TOE 模块设计方案

V1.0

文件状态	当前版本	*V1.0*
[√] 草稿 [ ] 正式发布	作　　者	李明
	完成日期	
	文档模板	
	密　　级	*内部使用*

变更历史

版本	完成日期	变更记录	作者	审核	批准
V1.0	20171222	初稿	李明		

1. 功能介绍

1.1. 背景介绍

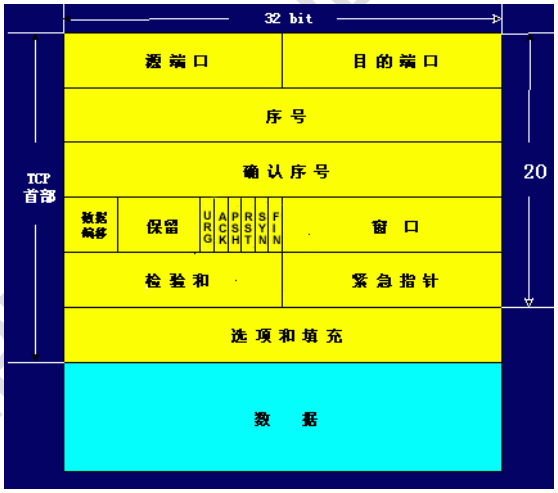
TCP 协议

TCP（Transmission Control Protocol 传输控制协议）是一种面向连接的、可靠的、基于字节流的传输层通信协议，由 IETF 的 RFC 793 定义。

在因特网协议族（Internet protocol suite）中，TCP 层是位于 IP 层之上，应用层之下的中间层。为不同主机的应用层之间提供可靠的、像管道一样的连接。

其基本原理如下:为了保证可靠传输，发送端给每个包一个序号，序号保证了接收端可以按序接收。然后接收端对已成功收到的包发回一个相应的确认（ACK）；如果发送端在合理的往返时延（RTT）内未收到确认，那么对应的数据包就被假设为已丢失，将会被进行重传。TCP 用一个校验和函数来检验数据是否有错误；在发送和接收时都要计算校验和。

其帧结构如下图所示



为了提供一个可靠的连接通道，TCP 实现了多种控制机制，是一个非常复杂的协议，略述如下：

- 为实现连接可靠的建立与终止，采用了三次握手建立连接，四次握手终止连接
- 为实现可靠传输，采用了确认机制、超时重传，乱序处理、
- 为实现高效传输，采用了滑动窗口机制
- 为实现拥塞控制，采用了慢启动、拥塞避免、快速恢复等算法
- 为减少小分组报文的产生，采用了数据捎带 ACK、Nagle 等方法
- 为增强可靠性，提供了窗口探查机制和保活机制

传统上，都是用软件的方式来处理 TCP 协议，这在网络负载较轻的时候运转良好。但近年来，随着网络负载越来越高，用软件来处理 TCP 给 CPU 造成了很大的负担。这时候，产生了一种基于硬件处理 TCP 协议的技术，即 TOE（TCP OFFLOAD ENGINE）

TOE

TOE 的主要功能就是将原来由软件处理的 TCP 和 UDP 协议从软件协议栈剥离，用硬件来实现，比如 FPGA 和 ASIC 等。

在 TOE 实现过程中，会遇到一系列的设计挑战

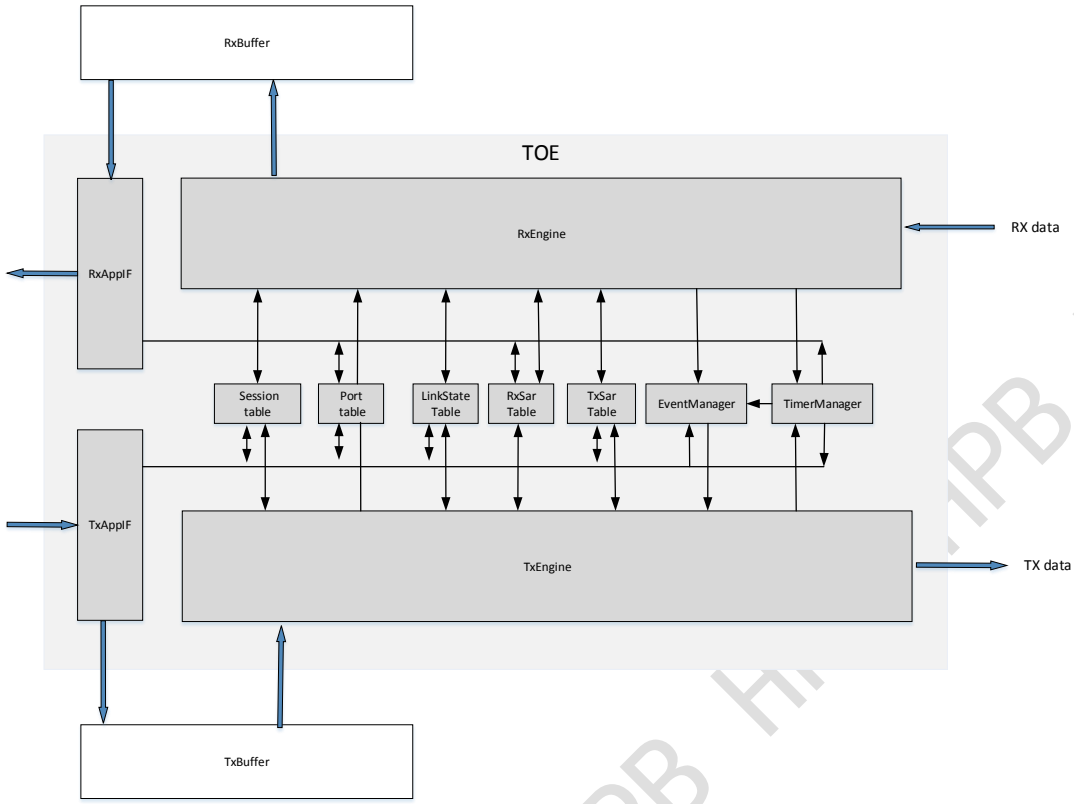
- 复杂的状态控制：连接控制、拥塞控制、报文数量控制
- 复杂的数据处理：分段、聚合、确认机制、重传机制
- 10K+连接状态维护
- 多单元之间的异步通信

1.2. 功能列表

- 实现 10K 条 TCP 连接的管理与实时处理
- 实现 RFC 793 标准链路状态机
- 实现滑动窗口数据传输机制
- 实现超时重传机制
- 实现乱序报文的处理
- 实现 ACK 延迟发送机制
- 实现 RFC 896 中建议的 Nagle 算法
- 实现慢启动及拥塞控制算法
- 实现保活机制

2. 整体设计

2.1. 整体框图



TcpTxUnit、TcpRxUnit 分别实现了主要的报文发送功能和报文接收功能，TxAppIF、RxAppIF 负责和应用程序接口。在收发单元之间是共用的数据表项，列举如下：

- 会话表：实现 3 元组(源/目的端口+目的/源 IP 地址+目的/源端口号)到内部 SessionID 的转换，内部操作以 SessionID 为索引进行操作
- 端口表：存储每个端口的配置和状态信息
- 链路状态表：存储每个连接的状态机（由 RFC 793 定义）
- 接收信息表：存储每个连接的接收控制信息，如接收序号、应用层读取指针
- 发送信息表：存储每个连接的发送控制信息，包括接收到的反向控制信息、本地发送控制信息
- 事件管理器：负责维护给发送侧的事件
- 定时管理器：负责维护一系列的计数器

