS: return init_load_purchase(papdu.p1);// 0x50消费初始化或者圈存初始化
        case condef.INS_LOAD:       return load();                     // 0x52圈存
        case condef.INS_PURCHASE:   return purchase();                 // 0x54消费
        case condef.INS_GET_BALANCE:return get_balance();              // 0x5c查询余额
    }
    ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED); // 0x6D00 表示 CLA错误
    return false;
}
```


因为圈存和消费的init方法的ins都一样，所以我们还需要**增加一个判断方法，利用两者p1参数不一样，来判断是** `init_load` **还是** `init _purchase` 。

```
private boolean init_load_purchase(short load_purchase){
    switch(load_purchase){
        case (short)0x00: return init_load();
        case (short)0x01: return init_purchase();
        default : return false;
    }
}
```


好了，入口写好了。我们开始看 `init_load` 的实现方法。

```
/*
 * 功能：圈存初始化命令的实现
 */
private boolean init_load() {
    short num,rc;

    if(papdu.cla != (byte)0x80)
        ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

    if(papdu.p1 != (byte)0x00 && papdu.p2 != (byte)0x02)
        ISOException.throwIt(ISO7816.SW_WRONG_P1P2);

    if(papdu.lc != (short)0x0B)
        ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

    if(EPfile == null)
        ISOException.throwIt(ISO7816.SW_FILE_NOT_FOUND);

    num = keyfile.findkey(papdu.pdata[0]);

    if(num == 0x00)
        ISOException.throwIt(ISO7816.SW_RECORD_NOT_FOUND);

    rc = EPfile.init4load(num, papdu.pdata);

    if(rc == 2)
        ISOException.throwIt((condef.SW_LOAD_FULL));

    papdu.le = (short)0x10;

    return true;
}
```

前面还是和读写数据时一样的操作，进行参数校对，这个部分已经很简单了。我们看到有一个`findkey`方法。点进去查看函数的具体内容。

```
/*
 * 功能：通过密钥标识符获取密钥记录号
 * 参数：tag 密钥标识符
 * 返回：记录号
 */
public short findkey(byte Tag){
    byte[] pbuf;
    if(recNum == 0)
        return 0;

    for(short i = 0;i < recNum;i ++){
        pbuf = (byte[])Key[i];
        if(pbuf[0] == Tag)
            return (short)(i + 1);
    }
    return 0;
}
```


查找方法比较简单，从Key数组中获取秘钥，存进pbuf中，如果存在就返回秘钥标识符，否则返回0。

回到`init_load`，在执行`fendkey`之后，是异常处理，验证是否查找失败。然后执行`init4load`方法。继续点进去查看。（**前方高能预警**）

```
/*
 * 功能：圈存初始化功能完成
 * 参数：num 密钥记录号，  data 命令报文中的数据段
 * 返回：0：  圈存初始化命令执行成功     2：圈存超过电子钱包最大限额
 */
public final short init4load(short num, byte[] data){
    short length,rc;

    Util.arrayCopyNonAtomic(data, (short)1, pTemp42, (short)0, (short)4);  //交易
    Util.arrayCopyNonAtomic(data, (short)5, pTemp81, (short)0, (short)6);  //终端

    //判断是否超额圈存
    rc = increase(pTemp42, false);
    if(rc != (short)0)
        return (short)2;

    //密钥获取
    length = keyfile.readkey(num, pTemp32);
    keyID = pTemp32[3];
    algID = pTemp32[4];
    Util.arrayCopyNonAtomic(pTemp32, (short)5, pTemp16, (short)0, length);

    //产生随机数
    RandData.GenerateSecureRnd();
    RandData.getRndValue(pTemp32, (short)0);

    //产生过程密钥
    Util.arrayCopyNonAtomic(EP_online, (short)0, pTemp32, (short)4, (short)2);
    pTemp32[6] = (byte)0x80;
    pTemp32[7] = (byte)0x00;

    EnCipher.gen_SESPK(pTemp16, pTemp32, (short)0, (short)8, pTemp82, (short)0);

    //产生MAC1
    Util.arrayCopyNonAtomic(EP_balance, (short)0, pTemp32, (short)0, (short)4);
    Util.arrayCopyNonAtomic(data, (short)1, pTemp32, (short)4, (short)4);
    pTemp32[8] = (byte)0x02;
    Util.arrayCopyNonAtomic(data, (short)5, pTemp32, (short)9, (short)6);
    Util.arrayCopyNonAtomic(pTemp32, (short)0, data, (short)0x00, (short)0x0F);
    EnCipher.gmac4(pTemp82, pTemp32, (short)0x0F, pTemp41);

    //响应数据
    Util.arrayCopyNonAtomic(EP_balance, (short)0, data, (short)0, (short)4);
    Util.arrayCopyNonAtomic(EP_online, (short)0, data,  (short)4, (short)2);
    data[6] = keyID;
    data[7] = algID;
    RandData.getRndValue(data, (short)8);
    Util.arrayCopyNonAtomic(pTemp41, (short)0, data, (short)12, (short)4);

    return 0;
}
```


回忆一下初始化的过程中，IC的操作部分是怎么操作的。

- **IC卡根据秘钥标识符寻找圈存秘钥**
- **生成过程秘钥。输入数据为[伪随机数||电子钱包联机交易序号||8000]，秘钥为圈存秘钥，使用3DES加密算法。**
- **生成MAC1。输入数据为[电子钱包余额（交易前）||交易金额||交易类型标识||终端机编号]，秘钥为过程秘钥，使用我们在上一次实现的MAC生成函数gmac4，计算出MAC1用来表明身份。**
- **IC卡返回[余额||联机交易序列号||秘钥版本号||算法标识||伪随机数||MAC1]。**

按照这个思路看源代码。（看代码的时候一**定要对着ppt的流程读**，我第一次读源代码就是单纯对着IDE读，结果读的很爽，但是读完了只知道每个基本操作在干嘛，但是整个操作流程还是一脸懵逼。）

开始看代码。

首先从`data`中提取交易金额，终端编号，存进`pTemp42`和`pTemp81`。

然后判断是否超额。我们继续点进去看一下`increase`的源代码。

```
/*
 * 功能：电子钱包金额的增加
 * 参数：data 所增加的金额
 *      flag 是否真正增加电子钱包余额
 * 返回：圈存后，余额是否超过最大限额
 */
public final short increase(byte[] data, boolean flag){
    short i, t1, t2, ads;

    ads = (short)0;
    for(i = 3; i >= 0; i --){
        t1 = (short)(EP_balance[(short)i] & 0xFF);
        t2 = (short)(data[i] & 0xFF);

        t1 = (short)(t1 + t2 + ads);
        if(flag)
            EP_balance[(short)i] = (byte)(t1 % 256);
        ads = (short)(t1 / 256);
    }
    return ads;
}
```


从`EP_ balance`中和`data`中依次取出一个字节，将其相加再和一个`ads`(进位标志符)相加，如果`flag`为真，就改写`EP _balance`中的值，然后更新`ads`。最终返回进位标志位`ads`。整个就是大数加法的思想方法。但是`EP _balance`是啥？？？我们再点进去看源代码。

```
    //内部数据元
    private byte[] EP_balance;          //电子钱包余额
    private byte[] EP_offline;          //电子钱包联机交易序号
    private byte[] EP_online;           //电子钱包脱机交易序号

    byte keyID;         //密钥版本号
    byte algID;         //算法标识符

    //安全系统设计
    private Randgenerator RandData;             //随机数产生
    private PenCipher EnCipher;                 //数据加解密方式实现
/**
 * 下面数据是计算时需要用到的临时过程数据
 */
    //临时计算数据
    //4个字节的临时计算数据
    private byte[] pTemp41;
    private byte[] pTemp42;

    //8个字节的临时计算数据
    private byte[] pTemp81;
    private byte[] pTemp82;

    //32个字节的临时计算数据
    private byte[] pTemp16;
    private byte[] pTemp32;
```


是当前电子钱包的余额。因此**整个`increase`就是判断当前余额加上一个圈存值，如果超额（结果超过4bytes），就会返回1（那个进位标志位`ads`）。否则返回0。如果传入的第二个参数为false，就不会更新余额，否则会执行余额更新操**作。

好，到这里我们知道了`rc = increase(pTemp42, false);`部分的意义了。再往下看密码取位部分。

首先使用`readkey`方法。点进去查看。

```
/*
 * 功能：通过密钥记录号读取密钥
 * 参数：num 密钥记录号 data 所读取到的密钥缓冲区
 * 返回：密钥的长度
 */
public short readkey(short num, byte[] data){
    byte[] pdata;
    pdata = (byte[])Key[num - 1];
    Util.arrayCopyNonAtomic(pdata, (short)2, data, (short)0, (short)(
        pdata[1]));
    return (short)(pdata[1] + 5);
}
```


这里需要对`Key`的结构有一定的了解了。我们点进去查看`Key`这个类定义和实现。

