

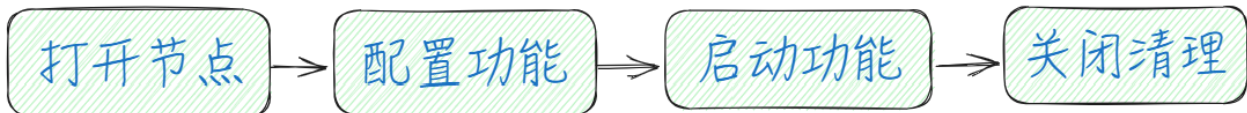
How to Use 1394B

在航电领域，1394是一个非常重要的通信总线，用好它是非常有必要的。

0. 前言

对于1394，所有操作都是围绕节点进行的，在我们开发过程中，大多数只会遇到CC和RN这两个节点。

对于这两个节点，它们有一套相同的操作逻辑，如下所示：



1. 打开节点。
2. 配置收发参数。
3. 开启收发功能。
4. 结束关闭清理。

因此，我们要对CC和RN都要做这些操作。

一. 打开并初始化节点

在程序中我们通常会定义设备以及节点的编号，同时根据需要定义好节点的句柄并初始化为nullptr，如下所示：

```
static const int RN_NODE_NO = 1; // RN节点编号
static const int CC_NODE_NO = 0; // CC节点编号
static const int DEVICE_NO = 0; // 设备编号

_TNF_Node_Struct* hCcNode = nullptr; // CC节点句柄
_TNF_Node_Struct* hRnNode = nullptr; // RN节点句柄
```

要想打开节点，必须使用以下函数，传入节点描述，节点号和设备号，节点描述我们大多数不需要，因此可以填nullptr，打开成功便可返回完整的节点句柄。

```
TNFU32 deviceNo = DEVICE_NO;
TNFU32 ccNodeNo = CC_NODE_NO;

hCcNode = static_cast<_TNF_Node_Struct*>(Mil1394_XT_OPEN(nullptr, &deviceNo,
ccNodeNo));
if (hCcNode == nullptr) {
    // 【错误】CC节点打开失败
    return false;
}
```

```
// RN节点同理
```

在打开后，我们需要使用某种方式尽量等待一段时间，这样确保节点可以完全打开。

二.配置消息参数

节点打开之后便是最重要的部分，配置参数。

对于不同的节点，它们都有相同的一部分，那便是设置端口速率和发送速率：

```
// 端口速率
enum PortSpeed {
    PORTSPEED_S100 = 0,          // 100Mbps
    PORTSPEED_S200 = 1,          // 200Mbps
    PORTSPEED_S400 = 2           // 400Mbps
};

// 使能/失能
enum Ability {
    ABILITY_DISABLE = 0,         // 禁用功能
    ABILITY_ENABLE = 1           // 启用功能
};

// 通信参数定义
static const int COMM_SPEED      = PORTSPEED_S400;          // 通信速率: 400Mbps

static const int CC2RN_ASYNC_MSG_ID   = 0x1001;            // 异步消息ID
static const int CC2RN_ASYNC_PAYLOAD_LEN = 256;            // 异步消息载荷长度
static const int RN2CC_ASYNC_MSG_ID   = 0x2001;            // 异步消息ID
static const int RN2CC_ASYNC_PAYLOAD_LEN = 128;            // 异步消息载荷长度

// 设置通信端口速率为S400(400Mbps)
if (Mil1394_Port_Speed_Set(hCcNode, COMM_SPEED) != OK) {
    // 【错误】设置CC节点通信速率失败
    Mil1394_Close(hCcNode);
    hCcNode = nullptr;
    return false;
}

// 设置消息发送速率为S400(400Mbps)
if (Mil1394_SEND_Speed_Set(hCcNode, COMM_SPEED) != OK) {
    // 【错误】设置CC节点发送速率失败
    Mil1394_Close(hCcNode);
    hCcNode = nullptr;
    return false;
}
```