收发单元通过上述的表项进行读取和处理，来实现 TCP 定义的各种功能。

结构设计上有如下特点：

- 接收侧为接收数据驱动
- 发送侧为事件驱动，驱动源来自接收侧、应用接口及计数器中断
- 设计多个表项，每个表项可以有多个访问源
- 表项的多个访问源通过各自的 FIFO 实现排队等待
- 引入表项锁定机制，解决多个源访问同一表项时的完整性问题

2.2. 接口说明

2.2.1. 应用层接口

接口列表如下

	Control/Info		Data	
Function	**TOE -> APP**	**APP -> TOE**	**TOE -> APP**	**APP -> TOE**
Listen	Listen require	Listen response		
Open	Open require	Open response		
Close	Close require			
Receive data	Notification	Rx data require	Rx data response Rx data	
Transmit data	Tx data response	Tx data require		Tx data

端口监听接口

- 监听请求
  端口号：16bit

- 监听响应
  是否成功指示：1bit

打开会话接口

- 打开请求

```
struct ipTuple
{
    ap_uint<32> ip_address;         //目的IP
    ap_uint<16> ip_port;            //目的端口
};
```

- 打开响应

```
struct openStatus
{
    ap_uint<16> sessionID;          //会话ID
```

```
    bool        success;            //是否成功
};
```

会话关闭接口

- 关闭请求
  会话 ID：16bit

数据接收接口

- 数据接收通知

```
struct appNotification
{
  ap_uint<16>        sessionID;      //会话ID
  ap_uint<16>        length;         //数据长度
  ap_uint<32>        ipAddress;      //源IP地址
  ap_uint<16>        dstPort;        //目的端口
  bool               closed;         //会话关闭指示
  };
```

- 接收数据请求

```
struct appReadRequest
{
    ap_uint<16> sessionID;          //会话ID
    ap_uint<16> length;             //接收数据长度
};
```

- 接收数据响应
  会话 ID：16bit

- 接收数据
  64 位 axis 接口

数据发送接口

- 发送数据请求

```
struct appTxMeta
{
```

```
    ap_uint<16> sessionID;    //会话ID
    ap_uint<16> length;       //发送数据长度
  };
```

- 发送数据响应

  响应值：17bit

  ```
  ERROR_NOSPACE  = -1;
  ERROR_NOCONNCECTION  = -2;
  ```

  此值>0，表示可以发送的数据长度（？），目前为请求值返回

- 发送数据

  64 位 axis 接口

2.2.2. IP 层接口

接收数据接口

64 位数据的 axis 接口

发送数据接口

64 位数据的 axis 接口

2.2.3. 缓存接口

接收侧接口

和 Xilinx 的 DataMover 相连接，各接口需符合此 IP 接口格式

- 写命令

```
struct mmCmd
{
  ap_uint<23> bbt;
  ap_uint<1>  type;
  ap_uint<6>  dsa;
  ap_uint<1>  eof;
  ap_uint<1>  drr;
  ap_uint<32> saddr;
  ap_uint<4>  tag;
  ap_uint<4>  rsvd;
```

```
};
```

- 写数据：64 位数据的 axis 接口
- 写状态

```
struct mmStatus
{
   ap_uint<4>  tag;
   ap_uint<1>  interr;
   ap_uint<1>  decerr;
   ap_uint<1>  slverr;
   ap_uint<1>  okay;
};
```

- 读命令：同写命令
- 读数据：64 位数据的 axis 接口

发送侧接口

接口格式和接收侧相同

- 写命令
- 写数据
- 写状态
- 读命令
- 读数据

2.2.4. 查找表接口

- 查找请求接口

```
struct rtlSessionLookupRequest
{
    lookupSource       source;    //查找请求源
    fourTupleInternal  key;       //四元组
};

enum lookupSource {RX, TX_APP};
```

- 查找响应接口

```
struct rtlSessionLookupReply
{
    lookupSource       source;      //查找请求源
```

```
    ap_uint<14>        sessionID;   //会话ID
    bool               hit;         //是否命中
};
```

- 更新请求接口

```
struct rtlSessionUpdateRequest
{
  lookupSource       source;      //查找请求源
  lookupOp           op;          //操作码
  ap_uint<14>        value;       //会话ID
  fourTupleInternal   key;         //四元组
};

enum lookupOp {INSERT, DELETE};
```

- 更新响应接口

```
struct rtlSessionUpdateReply
{
    lookupSource       source;      //查找请求源
    lookupOp           op;          //操作码
    ap_uint<14>        sessionID;   //会话 ID
    } ;
```

2.3. 表项说明

会话表

- 功能：实现 4 元组(源 IP 地址+源端口+目的 IP 地址+目的端口号)到内部 SessionID 的转换，内部操作以 SessionID 为索引进行操作
- 索引：四元组
- 内容：SessionID，Valid

反向会话表

- 功能：实现内部 SessionID 到 4 元组(源 IP 地址+源端口+目的 IP 地址+目的端口号)的转换，用于发送单元
- 索引：SessionID ，范围 0-(MAX_SESSIONS-1)
- 内容：四元组，Valid

SessionIdFreeList

- 功能：sessionId 的资源池，用于分配 SessionId

监听端口表

- 功能：配置此端口是否允许监听，只监听小于 32768 的端口
- 索引：监听端口号，范围 0-32767
- 内容：1bit 表示是否监听

空闲端口表

- 功能：存储此端口是否空闲，只使用大于等于 32768 的端口
- 索引：自由端口号低 15 位，范围 0-32767
- 内容：1bit 表示是否空闲

链接状态表

- 功能：存储每个会话的链接状态机（由 RFC 793 定义）
- 索引：SessionID，范围 0-(MAX_SESSIONS-1)
- 内容：此会话的状态

```
enum sessionState {CLOSED, SYN_SENT, SYN_RECEIVED, ESTABLISHED, FIN_WAIT_1, FIN_WAIT_2,
CLOSING, TIME_WAIT, LAST_ACK};
```

接收控制表

- 功能：用于控制每个会话的数据接收
- 索引：SessionID，范围 0-(MAX_SESSIONS-1)
- 内容：

```
struct rxSarEntry
{
  ap_uint<32> recvd;    //接收序号，低 16 位为接收数据的写指针
  ap_uint<16> appd;     //应用层读取指针
};
```

发送控制表

- 功能：用于控制每个会话的数据发送
- 索引：SessionID，范围 0-(MAX_SESSIONS-1)
- 内容：包括接收到的反向控制信息、本地发送控制信息

```
struct txSarEntry
{
  ap_uint<32> ackd;                    //已确认过的序号，低 16 位为数据读指针上界
  ap_uint<32> not_ackd;                //未确认过的序号，低 16 位为数据读指针下界
  ap_uint<16> recv_window;             //对端的接收窗口
  ap_uint<16> cong_window;             //拥塞窗口
  ap_uint<16> slowstart_threshold;     //慢启动阈值
  ap_uint<16> app;                     //应用层数据写指针
  ap_uint<2> count;                    //ACK 重复计数
  bool      finReady;
  bool      finSent;
};
```

重发送定时器

- 功能：用于控制重发送功能
- 索引：SessionID，范围 0-(MAX_SESSIONS-1)
- 内容

```
struct retransmitTimerEntry
{
  ap_uint<32>       time;          //计时器
  ap_uint<3>        retries;       //重发次数
  bool          active;            //此计数器是否有效
  eventType     type;              //类型
};

enum eventType {TX, RT, ACK, SYN, SYN_ACK, FIN, RST, ACK_NODELAY};
```

