# 华为软件编程规范培训实例与练习

- 问题分类

1　　逻辑类问题（A类）－指设计、编码中出现的计算正确性和一致性、程序逻辑控制等方面出现的问题，在系统中起关键作用，将导致软件死机、功能正常实现等严重问题；

接口类问题（B类）－指设计、编码中出现的函数和环境、其他函数、全局/局部变量或数据变量之间的数据/控制传输不匹配的问题，在系统中起重要作用，将导致模块间配合失效等严重问题；

维护类问题（C类）－指设计、编码中出现的对软件系统的维护方便程度造成影响的问题，在系统中不起关键作用，但对系统后期维护造成不便或导致维护费用上升；

可测试性问题（D类）－指设计、编码中因考虑不周而导致后期系统可测试性差的问题。

- 处罚办法

问题发生率：

$$P=D/S$$

$$D=D_A+0.5D_B+0.25D_C$$

其中：

P　－问题发生率

D　－1个季度内错误总数

$D_A$ －1个季度内A类错误总数

$D_B$ －1个季度内B类错误总数

$D_C$ －1个季度内C类错误总数

S　－1个季度内收到问题报告单总数

1）当D≥3时，如果P≥3%，将进行警告处理，并予以公告；

2）当D≥5时，如果P≥5%，将进行罚款处理，并予以公告。

# 目　录

## 二、接口类代码问题



## 三、维护类代码问题



## 四、产品兼容性问题

## 五、版本控制问题



## 六、可测试性代码问题

# 一、逻辑类代码问题

## 1、变量/指针在使用前就必须初始化

**【案例1.1.1】**

C语言中最大的特色就是指针。指针的使用具有很强的技巧性和灵活性，但同时也带来了很大的危险性。在XXX的代码中有如下一端对指针的灵活使用：

... ...

```
_UC *puc_card_config_tab;
```

... ...

```
Get_Config_Table( AMP_CPM_CARD_CONFIG_TABLE,
                  &ul_card_config_num,
                  &puc_card_config_tab,
                  use_which_data_area
                );
```

... ...

```
b_middle_data_ok = generate_trans_middle_data_from_original_data(
                                puc_card_config_tab,
                                Ul_card_config_num)
```

.... ...

其中红色部分巧妙的利用指向指针的指针为指针puc_card_config_tab赋值,而在兰色部分使用该指针。但在Get_Config_Table函数中有可能失败返回而不给该指针赋值。因此，以后使用的可能是一个非法指针。

指针的使用是非常灵活的，同时也存在危险性，必须小心使用。指针使用的危险性举世共知。在新的编程思想中，指针基本上被禁止使用（JAVA中就是这样），至少也是被限制使用。而在我们交换机的程序中大量使用指针，并且有增无减。

2、防止指针/数组操作越界

**【案例1.2.1】**

在香港项目测试中，发现ISDN话机拨新业务号码时，若一位一位的拨至18位，不会有问题。但若先拨完号码再成组发送，会导致MPU死机。

处理过程：

查错过程很简单，按呼叫处理的过程检查代码，发现某一处的判断有误，本应为小于18的判断，写成了小于等于18。

结　论：

代码编写有误。

思考与启示：

1、极限测试必须注意，测试前应对某项设计的极限做好充分测试规划。

2、测试极限时还要注意多种业务接入点，本例为ISDN。对于交换机来说，任何一种业务都要分别在模拟话机、ISDN话机、V5话机、多种形式的话务台上做测试。对于中继的业务，则要充分考虑各种信令：TUP、ISUP、PRA、NO1、V5等等。

**【案例1.2.2】**

对某交换类进行计费测试，字冠011对应1号路由、1号子路由，有4个中继群11,12,13,14(都属于1#模块)，前后两个群分别构成自环。其中11,13群向为出中继,12,14群向为入中继，对这四个群分别进行计费设置，对出入中继都计费。电话60640001拨打01160010001两次，使四个群都有机会被计费，取话单后浏览话单发现对11群计费计次表话单出中继群号不正确，其它群的计次表中出中继群号正常。

处理过程：

与开发人员在测试组环境多次重复以上步骤，发现11群的计次表话单有时正

常，有时其出中继群号就为一个随机值，发生异常的频率比较高。为什么其它群的话单正常，唯独11群不正常呢？11群是四个群中最小的群，其中继计次表位于缓冲区的首位，打完电话后查询内存发现出中继群号在内存中是正确的，取完话单后再查就不正确了。

结　论：

话单池的一个备份指针Pool_head_1和中继计次表的头指针重合，影响到第一个中继计次表的计费。

思考与启示：

随机值的背后往往隐藏着指针问题，两块内存缓冲区的交界处比较容易出现问题，在编程时是应该注意的地方。

**【案例1.2.3】**

【正　　文】

在接入网产品A测试中，在内存数据库正常的情况下的各种数据库方面的操作都是正常的。为了进行数据库异常测试，于是将数据库内容人为地破坏了。发现在对数据库进行比较操作时，出现程序跑死了现象。

经过跟踪调试发现问题出现在如下一段代码中:

```
1    for(i=0; i<pSysHead->dbf_count; i++)
2    {
3                    pDBFat = (_NM_DBFAT_STRUC *)(NVDB_BASE +
DBFAT_OFFSET + i*DBFAT_LEN);
4        if(fat_check(pDBFat) != 0)
5        {
6                        pSysHead->system_flag = 0;
7                        head_sum();
8                        continue;
9                }
```

```
10                      if(strlen(dbf->dbf_name) != 0 && strncmp(dbf->dbf_name,
pDBFat->dbf_name, strlen(dbf->dbf_name)) == 0)
11                      {
12                          dbf_ptr1 = (_UC *)pDBFat->dbf_head;
13                          filesize = pDBFat->dbf_fsize;
14                              break;
15                  }
16          }
```

在测试时发现程序死在循环之中，得到的错误记录是"Bus Error"（总线出错），由此可以说明出现了内存操作异常。

经过跟踪变量值发现循环变量i的阀值pSysHead->dbf_count的数值为0xFFFFFFFF，该值是从被破坏的内存数据库中获取的，正常情况下该值小于127。而pDBFat是数据库的起始地址，如果pSysHead->dbf_count值异常过大，将导致pDBFat值超过最大内存地址值，随后进行的内存操作将导致内存操作越界错误，因而在测试过程中数据库破坏后就出现了主机死机的现象。

上面的问题解决起来很容易，只需在第一行代码中增加一个判断条件即可，如下：

```
for(i=0; i<pSysHead->dbf_coun && i < MAX_DB_NUM; i++)
   // MAX_DB_NUM=127
```

这样就保证了循环变量i的值在正常范围内，从而避免了对指针pDBFat进行内存越界的操作。

从上面的测试过程中，我们可以看到：如此严重的问题，仅仅是一个简单的错误引起的。实际上，系统的不稳定往往是由这些看似很简单的小错误导致的。这个问题给我们教训的是：在直接对内存地址进行操作时，一定要保证其值的合法性，否则容易引起内存操作越界，给系统的稳定性带来潜在的威胁。

**【案例1.2.4】**

近日在CDB并行测试中发现一个问题：我们需要的小区负荷话统结果总是为零，开始还以为小区负荷太小，于是加大短消息下发数量，但还为零，于是在程序中加入测试代码，把收到的数据在BAM上打印出来,

结果打印出来的数据正常,不可能为零,仔细查看相关代码,问题只可能在指针移位上有问题,果然在函数中发现一处比较隐蔽的错误。

```
/*  功能:一个BM模块内所有小区CDB侧广播消息忙闲情况          */
/*************************************************************/
void Cell_CBCH_Load_Static(struct MsgCB FAR *pMsg)
        {
        。。。

        memcpy((_UC *)&tmp_msg,pMsg,sizeof(tmp_msg));
        pMsg=pMsg+sizeof(tmp_msg);//sizeof(tmp_msg)=10;本意是想移动10个字节,可是实际上指针移动了10*sizeof(struct MsgCB)个字节;
        CellNum=tmp_msg.usCellNum;
        。。。
        }
1
```

所以结构指针传入函数后，如要进行指针移动操作，最好先将其转化为_UC型再说。总之指针操作要小心为上。

3、避免指针的非法引用

**【案例1.3.1】**

【正　　文】

在一次测试中，并没有记得做了什么操作，发现HONET系统的主机复位了，之后，系统又工作正常了。由于没有打开后台的跟踪窗口，当时查了半天没有眉目。过了半天，现象又出现了，而且这次是主机在反复复位，系统根本无法

正常工作了。

我凭记忆，判断应该是与当时正在测试的DSL板的端口配置有关。于是将板上所有端口配置为普通2B+D端口，重新加载在主机数据，现象消失。于是初步定位为主机在DSL端口处理过程中有重大错误。

我在新的数据上努力恢复原出问题的现象，却一直没有重现，于是恢复原数据，加载后立即重现。并注意到，当DSL端口激活时，主机复位。仔细比较两种数据的差别，发现出现主机复位问题的数据中DSL板配置了MNT/MLT端口，但是没有做DSL端口之间的半永久数据。

于是在程序中不断加打印语句，通过后台的DBWIN调试程序跟踪，最后终于定位为：每当执行到portdsl.c的DeviceDslMsgProc()函数中处理U口透传的

```
if ( SPC_STATE_OK == pSpcCB->bySpcState )
```

语句时，主机复位。但是该语句似乎并无不妥。

再分析整个函数，pSpcCB在函数前部分已经被赋值，

```
pSpcCB = SpcCB + (PortTable+index)->spcNo;
```

但由于得到 index 后，没有任何判断，导致若MNT/MLT端口没有做半永久，端口激活后，执行此部分函数，(PortTable+index)->spcNo 有可能为NULL_WORD，于是，运算后，pSpcCB 可能为非法值。此时主机在取进行判断，就不知会导致什么后果了。

其实，改起来很简单，只要在这两句前增加一个判断就行了。于是，修改代码为：

```
if ( (PortTable+index)->spcNo != NULL_WORD)
{
 pSpcCB = SpcCB + (PortTable+index)->spcNo;
 if ( SPC_STATE_OK == pSpcCB->bySpcState )
 {。。。}
}
```

修改后，问题不再重现。

经过分析可以发现，编译环境是有很大的容许空间的，若主机没有做充分的保护，很可能会有极严重的随即故障出现。所以编程时一定要考虑各种可能情

况；而测试中遇到此类死机问题，则要耐心的定位到具体是执行哪句代码时出现的，再进行分析。因为问题很隐蔽，直接分析海一样的代码是很难发现的。

4、变量类型定义错误

**【案例1.4.1】**

【正　　文】

在FRI板上建几条FRPVC，其DLCI类型分别为：10Bit/2bytes、10bit/3bytes、16bit/3bytes、17bit/4bytes、23bit/4bytes。相应的DLCI值为：16、234、991、126975、1234567，然后保存，重起MUX，观察PVC的恢复情况，结果DLCI值为16、234和991的PVC正确恢复，而DLCI=126975的PVC恢复的数据错误为61439，而DLCI=1234567的PVC完全没有恢复。

对于17/4类型，DLCI=126975的PVC在恢复时变成61439，根据这条线索，查找原因，发现126975-61439=65535，转化二进制就是10000000000000000，也就是说在数据恢复或保存时把原数据的第一个1给忽略了。此时第一个想法是：在程序处理中，把无符号长整型变量当作短整型变量处理了，为了证实这个判断，针对17bit/4bytes类型又重新设计测试用例：（1） 先建PVC，DLCI=65535，然后保存，重起MUX，观察PVC的恢复情况，发现PVC能够正确恢复；