值得注意的是，相互通信的节点必须设置相同的速率才能正常的通信。

同时，根据节点的具体设计，判断是否开启STOF接收，以CC节点为例，大多数情况下CC节点只会发送STOF消息，不会接收STOF消息，因此需要禁用掉接收功能：

```
// 设置CC节点工作模式 - 禁用STOF接收功能（因为CC节点在本应用中只负责发送STOF消息）
if (Mil1394_RCV_STOF_ENABLE(hCcNode, ABILITY_DISABLE) != OK) {
    // 【错误】设置CC节点工作模式失败
    Mil1394_Close(hCcNode);
    hCcNode = nullptr;
    return false;
}
```

最基础的设置完毕后，接下来便是各种消息的设置。

在1394中，有4种不同的消息：

- 总线复位消息
- STOF消息
- Asyn异步消息
- event事件消息

在这些消息中，我们用的最多的便是STOF消息和Asyn异步消息，接下来也只关心这两个消息。

STOF消息配置

对于STOF消息，它是周期性发送，因此需要我们指定发送周期，这里为其设定1秒的较大的周期，并且在接收方，消息并不一定会以非常准确的时间点到达，因此对于接收方还需要设置一下到达的误差范围，这里成为门限，具体设置如下所示：

```
static const int LARGE_PERIOD    = 1000000; // 大周期：1秒(1,000,000微秒)
static const int PERIOD_LIMIT    = 50000;   // 容错范围：50毫秒(50,000微秒)

// CC节点—设置STOF消息发送周期为大周期(1秒)
if (MSG_STOF_Period(hCcNode, LARGE_PERIOD) != OK) {
    // 【错误】设置CC节点STOF周期失败
    return false;
}

// RN节点—设置STOF消息接收周期（与CC节点保持一致）
if (MSG_STOF_Period(hRnNode, LARGE_PERIOD) != OK) {
    // 【错误】设置RN节点STOF周期失败
    return false;
}

// RN节点—设置STOF接收门限（容错范围）
if (MSG_RECV_STOF_limitPer(hRnNode, PERIOD_LIMIT) != OK) {
    // 【错误】设置RN节点STOF接收门限失败
    return false;
}
```

设置完周期相关之后便是消息内容相关东西。

对于1394中的消息的配置，它们都是通过消息配置结构体和消息数据包结构体两部分完成。

以STOF消息为例：对于STOF的发送，我们首先要使用 `_TNF_STOF_CFG_Struct` 对STOF消息的基本参数进行选择 and 设置，然后使用 `_MsgSTOF` 对STOF消息的数据进行填充，这两部分都有对应的加载操作，这样做之后便可以进行发送。具体代码如下所示：

```
_TNF_STOF_CFG_Struct cc2RnStofCfg;      // CC节点发送到RN节点的STOF配置结构体
_MsgSTOF              cc2RnStofPackage;  // CC节点发送到RN节点的STOF消息包

// 初始化并配置STOF配置结构体
memset(&cc2RnStofCfg, 0, sizeof(cc2RnStofCfg));
cc2RnStofCfg.STOFPeriod = LARGE_PERIOD;  // 设置大周期为1秒(1,000,000微秒)
cc2RnStofCfg.SysCntType = 1;             // 系统计数类型，用于时间同步
cc2RnStofCfg.STOFPayload4 = 0;           // 初始化负载4值
// 应用STOF配置到CC节点
if (MSG_STOF_SEND_CFG(hCcNode, &cc2RnStofCfg) != OK) {
    // 【错误】配置CC节点STOF消息失败
    return false;
}

// 初始化并设置STOF消息数据
memset(&cc2RnStofPackage, 0, sizeof(cc2RnStofPackage));
// 设置STOF消息的各个负载值
cc2RnStofPackage.STOFPayload0 = 0xCCCC0001; // 消息标识，用于RN节点识别CC节点消息
cc2RnStofPackage.STOFPayload1 = 0x00000000; // 时间戳低位，初始化为0
cc2RnStofPackage.STOFPayload2 = 0x00000000; // 时间戳高位，初始化为0
cc2RnStofPackage.STOFPayload3 = 0x00000000; // 消息计数器，初始化为0
cc2RnStofPackage.STOFPayload4 = 0x00000000; // 备用负载值
cc2RnStofPackage.STOFPayload5 = 0x00000000; // 备用负载值
cc2RnStofPackage.STOFPayload6 = 0x00000000; // 备用负载值
cc2RnStofPackage.STOFPayload7 = 0x00000000; // 备用负载值
cc2RnStofPackage.STOFPayload8 = 0x00000000; // 备用负载值
// 设置STOF消息发送数据
if (MSG_STOF_SEND_DATA_Set(hCcNode, &cc2RnStofPackage) != OK) {
    // 【错误】设置CC节点STOF消息数据失败
    return false;
}
```