探测定时器

- 功能：用于控制探测功能
- 索引：SessionID，范围 0-(MAX_SESSIONS-1)
- 内容

```
struct probe_timer_entry
{
  ap_uint<32>       time;          //计时器
  bool          active;            //是否有效
};
```

关闭定时器

- 功能：用于控制关闭测功能
- 索引：SessionID，范围 0-(MAX_SESSIONS-1)
- 内容

```
struct close_timer_entry
{
  ap_uint<32> time;
  bool      active;
};
```

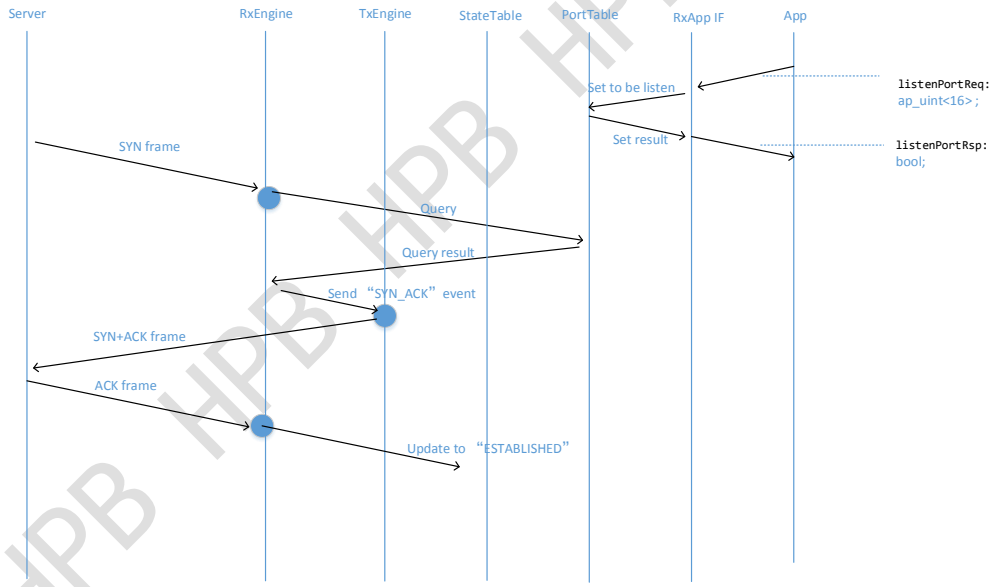
事件管理器：负责维护给发送侧的事件

2.4. 数据流

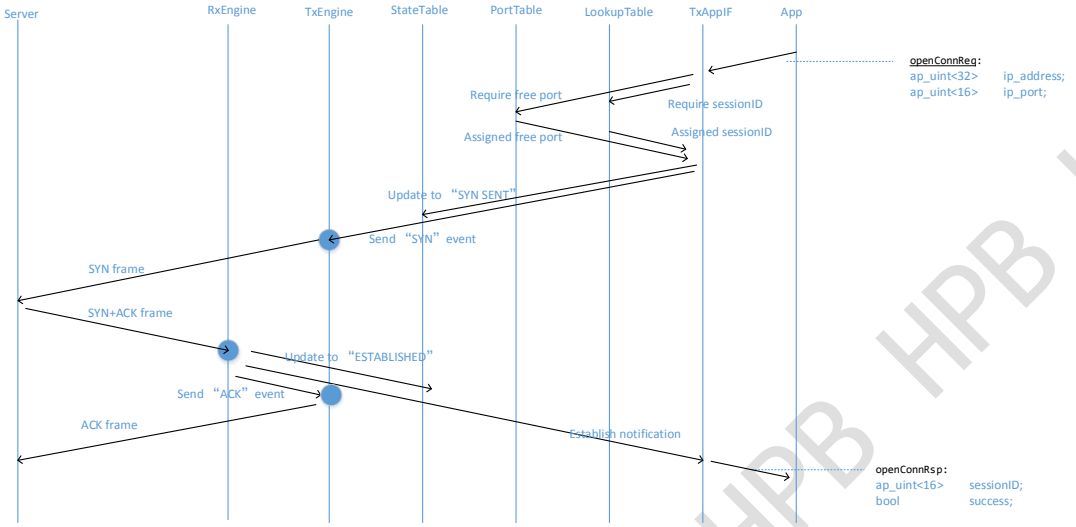
为了简化数据流的图示，在所有图中共有的部分未画出，即下图中圆圈部分：

- TxEngine 模块：收到 Event 之后，会读取反向会话表、rxSarTable、txSarTable；发送完毕之后会更新 txSarTable。
- RxEngine 模块：收到线路上的 TCP 帧后，会读取 portTable、会话表、stateTable、rxSarTable、txSarTable，处理完后更新 stateTable、rxSarTable、txSarTable。