（2）再建PVC，DLCI=65536，然后保存，重起MUX，观察PVC的恢复情况，此时PVC不能正确恢复。

至此基本可以断定原因就是出在这里。带着这个目的查看原代码，发现在以下代码中有问题：

```
int  _GetFrDlci( DWORD* dwDlci, char* str, DWORD dwDlciType, DWORD
dwPortType, DWORD dwSlotID, DWORD dwPortID)
{              DWORD    tempDlci;
    charszArg[80];
1   charszLine[80];
    ID  LowPVCEP;
              DWORD   dwDlciVal[5][2] =
```

```
        { {16,1007}, {16,1007}, {1024,64511},
          {2048,129023}, {131072,4194303} } ;
              . . .
}

typedef struct tagFrPppIntIWF
{
    . . .
    WORD  wHdlcPort;
    WORD  wHdlcDlci;
    WORD  wPeerHdlcDlci;
    WORD  wPeerOldAtmPort;
    . . .
}   SFrPppIntIWFData;

DWORD    SaveFrNetIntIWFData ( DWORD *pdwWritePoint )
{
    BYTE   bSlotID, bPeerSlotID;
    DWORD    dwCCID, dwPeerCCID;
    WORD  wHdlcPort, wAtmPort, wIci, wPeerIci, wPeerHdlcPort ;
    WORD  wCount;
              . . .
}

DWORD    SaveFrNetExtIWFData ( DWORD *pdwWritePoint )
{
    BYTE   bSlotID;
    DWORD    dwCCID, dwPeerCCID;
    WORD  wHdlcPort, wAtmPort, wIci ;
    WORD  wCount;
```

```
    . . .
unSevData.FrNetExtIWF[wCount].bSlotID  = bSlotID;
        unSevData.FrNetExtIWF[wCount].wHdlcPort    = wHdlcPort;
        unSevData.FrNetExtIWF[wCount].wHdlcDlci    =
gFrPVCEP[bSlotID ][ gFrPVCC[bSlotID][dwCCID].dwLoPVCEP ].dwDLCI;
        unSevData.FrNetExtIWF[wCount].wOldAtmPort      = wAtmPort;
        unSevData.FrNetExtIWF[wCount].wAtmDlci     =
gFrPVCEP[ bSlotID ][ gFrPVCC[bSlotID][dwCCID].dwHiPVCEP ].dwDLCI;
        unSevData.FrNetExtIWF[wCount].dwMapMode =
gFrPVCC[bSlotID][dwCCID].dwMapMode;
            . . .
         }


DWORD RestoreFrNetExtIWFData ( WORD wSlotID, BYTE *pReadPoint )
{
        WORD        wCount, wTotalNetIWF;
        BYTE  bSlotID, bHdlcDlciType, bAtmDlciType;
        WORD        wOldAtmPort, wAtmDlci, wHdlcPort, wHdlcDlci;
        DWORD       dwMapMode, dwCIR, dwBe;
        DWORD       dwCCID, dwResult, dwAtmPort;
        wTotalNetIWF = g_MuxData.SevDataSize.wFrNetExtIWFNum;
        . . .
}

DWORD RestoreFrHdlcIntIWFData ( WORD wSlotID, BYTE *pReadPoint )
{
    WORD  wCount, wTotalHdlcIWF;
    DWORD     dwCCID, dwPeerCCID, dwAtmPort, dwPeerAtmPort;
    DWORD     dwResult;
```

```
    BYTE    bSlotID, bPeerSlotID;
    WORD  wHdlcPort, wOldAtmPort, wCIR;
    WORD  wPeerHdlcPort, wPeerOldAtmPort;
                . . .
}
```

其中涉及DLCI值的变量都为WORD（即无符号短整型）类型，在程序的处理时，出现WORD和DWORD（无符号长整型）类型在一句中同时存在的情况，至此可以判断问题出在这里。由于DLCI值在不同类型时的取值范围不同，前三种类型的取值范围为16~991，第四种取值范围为2048~126975，第五种取值范围为131072~4194303，所以当采用前三种DLCI类型时，采用WORD类型最大值为65535，已经完全够用了；而对于第四种类型时，其取值在超过65535时，获取DLCI值的函数_GetFrDlci（）采用DWORD类型，而负责保存和恢复的两个函数SaveFrNetExtIWFData（）和RestoreFrNetExtIWFData（），都把DLCI的值当作WORD类型进行处理，因此导致DLCI取值越界，于是程序把原本为长整型的DLCI强制转换成整型，从而导致DLCI值在恢复时，比原数据小65536。而在程序运行过程中，这些数据保存在DRAM中，程序运行直接从DRAM中获取数据，程序不会出错；当FRI板复位或插拔后，需要从FLASH中读取数据，此时恢复函数的错误就表现出来。

另一个问题是为什么23/4类型的DLCI数据不能恢复？这是由于对于23/4类型的PVC，其DLCI的取值范围为：131072~4194303，而程序强制转换并恢复的数据最大只能是65535，所以这条PVC不能恢复。

至此，DLCI数据恢复出错的原因完全找到，解决的方法是将DLCI的类型改为DWORD类型。从这个案例可以看出，在程序开发中一个很低级的错误，将在实际工作中造成很严重的后果。

【案例1.4.2】

【正　　文】

在FRI板上建几条FRPVC，其DLCI类型分别为：10Bit/2bytes、

10bit/3bytes、16bit/3bytes、17bit/4bytes、23bit/4bytes。相应的DLCI值为：16、234、991、126975、1234567，然后保存，重起MUX，观察PVC的恢复情况，结果DLCI值为16、234和991的PVC正确恢复，而DLCI=126975的PVC恢复的数据错误为61439，而DLCI=1234567的PVC完全没有恢复。

对于17/4类型，DLCI=126975的PVC在恢复时变成61439，根据这条线索，查找原因，发现126975-61439=65535，转化二进制就是10000000000000000，也就是说在数据恢复或保存时把原数据的第一个1给忽略了。此时第一个想法是：在程序处理中，把无符号长整型变量当作短整型变量处理了，为了证实这个判断，针对17bit/4bytes类型又重新设计测试用例：（1） 先建PVC，DLCI=65535，然后保存，重起MUX，观察PVC的恢复情况，发现PVC能够正确恢复；
（2）再建PVC，DLCI=65536，然后保存，重起MUX，观察PVC的恢复情况，此时PVC不能正确恢复。
至此基本可以断定原因就是出在这里。带着这个目的查看原代码，发现在以下代码中有问题：

```
int  _GetFrDlci( DWORD* dwDlci, char* str, DWORD dwDlciType, DWORD
dwPortType, DWORD dwSlotID, DWORD dwPortID)
{              DWORD    tempDlci;
    charszArg[80];
    charszLine[80];
    ID  LowPVCEP;
              DWORD   dwDlciVal[5][2] =
        { {16,1007}, {16,1007}, {1024,64511},
          {2048,129023}, {131072,4194303} } ;
              . . .
}

typedef struct tagFrPppIntIWF
{
    . . .
    WORD  wHdlcPort;
```

```
    WORD  wHdlcDlci;
    WORD  wPeerHdlcDlci;
    WORD  wPeerOldAtmPort;
    . . .
}   SFrPppIntIWFData;

DWORD     SaveFrNetIntIWFData ( DWORD *pdwWritePoint )
{
    BYTE   bSlotID, bPeerSlotID;
    DWORD     dwCCID, dwPeerCCID;
    WORD  wHdlcPort, wAtmPort, wIci, wPeerIci, wPeerHdlcPort ;
    WORD  wCount;
              . . .
}

DWORD     SaveFrNetExtIWFData ( DWORD *pdwWritePoint )
{
    BYTE   bSlotID;
    DWORD     dwCCID, dwPeerCCID;
    WORD  wHdlcPort, wAtmPort, wIci ;
    WORD  wCount;
    . . .
unSevData.FrNetExtIWF[wCount].bSlotID  = bSlotID;
        unSevData.FrNetExtIWF[wCount].wHdlcPort    = wHdlcPort;
        unSevData.FrNetExtIWF[wCount].wHdlcDlci    =
gFrPVCEP[bSlotID ][ gFrPVCC[bSlotID][dwCCID].dwLoPVCEP ].dwDLCI;
        unSevData.FrNetExtIWF[wCount].wOldAtmPort      = wAtmPort;
        unSevData.FrNetExtIWF[wCount].wAtmDlci     =
gFrPVCEP[ bSlotID ][ gFrPVCC[bSlotID][dwCCID].dwHiPVCEP ].dwDLCI;
        unSevData.FrNetExtIWF[wCount].dwMapMode =
```

```
gFrPVCC[bSlotID][dwCCID].dwMapMode;
            . . .
        }


DWORD RestoreFrNetExtIWFData ( WORD wSlotID, BYTE *pReadPoint )
{
        WORD        wCount, wTotalNetIWF;
        BYTE  bSlotID, bHdlcDlciType, bAtmDlciType;
        WORD        wOldAtmPort, wAtmDlci, wHdlcPort, wHdlcDlci;
        DWORD       dwMapMode, dwCIR, dwBe;
        DWORD       dwCCID, dwResult, dwAtmPort;
        wTotalNetIWF = g_MuxData.SevDataSize.wFrNetExtIWFNum;
        . . .
}

DWORD RestoreFrHdlcIntIWFData ( WORD wSlotID, BYTE *pReadPoint )
{
    WORD  wCount, wTotalHdlcIWF;
    DWORD     dwCCID, dwPeerCCID, dwAtmPort, dwPeerAtmPort;
    DWORD     dwResult;
    BYTE    bSlotID, bPeerSlotID;
    WORD  wHdlcPort, wOldAtmPort, wCIR;
    WORD  wPeerHdlcPort, wPeerOldAtmPort;
              . . .
}
```

其中涉及DLCI值的变量都为WORD（即无符号短整型）类型，在程序的处理时，出现WORD和DWORD（无符号长整型）类型在一句中同时存在的情

况，至此可以判断问题出在这里。由于DLCI值在不同类型时的取值范围不同，前三种类型的取值范围为16~991，第四种取值范围为2048~126975，第五种取值范围为131072~4194303，所以当采用前三种DLCI类型时，采用WORD类型最大值为65535，已经完全够用了；而对于第四种类型时，其取值在超过65535时，获取DLCI值的函数_GetFrDlci（）采用DWORD类型，而负责保存和恢复的两个函数SaveFrNetExtIWFData（）和RestoreFrNetExtIWFData（），都把DLCI的值当作WORD类型进行处理，因此导致DLCI取值越界，于是程序把原本为长整型的DLCI强制转换成整型，从而导致DLCI值在恢复时，比原数据小65536。而在程序运行过程中，这些数据保存在DRAM中，程序运行直接从DRAM中获取数据，程序不会出错；当FRI板复位或插拔后，需要从FLASH中读取数据，此时恢复函数的错误就表现出来。

另一个问题是为什么23/4类型的DLCI数据不能恢复？这是由于对于23/4类型的PVC，其DLCI的取值范围为：131072~4194303，而程序强制转换并恢复的数据最大只能是65535，所以这条PVC不能恢复。

至此，DLCI数据恢复出错的原因完全找到，解决的方法是将DLCI的类型改为DWORD类型。从这个案例可以看出，在程序开发中一个很低级的错误，将在实际工作中造成很严重的后果。

5、正确使用逻辑与&&、屏蔽&操作符

**【案例1.5.1】**

【案例描述】：由于C语言中位与比求模效率高，因而系统设计时，对于模128的地方都改为与127，系统定义的宏为#define MOD128　127和#define W_MOD 127(定义的宏的名字易引起误解)，但实际程序中还是采取求模，从而引起发送窗口欲重发的和实际重发的不一致，最终导致链路复位此类严重问题，曾在定位此问题时花了不少时间。

【处理过程】：处理过程如下：

```
#define      MOD128              127   //队列长128，当队头到128时，上其返回。
#define      W_MOD               127 //发送窗口队列，意义同上。
```

在函数L2_TO_L1()中，有如下语句：

linkstate_ptr->SendWin.head = (head + 1) % W_MOD ;

这里当head=126时，SendWin.head = 0，这将造成发送窗口指针和队列窗口指针错位，造成链路复位;

另外，在重发函数void INVOKE_RETRANSMISSION(_US logic_link,_US n_r)中，有如下语句：

retran_num = (LinkState[logic_link].Vs + MOD128 - (_UC)n_r) % MOD128 ;

w_head = (LinkState[logic_link].SendWin.head + W_MOD - retran_num) % W_MOD ;

第一个语句求欲重发的消息包个数，第二个语句求重发的起始位置，当Vs小于n_r时，将造成实际重发数小于欲重发数，同时造成实际起始重发位置和欲重发起始位置错开，从而引起链路复位。上面三个语句应该做如下改动：

linkstate_ptr->SendWin.head = (head + 1) & W_MOD ;

retran_num = (LinkState[logic_link].Vs + MOD128 + 1 - (_UC)n_r) & MOD128 ;

w_head = (LinkState[logic_link].SendWin.head + W_MOD + 1 - retran_num) & W_MOD ;

【结　论】：由于链路通信对系统效率要求很高，算法采用效率最高的，但位与（&）和求模（%）这小小的区别，造成的竟是链路复位这种严重的错误。

【思考与启示】：对这类问题，大家在阅读代码或代码审查时一定要注意，仔细一点往往能发现问题，但在测试中来定位这种问题，花费的时间往往更长。

6、注意数据类型的匹配

**【案例1.6.1】**

【案例描述】

下面通过测试中的一个例子来说明这个问题：命令DSP N7C是用来显示NO7

电路状态的，其参数设备类型DID支持TUP和ISUP，参数信道号BSN支持多值输入（最多支持32路查询），正常情况下该命令没有问题。但试了非正常情况下，问题就出来了。