而对于STOF接收，在设置完周期后，便不用专门的参数配置。

值得注意的是，在我们进行STOF配置结构体时，除了周期，还有一个系统计数类型和负载4的初始化。当我们赋值1时，每发一次STOF消息，这个计数就会自动加1，而负载4就是其初始值。有了这个东西，我们便可以在接收时查看这个计数来判断一些东西。

在填充STOF的数据时，我们可以发现STOF的结构便是9个32字的负载，这些负载存储信息，我们会发现负载3也是消息计数器，这与之前的负载4有冲突，这是为什么呢？深入了解可以发现，负载4仅仅是设置了初始值，当设置完毕后，之后所有的计数本身会放在负载3上，这便是不同的负载都出现计数的原因。

STOF的结构如下图以及代码所示：



```
typedef struct {
    TNFU32  Header1394;    /* 1394协议头部 */
    TNFU32  STOFPayload0;  /* 载荷0 */
    TNFU32  STOFPayload1; /* 载荷1, 其中5、4位代表余度分支, 配置错误可能导致产品进入故障模式 */
    TNFU32  STOFPayload2; /* 载荷2 */
    TNFU32  STOFPayload3; /* 载荷3 */
    TNFU32  STOFPayload4; /* 载荷4, 通常用于系统计数 */
    TNFU32  STOFPayload5; /* 载荷5 */
    TNFU32  STOFPayload6; /* 载荷6 */
    TNFU32  STOFPayload7; /* 载荷7 */
    TNFU32  STOFPayload8; /* 载荷8 */
    TNFU32  STOFVPC;      /* 垂直奇偶校验值 */
} _MsgSTOF;
```

这样STOF消息就算配置完成。

Asyn异步消息配置

在1394中最常用的便是Asyn异步消息，最复杂的也是异步消息，因此配置异步消息也是极为复杂的。

配置Asyn异步消息时，除了消息配置结构体和消息数据包结构体还要添加额外的配置数组加载。配置数组的加载使用以下方式：

```
MSG_ASYNC_LoadCfg(节点句柄, 发送配置数量,接收配置数量,通道号,发送配置数组,接收配置数组);
```