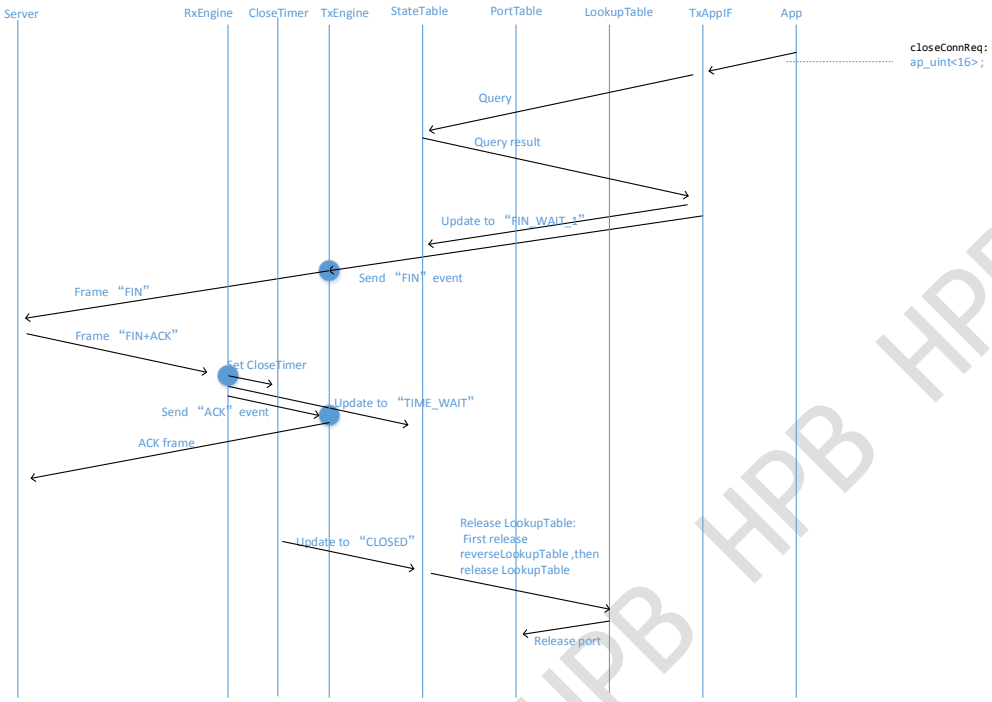
2.4.1. 监听端口



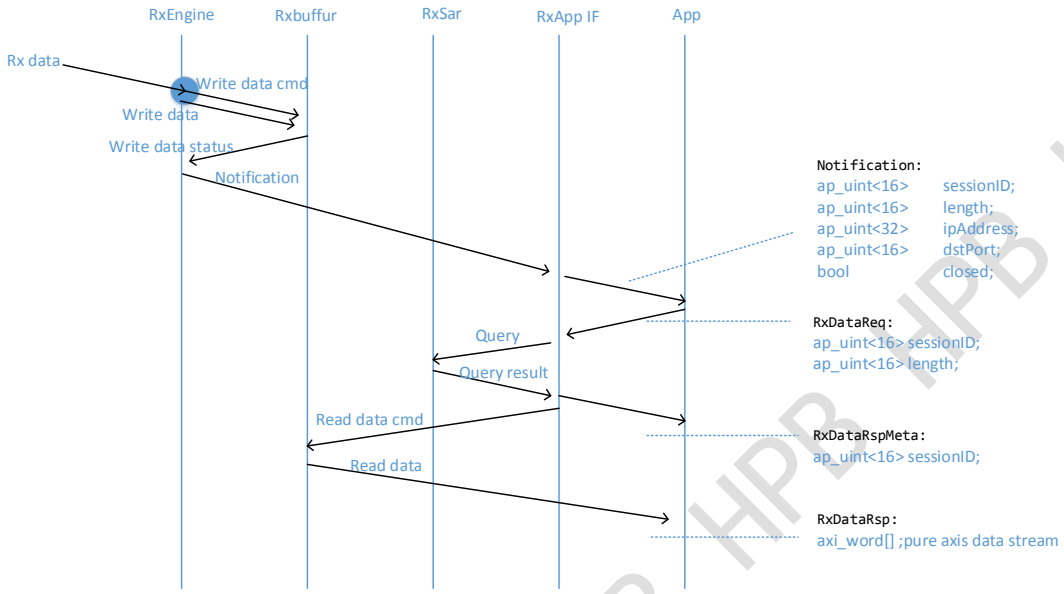
2.4.2. 打开端口



2.4.3. 关闭端口

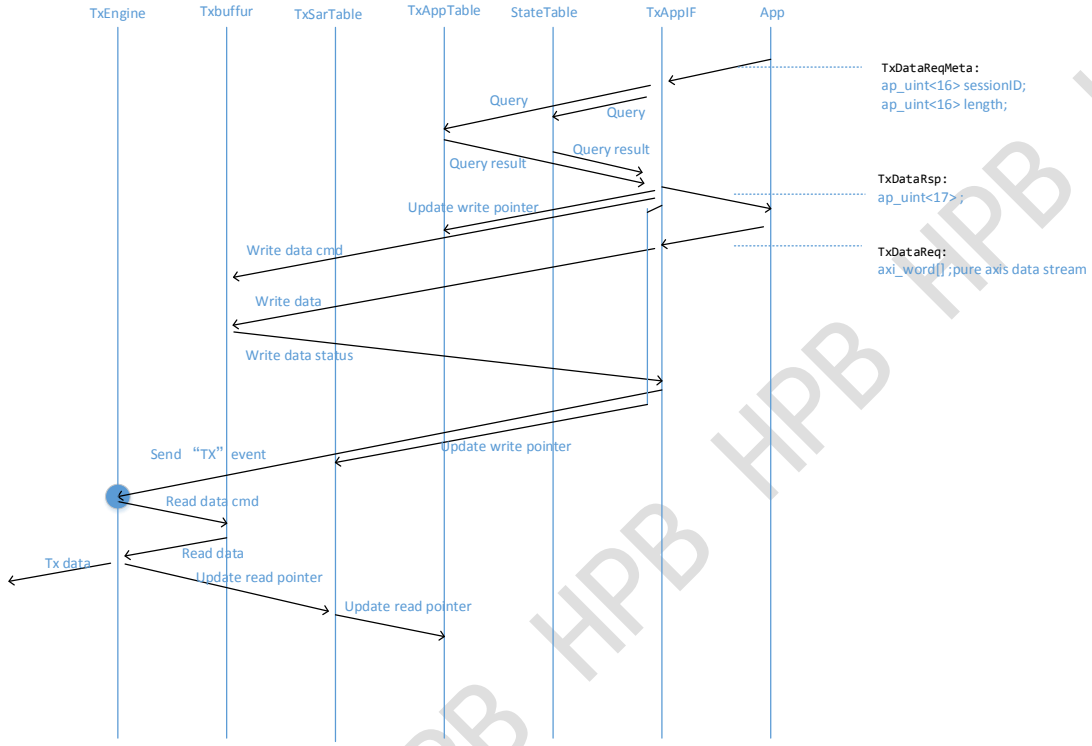


2.4.4. 接收数据



2.4.5. 发送数据

数据流如下图：



其中 txAppTable 为 txSarTable 部分内容的副本。内容如下：

```
struct txAppTableEntry
{
    ap_uint<16>     ackd;       //已确认序号，即读指针
    ap_uint<16>     mempt;      //写指针
};
```

2.4.6. 重发送

重发送机制主要涉及到 3 个模块：ReTransmitTimer、RxEngine、TxEngine。基本控制流是

- TxEngine：设置 ReTransmitTimer，进行重发操作
- RxEngine：清除 ReTransmitTimer；生成重发事件
- ReTransmitTimer：计数；生成重发事件

各模块中和重发送相关机制如下（文本框中为伪代码）：

RxEngine 中重发送机制

```
If (RxAck==1)
        if (RxAckSQ==TxNotAckSQ)
                ClearRetransmitTimer(sessionID,stop=1)
        else
                ClearRetransmitTimer(sessionID,stop=0)


If (only RxAck==1 )
         if (RxAckSQ==LastRxAckSQ && LastRxAckSQ !=TxNotAckSQ)
                  RepeatedRxAckSQCount++
         else
                  RepeatedRxAckSQCount=0

          if (RepeatedRxAckSQCount==3)
                   SendEventToTxEngine(sessionID,event=RT)
```

TxEngine 中重发送机制

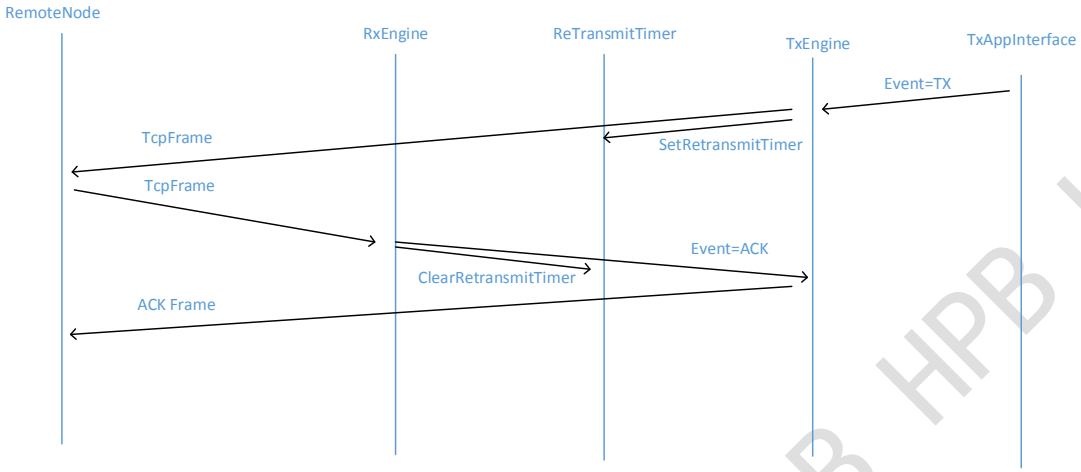
```
If ((Event=Tx || RT) && (send_length!=0))   ||   (Event=ACK ||ACK_NODELAY ||SYN || SYN_ACK ||FIN )
        SendTcpFrame
        SetRetransmitTimer(sessionID,eventType)
```