1、首先试BSN参数越界情况，即参数BSN超过32路查询，选了几个数据段，问题就出来了。对于0&&300和0&&256，该命令返回结果不一致，对前者认为参数越界，对后者返回执行成功。

2、对于参数DID，选定一种设备类型（TUP或ISUP），让参数BSN所包含的32路电路跨越TUP和ISUP，两次结果是不一致的。

【处理过程】

反馈到开发人员那里，第一个问题是BAM的问题，第二个问题是SM的问题。

【结　论】

1、为数据超出范围溢出造成，int值赋值给BYTE，造成数据丢失。

2、问题的产生是因为查询的第一个信道是TUP电路，但是却按ISUP电路查询。ISUP的维护处理函数判断第一个信道不是ISUP信道，认为整个的PCM不是ISUP类型的PCM，返回全部的电路状态为未安装。消息处理不合理。TUP也会产生如此错误。

【思考与启示】

我们的MML命令并不是无懈可击的，许多表面上的小问题，往往隐藏着代码的缺陷和错误。

**【案例1.6.2】**

【正　　文】

当我们使用PC-LINT检查代码时，会发现大量的数据类型不匹配的告警，大部分情况下，这种代码上存在的问题并不会引起程序功能实现上的错误，但有些情况下，也许会产生严重的问题：

一、不同数据类型变量之间赋值引起的问题，实际上，该类问题也可以分为几种情况：

1、直接赋值，比如，把一个WORD型变量赋给一个INT型变量，如果WORD型变量大于32767，INT型变量得到的就是一个负值了。

【例一】一次测试过程中发现，SDH送的告警在BAM调试窗口打印出红色提示：File(XXX),Line(XXX):Invalid alarm id ,from: 7, AlarmId: 65463

经过检查数据发现，并没有ID为65463的告警，分析上报的数据帧，发现上报的告警ID为B7，原来代码中有一处强制类型转换：

```
sdhAlmStru.AlarmId = (WORD)RecvBuffer[iTmpLen + 5];
```

char型强制转换成WORD型。B7就变成了FFB7，十进制就是65463。由于char是有符号型，B7的第8位为1，所以转换后为FFB7，而不是代码作者希望的00B7，如果第8位是0，或该变量是BYTE型，转换就不会有问题了。

2、函数形参和实参不一致，实际上和第一种情况本质上是一样的，只是表现的形式不太一样，这种情况也是代码中经常出现的问题,下面例子是测试中曾经发现的一个小问题：

【例二】在file01中的INT DebugMsgProc(char byMsg0, char byMsg1)函数，两个形参都是char型，而实际传入的参数都是BYTE型，结果函数中的如下语句：

```
PrintfE(PID_RED,″ %d ticks time out!″,byMsg1);
```

在byMsg1大于127时，输出错误的结果。

二、不同数据类型之间的比较操作

在循环终止条件的判断中，不同类型变量的比较操作是容易造成死循环错误的地方，同时也是开发人员容易忽视的地方，值得测试人员多加留意。下面两个例子是该类错误的两种典型情况：

【例三】file02文件中某函数中如下代码，可能造成死循环：

```
.....
    int i;
WORD *pCheck   =(WORD*)p;
WORD wCheckSum=*pCheck;
pCheck++;
```

```
for(i=1;i<dwLen/2;i++)
{
    wCheckSum^=(*pCheck);
    pCheck++;
}
//binlen had already word alignment
return (wCheckSum);
.....
```

该段代码是在DOS环境下用BC编译的，由于循环变量i是int型（2个字节），而dwLen是DWORD型（4个字节），如果dwLen大于65536，那么该函数就是死循环了。

上面的例子是不同类型变量之间直接比较操作，还有一种情况是函数的返回值与另一不同类型的变量比较，见下面例子：

【例四】file03.c文件中某函数中如下代码，

```
    while( ftell(fp)< Part[3])
{.....
                    }
```

ftell返回long型，而Part是DWORD型，有符号变量和无符号变量的比较，可能造成死循环。

类似的例子还有很多，类型不匹配的问题还有许多种情况，都是代码中的隐患，有时会造成严重的后果，需要引起足够的重视。对于该类问题，我们可以利用PC-LINT工具对代码进行细致的检查。

7、用于控制条件转移的表达式及取值范围是否书写正确

**【案例1.7.1】**

【案例描述】：

在测试主机MPU板倒换功能时，如果MPU备份充分，倒换前后对处于激活状态的电路应无影响，即不影响通话。但近期测试发现，如果两局通过DT板进行一号对接，MPU备份倒换却发生断话。具体现象为：如果DT板的第1个PCM系统电路为故障，则MPU倒换时复位该DT板，如果DT板的第2个PCM系统电路为故障，则MPU倒换时复位下一块DT。

【处理过程】：

据查，MPU倒换时会自动复位处于“故障”态的电路，但由于计算错误（多加了32），错复位了下一个PCM系统32路电路。

【结　论】：

如此严重问题为什么到今天才发现？因为我们在实验室中一般采用同一单板的2个PCM系统自环进行测试，则不会在某单板上有故障和空闲电路共存，自环屏蔽了错误。

【思考与启示】：

自环是在测试环境下常用的一种提高效率的手段，但一旦条件允许，我们的测试工作应尽量模拟网上的实际环境进行。

**【案例1.7.2】**

平时对计费功能进行测试的时候，浏览详细话单都是比较注意话单本身的正确性，并没有注意该命令对系统的影响。所以当浏览少量话单的时候，并没有发现该命令的异常。但是当时间的跨度较大时，详细话单数量较多，问题就出现了。执行如下命令：

LST AMA: TP=NRM, SD=1999&7&1, SA=YES;

当浏览了大约10万张详细话单后，终端与BAM的连接关闭。重建连接后，发现话单台的命令不能执行。观察BAM的性能，发现话单台仍占有CPU50%以上的利用率，说明原来的任务仍在执行。需要关一下话单台才能恢复正常。

重复上述步骤，当终端与BAM的连接尚未关闭时主动断开此次连接，结果同上。

反馈到开发人员那里，发现该现象与设计的初衷是相违背的。本来话单台控

制最多输出200张话单，这是为了防止过多话单的输出显示会增加BAM的开销，从而降低BAM的性能。查看一下源代码，问题就发现了。

话单台控制最多输出200张话单

程序如下

```
 while(timeCur <= timeEnd)
 {
timeCur += tsOneDay;//加一天

while(fileBill.Read(&rpt, sizeof(CBillReport)) ==
    sizeof(CBillReport))
{
                .....
                        //只输出满足条件的前200张话单
          if (++wBillCount == 200)
         {
    break;
         }
}//一个文件查询结束
}//所有文件查询结束
```

在话单输出200张之后，程序只退出一层循环，仍然会从下一天话单继续输出，导致向MML发帧过多，造成MML和话单台都被堵死。

修改ProcessQueryBill()函数

```
  //只输出满足条件的前200张话单
  if (++wBillCount == 200)
  {
        timeCur = timeEnd + tsOneDay;//退出第二层循环,
    while(timeCur <= timeEnd)
        break;
  }
```

作上述修改后问题就不再出现了。

一些MML命令从完成的功能来讲可能是没什么问题的，但其执行对系统性能的影响我们在测试时时往往给忽视了。在我们目前的BAM方案中，存在着多个终端协同工作，如果某个终端发出的命令在BAM中长时间独占着大部分系统资源，造成的后果是严重的。这是在设计时要避免的，在测试中要注意的问题。

**【案例1.7.3】**

【正　　文】

在判断模拟用户端口是否反极性时有这样一段程序：

```
if ( ( bsn >= g_wASL32StartPSN ) &&
    ( ( ( bsn - g_wASL32StartPSN ) % 32 ) == 15  ||  ( ( bsn -
g_wASL32StartPSN ) % 32 == 16 ) ) )
        return TRUE;
if ( ( bsn % 16 ) == 7 || ( bsn % 16 ) == 8 )
        return TRUE;
    return FALSE;
```

作者的本意是如果是32路用户板（蓝色字体判断），就看端口号是否是第15和16路，如果是，就是反极性端口，返回TRUE，否则就不是，应该返回FALSE。但代码表达的意思是：如果是32路用户板并且端口号是15或16就返回真值，否则还要执行下边语句。

当端口在32路用户板上，但端口号不是15或16时，不同的32路端口的起始地址g_wASL32StartPSN，会导致不同的非15、16端口被误认为是反极性端口。举个例子，当g_wASL32StartPSN的值为3000时，端口号为3000（第一块板上的第0个端口）就被认为是反极性端口，这与作者的意图完全相悖。

可以将代码修改如下：

```
if ( ( bsn >= g_wASL32StartPSN )
{
    if  ( ( ( bsn - g_wASL32StartPSN ) % 32 ) == 15  ||  ( ( bsn -
```

```
g_wASL32StartPSN ) % 32 == 16 ) ) )
                return TRUE;
            }

        else
            if ( ( bsn % 16 ) == 7 || ( bsn % 16 ) == 8 )
                return TRUE;
        return FALSE;
```

通过这个例子，我觉得在代码审查时应该留意在判断条件较多的情况下，每个输入是否都能正确输出，在单元测试、集成测试、系统测试时要针对边界值设计相应的测试用例。

判断条件较多时开发人员也应该适当分开写，既使代码更易读，又不容易出错。

8、条件分支处理是否有遗漏

**【案例1.8.1】**

【现　象】

在接入网主机程序的代码审查中，发现dbquery.c的DBQ_Init_ANType函数中如下代码段缺少应有的条件分支，在数据异常的情况下，会产生较严重的问题。

【处理过程】

该错误比较隐蔽，现在说明如下：

Max2B1QStatTime 最大统计时间

Max2B1QStatPortNum最大统计端口数

MAX_2B1Q_STAT_PSN　　最大统计内存分配数量

其中：Max2B1QStatTime（最大统计时间）和Max2B1QStatPortNum（最大统计 端口数）的乘积不能大于MAX_2B1Q_STAT_PSN

程序如下：

```
//查询数据库，获得Max2B1QStatTime的值
directQueryCond.tupleNo = 10;
error_code = DB_Query( RID_OTHERS_PARA_INFO, 1,
                       (LPDBCondition)&directQueryCond,
                       (BYTE FAR *)&tempstruct0 );
//查询数据库成功
if( error_code == DB_SUCCESS )
{
    //tempstruct0.data是数据库中为Max2B1QStatTime配置的值
    if ( tempstruct0.data > MAX_2B1Q_STAT_PSN )
        Max2B1QStatTime = MAX_2B1Q_STAT_PSN;
    else if ( tempstruct0.data != 0 )
        Max2B1QStatTime = tempstruct0.data;
}

//查询数据库，获得Max2B1QStatPortNum的值
directQueryCond.tupleNo = 11;
error_code = DB_Query( RID_OTHERS_PARA_INFO, 1,
                       (LPDBCondition)&directQueryCond,
                       (BYTE FAR *)&tempstruct0 );
//查询数据库成功
if( error_code == DB_SUCCESS )
{

    //tempstruct0.data为数据库中为Max2B1QStatPortNum配置的值，如果其缺省值和Max2B1QStatTime乘积值大于MAX_2B1Q_STAT_PSN的话：
    if ( (tempstruct0.data * Max2B1QStatTime) > MAX_2B1Q_STAT_PSN )
                    Max2B1QStatPortNum = MAX_2B1Q_STAT_PSN /
            Max2B1QStatTime;
```

```
                //如果在合理范围内且不为0的话：
                else if ( tempstruct0.data != 0 )
        Max2B1QStatPortNum = tempstruct0.data;
    }
```

此处if-else if 分支没有判断 值为0的情况，即数据库为Max2B1QStatPortNum配置的值为0： tempstruct0.data == 0，则Max2B1QStatPortNum就为缺省值32。

【结　论】

由于内存限制，Max2B1QStatTime（最大统计时间）和Max2B1QStatPortNum（最大统计端口数）的乘积不能大于MAX_2B1Q_STAT_PSN，