我们可以看到配置数量以及对应数量的配置数组，从驱动库中寻找后可以做出以下内容：

```
static const int CC2RN_ASYNC_MSG_ID    = 0x1001;    // 异步消息ID
static const int CC2RN_ASYNC_PAYLOAD_LEN = 256;      // 异步消息载荷长度
```

```

static const int RN2CC_ASYNC_MSG_ID    = 0x2001;           // 异步消息ID
static const int RN2CC_ASYNC_PAYLOAD_LEN = 128;           // 异步消息载荷长度

// CC节点Asyn消息配置数组(发1收1)
TNFU32 CcNodeTxConfigArr[5] = {
    //-控制字、1394头、消息ID、保留字段
    0x0000, TX_PACK1394HEADER(CC2RN_ASYNC_PAYLOAD_LEN, 1), CC2RN_ASYNC_MSG_ID,
    0x0000,
    0x8000
};
TNFU32 CcNodeRxConfigArr[3] = {
    //-控制字、消息ID、保留字段
    RX_PACKCONTROLWORD(RN2CC_ASYNC_PAYLOAD_LEN), RN2CC_ASYNC_MSG_ID, 0x0000
};

// RN节点Asyn消息配置数组(发1收1)
TNFU32 RnNodeTxConfigArr[5] = {
    //-控制字、1394头、消息ID、保留字段
    0x0000, TX_PACK1394HEADER(RN2CC_ASYNC_PAYLOAD_LEN, 1), RN2CC_ASYNC_MSG_ID,
    0x0000,
    0x8000
};
TNFU32 RnNodeRxConfigArr[3] = {
    //-控制字、消息ID、保留字段
    RX_PACKCONTROLWORD(CC2RN_ASYNC_PAYLOAD_LEN), CC2RN_ASYNC_MSG_ID, 0x0000
};

```

对于发送配置数组，其最少有5个元素组成：*控制字*，*1394头*，*消息ID*，*保留字段*，*结束标志*。这5个周游1394头和消息ID是需要我们来填充的，其余都是固定的。

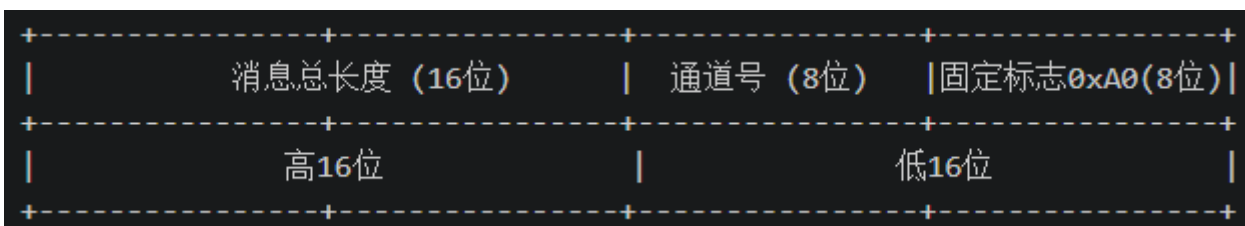
对于消息ID，我们提前设计好就行了，比较复杂的是1394头的计算。在这里，给出1394头的计算宏定义如下所示，其中需要我们传入*有效载荷长度（字节数）*和*通道号*这两个参数：

```

// 1394头计算
#define TX_PACK1394HEADER(payloadLength, channel) ((quint32)(payloadLength + 8 *
4) << 16) | ((quint32)channel << 8) | 0xA0

```

让我们想一想为什么要这样设计呢？



从驱动库中我们可以找到以上这样的结构，其是一个32位字，由（高16位的消息总长度），（中8位的通道号），（低8位的固定标志）组成，而消息总长度又由（有效载荷长度）和（1个32位字的头部长度）组成，因此这样我们便可以明白为什么要这样设计了。

同理，对于接收配置数组，其最少由3个元素组成：*接收控制字*，*消息ID*，*保留字段*。也只有接收控制字和消息ID需要我们填充。

对于消息ID，需要填入发送消息的消息ID，而接收控制字，其计算的宏定义也如下所示，我们需要传入的仍是*有效载荷长度*。

```
#define RX_PACKCONTROLWORD(payloadLength) (quint32)(payloadLength + 8 * 4) /  
(quint32)4
```

同样让我们想想为什么要这样设计？

接收控制字的意义是32位字的个数，经过反向操作我们便可以得到*有效载荷长度* + $8 * 4$ 的值，再反向计算便可以得到有效载荷长度。这样我们便可以推出要接收多少数据。这便是接收控制字的意义。