ReTransmitTimer 中重发送机制

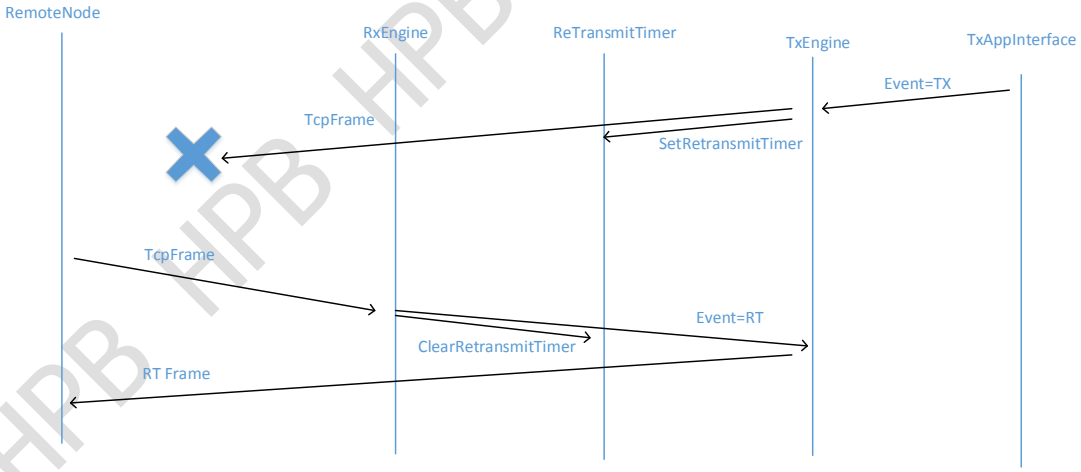
```
If (ReTransmitTimer.Active==1)
        If ( ReTransmitTimer.Timer==0)
                ReTransmitTimer.Active=0

                If (ReTransmitTimer.ReTries==4)
                        ReTransmitTimer.ReTries=0
                        NotifyStateTableToReleaseSession
                else
                        ReTransmitTimer.ReTries++
                        SendEventToTxEngine(sessionID,event=RT)
          else
                 ReTransmitTimer.Timer--



If(TxSetRetransmitTimer)
          if (ReTransmitTimer.Active==0)
                  ReTransmitTimer.Active=1
                  ReTransmitTimer.Timer =XXX    according to ReTransmitTimer.ReTries



If(RxClearRetransmitTimer)
          if (RxClearRetransmitTimer.Stop==1)
                  ReTransmitTimer.Active=0
                  ReTransmitTimer.Timer=0
                  ReTransmitTimer.ReTries=0
          else
                  ReTransmitTimer.Timer=Timer_1S
```

下面用流程图来表示 3 种典型的交互方式：

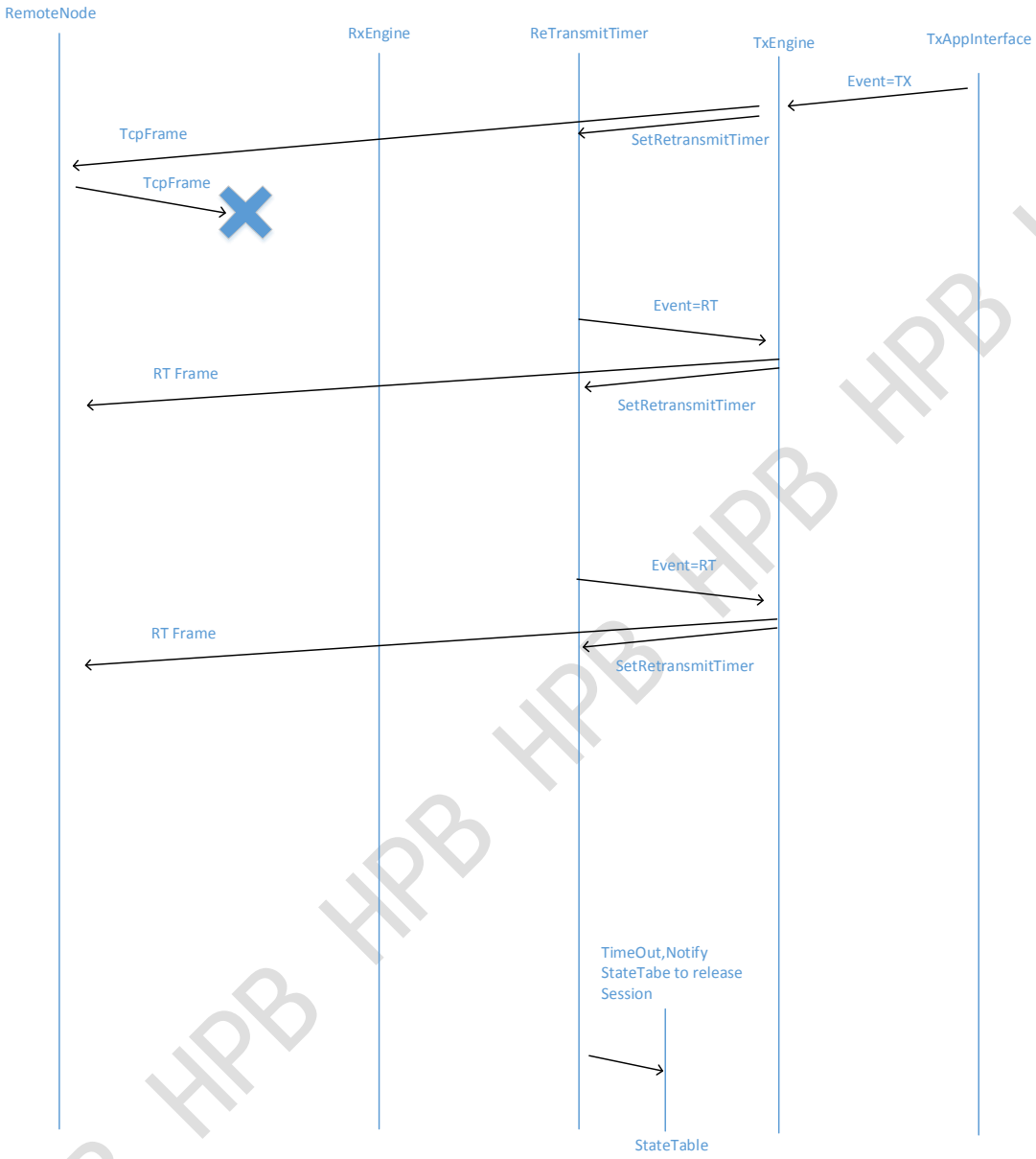
正常情况下



发送中断情况下



接收中断或收发均中断情况下



2.4.7. Ack 发送

Ack 机制主要涉及到 3 个模块： RxEngine、TxEngine、AckDelay。基本控制流是

- RxEngine：接收数据，生成 ACK、ACK_NODELAY 事件
- TxEngine：发送数据时捎带 ACK，否则进行 ACK、ACK_NODELAY 操作
- AckDelay：对 ACK 事件进行延迟处理

各模块中和ACK发送相关机制如下（文本框中为伪代码）：

RxEngine 中 ACK 机制

```
If (RxFLAG==SYN )
        Send event ”SYN_ACK” to TxEngine

Else If (RxFLAG==SYN_ACK)
        Send event ” ACK_NODELAY” to TxEngine

Else If (RxFLAG=FIN_ACK && (tcpState==FIN_WAIT_1 || tcpState==FIN_WAIT_2) )
        Send event ” ACK” to TxEngine

Else If (RxFLAG==ACK && RepeatedRxAckSQCount!=3 && RxPayloadLength!=0)
        Send event ” ACK” to TxEngine
```