如果从数据库中得到Max2B1QStatTime为MAX_2B1Q_STAT_PSN，而数据库中最大统计端口数恰好为0，由于上述代码没有对tempstruct0.data == 0的情况进行判断，Max2B1QStatPortNum为缺省值32，这样Max2B1QStatTime和Max2B1QStatPortNum乘积已经是32倍MAX_2B1Q_STAT_PSN了，远远超过了设计内存的限制。

造成这种错误的原因是判断语句对条件判断不完整。

【思考与启示】

在代码审查时，应该十分注意条件判断的的完备性。好多问题就是因为条件判断不完全造成的。

9、引用已释放的资源

**【案例1.9.1】**

【正　　文】

在计费测试的过程中，用呼叫器进行大话务量呼叫测试。30路话路通过TUP自环呼叫另外30路话路，计费数据的设定是这样的：通过计费情况索引对主叫计费，得到详细话单。首先保证计费数据设定的正确性，打了几次自环电话后，查看话单正常，则开始呼叫。

呼叫几万次后停止呼叫，取话单进行观察。发现这30路每次呼叫总会出现一张告警话单，其余话单正常，该告警话单相对于话路来说是随机出现的。

通知开发人员后，首先我们再次对计费数据进行了确认。某个用户在某次呼叫产生了告警话单，其上一次和下一次呼叫的计费情况都正常，两次呼叫之间的时间间隔只有几秒钟，排除了人为修改数据的可能。开发人员认为是CCB的问题，后来一查果然如此。

当中继选线发生了同抢需要重新选线时，CCB的reset_CCB_for_reseatch_called_location()就会把有关的呼叫信息清掉，造成计费情况分析失败，产生计费费用为0的告警话单。

更正reset_CCB_for_reseatch_called_location()中清除被叫信息的代码，重选中继时不清除被叫用户这部分属性。

思考与启示：

1、在计费测试过程中，对话单的观察很重要，不应该放过任何一个细小的疑点；

2、计费测试仅仅打几次电话往往达不到效果，越接近用户实际使用的情况越可能发现问题。

【案例1.9.2】

【案例描述】

在进行128模块V5用户CENTREX新业务测试时，偶然遇到一个怪现象：对群内一个V5ST用户只开放MCT权限，在进行恶意呼叫追查时，有一次报恶意呼叫追查成功音只报了一半，当正要报出恶意呼叫的号码时，业务中断重新回到通话态，随即重新追查一次，报“已申请其它新业务，本次申请不成功”。恶意呼叫追查与任何新业务都不会冲突，而且此用户也只有恶意呼叫追查有权，可以

肯定此时程序出问题了。为了重现，再次挂机，重新呼叫，应用此新业务，但这个现象一直没有出现。大约反复操作20遍，又出现了一次这样的情况，显然程序中可能存在某种问题。

【处理过程】

出现这个问题后，及时与开发人员A取得了联系，并一起试图重现这个问题，通过许多次的反复操作，又出现了一次这种情况。确认问题后，A表现出高度的责任心，马上驾调试环境，反复调测，终于在当天就逮住了狐狸尾巴：

1、当用户接听恶意呼叫者的电话, 并启动恶意呼叫追查业务后, 在V5_CR_VOICETONE状态下, 只要听MCT音的用户用脉冲方式拨任意一个数字, 则立即停止送MCT音, 而将用户切换回与恶意呼叫者的通话. 但是程序中没有对拨号类型作判断, 导致用户若用音频拨号也会作同样的处理。

2、除了取消此次MCT服务, 将用户切换回与恶意呼叫者的通话外, 如果不释放MCT_HANDLE, 由于每个模块只有一个这样的资源, 则下一次使用MCT业务的用户不能成功, 因为会在申请MCT_HANDLE时失败, V5模块和ST模块在这个地方处理都有问题, 没有将MCT_HANDLE释放掉, 对于V5用户会听新业务失败音, 对于ST用户会听音乐。

当不停的拨测V5用户的MCT业务时, 有时在听音时, 可能由于网板有杂音等原因(或用户碰了话机的按键), 导致DTR收到一位号, 则会立即停止此次MCT服务, 用户会听到MCT送音突然中断, 然后恢复了与恶意呼叫者的通话. 而下次再用MCT时, 由于上面所述的原因, 会听到新业务失败音, 此次失败后, 无论MCT_HANDLE分配成功与否, 该用户的MCT标志都被置为1, 所以在用户挂机时, 会将该模块唯一的MCT_HANDLE资源释放掉. 则以后该功能又可以正常实现。

在追查这个问题时，开发人员A又发现了一个可能导致死机的严重问题：在用户启动MCT服务，正在听报追查号码的MCT音时，若恶意用户此时挂机，CCB的处理中，只针对ST用户送DISCONNECT，而对V5ST用户送的是RELEASE消息，这导致V5X收到此消息后，将该V5ST用户的cr2清除掉，V5_USER_TALBE[ ]. cr2变为0xFFFF，这样在V5_CR_VOICETONE超时后，

程序中会检查cr2的状态是否为HOLD，当取cr2的内容时，由于cr2已被清除，会发生指针越界的GP错误。

【结　论】

通过调测发现、定位并解决问题。

【思考与启示】

我们平常一些熟视无睹的业务或按正常流程操作没有问题的业务，不能保证它就一定没有问题，要善于抓住一丝一毫的异常现象。对于很难重现的问题千万不要轻易放过，我们网上设备所出的问题很多都是一些在实验室难以出现或很难重现的一些问题，一些显而易见的问题一般都可消灭在实验室，难就难在消灭一些隐藏很深的问题。说老实话，我们的产品还有许多问题 ，需要我们扎扎实实锲而不舍的工作。

10、分配资源是否已正确释放

**【案例1.10.1】**

【正　　文】

在对接入网A产品的网管软件测试中，发现了一个WINDSOWS资源损耗的的问题：当网管软件运行几天后，WINDOWS总会出现“资源不够”的告警提示。如果网管软件不关掉再重新启动的话，就会出现WINDOWS资源完全耗尽的现象，最终网管系统反应很慢，无法正常工作。

从现象上可以判断出，网管软件存在隐蔽的内存泄露或资源不释放的问题，并且这种资源耗尽是一个缓慢的过程。如何定位这个问题呢？

定位这种问题可以利用WINDOWS中的一个系统资源监视工具。打开Windows的“附件/系统工具/资源状况”，这是一个系统资源、用户资源、和GDI资源的实时监视工具。

工具有了，那么如何发现导致不断消耗资源的特定操作呢？

首先和开发人员共同探讨，列出几个最可能消耗资源的操作和一些操作组合，这样就缩小了监视范围，避免没有范围的碰运气，否则如大海捞针。

监视前，首先重新启动WINDOWS，最好不运行其他的程序，打开“系统状况”这个监视工具，然后运行网管软件，记下此时的资源状况数据。

然后针对一个可疑的操作，快速大量地重复进行。这种重复性的操作可以利用QArun测试工具执行，QArun可以记录操作者的一次操作步骤，然后按照设定的次数重复操作。操作后，观察此时的资源状况，并记下此时的数据，与操作前的数据比较，如果操作前后的数据数据没有变化或变化很小，可排除此项操作，否则就可断定此项操作会引起资源耗尽。

对其它可疑的操作和操作组合重复以上过程。

通过以上的步骤，终于找出引起资源耗尽的罪魁祸首。分析相应部分的代码，发现引起资源耗尽原因有：内存泄露，画笔和画刷资源用完后未释放等。

**【案例1.10.2】**

【正　　文】

某产品后台软件版本，是用C++写的，程序员在写代码时，经常在构造函数中申请一块内存，而不释放，在程序其他代码中也经常只管申请，不管释放。例如:

```
 void WarnSvr::SaveWarnData()
{
      .....

          for(int m=0;m<RecordsInBuffer[EVENT_ALARM];m++)
          {
            HISTORY_FILTER_INDEX* item=
                    new HISTORY_FILTER_INDEX;
            item->Csn=Buffer[EVENT_ALARM][m].Csn;

                                    item->Position=m
                    +(RecordsInHistoryFile-RecordsInBuffer[EVENT_ALARM]);
```

```
//If a warn with a certain Csn is not in EventFilterIndex
//it is not necessary to be added to    HistoryFilterIndex
int item_total=EventFilterIndex.GetItemsInContainer();
BOOL find_flag=false;
for(int k=0;k<item_total;k++)
 if(EventFilterIndex[k]->Csn==item->Csn)
      {
      find_flag=true;
      break;
      }
if(find_flag)
   {
     HistoryFilterIndex.Add(item);
     if(HistoryFilterIndex.IsFull())
       ClearIndexEntry();
  }
//建议在此处加上:
//                                  else
//                                                delete item;
}。
```

有的程序员认为，后台运行的环境有大量内存，几个字节的浪费不会造成死机等重大事故。然而，长时间累计起来，必然会造成资源紧张而出现故障。

实际上，这种思想是造成我们产品不稳定的原因之一。我们的主机在网上能运行几个月几年，大家对内存的分配释放较敏感，而我们的后台产品往往只能正常运行几天。这个地方不注意也是原因之一吧。

**【案例1.10.3】**

【正　　文】

在进行代码审查过程中，造成内存泄漏的代码比较多。下面举几种常见的内

存泄漏错误，供测试人员在代码审查中参考：

1. 函数有多个出口时，没有在每个出口处对动态申请的内存进行释放。一般在异常处理时容易出现这种错误。下面的代码段就是这样的例子：

```
.....
        pRecord = new char[pTable->GetRecordLength()];
    assert(pRecord != NULL);

    if (pTable->GoTop(FALSE) != DBIERR_NONE)
       return; // 如果从这里返回，pRecord将得不到释放
     .....
        pTable->Close();
    delete[] pRecord;
      }
```

2. 给指针赋值时，没有检查指针是否为空，如果指针不为空，那么指针原来指向的内存将丢失。请看如下代码段：

```
....
    struct FileInfo * pdbffile = new struct FileInfo;
    pdbffile->pfileinfo = new struct ffblk;
    pdbffile->srcname = srcRootPath;
    pdbffile->desname = desRootPath;
    pdbffile->prev = NULL;
    pfile = pdbffile;
//赋值之前没有检查一下pfile是否为空，如果不为空，会造成pfile指向的内存丢失。
    dbf_start_needed = FALSE;
    dbf_Finish = FALSE;
    flag_begined = TRUE;
```

```
    if(FALSE == Copy(TRUE))
    {
        dbf_start_needed = TRUE;
        WarnMsgOut("Error occurs while copying files in directory
<dbf>,trying again.");
    }
    }
```

3. 连续二次内存动态分配，在第二次分配失败时，忘记释放第一次已经申请到的内存。

```
    ....
    pMsgDB_DEV = (PDBDevMsg)GetBuff( sizeof( DBDevMsg ), __LINE__);
    if( pMsgDB_DEV == NULL )
        return;

    pMsgDBApp_To_Logic = (LPDBSelfMsg)GetBuff( sizeof(DBSelfMsg),
__LINE__ );
    if( pMsgDBApp_To_Logic == NULL )
        return;//此处返回造成pMsgDB_DEV指向的内存丢失
    ....
```

4.代码中缺少应有的条件分支处理，导致程序未执行任何操作而退出时，也可能没有释放应释放的内存，这种情况一般是缺少应有的else分支，或switch语句的default分支没有应有的处理。

```
static void    OncePowerCmdHandle( struct HT_Appmsg    * msg )
{
    ... ...
        pPower_test_answer =(struct _oncepower_test_answer
*)GetBuff(sizeof(struct _oncepower_test_answer),__LINE__);
    if( pPower_test_answer == NULL_PTR )
```

```
        return;
    ... ...
    if (TSS_State[testpsn].state == TEST_DEV_BUSY ||
        TSS_State[testpsn].state == TEST_DEV_ERROR )
    {...
    }
    else if (TSS_State[testpsn].state == TEST_DEV_IDLE )
    {...
    }
    // 缺少 else 分支，可能造成 pPower_test_answer 得不到释放
}
```

造成内存泄漏的情况很多，以上是几种典型的情况。

虽然内存泄露一般出现在异常情况下，毕竟给系统造成很大的隐患，使系统的健壮性降低。测试人员在作代码审查时，对上述几种情况要尤其注意。

**【案例1.10.4】**

【正　　文】在进行SAR的PDU包发收的测试过程中要同时考虑几个边界值,即发送包大小范围[0-Nmax],SAR的PDU包接收的最大值Kmax,MBUF块的大小M.在实测中,将SAR的PDU包接收的最大值设为2000(Kmax=2000B), MBUF的块长设为512(M = 512B),则发送包大小的正确分支的取值为下限0,上限Nmax=2000,然后在0与2000之间随机取若干值,再考虑MBUF的块长,还可增加M倍数的若干选值及其附近值.以上是测试的一般思路,但由于很偶然的机会选择包长2000,及Kmax=2000B,才发现问题.原因如下:

MBUF块长512,但块中实际存放数据的只有500(MBUF头上有2个长字,尾部有1个长字共12B只用于块控制),而发送的包长正好是500的整数倍4,由于是整数倍,所以SAR(BT8230)从FREE链上摘成5个MBUF(原因从略),而SAR驱动只知道有4个MBUF,这样到上层用户时,只释放4个MBUF,从而漏掉1个MBUF,经过很短一段时间后,内存即被耗尽.(此问题非常严重,因为在实际运用中,是500的整数倍的PDU

包的概率较小,但一旦出现就会发生一次内存泄漏,这样经过若干天或若干月的运行后会使系统崩溃)

以前未发现此问题的原因是因为原来使用的缓冲块长为2048,减去12B的控制信息,实际存放数据的长度为2036.由于只考虑了2048这个值,忽略了2036,所以在选取上下限中的若干值时,选取包的长度是2036的倍数的概率就非常小,因而未发现该问题.