```
public short size;          //记录的最大存储数量
public short recNum;        //当前所存储的记录数量
private Object[] Key;       //密钥记录，一个指针数组

public KeyFile(){
    size = 4;
    recNum = 0;
    Key = new Object[size];
}

/*
 * 功能：添加密
 * 参数：tag 密钥标识符；length 数值的长度；value 数值（5个字节的密钥头+16个字节的密钥值）
 * 返回：无
 */
public void addkey(byte tag, short length, byte[] value){
    byte[] pbuf;

    pbuf = new byte[23];
    Key[recNum] = pbuf;
    pbuf[0] = tag;
    pbuf[1] = (byte)length;
    Util.arrayCopyNonAtomic(value, (short)0, pbuf, (short)2, length);
    recNum ++;
}
```


所以`Key`中存放的结构为2bytes信息位（分别为1byte的`pbuf`，1byte的`length`），5bytes的秘钥头，以及16bytes的秘钥值。其中length是addKey传入value的长度，为表头+秘钥的长度。

回到`readkey`，`readkey`就是将秘钥取出来，然后将秘钥表头中的`value`值取出来（5bytes表头+16bytes秘钥），**返回长度为秘钥的实际长度（减掉了表头长度）**。

回到init4load，所以这4行代码的意义就是

- **按照秘钥获取部分就是按照秘钥标识符获取bytes秘钥头+16bytes秘钥存进pTemp32**
- **从秘钥头第4个bytes获取秘钥版本号，从第5个bytes获取算法标识符**
- **从获取pTemp32中将秘钥的实际值（从第5位开始读取秘钥长度个bytes），取出来存进pTemp16。**

再往下读随机数产生的代码。

```
/*
 * 功能：生成随机数
 * 参数：无
 * 返回：无
 */
public final void GenerateSecureRnd(){
    rd.generateData(v, (short)0, (short)size);
}
```


调用`rd.generateData`方法，对传入的参数进行操作。我们在查看generateData的实现方法的时候，已经进入.class文件，没有发现有意义的信息。于是折返查看**v和size的信息**。

```
private byte size;      //随机数的长度
private byte[] v;       //随机数的值
private RandomData rd; //随机数的产生机制

public Randgenerator(){
    size = (byte)4;
    v = JCSystem.makeTransientByteArray((short)4, JCSystem.CLEAR_ON_DESELECT);
    rd = RandomData.getInstance(RandomData.ALG_SECURE_RANDOM);
}

/*
 * 功能：生成随机数
 * 参数：无
 * 返回：无
 */
public final void GenerateSecureRnd(){
    rd.generateData(v, (short)0, (short)size);
}
```


根据注释，我们可以大概知道`GenerateSecureRnd`的意义就是**根据特定的随机数产生机制产生随机数，然后写进v**。

```
/*
 * 功能：获取随机数
 * 参数：bOff 所获取的数据的偏移量； bf 所存储的数据的值
 * 返回：随机数的长度
 */
public final byte getRndValue(byte[] bf, short bOff){
    Util.arrayCopyNonAtomic(v, (short)0, bf, bOff, (short)size);
    return size;
}
```


`getRndValue`方法就是**将随机数v写入参数bf中，偏移位为bOff**。

回到`init4load`，产生随机数这两行的意义就是**产生随机数**，然后将随机数写进`pTemp32`的[0:3]位。

接下来，将`EP_online`(电子钱包脱机交易序号，之前分析图片中出现过)写入`pTemp32[4:5]`，将`0x8000`写入`pTemp32[6:7]`，调用上一次实验中实现的`3des`加密算法，秘钥为上面得到的圈存秘钥（存在`pTemp16`中）**产生过程秘钥**，写入`pTemp82`。

回看前面分析的圈存初始化的第一步和第二步，**①IC卡根据秘钥标识符寻找圈存秘钥；②生成过程秘钥。输入数据为[伪随机数||电子钱包联机交易序号||8000]，秘钥为圈存秘钥，使用3DES加密算法。**是不是一模一样？

好，我们再往下看。

产生`MAC1`。首先分别往`pTemp32`中写入`EP_balance`（余额），`data[1:4]`（交易金融，对着那一页PPT找`data`结构就知道了），0x02，`data[5:10]`（终端机编号，一样看ppt中的`data`结构），然后将`pTemp32`中的内容复制到`data`（不知道这里写入data有什么意义，因为在响应数据部分又会被写一次。）。然后**使用上次实验实现的`gmac4`，输入数据为`pTemp32`，秘钥为上一步得到的存在`pTemp82`的过程秘钥，得到的`mac1`存在`pTemp41`中**。

同样回看前面分析的圈存初始化的第三步。**生成MAC1。输入数据为[电子钱包余额（交易前）||交易金额||交易类型标识||终端机编号]，秘钥为过程秘钥，使用我们在上一次实现的MAC生成函数gmac4，计算出MAC1用来表明身份**。是不是对上了？

然后将`EP_balance`（余额），`EP _online`(电子钱包脱机交易序号），`keyID`（秘钥版本号），`algID`（算法标识符），**随机数，以及mac1写进data**。

回看圈存初始化的第四步，**IC卡返回[余额||联机交易序列号||秘钥版本号||算法标识||伪随机数||MAC1]**。一模一样。

到这里`init4load`也结束了，往上回到`init_load`，`papdu.pdata`已经在`init4load`中设置完成，在`papdu.le`（理想的下一次指令中的数据长度）写为`0x10`。然后`purse.init _load`结束。

光分析这一步我写了一个半钟，当时在读源代码的时候花的时间更久。**但是这个步骤不能跳过，它让我们对整个IC卡的业务逻辑和方法实现以及调用打下了基础。我们在实现其他方法的时候，才能更加得心应手。**

再看`purse.load`方法吧。这一步实现了**圈存**。

```
/*
 * 功能：圈存命令的实现
 */
private boolean load() {
    short rc;

    if(papdu.cla != (byte)0x80)
        ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

    if(papdu.p1 != (byte)0x00 && papdu.p2 != (byte)0x00)
        ISOException.throwIt(ISO7816.SW_WRONG_P1P2);

    if(EPfile == null)
        ISOException.throwIt(ISO7816.SW_FILE_NOT_FOUND);

    if(papdu.lc != (short)0x0B)
        ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

    rc = EPfile.load(papdu.pdata);

    if(rc == 1)
        ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
    else if(rc == 2)
        ISOException.throwIt(condef.SW_LOAD_FULL);
    else if(rc == 3)
        ISOException.throwIt(ISO7816.SW_RECORD_NOT_FOUND);

    papdu.le = (short)4;

    return true;
}
```


前面的就不详说了，它调用了`EPfile.load`方法，我们点进去查看。

```
/*
 * 功能：圈存功能的完成
 * 参数：data 命令报文中的数据段
 * 返回：0 圈存命令执行成功；1 MAC2校验错误；  2 圈存超过最大限额；3 密钥未找到
 */
public final short load(byte[] data){
    short rc;

    Util.arrayCopyNonAtomic(pTemp42, (short)0, pTemp32, (short)0, (short)4);
                //交易金额
    pTemp32[4] = (byte)0x02;
                                                                    //交易标识
    Util.arrayCopyNonAtomic(pTemp81, (short)0, pTemp32, (short)5, (short)6);
                //终端机编号
    Util.arrayCopyNonAtomic(data, (short)0, pTemp32, (short)11, (short)7);
                 //交易日期与时间
    EnCipher.gmac4(pTemp82, pTemp32, (short)0x12, pTemp41);

    //检验MAC2
    if(Util.arrayCompare(data, (short)7, pTemp41, (short)0, (short)4) != (byte)
        0x00)
        return (short)1;

    //电子钱包数目增加
    rc = increase(pTemp42, true);
    if(rc != (short)0)
        return 2;

    //TAC数据
    Util.arrayCopyNonAtomic(EP_balance, (short)0, pTemp32, (short)0, (short)4);
            //电子钱包余额
    Util.arrayCopyNonAtomic(EP_online, (short)0, pTemp32, (short)4, (short)2);
             //电子钱包联机交易序号
    Util.arrayCopyNonAtomic(pTemp42, (short)0, pTemp32, (short)6, (short)4);
                //交易金额
    pTemp32[10] = (byte)0x02;
                                                                    //交易类型
    Util.arrayCopyNonAtomic(pTemp81, (short)0, pTemp32, (short)11, (short)6);
                //终端机编号
    Util.arrayCopyNonAtomic(data, (short)0, pTemp32, (short)17, (short)7);
                 //交易日期与时间

    //联机交易序号加1
    rc = Util.makeShort(EP_online[0], EP_online[1]);
    rc ++;
    if(rc > (short)256)
        rc = (short)1;
    Util.setShort(EP_online, (short)0, rc);