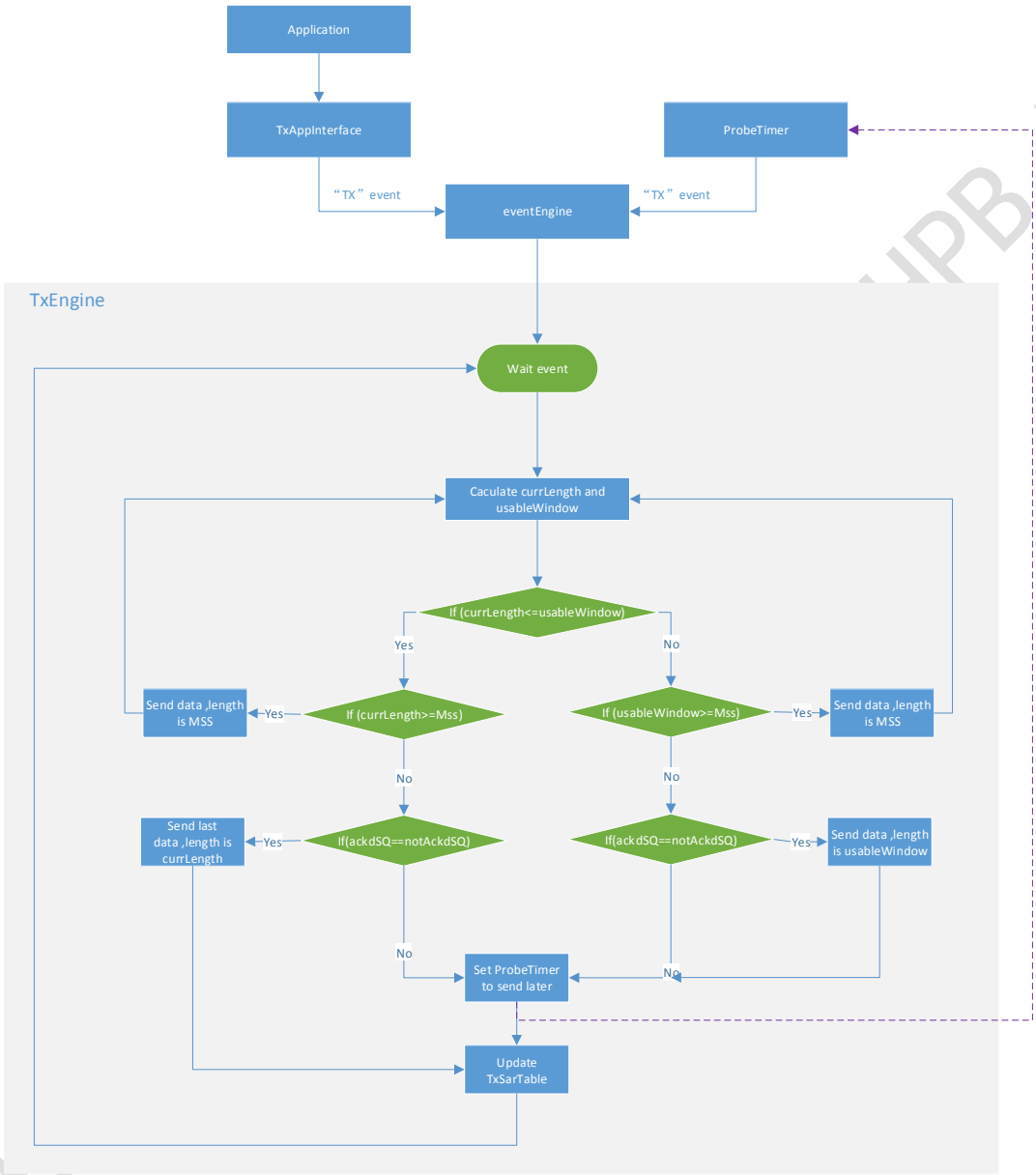
TxEngine 中 ACK 机制

```
If (Event==TX || RT )
        Send frame with data and ACK=1

Else If (Event==SYN )
        Send frame with SYN=1 and ACK=0

Else If (Event==SYN_ACK )
      Send frame with SYN=1 and ACK=1

Else If (Event==FIN )
      Send frame with FIN=1 and ACK=1

Else If (Event==ACK || ACK_NODELAY )
      Send frame with ACK=1
```

ACK_DELAY 中机制

```
If (ReceiveEventFromRxEngine)
        If(Event==ACK )
                Set AckDelayTimer to 64us
        else
                Transmit this event to TxEngine
                Close AckDelayTimer

Else if (AckDelayTimer timeout)
                Generate event “ACK” to TxEngine

Else
                AckDelayTimer--
```

2.4.8. Nagle 算法


Application
TxAppInterface
ProbeTimer
“TX” event
eventEngine
“TX” event
TxEngine
Wait event
Caculate currLength and usableWindow
If (currLength<=usableWindow)
Yes
No
Send data ,length is MSS
Yes
If (currLength>=Mss)
If (usableWindow>=Mss)
Yes
Send data ,length is MSS
No
No
Send last data ,length is currLength
Yes
If(ackdSQ==notAckdSQ)
If(ackdSQ==notAckdSQ)
Yes
Send data ,length is usableWindow
No
No
Set ProbeTimer to send later
Update TxSarTable

2.4.9. 慢启动及拥塞控制

此机制主要涉及到 3 个模块： RxEngine、TxEngine、txSarTable。基本控制流是

- RxEngine：接收数据，更新接收窗口和拥塞控制窗口
- TxEngine：初始化 txSarTable；重发时更新慢启动阈值
- txSarTable：根据 txEngine、rxEngine 指令更新各字段；进行最小窗口计算

各模块中和ACK发送相关机制如下（文本框中为伪代码）：

txSarTable 中拥塞控制机制

```
If (receive Update Command from txEngine)
                if (command == init)
                        //slow start
                         sarTable.cong_window = 0x3908      (10Mss)
                         sarTable.slowstart_threshold = 0xFFFF
                else if (command=isRTQuery)
                         //slow start when ReTransmit caused by timeout or repeated ack
                         sarTable.cong_window = 0x3908
                         sarTable.slowstart_threshold = new value from txEngine

If (receive Update Command from rxEngine)
                 sarTable.cong_window = new value from rxEngine
                sarTable.recv_window = new value from rxEngine

If (receive Query command from txEngine)
                return minWindow=min(cong_window,recv_window)
```