由于测试中一般很难将取值范围中的所有值覆盖全,所以在选取上下限中的若干取值时要格外仔细,考虑的方面尽可能全,因为很有可能其中某些值就是测试边界值.凡是涉及的数字尽量选取,象该例中正确分支的测试边界为0,2000,512及其整数倍,500 及其整数倍,12 及其整数倍等值,它们是必测的边界值,而非可测可不测的随机选取的所谓若干选值.

**【案例1.10.5】**

【正　　文】

ABIS.CPP中的函数rel_ABIS_CCB_conn( )中，在进行消息链表Msg_Queue[ces]的拆链操作时，对于相应的CCB只进行了一次拆链操作，即只拆除了一个节点，如果出现该CCB对应的消息节点不止一个的情况就会出现大量节点不能释放的问题。

```
if( Msg_Queue[ces].msghead != NULL_PTR )//message buffer notempty
{
        //get first message record
        pMsgRecord = Msg_Queue[ces].msghead;
        //release buffer-messages concerning with ccb_no
        for( index = 0; index < MSGBUFFERNUM; index++ )
        {
            //这里要对pMsgRecord的值进行判断
              if( (pMsgRecord != NULL_PTR) && pMsgRecord->CCB_no ==
                ccb_no )
            {
                 //free the message buffer
```

```
            if( pMsgRecord == Msg_Queue[ces].msghead )//head
            Msg_Queue[ces].msghead = pMsgRecord->pnext;
            else if( pMsgRecord == Msg_Queue[ces].msgtail )//tail
            {
                  Msg_Queue[ces].msgtail = pPrevMsgRecord;
                  Msg_Queue[ces].msgtail->pnext = NULL_PTR;
            }
            else//not head and tail
            {
                  pPrevMsgRecord->pnext = pMsgRecord->pnext;
            }
            //put buffer back to buffer pool
            if( Msg_Buffer.empty_num == 0 )
            {
                  Msg_Buffer.linkhead = Msg_Buffer.linktail = pMsgRecord;
                  pMsgRecord->pnext = NULL_PTR;//这里将

pMsgRecord->pnext置为空
                  Msg_Buffer.empty_num++;
            }
            else
            {
                  Msg_Buffer.linktail->pnext = pMsgRecord;
                  pMsgRecord->pnext = NULL_PTR;//这里将

pMsgRecord->pnext置为空
                  Msg_Buffer.linktail = pMsgRecord;
                  Msg_Buffer.empty_num++;
            }
      }
```

```
else if( pMsgRecord == NULL_PTR )
                    break;//end of if
//get next message record
   pPrevMsgRecord = pMsgRecord;
   pMsgRecord = pMsgRecord->pnext;//这时pMsgRecord为
```

NULL_PTR将跳出for循环语句

```
    }//end of for
}//end of if
```

这里在拆除一个节点后导致pMsgRecord为NULL_PTR，再进行判断时将会跳出循环，这样将不能保证所有与同一个CCB有关的节点均被拆除，这时如果与同一个CCB对应的消息节点不止一个则这些消息节点均无法释放，造成可用的节点数不断减少，直接影响系统的建链过程，给系统的稳定带来隐患。

后与开发人员联系，根据这段算法编写小程序验证了该问题，并提出了相应的解决方案，消除了该隐患。

**【案例1.10.6】**

【正　　文】

1、建立一个呼叫，并保持通话。在AM控存监控操作界面中观察通话建立在哪一块FBI板上。

2、将有通话的FBI板拔出，观察通话情况，此时话音中断，但信令仍然保持。观察AM控存监控操作界面和E3M板2K网界面，发现AM侧因为检测到光纤已断，将通话在CTN、E3M板上占用的时隙置为空闲，即在AM控存监控操作界面和E3M板2K网界面观察不到时隙占用情况。

3、分别在30秒、1分钟、3分钟时将拔出的FBI板插回原槽位，发现每次插回FBI板后话音立即恢复。

4、观察BAM上的打印消息，发现打印的各模块占用CTN板大HW上DM时隙的空闲个数比较混乱。打印消息如下图所示：

其中：

1） 由于模块1、2、3、4各占用CTN板上两条大HW，每个DM时隙个数为256（即由两条大HW的两个DM组成，由于与OPT相联的大HW上有两个保留时隙，因此此DM上空闲时隙个数为：254。

2） 由于E3M板只与一条大HW相联，故每个DM上空闲的时隙个数为：

```
Modu-233 :Extern SM_Module 207 DM idle_count = 128 - 128 - 128 - 128
Modu-233 :Extern SM_Module 1 DM idle_count = 255 - 255 - 254 - 255
Modu-233 :Extern SM_Module 2 DM idle_count = 253 - 253 - 254 - 255
Modu-233 :Extern SM_Module 3 DM idle_count = 256 - 256 - 254 - 256
Modu-233 :Extern SM_Module 4 DM idle_count = 256 - 256 - 254 - 256
Modu-233 :Extern SM_Module 200 DM idle_count = 128 - 128 - 128 - 128
Modu-233 :Extern SM_Module 201 DM idle_count = 128 - 128 - 128 - 128
Modu-233 :Extern SM_Module 202 DM idle_count = 128 - 128 - 128 - 128
Modu-233 :Extern SM_Module 203 DM idle_count = 128 - 128 - 128 - 128
Modu-233 :Extern SM_Module 204 DM idle_count = 128 - 128 - 128 - 128
Modu-233 :Extern SM_Module 205 DM idle_count = 128 - 128 - 128 - 128
Modu-232 :Extern SM_Module 200 DM idle_count = 128 - 128 - 128 - 128
Modu-233 :Extern SM_Module 206 DM idle_count = 128 - 128 - 128 - 128
Modu-233 :Extern SM_Module 207 DM idle_count = 128 - 128 - 128 - 128
Modu-233 :Extern SM_Module 1 DM idle_count = 255 - 255 - 254 - 255
Modu-233 :Extern SM_Module 2 DM idle_count = 253 - 253 - 254 - 255
Modu-233 :Extern SM_Module 3 DM idle_count = 256 - 256 - 254 - 256
Modu-233 :Extern SM_Module 4 DM idle_count = 256 - 256 - 254 - 256
```

128。

本现象对应2个问题：idle_count打印混乱，BM释放故障光路的时隙和对应的CCB、无线信道等资源。

1、idle_count打印混乱是由于函数restore_one_hw中的一些处理不当造成的，以前被当作B型机的历史遗留问题没有重视；

2、B2模块有2条光路，如果断掉其中一条，模块状态不会改变，原B型机程序对此不作任何处理，但应该增加这个功能，以免光路故障导致资源吊死。

解决方法：

问题一： 将函数restore_one_hw中原代码作如下改动：

```
                    mod_dm[mod][i].tail.tsn = idle_dm_head + 125;
    ( idle_dm_head == 384 ) ?
        mod_dm[mod][i].idle_count += TS_PER_DM - 1:
        mod_dm[mod][i].idle_count += TS_PER_DM - 1;
```

改为：

```
if ( idle_dm_head != 384 )
{
                    mod_dm[mod][i].tail.tsn = idle_dm_head + 127;
    mod_dm[mod][i].idle_count += TS_PER_DM;
}
```

```
else
{
        mod_dm[mod][i].tail.tsn = idle_dm_head + 126;
        mod_dm[mod][i].idle_count += TS_PER_DM - 1;
}
```

问题二分析如下：

目前的模块状态是由IPATH调用DBMS模块的边检查实现的，只要存在一条可用的光路，即认为相邻模块为正常，对于具体的OPT板上的时隙状态的维护没有与呼叫控制的接口。具体的OPT板状态功能的检测是由IPATH完成的，在BM侧没有专门维护OPT和MC2板的模块，将转交OS组处理。

总结：

在拔出FBC板后，通话话音被中断，AM/CM侧已将与被拔出的FBC板相关的资源全部置为不可用，此时BM侧主机程序也应该与AM/CM侧一致，释放掉所占用的资源，并将原通话的信令连接断开。这可能是由于不同模块的开发人员缺少相互间了解而造成的，即AM/CM侧与BM侧开发人员交流不够。作为测试人员对类似两个或多个模块相关的部分应该充分进行测试，不要想当然，往往是看起来不可能出问题的地方也容易测出问题。

**【案例1.10.7】**

在进行有关排队指示的系统测试中，先闭塞掉基站的所有业务信道TCH，进行呼叫，再直接挂机或超时释放，发现TC存在中继资源吊死的问题。

由于此问题重现，后经定位分析，发现是ccb超时后收到AIR发来的clear cmd，进入 rel_one_bm_res( )时，由于ccb所登记的CIC还放在pre_occupied_res，并没有放入occuped_res，而rel_one_bm_res()只对存入occuped_res的CIC进行判断，并向AIE发UNBOOKCIC，而没有对存入pre_occupied_res的CIC进行判断，并UNBOOK掉，导致TC的中继资源吊死。应在超时函数或释放函数中对pre_occupied_res的CIC进行处理。

在此过程中，CIC资源还存放在老CCB的pre_occupied_res中，在超时函数或释放函数中均未对pre_occupied_res中的CIC进行处理（即向AIE UNBOOK），导致TC中继资源吊死。

在超时函数RR_time_out()中timer_name为TN_WAIT_ASS_READY时，和释放函数rel_one_bam_res()中增加对CCB的pre_occupied_res中的CIC的判断和释放处理。

在使用资源同时，就要周密地考虑好资源的释放问题，只有这样，才能使我们的系统不断地稳定下来。

资源的释放对于我们的交换机来说是至关重要的，一点点的疏忽都可能最终使我们的交换机因为无资源使用而死掉，要知道，“千里长堤，毁于蚁穴”。

11、防止资源的重复释放

**【案例1.11.1】**

【正　　文】

当进行大话务量呼叫时，在统计代码中出现AIE收到UNBOOK CIC消息时，发现自身电路状态为空闲，出现一个断言。这说明AIE电路电路被误释放了。

这个问题出现的原因有以下几种：

1. RR可能发错了电路号，导致AIE状态错误。

2. AIE可能发起资源核查，失败后将本控制表项释放了。

3. RR可能发起了重复释放操作，导致AIE的某个表项连续收到两个UNBOOK消息。

分析完了可能的情况，就要一一分析定位。

在可能原因一发生的情况下，RR发来的UNBOOK消息所带的AIR连接号和模块号会错误，导致我们会出现断言。而在测试数据结果文件中，没有出现这个断言，因此可能原因一不成立。

在可能原因二发生的情况下，AIE收到资源核查失败消息的数目应该不是零。但是实际情况下统计结果中收到资源核查失败消息的个数为零，说明情况二

也不成立。

由上分析，这个问题只可能是由于RR重复释放造成的。但是为何会发生重复释放，这需要进行进一步分析。

从呼叫的正常流程来看，是不会产生重复释放的，因此我们怀疑该问题与异常流程有关。从统计代码中查找异常流程，发现该次统计中BSC内切换流程多次出现问题，具体原因是由于切换过程中在目标小区申请不到信道，产生切换失败造成的。因此集中研究这个流程，发现存在问题如下：