    //TAC的计算
    short length, num;
    num = keyfile.findKeyByType((byte)0x34);
    length = keyfile.readkey(num, pTemp16);

    if(length == 0)
        return (short)3;

    Util.arrayCopyNonAtomic(pTemp16, (short)5, pTemp82, (short)0, (short)8);

    EnCipher.xorblock8(pTemp82, pTemp16, (short)13);
    EnCipher.gmac4(pTemp82, pTemp32, (short)0x18, data);

    return (short)0;          http://blog.csdn.net/Michael1753951
}
```

回想在一开始分析的时候，**step4中，IC卡对接收到的终端机指令是怎么处理的**。

step4：IC使用同样的算法计算MAC2，如果计算结果和终端返回的MAC2一致，就说明终端的身份合法。IC卡就会执行圈存命令。同时返回TAC。

查看MAC2的产生方法：输入信息为[交易金额||交易类型标识||终端机编号||交易日期（主机）||交易时间（主机）]，秘钥为过程秘钥，加密算法为为gmac4。

查看TAC的产生方法：输入数据为[电子钱包余额（交易后）||电子钱包联机交易序号（加 1 前）||交易金额||交易类型标识||终端机编号||交易日期（主机）||交易时间（主机）]，密钥为TAC密码最左8个字节与TAC密码最右8个字节异或的结果。

通过源代码和步骤分析我们可以得出，`EPFile.load`整个过程就是将step4实现的过程，不过需要注意的是，`EPFile.load`**中不少参数使用的是**`EPFile.init4load`**中存下来放在**`pTemp`**中的值，这也就从逻辑上说明为什么我们在执行消费命令前必须执行消费初始化命令，保证了安全性。**返回`purse.load`，校验异常，设置`papdu.le`，`purse.load`也就结束了。

整个阅读过程需要不断跳转代码，也需要不断在代码和PPT《04 电子钱包的功能》和word《实验3文档》。同时因为pTemp实在太多，我们最好能够在一旁做笔记，记录下来每一个pTemp存放了哪些值，以及它们的作用，这样我们才能在实现消费和查询的时候，使用起来更加方便。

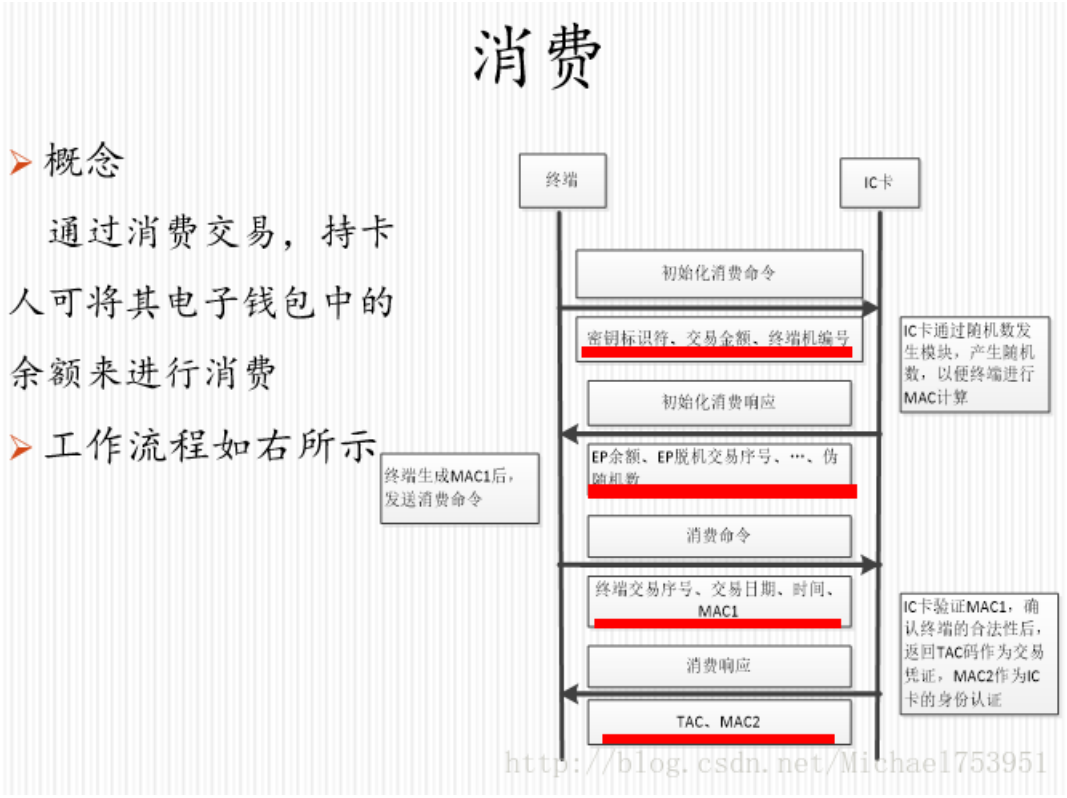
附上我记录的在init4load和load中各种变量的变化情况以及pTemp们的存在意义。

```
/**
 * 下面数据是计算时需要用到的临时过程数据
 */
    //临时计算数据
    //4个字节的临时计算数据
    private byte[] pTemp41[0:3]; //mac或者tac
    private byte[] pTemp42[0:3]; //交易金额

    //8个字节的临时计算数据
    private byte[] pTemp81[0:5]; //终端机编号
    private byte[] pTemp82[0:7]; //过程秘钥

    //32个字节的临时计算数据
    private byte[] pTemp16[0:15];//圈存秘钥
    private byte[] pTemp32; //输入数据

    init4load中
    //pTemp42 交易金额
    //pTemp81 终端机编号
    //pTemp32 查询圈存秘钥返回的序列
    //pTemp16 所查找到的圈存密钥
    //pTemp32 随机数+EP_online[0:3]+0x8000（输入）
    //pTemp82 过程秘钥
    //pTemp32 EP_balance[0:3]+data[1:4]+0x02+data[5:10]
    //pTemp41 mac1