这样就清楚了这样配置的原因。同时也完成了Asyn异步消息的接收配置。

而对于Asyn异步消息的发送配置，遵循之前的方式即可，定义配置结构体变量，填充相应内容，应用配置即可，代码如下所示：

```
// 初始化并配置异步消息配置结构体  
memset(&cc2RnAsynCfg, 0, sizeof(cc2RnAsynCfg));  
cc2RnAsynCfg.Header1394 = TX_PACK1394HEADER(CC2RN_ASYNC_PAYLOAD_LEN, 1); // 设置  
1394头  
cc2RnAsynCfg.MessageID = CC2RN_ASYNC_MSG_ID; // 设置消息ID  
cc2RnAsynCfg.HeartBeatStyle = 1; // 心跳样式：步进方式  
cc2RnAsynCfg.HeartBeatEnable = 1; // 启用心跳  
cc2RnAsynCfg.HeartBeatStep = 1; // 心跳步长  
cc2RnAsynCfg.HeartBeatinitValue = 0; // 心跳初始值  
cc2RnAsynCfg.SoftVPCenable = 1; // 启用软件VPC  
  
// 应用异步消息配置到CC节点（索引0）  
if (MSG_ASYNC_SEND_CFG(hCcNode, 0, &cc2RnAsynCfg) != OK) {  
    // 【错误】配置CC节点异步消息失败  
    return false;  
}
```

而对于Asyn异步消息的数据填充，我们只需要定义数据包结构，根据我们数据的有效长度进行填充，我们在设置时尽量选择4的整数倍，原因很简单，因为这里的有效长度都是以字节为单位的，但是数据填充是以32位字位单位的因此，在502个32位字之内尽可能的选择完整的数据。代码如下所示

```
// 初始化并设置异步消息数据  
memset(&cc2RnAsynPackage, 0, sizeof(cc2RnAsynPackage));  
cc2RnAsynPackage.Header1394 = TX_PACK1394HEADER(CC2RN_ASYNC_PAYLOAD_LEN, 1); //  
设置1394头  
cc2RnAsynPackage.MessageID = CC2RN_ASYNC_MSG_ID; //  
设置消息ID  
cc2RnAsynPackage.payloadLen = CC2RN_ASYNC_PAYLOAD_LEN; //
```


设置有效载荷长度

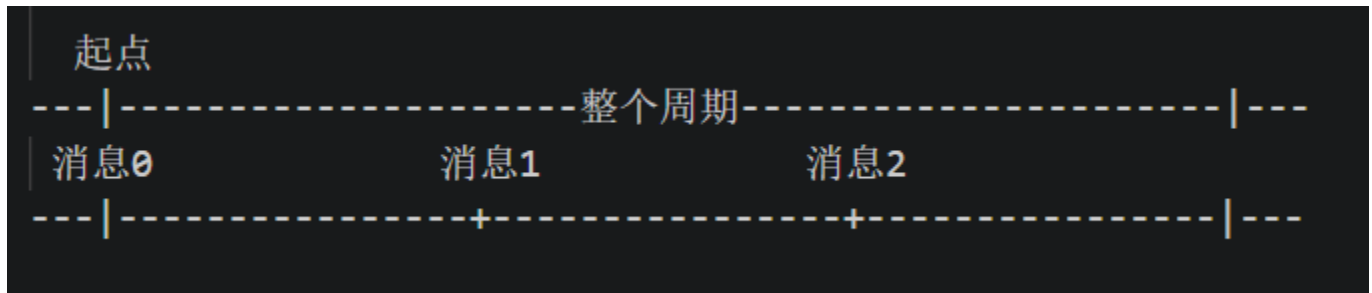
```
// 填充异步消息数据载荷
for (int i = 0; i < CC2RN_ASYNC_PAYLOAD_LEN/4 && i < 502; i++) {
    cc2RnAsynPackage.msgData[i] = 0xABCD0000 + i; // 填充递增数据
}
```