当原小区向目标小区发送内部切换请求消息时，带来了AIR和AIE的各项信息，而目标小区收到这些信息后就将之保存在自身的占用资源中。如果目标侧申请信道失败，就会向源侧发内部切换拒绝消息，而后产生本地释放。由于在释放前目标侧RR没有将占用资源中的AIR和AIE信息清除，因此导致重复释放时对AIR和AIE发起了释放操作。由于AIR释放时有保护机制，所以不会产生问题，而AIE没有保护机制，新CCB就将AIE电路释放掉了。而后当老CCB在通话结束后发起释放时，就产生了重复释放。

从上面分析可以看出，这个问题是由于RR释放流程的错误造成的，因此，我们要对此加以修改，在新CCB释放前将AIR和AIE信息从预占资源中清除。

RR的释放是一个非常复杂的过程，如何正确的整理资源，确保资源的合理释放，这是摆在我们面前的一个艰巨的问题，我们要仔细分析各种可能发生的情况，正确释放各种资源，即不会吊死资源，也不会产生重复释放。

## 12、公共资源的互斥性和竞用性

**【案例1.12.1】**

【正　　文】

试验环境：CPX8216 CPCI 机架、vxWorks 操作系统、Tornado1.0.1 调试环境

测试用例：测试板间通信性能。从接口板A向接口板B循环发送消息，通过超级终端观察消息的收发情况。

测试结果：每发送一定数量的消息帧后，会出现发送地址出错现象。

原因分析：接收板回送缓冲区指针给发送板，是采用memcpy单字节拷贝的方式。若发送速度快于接收速度，两板竞用发送板系统总线访问缓冲区指针所在

的共享内存，导致数据访问冲突。memcpy过程被打断，即出现发送板读发送地址出错现象。

采用四字节拷贝函数bcopyLongs传送发送缓冲区指针，问题解决。

共享内存的访问设计，除了考虑互斥外，还有总线竞用问题。

**【案例1.12.2】**

【正　　文】

问题描述：

在进行主BCCH载频互助新功能开发的并行联调测试的过程中，发现了以下的问题：在数管台设置“TRX倒换是否允许”为“是”，进行设定整表后，关闭基站其中配有4个TRX的小区的主BCCH所在的TRX电源，发现对应小区重新初始化并成功，也就是载频互助成功。这个时候从后台对该小区所在的站点进行4级复位，同时重新打开之前关闭的该小区的原配主BCCH所在TRX的电源，发现对应小区初始化失败。

问题定位：

在问题定位开始，先是查看了载频互助相关代码在站点初始化流程中的处理。BTSM程序初始化过程中，先是判断这一次初始化之前是否发生过载频互助，若发生过，再判断原配主BCCH（即数据库中实际配置的主BCCH所在的TRX）是否已经恢复（即能正常建立TEI，能正常设置该TRX对应的RC属性，总之能正常开工）。若载频互助发生过，且原配主BCCH所在的TRX（CoTRXGroupForBts[BtsNo].MainTRX）已经恢复，即把之前进行互助的TRX（CoTRXGroupForBts[BtsNo].AidTRX）的数据和原配的主BCCH所在TRX的数据交换回来，并重新进行初始化。表面上看原理应该没有什么逻辑错误，怎么会出现初始化不成功呢？

我们对程序中的每一个可能导致该问题的变量加打印调试程序，然后重现该问题，终于在打印出来的信息中发现在载频互助发生后其互助的主BCCH所在的TRX与实际数据配置主BCCH所在的TRX为同一TRX，这有问题，因为载频互助的实质就是实际数据配置主BCCH所在的TRX不能正常开工而借用其他TRX作为主BCCH。于是我们根据此线索查询了所有BTSM的程序，没有发现问题的

根源。于是我们查了最近合进版本的相关模块的程序，终于找出了问题的根源所在。

在载频互助程序中以全局变量ptrBTS_CONFIG_MAP[BtsNo].TRX_no_BCCH_in表示当前实际运行的主BCCH所在的TRX号，是随时变化的；以CoTRXGroupForBts[BtsNo].MainTRX表示原配的主BCCH所在的TRX号，是固定的。两者在系统开工的系统开工的接口函数FetchOneSiteConfig（ ）中赋了相同的值：该函数的409行有赋值语句CoTRXGroupForBts[BTS_no_temp].MainTRX = ptrBTS_CONFIG_MAP[BTS_no_temp].TRX_no_BCCH_in。以前函数FetchOneSiteConfig（）只是在系统开工时才调用过一次，故CoTRXGroupForBts[BTS_no_temp].MainTRX 在系统开工以后是不变的，但是在DBMI同步开发的整改中，作了如下处理：在每一次数据动态设定后，先判断站点下有没有发生过载频互助，若发生过则试图先把目前进行互助的TRX的数据与实际数据配置成主BCCH的TRX的数据倒换回来，然后进行站点初始化。问题就出现在这，在DBMI中认为DB中原配的主BCCH的TRX是ptrBTS_CONFIG_MAP[BTS_no_temp].TRX_no_BCCH_in，而且每次进行站点初始化时都调用函数FetchOneSiteConfig（），这样将导致CoTRXGroupForBts[BTS_no_temp].MainTRX的值与DB中实际原配主BCCH所在TRX不一致，从而导致主BCCH的相关数据倒换是出现错误，最终导致相应小区初始化不成功。

收获及反思：这个问题的出现是因为主机程序两个功能模块DBMI与BTSM之间的开发缺少相互沟通引起，如果在开发之前两个模块的的开发人员先约定好各个全局变量的用途，如果DBMI与BTSM两个功能模块都认为CoTRXGroupForBts[BTS_no_temp].MainTRX是实际数据的原来的主BCCH所在的TRX号，那么就不会出现以上问题。现在BSC主机程序的各个功能模块都同时合进了许多代码，各个功能模块之间的联系与冲突肯定会存在，这就需要开发人员在开发设计方案时就相互沟通，否则以上由于功能模块间的的冲突引起的问题肯定会存在，而且可能不那么明显的暴露了出来。我们的产品埋伏的炸弹的机会就越多。

## 二、接口类代码问题

### 1、对函数参数进行有效性检查

**【案例2.1.1】**

【案例描述】

某交换类产品BAM后台管理系统使用了注册表存储后台系统数据自动备份时间；在对数据自动备份进行系统测试时考虑到注册表中数值的任意性，很有可能被测系统没有对注册表中存储的值作相应的合法性检测，从而有可能对系统产生不良影响。

【处理过程】

通过审查源程序及实际验证，发现果然存在问题。BAM有关数据自动备份程序程序在得到该项值后只做了简单处理，没有对时间进行合法性验证。若自动备份操作发生在前后台通讯的高峰期或者是在设置数据需要对数据库进行操作时，对系统可能会产生重大影响。

【结　论】

系统使用注册表中的数据同样要进行各种情况下的测试，包括合法的和不合法的数据设置。

【思考与启示】

系统的数据输入有多种渠道，不仅包括人机命令、系统INI文件，还包括注册表等其它途径；在测试时对各种情况都要进行全方位的测试，从而保证系统的可靠性。

**【案例2.1.2】**

【案例描述】

设计规定07的TSS板是不支持数字用户内，外线强测的，在对数字用户内，外线测试正常后，有意强测一正在通话的数字用户，却发现返回报告：

用户外线测试

---

测试时间 = 1999-06-03 16:48:06

号首集 = 0

用户号码 = 6540136

BAM测试状态 = 正常

主机测试状态 = 正常

AVAG = 0.08

AVBG = 0.29

AVAB = -0.21

DVAG = 0.62

DVBG = 0.62

DVAB = 0.00

RAG = >10M

RBG = >10M

RAB = >10M

RL = >10M

CAG = 0.001

CBG = 0.001

CAB = 0.001

结论 = 断线

结论断线肯定是错误的，测试期间查询TSS状态，显示TSS空闲，可见主机返回的报告是错误的。

【处理过程】

修改命令ADD RTSTI对应的存储过程，问题解决。

【结　论】

用户外线测试主机没有判用户是否为数字用户。

【思考与启示】

测试一项功能，既要测试系统提供的功能要满足要求，系统没有提供的功能看是否误执行了，导致错误的结论。

**【案例2.1.3】**

【正　　文】

一般开发人员认为函数的参数很简单，但在实际设计和调用函数时，很容易犯一些参数方面的错误。下面就一些例子进行一些分析，希望能引起大家的重视。

1、函数设计中，使用函数内局部变量保存下次函数重新调用时需要的保留值。

```
void SlaverDownLoadProc( WORD wMsgLength , void *pTempMsgAddr )
{
    void *pTempTargetAddr ;

    SSLAVERLOADMSG *pSTempSlaverLoadMsg = ( SSLAVERLOADMSG
* )pTempMsgAddr;

    if ( COMMON_BOARD_LOAD_PROGRAM ==
pSTempSlaverLoadMsg->m_ucCmd )
        pTempTargetAddr = ( void * )SDRAM_LOAD_PROG_START_ADDR ;
#ifndef MMX   /* MMX板的数据不用加载，用备份方式*/
    else if ( COMMON_BOARD_LOAD_FPGA ==
pSTempSlaverLoadMsg->m_ucCmd )
        pTempTargetAddr = ( void * )SDRAM_FPGA_START_ADDR ;
            ...

              case BEGIN_LOAD :
            if (   ( LOAD_MIDDLE_FRAME ==
pSTempSlaverLoadMsg->m_ucLoadCmd )
```

```
        &&( ( dwRecFrameNum % 0xFF ) ==
pSTempSlaverLoadMsg->m_ucOrderNo )
        )
    {
    /*  序号和帧的命令字都是合法的，保存BUFFER */
        memcpy( ( BYTE * )pTempTargetAddr , ( BYTE * )pSTempSlaverLoadMsg
+ 5 , wMsgLength - 5 ) ;
    /* 5 = m_ucBoardType + m_ucCmd + m_ucSerialNo + m_ucLoadCmd +
m_ucOrderNo */
        dwRecFrameNum ++ ;
                ...
```

这里每收到一帧都要调用此函数，而每次进入时，函数将pTempTargetAddr赋值为两个固定值中的一个，导致收到的新帧将上一次的帧数据覆盖。显然，函数将本应作为全局变量的参数pTempTargetAddr错误地定义为局部变量。

2、当函数的输入参数较多，并且互相之间有联系时，是较易出错的地方。

在测试串口任务时，我们发现一个错误：当加入新用户时，当用户名输入到四十一个字符时，程序死掉。经过调试跟踪，发现是在input函数中调用GetString，但是在对第二个参数赋值时，没有搞清楚参数之间的关系。第一个参数（传入的指针）指向一长度为40字符的字符串，而在对第二个参数时，错误地将字符串长度设置为41。导致串口接收第41个字符时，程序越界操作，导致死机。下面是函数GetString的声明：

```
    signed long GetString
    (
        char * szString,            /* OUT:  字符串指针  */
        WORD wSize                  /* IN:  指定的字符串长度  */
    )
    1
```

当然有关函数参数的错误不止这些，例如未在模块接口函数内部检验传入参数的合法性、指针参数的有效性、参数未赋值就使用、参数类型不匹配或考虑不

周导致上溢下溢等。这些都是编程者容易出错的地方，亦是我们大家在走读代码时，需要特别注意的地方。

**【案例2.1.4】**

【正　　文】

在交换机的V5协议测试中，有一个相当常用又相当简单的测试项目，“交换机对某个V5接口发起主备倒换命令”。这在所有的V5测试中都会很顺利地通过的项目，在以前这个项目也测试过许多次，也从来没有遇到过异常情况。

可是在一次测试中却遇到麻烦，在交换机侧输入了参数“模块号”、“V5接口号”、“逻辑C通道ID”之后，发起主备链路倒换的命令，操作的结果居然是“无效的V5端口”。

这种提示令人感到很诧异，因为从AN侧能够正常发起主备链路倒换，从LE侧也能正常发起该V5端口的指定链路倒换，而且系统也完全正常，可见数据配置并没有错误。排除了人为的错误之后，我把重点放到了这三个输入参数上：“模块号”，“V5接口号”均是非常常规的数值，应该没有问题；“逻辑C通道ID”数值比较大，但是仍然在协议规定的65535之内，应该为有效值。想到常规配置数据时“逻辑C通道ID”值一直配得比较小，或许问题就出在此。

于是把“逻辑C通道ID”值改小，再作同样的操作，操作成功。到此很显然是这个参数的有效值范围定义有误，再细细检查，发现它只在单字节范围内有效，必然是该参数定义的类型有误。原因是：在函数OAM_Swap_Communication_Link（_UC v5_interface，_UC logic_c_id）里将逻辑C通道定义为字符型导致，改为_US型即解决。