    load中
    //pTemp32 余额+0x02+终端机编号+交易日期+交易时间
    //pTemp41 mac2
    //pTemp32 余额（交易后）+电子钱包联机交易序号+交易金额+0x02+日期+时间
    //pTemp16 所查找到的圈存密钥
```


大部分pTemp其实是有固定意义的（结合PPT中传入的各种数据长度，我们就不难理解为什么了）。**pTemp32一般作为中间变量，用来进行3des加密或者gmac加密。**

谨记这一点，结合我们在上一步中总结下来的经验。我们就可以着手实现init_purchase和purchase以及get _balance了。

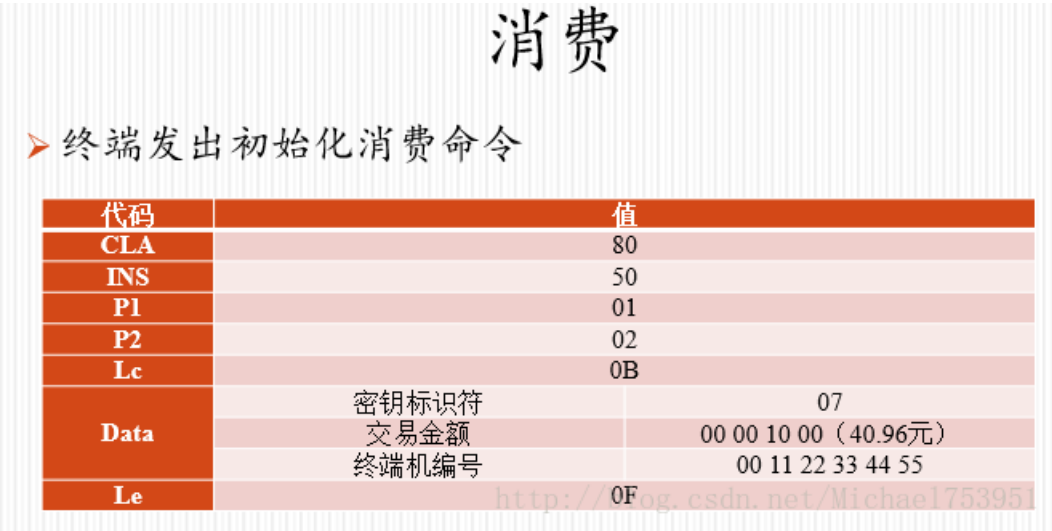
查看消费的流程图。



我们可以总结出消费的如下流程

- **终端发送消息初始化**
- **IC响应初始化，发回随机数**
- **终端产生MAC1，证明自己的身份，将交易信息发给IC卡**
- **IC卡验证终端机的合法性，计算MAC2和TAC，返回给终端作为身份凭证和消费凭证**

然后一步一步详细分析。



step1：圈存机发送的初始信息如下所示。消息中包含了**密钥标识符，交易金额，终端机编号**。（整个消息串中，除了p1、le对比圈存初始化指令有区别以外，并没有其他区别）。

IC 卡对初始化消费命令进行处理

1、IC 卡根据密钥标识符，在密钥文件中查找该密钥标识符对应的消费密钥，如果找不到，就返回状态字“9403”，表示不存在相对应的密钥。如果找到的话，就进行以下的处理

2、IC 卡生成随机数后，进行以下处理

3、IC 卡检查电子钱包余额是否大于或等于交易金额。如果小于交易金额，则回送状态字 9401，表示资金不足。（注：这一步，我并没有完成，这也将在其后的设计中，进一步完善）

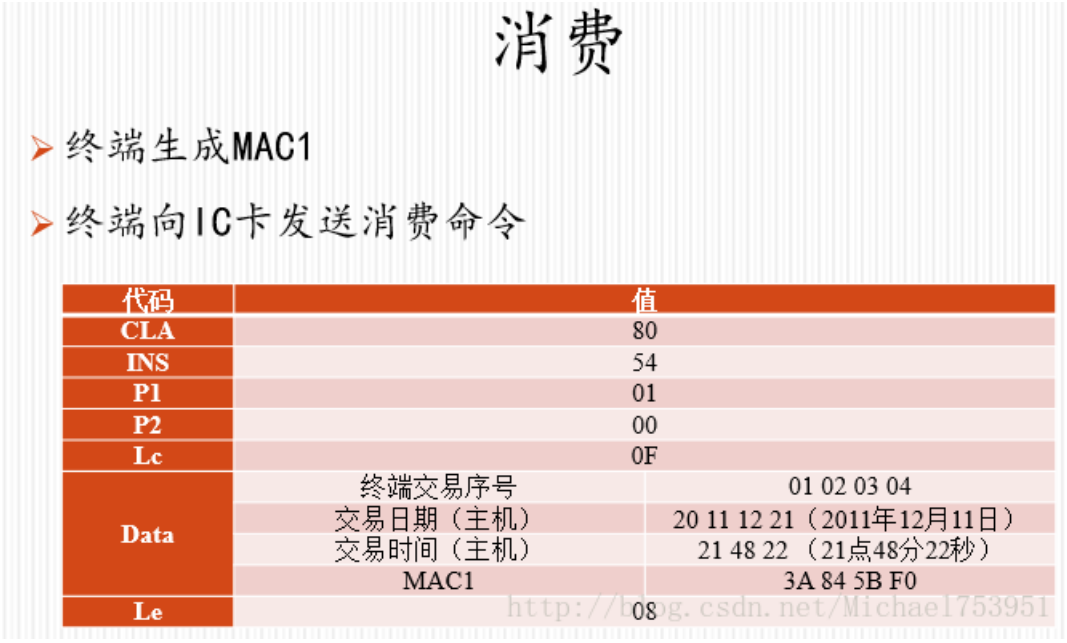
4、在进行这些操作后，IC 卡将返回相应的数据

(注：在我的设计中，电子钱包余额以 4 个字节来进行存储)

说明	值
EP 余额	00·00·10·00
EP 脱机交易序列号	00·00
透支限额	00·00·00
密钥版本号 DPK	01
算法标识 DPK	00
伪随机数（IC 卡）	29·88·AE·5A

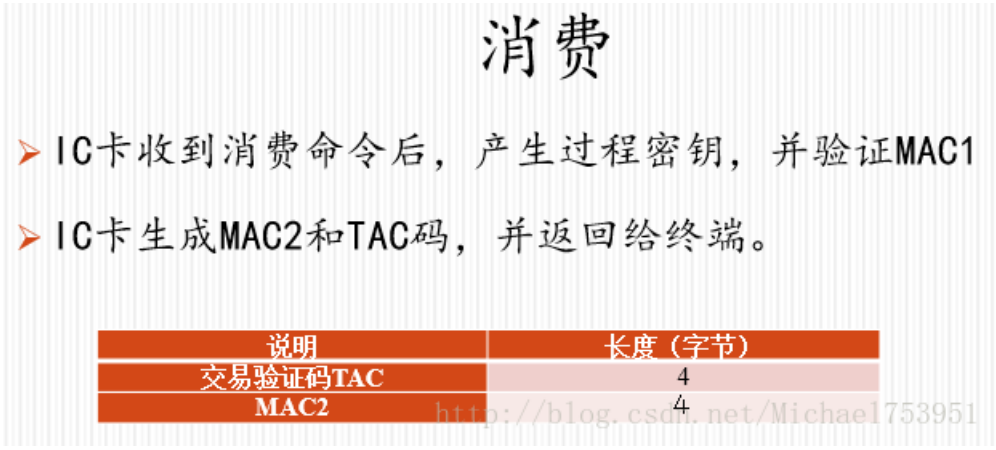
step2：

- **IC卡根据秘钥标识符寻找圈存秘钥**
- **生成随机数**
- **检查余额是否足够支付本次交易**
- **返回[余额||脱机交易序号||透支限额||秘钥版本号||算法标识||伪随机数]**



代码	值	
CLA	80	
INS	54	
P1	01	
P2	00	
Lc	0F	
Data	终端交易序号	01 02 03 04
	交易日期（主机）	20 11 12 21（2011年12月11日）
	交易时间（主机）	21 48 22 （21点48分22秒）
	MAC1	3A 84 5B F0
Le	08	

step3: **终端将命令响应数据传送给主机，主机利用消费主密钥产生消费子密钥，并生成MAC1。然后终端向IC发送[交易序列号||交易日期||交易时间||MAC1]。（和圈存中发送的信息中多了一个交易序列号）。**



说明	长度（字节）
交易验证码TAC	4
MAC2	4

IC 卡收到消费命令后，利用所查找到的密钥产生过程密钥。其输入的数据为伪随机数||电子钱包脱机交易序号||终端交易序号的最右两个字节，密钥为所查找到的消费密钥。
IC 卡利用过程密钥生成 MAC1。其输入数据为交易金额||交易类型标识(0x06)||终端机编号||交易日期（主机）||交易时间（主机）。与消费命令传送的 MAC1 进行比较，如果相同，则 MAC1 有效
IC 卡将电子钱包脱机交易序号加 1，并且把电子钱包的余额减去交易金额。
在进行上述处理后，IC 卡利用过程密钥生成 MAC2。其输入数据为交易金额，密码为过程密钥。
IC 卡生成 TAC 码。TAC 码的生成方式和 MAC 码的生成方式一致。其输入的数据：交易金额||交易类型标识||终端机编号||终端交易序号||交易日期（主机）||交易时间（主机）。密钥为 TAC 密码最左 8 个字节与 TAC 密码最右 8 个字节异或的结果。
、→IC 卡将 TAC 码和 MAC2 返回给终端。至此，IC 卡端的消费交易已经完成了。

说明	长度（字节）
交易验证码 TAC	4
MAC2	4

step4：

- **IC卡根据消费秘钥生成过程秘钥**
- **利用过程秘钥生成MAC1**
- **交易序列号+1，将消费金额从卡中扣除。**
- **IC卡生成MAC2（证明自己身份），和TAC（证明工作完成）。**
- **返回TAC和MAC2给终端**

首先我们填`purse`中的`init_purchase`函数。（这一步可以先从`init _load`中复制下来，然后稍微修改一下`p1`和`le`的值，以及调用的函数改成`init4purchase`，原因我刚刚在`step1`中说过了）。

```
/*
 * 功能：消费初始化的实现
 */
private boolean init_purchase(){
    short num,rc = 0;

    if(papdu.cla != (byte)0x80)
        ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

    if(papdu.ins != (byte)0x50)
        ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);

    //p1应该为0x01，p2应该为0x02
    if(papdu.p1 != (byte)0x01 && papdu.p2 != (byte)0x02)
        ISOException.throwIt(ISO7816.SW_WRONG_P1P2);

    if(papdu.lc != (short)0x0B)
        ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

    //根据tag寻找密钥返回密钥的记录号
    if(EPfile == null)
        ISOException.throwIt(ISO7816.SW_FILE_NOT_FOUND);

    // 如果EPfile存在，就不用在判断Keyfile是否存在了，因为Keyfile存在是
    num = keyfile.findkey(papdu.pdata[0]);

    /* 找不到相应的秘钥
     * IC卡根据密钥标识符，在密钥文件中查找该密钥标识符对应的消费密钥
     * 如果找不到，就返回状态字“9403”，表示不存在相对应的密钥。
     * 如果找到的话，就进行以下的处理
     */
    if(num == 0x00)
        ISOException.throwIt(ISO7816.SW_RECORD_NOT_FOUND);

    //调用默认的init4purchase，其他的类似
    rc = EPfile.init4purchase(num, papdu.pdata);

    if(rc == 2)
        ISOException.throwIt((condef.SW_BALANCE_NOT_ENOUGH));