当数据填充完毕后，就要进行总的数据的设置，设置的代码如下所示：

```
// 设置异步消息发送数据
if (MSG_ASYNC_SEND_DATA_Set(hCcNode, 0, 0, reinterpret_cast<TNFU32*>
(&cc2RnAsynPackage)) != OK) {
    // 【错误】设置CC节点异步消息数据失败
    return false;
}
```

在设置中我们可以发现有两个0作为参数，这是怎么回事？好像没有什么是0，这两个0该怎么解释呢？回顾之前的异步消息配置数组的加载，是不是写了消息的数量，而这里的第一个0便是消息的索引，由于我们只有一个消息，所以直接写为0，但是如果有多条消息时，就必须使用循环的方式来对每个消息进行设置。那第二个0是什么？

我们可以想象一下，如果有3条消息在同一时刻发送，会发生什么？会在总线上拥挤导致三个都发送不了，因此，就需要某种机制将消息分开发送，这便是第二个0的含义——消息偏移。如下图所示，每个消息以起点为基础向后延迟一定时间发送，这样就可以保证消息不会拥挤。第一个消息偏移为0，因此这里输入0。



综上所述以及浏览驱动库，我们可以找到Asyn异步消息的结构，如下图以及代码所示：

Header1394	MessageID	Security	NodeID
payloadLen	healthStat	heartBeat	msgData[0]
msgData[1]	msgData[2]	...	msgData[N-1]

```
typedef struct {
    TNFU32  Header1394;    /* 1394协议头部 */
    TNFU32  MessageID;     /* 消息ID, 用于标识消息类型 */
    TNFU32  Security;      /* 安全相关字段 */
    TNFU32  NodeID;        /* 节点ID */
    TNFU32  payloadLen;    /* 有效载荷长度 */
    TNFU32  healthStat;    /* 健康状态 */
    TNFU32  heartBeat;     /* 心跳值 */
    TNFU32  msgData[502]; /* 消息数据, 最大支持502个32位字 */
}_MsgAsyn;
```

这样，Asyn异步消息的发送也配置完成了。

到这里几乎所有的配置就已经完成了。接下来便是功能的开启。

三.开启消息收发

对于消息收发的启动，我们只需要记住，先接收，再发送即可，而具体的代码如下所示：

```
// 启用消息接收，以启动消息支持功能
if (MSG_RECV_Ctrl(hCcNode, ABILITY_ENABLE) != OK) {
    // 【错误】启用CC节点消息接收失败
    return false;
}

// 启动STOF消息周期性发送功能
if (MSG_STOF_SEND_Ctrl(hCcNode, ABILITY_ENABLE, LARGE_PERIOD, 0, 0) != OK) {
    // 【错误】启动CC节点STOF消息发送失败
    return false;
}

// 启动所有异步消息周期性发送功能
```

```
if (MSG_ASYNC_SEND_ALLCtrl(hRnNode, ABILITY_ENABLE) != OK) {  
    // 【错误】启动CC节点异步消息发送失败  
    return false;  
}
```

这样设置之后1394相关便会自行根据设置发送和接收消息。

四.获取并处理消息

对于STOF消息和Asyn异步消息，它们都有两种大的方法来接收消息，**阻塞式**和**非阻塞式**。

阻塞式接收

当我们主动调用以下方法时，系统会进行阻塞式的接收，除非超时或接收到消息，否则就会阻塞程序。

STOF消息阻塞式接收代码如下所示：

```
// 调用API接收STOF消息，使用指定的超时时间  
if (MSG_STOF_RECV(hRnNode, &receivedStofPackage, timeoutMs) == OK) {  
    // 【接收】成功接收到STOF消息  
    printf(" Payload0: 0x%08X\n", receivedStofPackage.STOFPayload0);  
}  
else {  
    // 【超时】STOF消息接收超时  
}
```