推而广之，在终端的功能测试中，对输入参数值的测试，应该尽量覆盖所有的有效值。在正常情况下，如果输入值在为有效值，则应该得出正确的结果；如果输入值为非法值，则系统应该给出错误提示，并且不予执行。 对于参数值有效性的判断，特别应该注意一些特殊值和临界值，比如在字节和双字节处等等。

**【案例2.1.5】**

现象与分析：

在回归“载频配置表及跳频数据表中有零频点或重复频点时，MA编码不正确”问题单时。发现当载频配置表及跳频数据表中有重复频点或零频点时，带跳频的呼叫不成功。

这时首先考虑系统消息发的是否正确，观察系统消息一，发现所带的CA表没有错误，已经过滤掉了重复频点和零频点。

在考虑指配命令所带的MA值是否正确，结果发现MA编码也是正确的，也已经过滤掉重复频点和零频点。

经过以上初步分析以后，开始怀疑给基站下配置时是否也正确过滤了无效频点。然后查阅代码，发现果然 BTSM在对基站初始化设置载频属性及设置信道属性时，没有对载频配置表及跳频数据表中频点的有效性做检查，以致于表中出现非法频点时跳频呼叫不成功。

回顾与反思：

这个问题的发现并没有用什么特殊的分析方法，只不过是一般的“排除法”，而且这个问题也不是隐藏的非常之深，但是我觉得这个问题的发现暴露了我们在开发过程中的一个问题。那就是：如何实现各个模块之间的有效沟通，避免因为某一模块的修改而引起其他模块的连锁反映。

本来RR和BTSM都没有考虑重复频点和零频点的情况，大家都寄托与数据配置的正确，却也相安无事。RR首先增加了对非法频点的过滤，本来是件好事，使我们的系统变的更加坚强，但是BTSM并不知道这一变化，于是在错误数据面前束手无策，反而起到了相反的效果。

公司常说“下一道工序是上一道工序的上帝”，是不是可以引申为“其他相关模块是本模块的上帝”。试想如果各个模块多想想自己的改动是否给别人带来不良影响，多及时了解一下其他模块的最新进展。那么此类问题将不再会发生。

**【案例2.1.6】**

【正　　文】

现象与分析：

在动态数据配置测试增加小区时，发现小区不能正常初始化。观察ABIS_BTSM接口跟踪窗口，没有关于初始化流程的相关消息。经过分析发现是调用函数。

```
BTSM_make_send_site_config(_UC site_no , _UC ObjectClass , _US BTS_no ,_UC TRX_no , _UC OperMode)
{
    ……
    //判断合法性
    if ((site_no >= MAX_SITE_NUM)
        || (BTS_no >= MAX_BTS_NUM)
        || (TRX_no >= MAX_TRX_A_BTS))
    {
        ASSERT(FALSE);
        return (FALSE);
    }
    ……
}
```

由于进行小区级的操作调用该函数时，TRX_NO以0xFF带入，则在函数合法性检查时就会返回。从而引发该错误。

回顾与反思：

这是一个比较普通的错误，但是回顾它产生和解决的过程，觉得很有启发，也引起了我对编程规范的一些思考。

可以说该错误的引起是与编程规范有关,本来函数BTSM_make_send_site_config()没有对 BTS_no和TRX_no 的合法性判断，在代码审查时，我们考虑到这不符合编程规范中的“检查所有参数输入的有效性”这一条，于是提出请开发人员增加对参数的有效性检查。

但是我们和开发人员都忽略的一个问题，那就是，该函数被不同级别

的对象调用，当进行小区级操作时，TRX_NO以0xFF带入，则在函数合法性检查时就会返回，引起上述错误。

让我们再回头看看这个函数，发现依然不符合编程规范，至少违反了“不要设计多用途面面俱到的函数”这一条。

查看我们的代码，类似的问题还有不少，而且诸如：函数入口参数不加有效性检查、函数返回值不加处理等问题也可以找到。这些问题就象一颗颗隐型炸弹，在稍微不注意时就会带来严重后果。

正如前文所述，测试和开发人员都会因为“代码熟悉程度”、“代码编制时间太久有所遗忘”等诸如此类的原因而忽略一些问题。这时，良好的程序风格和编程规范就会成为一把强有力的保护伞。试想如果本文所提到的函数从设计到开发都严格按照规范来进行，那么这个问题就没有产生的土壤，如果我们的每一段代码的编写都严格遵守规范，那么我们的系统将变的有多么坚不可摧。

2、注意多出口函数的处理

**【案例2.2.1】**

摘　　要：如果函数存在多个出口，应注意函数对各个出口的处理。

问题描述：根据函数功能的需求，被测函数在函数体开始时保存了当前系统任务模式，并设置新的任务模式为不可抢占模式，在函数返回时应该恢复任务的旧模式，否则会影响整个系统功能的实现。可是问题就出在这里。由于函数有复杂的分支结构，有多个出口，有的出口对任务模式进行了恢复，有的出口没有恢复，这样就可能导致函数返回后系统任务模式被改变的问题。

问题分析：此问题出现的原因在于函数有多个出口，而程序员往往注重各个分支功能的实现的细节，而忽视了或者是忘记了在函数出口应做的收尾工作。这就象打一场战争一样，战争胜利了，还要打扫战场，开总结大会，否则就会象李自成一样，仗打完了就开始享受了，最后前功尽弃。毛主席说得好：“宜将剩勇追穷寇，不可沽名学霸王”。所以，函数出口的处理应充分重视。

案例意义：这类问题很常见，对于有多个出口的函数，测试时应充分构造测试例，采用分支覆盖的测试方法对函数各个出口的环境恢复、资源释放情况进行观察。对于编程人员来说，如果使函数有统一出口，可有效避免或减少类似

问题。

## 三、维护类代码问题

### 1、 统一枚举类型的使用

**【案例3.1.1】**

【正　　文】

在对基站告警屏蔽的测试中，偶尔使用一个告警ID测试，基站出乎意料地报错：消息与物理位置不一致；通过跟踪消息，发现主机消息中使用的单板类型值与基站的不一致；查看主机程序得知：对基站的单板类型（TRX）的定义中：告警台和告警屏蔽使用0X23（BCID_CUI）,而据基站开发人员称，他们使用的是0X09（BCID_TRX）。

该问题给我们的启示是：测试中我们应尽可能地遍历实际可能的情况；另外希望BTS和BSC的开发人员间的协作更加精密无懈可击！

### 2、 注释量至少占代码总量的20％

**【案例3.2.1】对XXX产品BAM某版本部分代码注释量的统计**

注释比例统计　　比例（

==========

======

0.000(　0/ 2160)　　0.000(　0/　352)　　0.000(　0/　48)

hlr\source\include\aa.h

0.000(　0/ 1317)　　0.000(　0/　137)　　0.000(　0/　64)

newalarm\source\include\alarmerr.h

0.080(　228/ 2841)　　0.115(　49/　426)　　0.030(　7/　230)

newalarm\source\include\alarmrec.h

0.222(　220/　988)　　0.213(　37/　173)　　0.115(　11/　95)

newconfig\src\include\alarmrec.h

0.000(　0/　329)　　0.000(　0/　27)　　0.000(　0/　24)

newfhlr\source\include\alarmrec.h

0.151(　691/ 4563)　　0.242(　128/　528)　　0.114(　26/　228)

hlr\source\assembler.c

0.229( 631/ 2748) 0.361( 113/ 313) 0.160( 22/ 137)

newalarm\source\assembler.c

==========

======

0.088(213896/2423601) 0.122(32953/268773) 0.082(10475/127119) 总计（整个项目）

8% 12% 8%

## 四、产品兼容性问题

### 1、系统配置、命令方式

**【案例4.1.1】**

事故现象：有时RPU A不能被其他网络设备访问，从其他主机或RPU板上PING RPU A，不通。RPU A自己PING自己，也不通。RPU板复位后，恢复正常。此问题偶然出现。

问题分析步骤：

首先分析造成RPU板网络不通的原因通常有以下几种：

A) 物理连接的问题(如网口坏)；

B) 以太网设置为自环工作方式；

C) 以太网链路层协议有误，Ethernet II、SNAP不一致；

D) MAC地址非法(如为广播地址或首位不是偶数等)；

E) MAC或IP地址与其它网络设备重复；

F) IP协议设置为不转发；

G) 设置了防火墙；

经过检查，可以完全排除A、B、C、D、E、F这几种原因，而以太网口也没有设置任何防火墙规则，默认的包过滤设置为允许通过。但不能排除是防火墙的原因。

为了验证是否是防火墙造成的，打开RPU A的防火墙调试信息。果然，发现是防火墙有限制，采用了防火墙规则2，该规则限制访问网络设备。

可是以太网口的防火墙没有配置任何规则，从理论上来说，它应该只适用规则0，即默认规则。是什么原因使以太网口采用了2号规则呢？

进一步观察调试信息，发现以太网口所分配的内部用户号为49！这时我们想到防火墙所用的内部用户号可能与DMU通道号有直接对应关系。经过试验，果然发现对应普通拨号来说，防火墙内部用户号就等于用户所占用的DMU通道号。

这时，把RPU板上的所有DMU通道闭塞，只保留49号DMU通道，用户拨号上网，让它占用第49号通道，并使该用户采用第3号防火墙规则。这是，以太网口同样也该为收第3号防火墙规则限制。

最后，我们从头重复一次刚才的过程：

1、复位RPU板后，用户没有拨号上网，RPU板以太网口所采用的防火墙的内部用户号为49，采用第0号防火墙规则。

2、第49号MODEM有用户上网，且该用户采用的防火墙规则为X(0<X<=50)，则以太网口防火墙规则也相应为X。

至此，问题已经查明，防火墙对以太网口处理不当，不应该分配内部用户号为49给以太网口，以致与第49号DMU通道形成不应有的关联。这给系统运行带来极大隐患。

**【案例4.1.2】**

数据通信某产品，在进行终端并行测试的过程中发现一很奇怪的问题。先通过TELNET或NETTERM登录到主机系统上，然后在两个TELNET终端同时以大包PING主机，PING XXX。XXX。XXX。XXX　，包长为1000个Bytes，其中一个终端收到两个应答后，不再有任何反应，另一个终端收到一个应答后，显示超时，此后不再有任何反应。重新TELNET登录，还可以登录一个TELNET终端，但试图再登录第二个TELNET终端失败（此系统共支持3个TELNET终端）。测试人员怀疑此前登录的两个TELNET任务已经死掉。于是通过超级终端登录到串口，打开TRACE信息，TRACE信息显示当前已有3个TELNET任务处于运行状态，但实际上此时已只有一个TELNET终端可用，另两个TELNET终端已经没有反应了，而且确定再没有其它人登录到此交换机系统上，至此，确定是前述两个

TELNET任务已经死掉。但是，在与开发人员一起重现此问题时，奇怪的现象发生了，在有的机架上重复上述测试步骤，问题每次都能重现，而在有的机架上重复上述测试步骤，却非常正常，没有任何问题。开发人员仔细检查程序也无法发现问题症结所在。此问题持续多日无法解决。最后，经众开发人员会诊，怀疑是pSOS系统配置的问题，经比较两个产生不同现象的机架上pSoS系统的系统配置文件，发现其中关于pNA+的Buffer配置部分，某些配置不同。将配置改为无问题的机架上的系统配置后，进行测试，问题消失。若恢复原配置则问题重现。因此，此问题最后确定为系统配置有误。

由此案例，我们可得到一个经验，那就是，当系统在不同机架上运行现象不同时，除考虑其它可能原因外，还应考虑是否操作系统配置不当。另外，此问题还带出了一个附加的问题，那就是，我们的版本管理存在较大的问题，同为测试机架，但不同的机架上运行的程序版本却不一样。

2、设备对接

**【案例4.2.1】**

【正　　文】

测试环境：

A8010 Refiner通过中继线与Bell 1240交换机进行对接。

现象描述：

从Bell 1240交换机引出PRI中继线接入到Refiner的E1接入口上，接通后看Refiner RPU板的中继灯为灭状态，但用电话进行测试，线路不通为忙音。

原因分析：

问题的原因可能有两点：

1、Bell 1240未送主叫与被叫号码。

2、Bell 1240交换机PRI选路方式与Refiner接入服务器不同 。

解决办法：

(1) A8010 Refiner配置好数据后,RPU板正常运转,中继显示正常，但用电话拨号后为忙音，后用Debug 进行调试发现Bell 1240交换机未将用户被叫号码送来，而且主叫号码也不对，经与与交换机人员联系发现对方的传送的主叫全部为映射的虚拟号码，只有将接入服务器的主叫号码改为此虚拟号码才行，被叫配置后可正常传送至接入服务器。.