    //init_purchase初始化期待的LE为0x0F，而不是0x10
    papdu.le = (short)0x0F;

    return true;
}
```


接着我们需要实现 `EPFile.init4purchase`。同样的，我们只需要按照step2步骤，从 `init4load` 中，按需复制代码就行。

```
/*
 * 功能：消费初始化命令的完成
 * 参数：num 密钥记录号； data 命令报文中的数据段
 * 返回：0 命令执行成功；2 消费超额
 */
public final short init4purchase(short num, byte[] data){
    short length,rc;

    //pTemp42用来存放交易金额
    //pTemp81用来存放终端机编号
    Util.arrayCopyNonAtomic(data, (short)1, pTemp42, (short)0, (short)4);  //
    Util.arrayCopyNonAtomic(data, (short)5, pTemp81, (short)0, (short)6);  //

    /*
     * 判断是否超额圈存
     * IC卡检查电子钱包余额是否大于或等于交易金额
     * 如果小于交易金额，则回送状态字2，表示资金不足。
     */
    rc = increase(pTemp42, false);
    if(rc != (short)0)
        return (short)2;

    /*
     * 密钥获取
     *
     * keyfile中显示，取到的秘钥存在pTemp32中，返回伪所查找到的圈存密钥的长度
     * pTemp32结构：5个字节的密钥头+16个字节的密钥值
     * 具体结构为前3个byte未知，密钥版本号 1byte，算法标识 1byte，所查找到的圈存密
     * keyID密钥版本号
     * algID算法标识符
     * pTemp16所查找到的圈存密钥
     */
    length = keyfile.readkey(num, pTemp32);
    keyID = pTemp32[3];
    algID = pTemp32[4];
    Util.arrayCopyNonAtomic(pTemp32, (short)5, pTemp16, (short)0, length);

    /*
     * 产生随机数
     * RandData.GenerateSecureRnd()函数功能，产生一个4bytes的随机数
     * RandData.getRndValue(pTemp32, (short)0)功能，将随机数写进pTemp32[0:3]
     */
    RandData.GenerateSecureRnd();
    RandData.getRndValue(pTemp32, (short)0);

    /*
     * IC卡生成随机数后，进行以下处理
     * 在进行这些操作后，IC卡将返回相应的数据
     * 余额4bytes | 脱机交易序号2bytes | 透支限额3bytes | 密钥版本 1byte | 算法标
     */
    byte[] overdraft = {0x00, 0x00, 0x00};
    Util.arrayCopyNonAtomic(EP_balance, (short)0, data, (short)0, (short)4);
    Util.arrayCopyNonAtomic(EP_offline, (short)0, data, (short)4, (short)2);
    Util.arrayCopyNonAtomic(overdraft, (short)0, data, (short)6, (short)3);
    data[9] = keyID;
    data[10] = algID;
    Util.arrayCopyNonAtomic(pTemp32, (short)0, data, (short)11, (short)4);

    return 0;

}
```


同时我们需要对照并且记录下来，每一个pTemp使用是否正确，还要记录init4purchase中每一步执行完成以后pTemp中数据对应的值。

```
init4purchase
//pTemp42 用来存放交易金额
//pTemp81 用来存放终端机编号
//pTemp32 查询圈存秘钥返回的序列
//pTemp16 所查找到的圈存密钥
//pTemp32 放随机数
```


然后我们再实现 `EPFile.purchase` 方法。实现过程比较复杂，但是每执行一步，就更新所有参数的状态，**按照我们之前整理出来的 `pTemp` 用法，使用复制粘贴大法，从 `EPFile.init4load` 和 `EPFile.load` 中可以提取大量的重复代码下来。**

需要注意的是， `load` 中是余额加上交易金额， `purchase` 中就应该是减去交易金额。因此**我们必须手动实现 `decrease` 方法。**

```
/*
 * 功能：电子钱包金额减少
 * 参数：data 消费的金额；flag 是否真正扣减电子钱包余额
 * 返回：消费是否超额
 */
public final short decrease(byte[] data, boolean flag){
    short i, t1, t2, dec;

    dec = (short)0;
    for(i = 3; i >= 0; i --){
        t1 = (short)(EP_balance[(short)i] & 0xFF);  //取出一个字节
        t2 = (short)(data[i] & 0xFF);               //取出一个字节

        t1 = (short)(t1 - t2 - dec);                //对一个字节进行加法运算
        if(flag)
            EP_balance[(short)i] = (byte)((t1+256) % 256);//更新
        dec = (short) (t1<0?1:0);                   //借位标记，此处网上流传代码有误
    }
    return dec;
}
```


实现方法和 `increase` 类似，从后向前，一次取1byte，进行减法，同时更新借位，最后返回借位即可。

这里有一个小插曲，在计算MAC2的时候，一般不会出什么问题，但是在计算TAC的时候，就会出问题，这个坑在我最后debug的时候才发现（为了找这个bug整整debug了一个多钟= =）。于是我重新改变策略，使用在MAC2生成并且存进data中以后，将pTemp32新开一个数组（JCSystem.makeTransientByteArray），然后重新作为中间变量进行操作，但是不知道为什么，这样操作的结果还是错误的。于是就按照网上流传的代码，在计算TAC的时候，新申明了一个temp数组用来作为临时变量。

最终实现的代码如图。

```
/*
 * 功能：消费命令的实现
 * 参数：data 命令报文中的数据段
 * 返回：0 命令执行成功； 1 MAC校验错误 2 消费超额； 3 密钥未找到
 */
public final short purchase(byte[] data){
    short rc;
    /*
     * IC卡收到圈存命令后，用所查找到的密钥产生过程密钥
     * 其输入数据为伪随机数||电子钱包脱机交易序号||终端交易序号的最右两个字节
     * 密钥为所查找到的消费密钥
     *
     * 在init4load中，将pTemp32 4bytes用来存放随即变量，将其写进pTemp32[0:3]
     * 电子钱包脱机交易序号2bytes在EP_offline中，将其写进pTemp32[4:5]
     * 终端交易序号在data中，取出就行
     *
     * 对输入pTemp32[0:7]使用消费秘钥pTemp16[0:4]得到过程秘钥，写进pTemp82
     */

    //Util.arrayCopyNonAtomic(pTemp32, (short)0, pTemp32, (short)0, (short)4)
    Util.arrayCopyNonAtomic(EP_offline, (short)0, pTemp32, (short)4, (short)2
    Util.arrayCopyNonAtomic(data, (short)2, pTemp32, (short)6, (short)2);
```

```
//生成过程秘钥
EnCipher.gen_SESPK(pTemp16, pTemp32, (short)0, (short)8, pTemp82, (short)

/*
 * IC卡利用过程密钥生成MAC1。
 * 输入数据：交易金额||交易类型标识(0x06)||终端机编号||交易日期（主机）||交易
 * 与消费命令传送的MAC1进行比较，如果相同，则MAC1有效,如果不一致就返回0x01
 */
Util.arrayCopyNonAtomic(pTemp42, (short)0, pTemp32, (short)0, (short)4);
pTemp32[4] = (byte)0x06;
Util.arrayCopyNonAtomic(pTemp81, (short)0, pTemp32, (short)5, (short)6);
Util.arrayCopyNonAtomic(data, (short)4, pTemp32, (short)11, (short)7);
EnCipher.gmac4(pTemp82, pTemp32, (short)0x12, pTemp41);

if(Util.arrayCompare(data, (short)11, pTemp41, (short)0, (short)4) != (by
    return (short)0x01;    //不相同则返回1

 // IC卡将电子钱包脱机交易序号加1
rc = Util.makeShort(EP_offline[0], EP_offline[1]);
rc ++;
if(rc > (short)256)
    rc = (short)1;
Util.setShort(EP_offline, (short)0, rc);