TxEngine 中机制

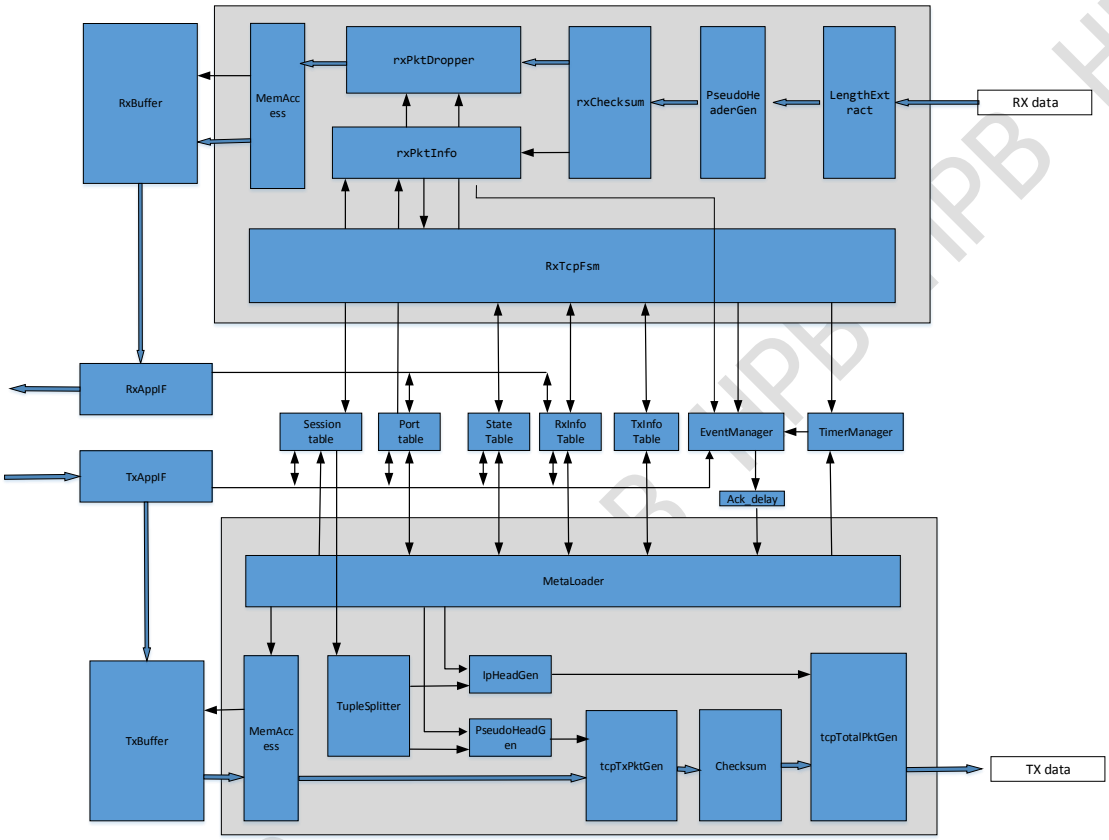
```
If (Event==RT )
        Re transmit user data
        If (is first ReTransmit)
                txSarTable.slowstart_threshold = max(currLength/2,2*MSS)
```

RxEngine 中机制

```
If (RxAckSQ != LastRxAckSQ    || LastRxAckSQ == TxNotAckSQ)
        If (txSar.cong_window <= txSar.slowstart_threshold)
                txSar.cong_window +=MSS      //this is slow start
        else
                txSar.cong_window += 1/cong_window          //this is Congestion Avoidance
```

3. 详细设计

3.1. 设计框图

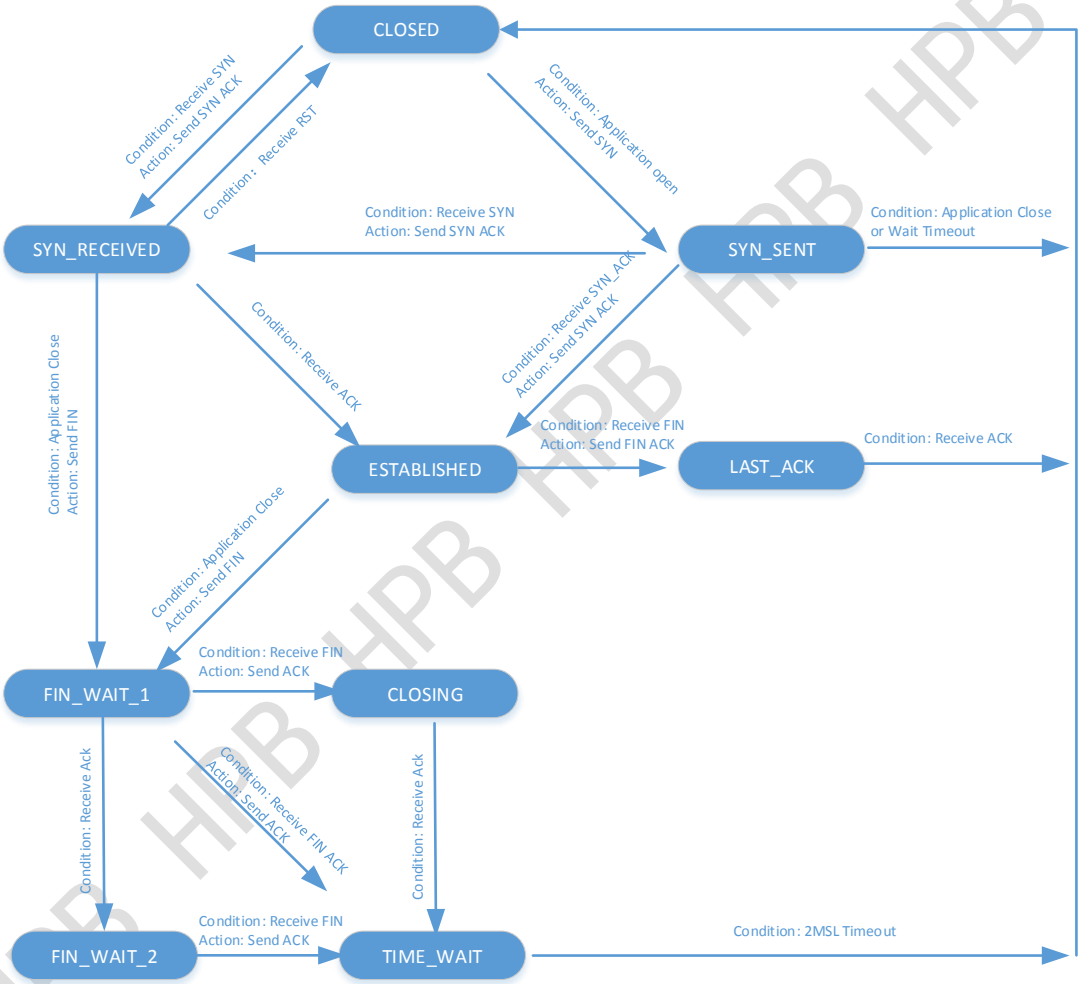


3.2. 核心状态机

下图是每个会话的状态机，和 RFC 793 标准中的状态机基本一致。为了简化设计，相比 RFC 793，做了两处状态合并：

- `LISTEN` 状态和并到 `CLOSED` 状态中
- `CLOSE_WAIT` 状态合并进 `LAST_ACK` 状态中

这两处都是和应用层接口侧的调整，并不影响和会话方的交互。



此状态机中涉及到三种驱动源，接收数据、定时器、应用层命令，因此在结构设计上并不是在一处完成。其中和应用层相关的状态变化由发送侧应用层接口模块 `tx_app_interface` 完成，包括：