(2) 数据配置正常后，发现DMU可进行选路但一选就断开，经调试发现Bell 1240交换机的PRI选路方式为Channel方式，Refiner接入服务器的默认选路方式为Ts-map方式，经更改配置后用户可正常接入。

启示与分析

由于我们自己的测试环境一般都是我们的交换机，我们的接入服务器，因此我们的交换机与接入设备上运行，数据都是配置好的，环境是稳定的，到了与不是我们的设备进行对接时才发现并不一定别人的设备都与我们完全兼容，有许多东西都是在问题发生后才认识到的(如PRI的选路方式)，以后在测试中一定要深入细致的测到每一个细小的问题，并且要近可能的与其它厂家的设备进行对接测试，只有这样才能保证我们产品在卖到局方后的正常稳定运行。

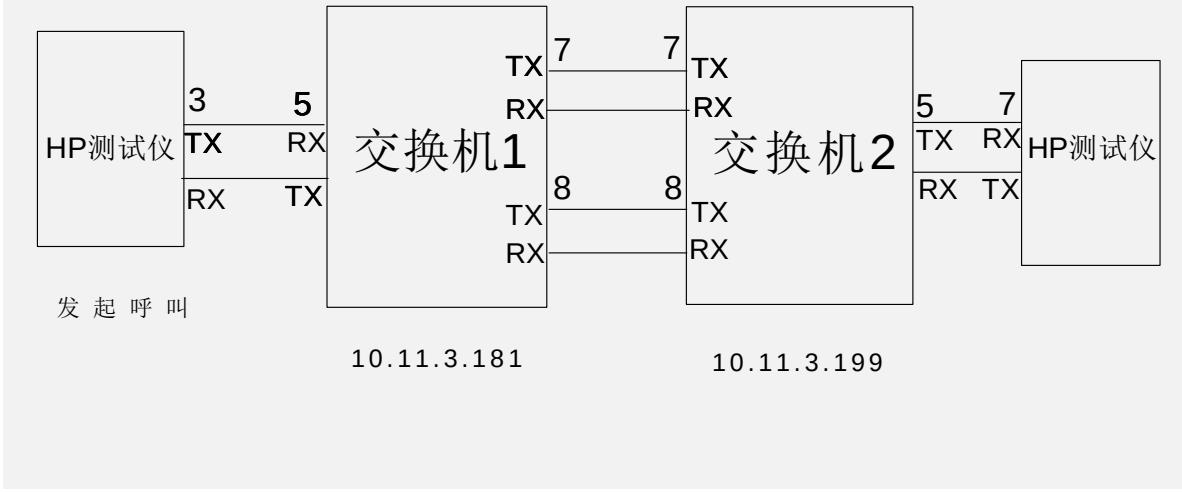
3、其他

**【案例4.3.1】**

【正　　文】

负荷分担的基本思想是通过每个链路中的带宽来均匀分布流量，目前还没有考虑PMP，2M UNI，34UNI，没有考虑优先级和百分比，况且对于同一局向的路由表的地址长度一定要相等，目前对于BEST EFFORT呼叫仅分配150K的带宽（UBR业务），此参数可在static void GetBandwidth( STraffic *sTempTraffic, DWORD *pdwBandwidth )函数细调，为了达到统计均匀，此参数要合适。如有二条负荷分担路由A、B，A中已建了10M的PVC，如果上述参数太少，则所有BEST EFFORT呼叫都在B上，而不会到A上，如果选取150K，则在B上有70个呼叫（BEST EFFORT）后，就在A、B上同时都分担呼叫。

根据上述负荷分担的基本思想进行了负荷分担 的功能测试。测试中的线路




连接图如下图所示。

HP测试仪设置Forword/Backword　Peak Cell Rate为1000cells/s，测试仪的3口向7口发起呼叫（信令类型为UNI3.1）。在没有发起呼叫前，将交换机 1、2的5、7和8口的ILMI和信令均激活，这三个口不建立与其它光口的链路，此时这三个口的已用带宽（Used Band-In/Out）为2000cellsps （sh port 可以看到已用带宽为信令链路所占）。现在由测试仪3口发起一次呼叫，sh port 5、7可以看到已用带宽由2000变为3000，同时建立起一条5口和7口间的SVC。再发起第二次呼叫，sh port 8可以看到已用带宽由2000变为3000，而 5口的已用带宽由3000变为4000，同时增加了一条5口和8口间的SVC。如此不断地发起呼叫，可以看到7口和8口的已用带宽是交替在增长的，这样就证明了负荷分担功能的实现。

Radium.MPU%sh po 7

Port status　　: Active

Band width : STM_1

O/E attribute : Optical

Clock attribute : Source timing

Type : UNI

Loopback mode : No loop

Alarm status : NODEFECT

Max Band-In : 353207

Max Band-Out : 353207

Used Band-In : 2000

Used Band-Out : 2000

Max VPCs : 100

Max VCCs : 1900

Used PVPs : 0

Used PVCs : 2

Used SVCs : 331

Test mode : off

在测试的过程中出现了如下的现象：交换机1的7口和交换机2的7、8口的ILMI注册成功，三个口都获得了对端的网络前缀，只有交换机1的8口的ILMI状态为VERIFYING，却没有获得对端的网络前缀，从LOG信息看，该口的ILMI注册过程正确。经查该口的ILMI协议版本为3.1，信令版本为UNI3.1，而其它三个口的ILMI协议版本为4.0， 信令版本为IISP，发起的所有呼叫都从交换机1的7口中继到交换机2，8口没有进行分担。将交换机1的8口的ILMI协议版本设置为4.0，8口即可进行负荷分担。这说明ILMI协议版本没有实现自适应的功能。后与开发人员沟通后得知，我们交换机的ILMI4.0版本可以自适应其它厂家的ILMI3.1版本，但不自适应我们自己交换机的ILMI3.1版本，对端网络前缀的获得也仅限于ILMI4.0 ，并且是非协议规定的，是由我们自己设计的，设计中没有考虑ILMI3.1对获得对端网络前缀的支持。在实际开局中，ILMI是不激活的，信令版本是由手工设置为IISP的，以实现与其它厂家的产品的互通。

测试结果说明负荷分担功能已经正确地实现了，但条件是ILMI的版本必须为

4.0。

## 五、版本控制问题

1、新老代码中同一全局变量不一致

**【案例5.1.1】**

【正　　文】

开始时，RPU板的网口地址为202.2.68.56，通过以太网口对RPU板上的软件和数据进行加载，RPU板运行正常，程序和数据全部正确；改变RPU板的地址（10.2.68.56），进行用户呼叫接入测试，发现用户正确输入的用户名和口令不能验证通过，在RPU板的telnet窗口中打开RADIUS调试开关，显示认证请求报文和计费报文发往正确的RADIUS服务器，但是一直没有收到应答，局方维护人员查看RADIUS服务器，也找不到任何记录，验证和计费始终不能成功。

接下来通过以下几个途径尝试定位问题：

1、在另一块RPU板上加载老版本的程序（注意这块RPU板的地址已经是10.2.68.57），重新测试，认证和计费正常进行，没有出现以上问题，可以排除对方RADIUS服务器存在问题的可能。

2、可能出现问题的地方是RADIUS报文的发送和接受，考虑到RADIUS报文是以UDP包的方式传送，所以在RPU的telnet中打开UDP报文的调试开关，跟踪RADIUS报文，发现RADIUS报文已经发送，但是在其填充的源地址（Source IP Address）字段的值是202.2.68.56，由此可以定位问题，是RADIUS模块在填充发送的RADIUS报文的源地址时仍然用RPU板改变地址前的老地址，导致RADIUS服务器发送的应答报文不能正确返回。

问题已经定位，时间紧迫，不可能立即修改程序，所以只能采取重新复位所有的RPU板的方法来暂时回避问题再次发生。

启　示：

1、软件全局变量的更改，要考虑到每一个模块受其影响的可能；

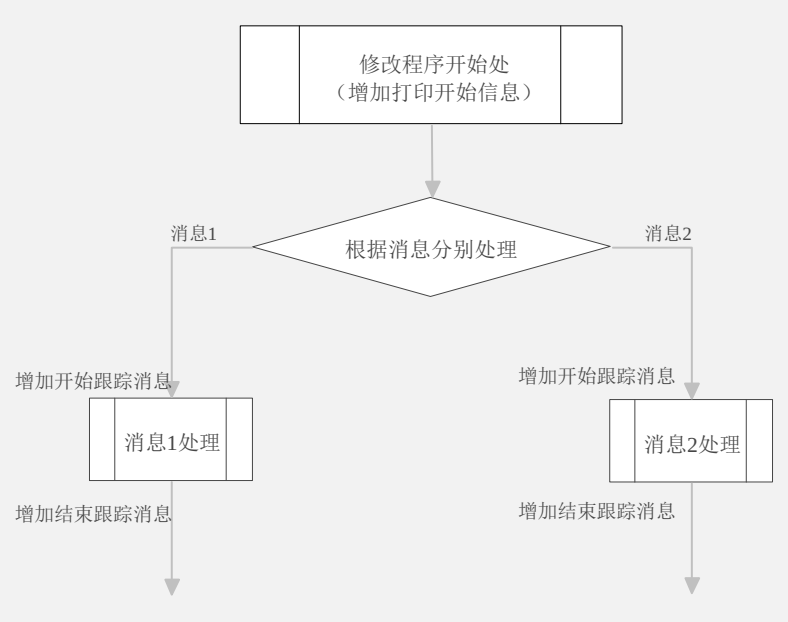
2、测试人员在测试中要尽可能地多考虑到各种情况下的边缘取值。

## 六、可测试性代码问题

### 1、调试信息/打印信息的正确性

**【案例6.1.1】**

【正　　文】为制作软件呼叫器，对SPT板的放音程序进行了修改，但是调试中发现进行呼叫后一段时间交换机自动重启。于是在修改的代码开始处增加了打印消息，跟踪程序流程。程序结构如下：




再次跟踪打印信息，发现消息打印消息1开始后就会发生重启现象，多次重试，发现消息1的结束跟踪打印有时有有时无，而消息2的打印一直没有跟踪到。初步判断程序错误发生在对消息1的处理于是集中力量检查对消息1的处理，但经过长时间的检查，并未在此处理流程中发现错误。于是在消息1的开始处直接RETURN，屏蔽掉消息1的处理。再次测试，发现重启现象依旧，偶尔能够看到函数入口处的打印消息。

为什么会出现这样的情况呢？什么情况导致重启呢？

从修改的情况看，可能的地方只有此两条消息处，会不会是第二条消息的处理造成的？虽然从现象看不太可能。报着试一试的想法在消息2的入口处也加了一RETURN，居然重启现象不重现了。

立即检查消息2的处理，很快发现在处理的开始由于指针使用错误导致程序死机重启。

错误是简单的，但是根据现象却使人得到错误判断。为什么消息2导致死机但其前的打印消息为什么后台看不见呢？

考虑一下交换机的消息打印机制，我们会发现，在程序中的打印消息并不是马上在后台上显示，消息从主机传到后台需要一定的时间。如果用Printf打印消息不久后，程序就发生严重错误而导致程序死机，而打印消息还没有来得及发送到后台，当然就不会出现我们想“应该有的”打印消息。

【总结】

1、由于环境的特殊性，我们常常认为“应该”的事件并没有发生，导致我们对问题迷惑不解；

2、“表面现象往往是骗人的”，要找到问题的实质可能需要绕过明显摆在我们面前的“表面现象”，从另一个角度考虑一下；

## 案例与练习第二部分

# 练习

用户login模块的编制：首先password.txt记录了注册用户的id和口令，password.txt文件每一行格式为：Id%%%%Password

其中：

Id: (字母+数字) 最多16 字符

Password: (字母+数字) 最多16字符

要求用户输入id/口令时，程序通过查找password.txt，检查用户是否存在，是否合法，直接输出相应结果；当用户直接回车时，程序退出。要求考虑一般的异常。

**规则：**

1. 小组来完成；
2. 首先花30--40分钟画流程图；
3. 然后30－40分钟编码，用C语言，人人都编；
4. 30-45分钟小组内部讨论，选择最有代表性的，认为比较好的，先自行评点；
5. 45分钟--1小时，集体评审；

   1〕不符合规范的是部分------〉集体自醒；

   2〕符合规范的----------〉加强巩固；
6. 要求严格按软件编程规范来进行编码、评审；
7. 保留代码，下一节要用。