//并且把电子钱包的余额减去交易金额
rc = decrease(pTemp42, true);
if(rc != (short)0)
    return 2;
```



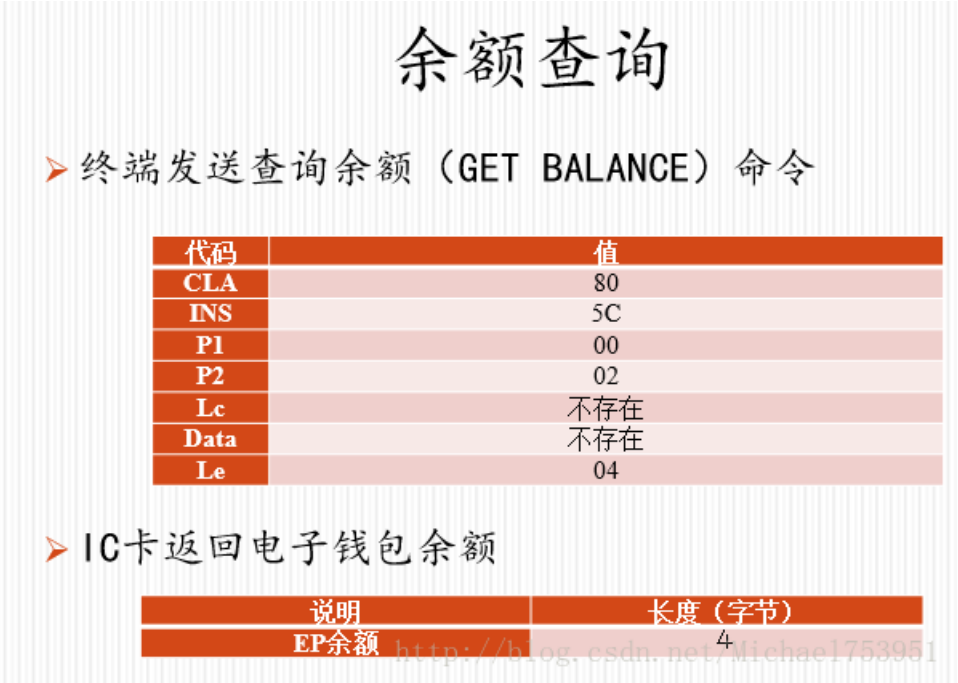
```
/*
 * 在进行上述处理后，IC卡利用过程密钥生成MAC2。
 * 其输入数据为交易金额，密码为过程密钥。
 */
//byte[] temp = JCSystem.makeTransientByteArray((short)8, JCSystem.CLEAR_
//Util.arrayCopyNonAtomic(pTemp16, (short)13, temp, (short)0, (short)8);/
//pTemp32 = JCSystem.makeTransientByteArray((short)32, JCSystem.CLEAR_ON_
Util.arrayCopyNonAtomic(pTemp42, (short)0, pTemp32, (short)0, (short)4);
EnCipher.gmac4(pTemp82, pTemp32, (short)0x4, pTemp41); //pTemp32被拓展

Util.arrayCopyNonAtomic(pTemp41, (short)0, data, (short)4, (short)4); //

/*
 *  IC卡生成TAC码。
 * TAC码的生成方式和MAC码的生成方式一致。
 * 输入：交易金额||交易类型标识||终端机编号||终端交易序号||交易日期（主机）||
 * 密钥为TAC密码最左8个字节与TAC密码最右8个字节异或的结果。
 */
//byte[] pTemp32 = JCSystem.makeTransientByteArray((short)32, JCSystem.CL
Util.arrayCopyNonAtomic(pTemp42, (short)0, pTemp32, (short)0, (short)4);
pTemp32[4] = 0x06;
Util.arrayCopyNonAtomic(pTemp81, (short)0, pTemp32, (short)5, (short)6);
Util.arrayCopyNonAtomic(data, (short)0, pTemp32, (short)11, (short)4);
Util.arrayCopyNonAtomic(data, (short)4, pTemp32, (short)15, (short)7);

short length, num;
num = keyfile.findKeyByType((byte)0x34);//为什么只找0x34？？？？？
length = keyfile.readkey(num, pTemp16);

if(length == 0)
    return (short)3;

//取秘钥的前8位，写进pTemp82[0:7]，覆盖掉过程秘钥
Util.arrayCopyNonAtomic(pTemp16, (short)5, pTemp82, (short)0, (short)8);/
//key再搞搞事情，计算tac，写进data
EnCipher.xorblock8(pTemp82, pTemp16, (short)13);//密钥左8位和右8位异或得到


//得到tac同时返回tac给终端
byte[] temp = JCSystem.makeTransientByteArray((short)8, JCSystem.CLEAR_ON
Util.arrayCopyNonAtomic(pTemp16, (short)13, temp, (short)0, (short)8);//p

//EnCipher.gmac4(pTemp82, pTemp32, (short)0x22, pTemp41);//得到tac，这个地
EnCipher.gmac4(temp, pTemp32, (short)22, data);//得到tac直接复制给data返回
//Util.arrayCopyNonAtomic(pTemp41, (short)0, data, (short)0, (short)4);
```

```
    //ISOException.throwIt((short) 0x1234);
    return 0;
}
```

参数变化的记录如图。
====更新，findKeyByType会传入0x34作为参数是因为在《实验2文档》已经说明，0x34是TAC秘钥的标识符。而我们本次操作就是为了生成TAC，故而才会使用0x34作为参数。

```
purchase
//pTemp32 输入
//pTemp82; //过程秘钥
//pTemp41 mac1
//pTemp16 返回的秘钥序列
//pTemp82 tac圈存序列
//pTemp82 异或得到的
```

最后一个就是`get_balance`了。



直接从`EP_balance`里面读出来就行了，一行代码搞定。

```
/*
 * 功能：电子钱包余额获取
 * 参数：data 电子钱包余额的缓冲区
 * 返回：0
 */
public final short get_balance(byte[] data){
    Util.arrayCopyNonAtomic(EP_balance, (short)0, data, (short)0, (short)4);
    return 0;
}
```

到这里，整个实验中需要我们填写的功能函数全部填写完成了。试验过程中，了解业务逻辑是一个很重要的过程，阅读源码，了解源码中每一个函数的具体意义，以及每一个参数中，存放的信息有哪些，这些都最好都在一个txt或者什么地方器记录下来，方便自己查阅和提取（毕竟一个没有debug的IDE，你不能要求什么，只能人脑debug）.另一方面，每实现一个功能，就尽可能写清楚这个部分的功能，同时写清楚那些信息在哪些位置，这样你在实现的过程中才不会头晕脑胀。也能够方便你最后debug找错误信息从哪来。

验证实验

验证之前，我们还需要添加新的参数信息。

```
package purse;
/**
 * 已给出部分常数值，其他根据需要自行添加
 */
public class condef {
    //----------------- INS Byte -------------------
    final static byte INS_CREATE_FILE    = (byte)0xE0;   //文件建立命令的INS值
    final static byte INS_WRITE_KEY      = (byte)0xD4;   //写入密钥命令的INS值
    final static byte INS_WRITE_BIN      = (byte)0xD6;   //写入二进制命令的INS值
    final static byte INS_READ_BIN       = (byte)0xB0;   //读取二进制命令的INS值
    final static byte INS_NIIT_TRANS     = (byte)0x50;   //初始化圈存和初始化消费命令的INS值
    final static byte INS_LOAD           = (byte)0x52;   //圈存命令的INS值
    final static byte INS_PURCHASE       = (byte)0x54;   //消费命令的INS值
    final static byte INS_GET_BALANCE    = (byte)0x5C;   //查询余额命令的INS值

    //－－－－－－－－－－－－－－－－  FILE TYPE Byte ----------------
    final static byte KEY_FILE           = (byte)0x3F;   //密钥文件的文件类型
    final static byte CARD_FILE          = (byte)0x38;   //应用基本文件的文件类型
    final static byte PERSON_FILE        = (byte)0x39;   //持卡人基本文件 的文件类型
    final static byte EP_FILE            = (byte)0x2F;   //电子钱包文件的文件类型

    //------------------------- SW ----------------------
    final static short SW_LOAD_FULL          = (short)0x9501;   //圈存超额
    final static short SW_BALANCE_NOT_ENOUGH = (short)0x9401;   //金额不足
}
```


以及一个的天坑的地方。。。。。。。这个地方我用throw 0x1234大法，从init_purchase的return 0之前，一直回退到Purse中的if(papdu.APDUContainData())才找到问题。

```
/*
 * 功能：判断APDU命令是包含数据
 * 参数：无
 * 返回：APDU命令包含数据的判断
 */
public boolean APDUContainData(){
    switch(ins){
    case condef.INS_CREATE_FILE:
    case condef.INS_LOAD:
    case condef.INS_NIIT_TRANS:
    case condef.INS_WRITE_KEY:
    case condef.INS_WRITE_BIN:
    case condef.INS_PURCHASE: // 真-x了狗了