Asyn异步消息阻塞式接收代码如下所示：

```
// 调用API接收异步消息，使用指定的超时时间和消息ID  
if (MSG_RECV_Packet_Asyn(hRnNode, ASYNC_MSG_ID, &receivedAsynPackage, nullptr) ==  
OK) {  
    // 【接收】成功接收到异步消息  
    printf(" MessageID: 0x%08X\n", receivedAsynPackage.MessageID);  
    printf(" PayloadLen: %d 字节\n", receivedAsynPackage.payloadLen);  
}  
else {  
    // 【超时】异步消息接收超时  
}
```

相对于阻塞式，我更推荐非阻塞式的接收方式。

非阻塞式接收

对于4种消息，它们的非阻塞式接收是同一个方法，那便是**接收消息队列**，我们获取消息队列，遍历队列内容，获取消息，根据消息类型字段确认消息，再执行相应的处理函数。

代码如下所示：

```

void RNnodeWork::pollAndProcessMessages() {
    // 检查是否有消息待处理
    TNFU32 msgCount = 0;
    _RcvMsgList* msgList = nullptr;
    _RcvMsgList* msgLastList = nullptr;

    // 获取消息列表
    if (MSG_RECV_list(hRnNode, &msgCount, &msgList, &msgLastList) == OK) {
        if (msgCount > 0) {
            // 遍历消息列表并处理每条消息
            _RcvMsgList* currentMsg = msgList;
            for (TNFU32 i = 0; i < msgCount && currentMsg != nullptr; i++) {
                processMessage(currentMsg);
                currentMsg = currentMsg->next;
            }
        }
        else {
            // 没有消息待处理
        }
    }
    else {
        // 获取消息列表失败
    }
}

// 处理消息
void RNnodeWork::processMessage(_RcvMsgList *msgList) {
    // 确保是一条可用的消息
    if (msgList == nullptr || msgList->msgstateAddr == nullptr) {
        return;
    }
    // 获取消息状态
    _MsgState* msgState = msgList->msgstateAddr;
    // 根据消息类型进行处理
    switch (msgState->MessageTYPE) {
        case MSG_BUSRESET:
            break;
        case MSG_STOF:
            memset(&receivedStofPackage, 0, sizeof(receivedStofPackage));
            if (MSG_RECV_Packet_STOF(hRnNode, &receivedStofPackage, msgState) ==
OK) {
                //
            }
            break;
        case MSG_ASYNC:
            memset(&receivedAsynPackage, 0, sizeof(receivedAsynPackage));
            if (MSG_RECV_Packet_Asyn(hRnNode, msgState->MessageID,
&receivedAsynPackage, msgState) == OK) {
                //
            }
            break;
        case MSG_EVENT:
            break;
    }
}

```

```

        default:
            break;
    }
}

```

为什么可以使用这种方法，查看驱动库可以看到_RcvMsgLis的结构，其中有几个很关键的成员：**消息状态**,**消息数据指针**,**链表结构**这几个重要成员。

```

// 消息队列链表结构体
typedef struct {
    _MsgState* msgstateAddr; // 消息状态（包含类型信息）
    void* msgAddr;           // 消息数据指针
    TNFU32 listID;
    struct _RcvMsgList* next; // 链表结构
} _RcvMsgList;

```

消息数据指针通过**void***指向具体的消息数据，消息状态保存消息的各个类型数据等，如下所示，其中最重要的便是**MessageTYPE**来判断消息的类型。

```

// 消息状态
typedef struct {
    /* 时间标签字段 */
    TNFU32 RTC :16; /* 16位相对STOF的时间标签 */
    /* 消息属性字段 */
    TNFU32 MessageTYPE :2; /* 消息类型 0:总线复位 1:STOF 2:异步流 3:事件 */
    TNFU32 ser1 :6; /* 保留位 */
    TNFU32 STOFLIMITErr :1; /* STOF包周期超出门限范围错误 */
    TNFU32 SVPCERR :1; /* 软件VPC校验错误 */
    TNFU32 Speed :2; /* 消息包速率 0: S100, 1: S200, 2: S400 */
    /* 错误标志字段 */
    TNFU32 CRCERR :1; /* 循环冗余校验错误 */
    TNFU32 VPCERR :1; /* 垂直奇偶校验错误 */
    TNFU32 LenERR :1; /* 长度错误标志 - 与配置接收长度不一致 */
    TNFU32 PacketFlag :1; /* 1表示接收的消息, 0表示自己发送的消息 */
    TNFU32 LRTCH; /* 64位时间标签(高32位) */
    TNFU32 LRTCL; /* 64位时间标签(低32位) */
    /* 1394头字段 */
    TNFU32 Headerflag :8; /* 头部标志 */
    TNFU32 HeaderChannel :6; /* 头部通道号 */
    TNFU32 Headerser :2; /* 头部序列号 */
    TNFU32 Headerlength :16; /* 头部长度 */
    TNFU32 MessageID; /* 消息ID */
} _MsgState;