        return true;
    }
    return false;
}
```


其他的地方应该没什么需要添加或者修改了，按照之前上一次的试验方法，我们可以开始进行debug了。

首先还是新建一个txt，因为我们需要人脑担任终端机的角色，所以**提前将必要的操作流程和模板写下来存在txt里面**，写指令的时候也能迅速一点。

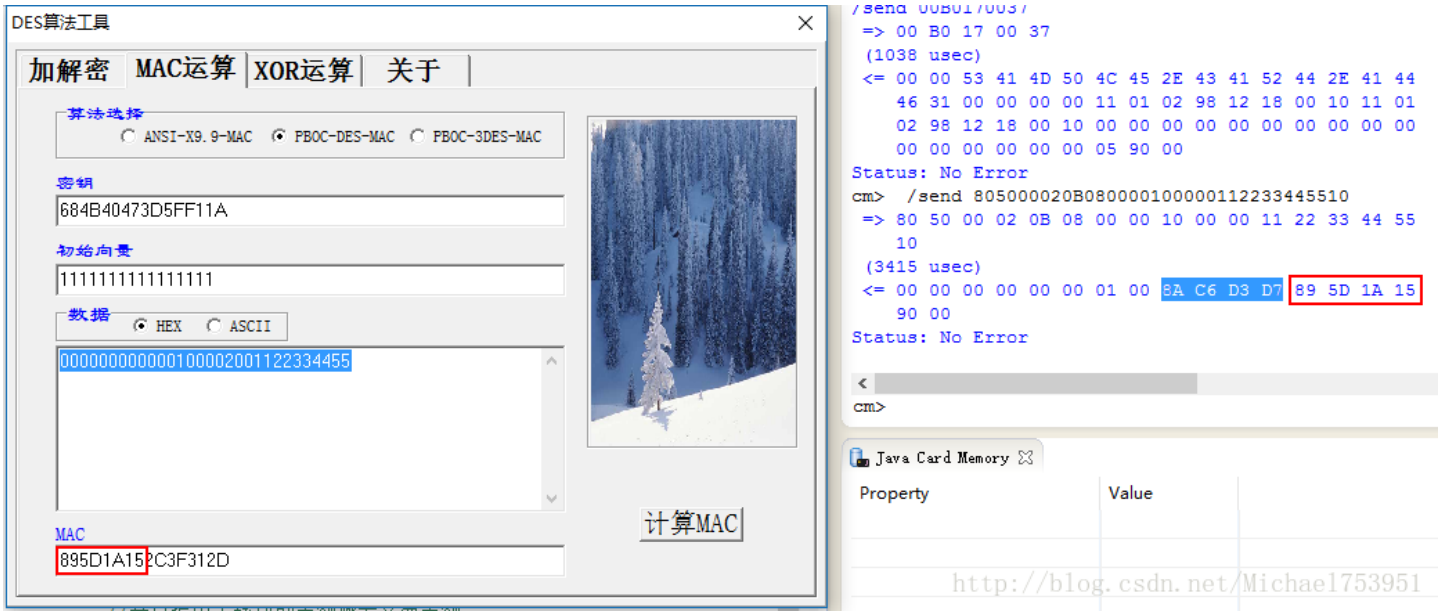
```
#//选择电子钱包文件
/select 1235318401

#//建立密钥文件
/send 80e00000073fffffffffffff

#//建立发卡方基本信息文件
/send 80e000160738001effffffff

#//建立持卡人基本信息文件
/send 80e0001707390037ffffffff

#//建立电子钱包文件
/send 80e00018072Fffffffffffff

#//写入密钥文件
#//增加消费密钥
/send 80d40007153ef0f0010009F4ACB09131420B8FE1B4CC007AC52B
#//增加圈存密钥
/send 80d40008153ff0f00100EB9BC6DCDF74FF4E4B43F2E34A6727B6
#//增加TAC密钥
/send 80d400061534f0f09000CEB726EDC01B793BC37DC09E2F768534

#//写入发卡方基本信息
/send 00D616001E626400223333000103010001200108170000000120010101200112315566

#//写入持卡人基本信息
/send 00D617003700053414D504C452E434152442E414446310000000011010298121800101101029812
8001000000000000000000000000000000000005

#//读取发卡方基本信息
/send 00B016001E

#//读取持卡人基本信息
/send 00B0170037

load指令
#//init_load, 001122334455号机准备圈存40.96元
/send 80500002080800010000011223344551O
#返回值余额4bytes | 联机交易序列号2bytes | 秘钥版本号1byte | 算法标识1byte |
伪随机数4bytes | mac14bytes
#eg: 00 00 00 00 | 00 00 | 01 | 00 | [伪随机数] | [mac1]
    90 00
#圈存秘钥：EB9BC6DCDF74FF4E4B43F2E34A6727B6
#验证：输入数据[伪随机数]00008000，得到过程秘钥


#/终端机发回确认，认识证明自己
#mac2输入数据为交易金额||交易类型标识||终端机编号||交易日期（主机）||
交易时间（主机）。密钥为过程密钥。
#mac2输入数据为00001000020011223344552011122121 4822
/send 80520000B20111221214822【mac2】04

#/生成tac的输入
00001000000000001000020011223344552011122121 4822

purchase指令
#//初始化
/send 80500102OB070000100000112233445 50F
#返回值00 00 10 00 | 00 00 | 00 00 00 | 01 | 00 | E2 83 5C 59 90 00
# 余额 | 脱机交易序列号 | 透支额度 | 密钥版本号 | 算法标识 | 伪随机数

#验证
#过程秘钥输入的数据为伪随机数||电子钱包脱机交易序号||
终端交易序号的最右两个字节，密钥为所查找到的消费密钥
#输入：【伪随机】 | 00 00 | 03 04
#秘钥：09F4ACB09131420B8FE1B4CC007AC52B
#接下来IC卡利用过程密钥生成MAC1。其输入数据为交易金额||交易类型标识(
0x06)||终端机编号||交易日期（主机）||
交易时间（主机）。与消费命令传送的MAC1进行比较，如果相同，则MAC1有效
#输入00001000060011223344552011122121 4822
#计算出来以后
/send 805401000F0102030420111221214822【mac1】08