```

这样我们就很清楚他的接收机制了。

除了收发的功能，1394还提供了消息计数功能，因为在收发的过程中，计数是一个非常重要的东西，我们可以根据计数来判断消息的连续和准确性。

1394提供了以下这些消息计数类型，帮助我们快速获得相应的计数：

```
// 消息计数
enum TNF_Msg_CNT{
    BusResetCNT = 1U,           /* 总线复位计数 */
    STOF_SendCNT,               /* STOF包发送计数 */
    STOF_RecvCNT,               /* STOF包接收计数 */
    Asyn_SendCNT,               /* 异步消息发送计数 */
    Asyn_RecvCNT,               /* 异步消息接收计数 */
    ERR_Asyn_HCRCNT,            /* 异步消息头CRC错误计数 */
    ERR_Asyn_MsgIDCNT,          /* 异步消息ID错误计数 */
    ERR_Asyn_DCRCNT,            /* 异步消息数据CRC错误计数 */
    ERR_Asyn_VPCNT,             /* 异步消息VPC错误计数 */
    Event_SendCNT,              /* 事件消息发送计数 */
    Event_SendErrCNT,           /* 事件消息发送错误计数 */
    Event_RecvCNT,              /* 事件消息接收计数 */
    ERR_Event_DCRCNT,           /* 事件消息数据CRC错误计数 */
    ERR_Event_VPCNT,            /* 事件消息VPC错误计数 */
    ERR_STOF_VPCNT,             /* STOF消息VPC错误计数 */
    ERR_STOF_DCRCNT,            /* STOF消息数据CRC错误计数 */
    ERR_Asyn_SVPCNT,            /* 异步流软件VPC错误计数 */
    ERR_Event_SVPCNT,           /* 事件消息软件VPC错误计数 */
    ERR_TopicIDCNT              /* 主题ID错误计数 */
};
```

而获取的方法也非常的简单，只需要调用方法传入对应的索引便可直接获取，如下所示：

```
// 获取STOF接收计数
TNFU32 stofRecvCount = 0;
Mil1394_MSG_Cnt_Get(hRnNode, STOF_RecvCNT, &stofRecvCount);

// 获取STOF发送计数
TNFU32 stofSendCount = 0;
Mil1394_MSG_Cnt_Get(hCcNode, STOF_SendCNT, &stofSendCount);

// 获取错误计数
TNFU32 errCounts[ERR_TopicIDCNT + 1] = {0};
Mil1394_MSG_Cnt_Get(hNode, ERR_Asyn_HCRCNT, &errCounts[ERR_Asyn_HCRCNT]);
Mil1394_MSG_Cnt_Get(hNode, ERR_Asyn_MsgIDCNT, &errCounts[ERR_Asyn_MsgIDCNT]);
Mil1394_MSG_Cnt_Get(hNode, ERR_Asyn_DCRCNT, &errCounts[ERR_Asyn_DCRCNT]);
Mil1394_MSG_Cnt_Get(hNode, ERR_Asyn_VPCNT, &errCounts[ERR_Asyn_VPCNT]);
```

到这里，1394常用的内容就完成了。