#/获取余额
/send 805C000204
```


首先实验一中的初始化脚本。

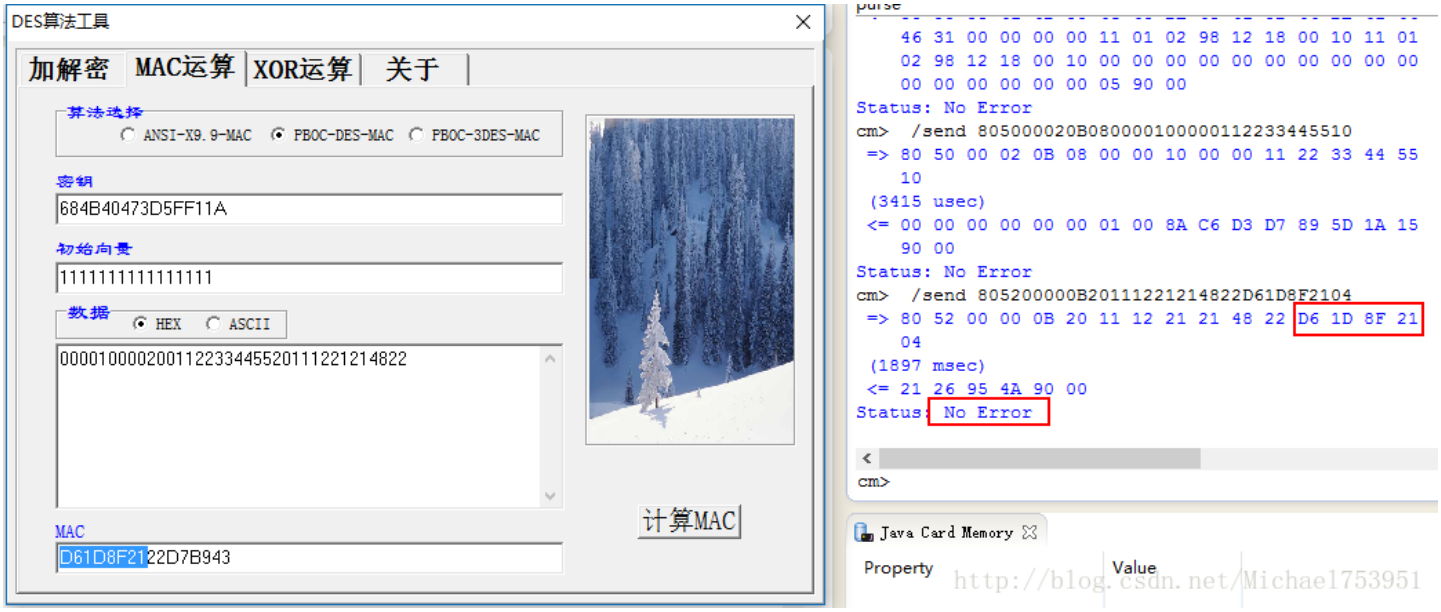
然后模仿终端机输入圈存初始化指令。`/send 80500002OB080000100000112233445510`

使用**[伪随机数||电子钱包联机交易序号||8000]**作为数据，**圈存秘钥**作为秘钥，使用**3DES**获得**过程秘钥**。

然后**使用过程秘药作为秘钥**，输入自定义好的初始向量，以及数据**[电子钱包余额（交易前）||交易金额||交易类型标识(0x02)||终端机编号]**作为输入，生成**mac1**，和**IC卡返回的mac1校对，发现一致**。



接着使用**[电子钱包余额（交易前）||交易金额||交易类型标识(0x02)||终端机编号]**作为数据，**过程秘钥**作为秘钥，生成mac2。夹杂其他信息一起发出去。

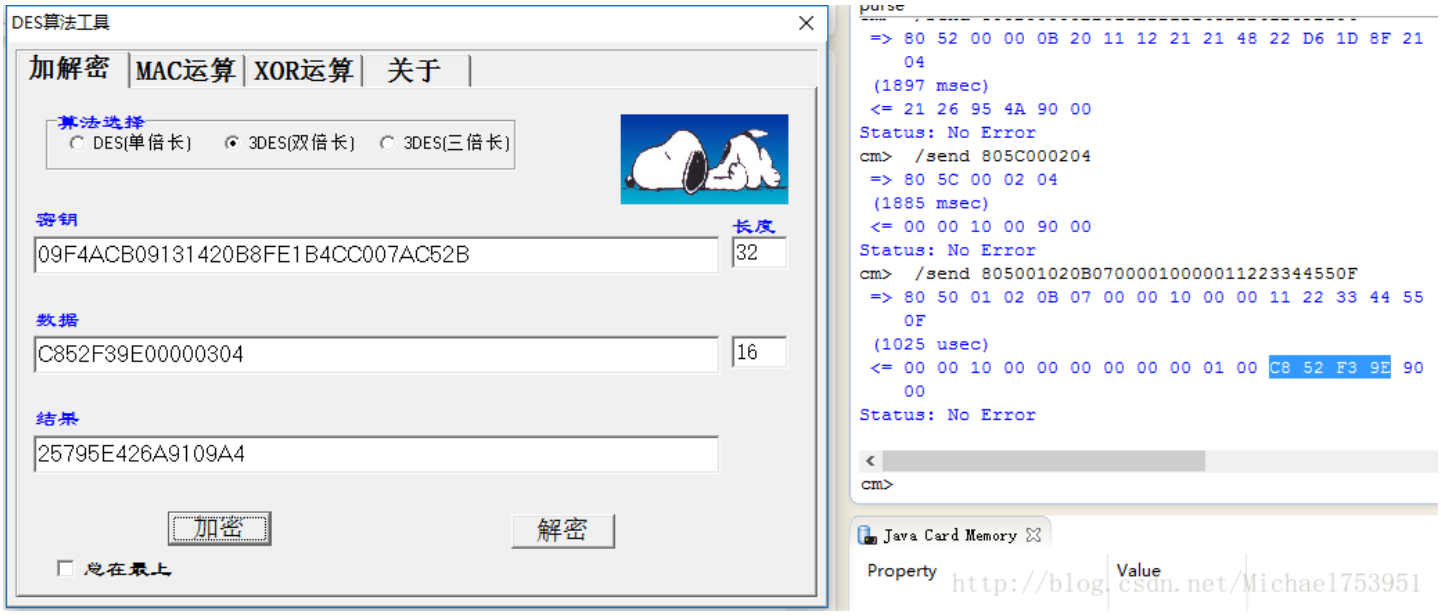


返回`TAC+9000`，所以我们认为`init_load`和`load`成功。

接下来执行查询指令，返回`00 00 10 00 90 00`。说明卡里面已经有了`00001000`，我们圈存确实成功了。查询也成功了。

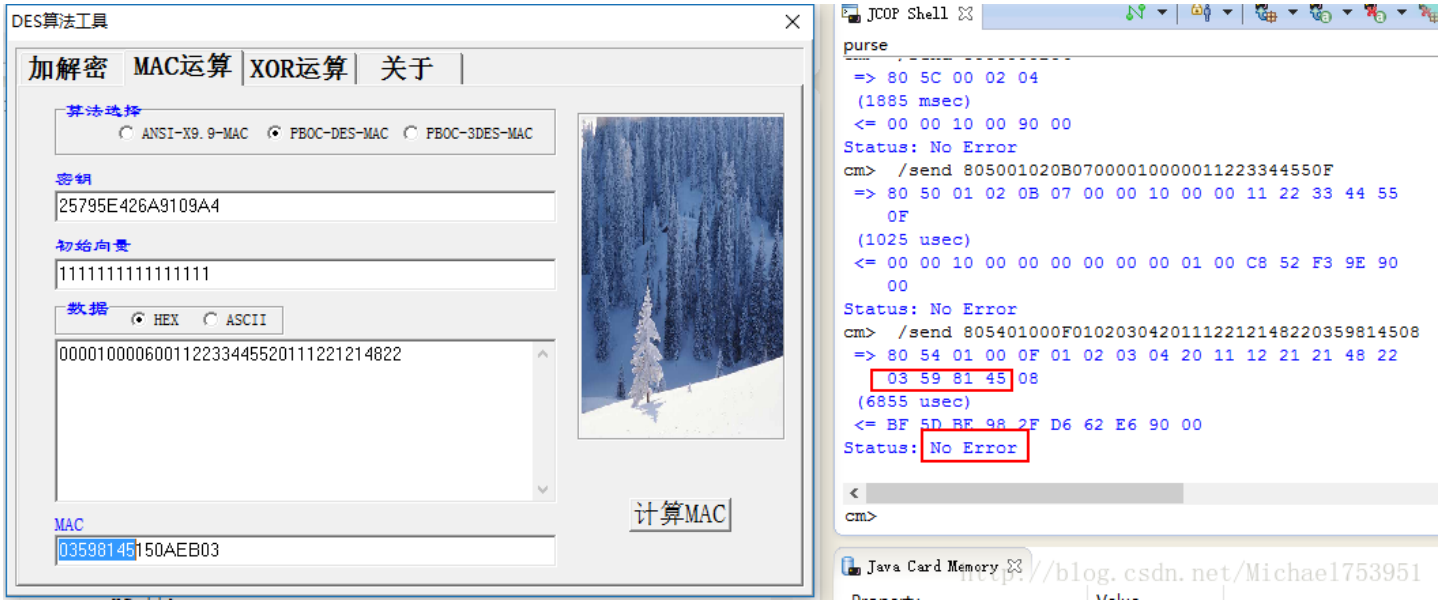
```
cm>  /send 805C000204
 => 80 5C 00 02 04
 (1885 msec)
 <= 00 00 10 00 90 00
Status: No Error
```

接下来执行消费指令，`/send 805001020B0700001000001122334455 0F`



返回得到伪随机数。接着我们将**[伪随机数||电子钱包脱机交易序号||终端交易序号的最右两个字节]**作为数据，**消费秘钥**作为秘钥，使用**3DES**得到**圈存秘钥**。

使用**[交易金额||交易类型标识(0x06)||终端机编号||交易日期（主机）||交易时间（主机）]**作为数据，**过程秘钥**最为秘钥生成**mac1**。



返回`MAC2+TAC+9000`。故而我们认为本次`init_purchase`和`purchase`成功。

为了防止意外，我们再次执行查询余额的指令。

```
 => 80 50 01 02 0B 07 00 00 10 00 00 11 22 33 44 55    .P..........."3
    0F                                                 .
 (1025 usec)
 <= 00 00 10 00 00 00 00 00 00 01 00 C8 52 F3 9E 90    ............R.
    00                                                 .
Status: No Error
cm>  /send 805401000F01020304201112212148220359814508
 => 80 54 01 00 0F 01 02 03 04 20 11 12 21 21 48 22    .T....... ..!!H"
    03 59 81 45 08                                     .Y.E.
 (6855 usec)
 <= BF 5D BE 98 2F D6 62 E6 90 00                      .]../.b...
Status: No Error
cm>  /send 805C000204
 => 80 5C 00 02 04                                     .\...
 (623778 nsec)
 <= 00 00 00 00 90 00                                  ......
Status: No Error
```


结果显示余额为`00000000`，返回`9000`。故而我们认为，查询消费圈存三大功能都已经能够正确执行了。

```
 => 80 54 01 00 0F 01 02 03 04 20 11 12 21 21 48 22    .T....... ..!!
    88 74 51 40 08                                     .tQ@.
 (4586 usec)
 <= 69 82                                              i.
Status: Security condition not satisfied
cm>  /send 805001020B0700000001001122334455OF
 => 80 50 01 02 0B 07 00 00 00 01 00 11 22 33 44 55    .P..........."3
    0F                                                 .
 (907728 nsec)
 <= 00 00 20 00 00 01 00 00 00 01 00 33 9A B5 C3 90    .. ........3..
    00                                                 .
Status: No Error
cm>  /send 805401000F010203042011122121482220D5EF77A08
 => 80 54 01 00 0F 01 02 03 04 20 11 12 21 21 48 22    .T....... ..!!
    0D 5E F7 7A 08                                     .^.z.
 (7874 usec)
 <= B5 96 19 12 F0 B7 DC 70 90 00                      .......p..
Status: No Error
cm>  /send 805C000204
 => 80 5C 00 02 04                                     .\...
 (573602 nsec)
 <= 00 00 1F FF 90 00                                  ......
Status: No Error
```


最后测试一下**多次存取操作**。没什么问题，故而我们认为，本次实验成功。

2017/4/25

更新，线下已校验多次存取（复杂数据），以及TAC码（我喜欢称其工单码）检验。经检验，不存在明显问题。