- `CLOSED -> SYN_SENT`
- `SYN_SENT -> CLOSED`
- `SYN_RECEIVED -> FIN_WAIT_1`
- `ESTABLISHED -> FIN_WAIT_1`

CloseTimer（关闭定时器）和 StateTable 之间完成如下状态变化

- TIME_WAIT -> CLOSED

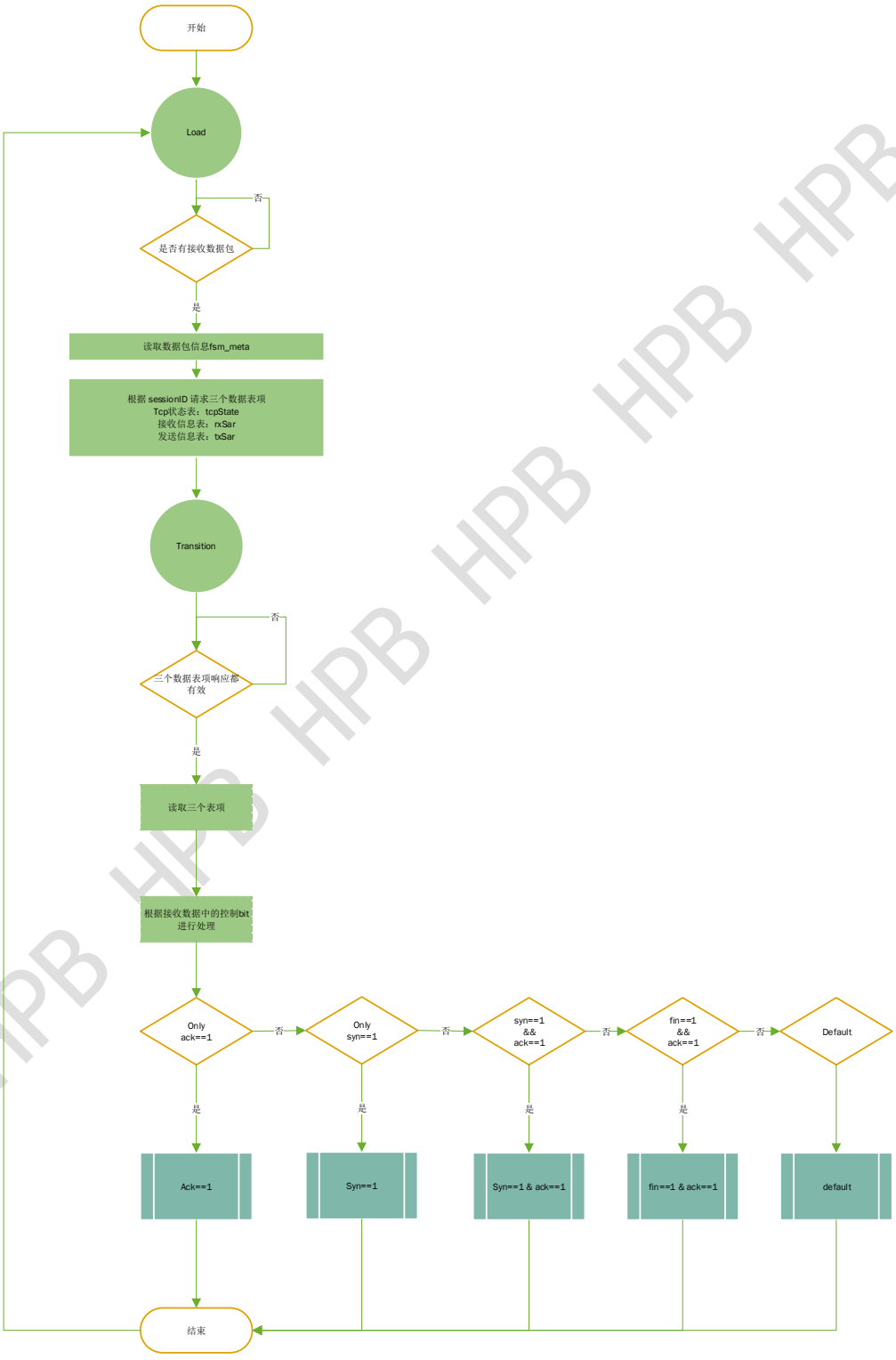
RtTimer（重发送定时器）和 StateTable 之间完成超时情况下的状态变化，除了 TIME_WAIT 和 CLOSED 状态之外，其它状态在 RtTimer 超时情况下都会变为 CLOSED，

- （任意状态） -> CLOSED

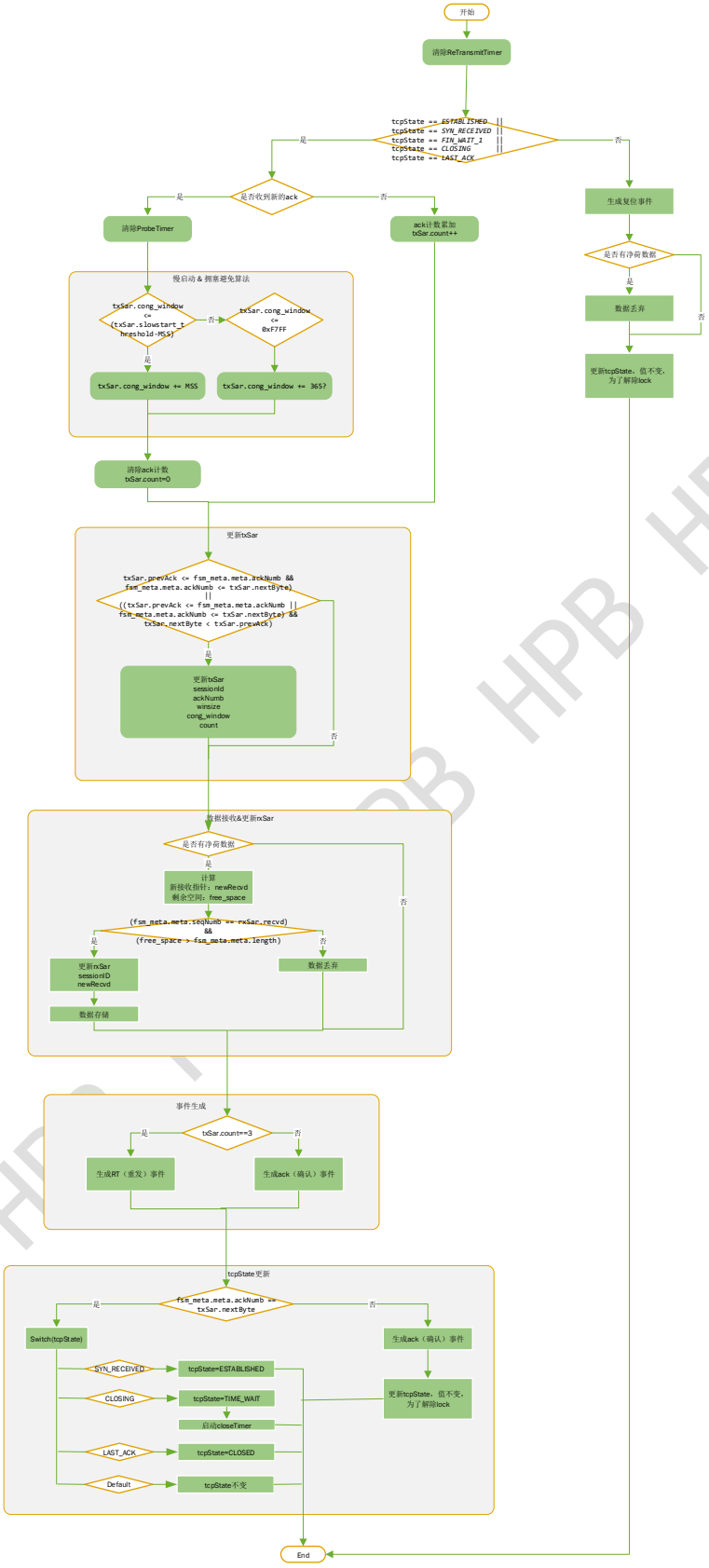
其它状态变化为数据驱动，在接收单元的 rxTcpFSM 中完成。

3.3. RxTcpFSM

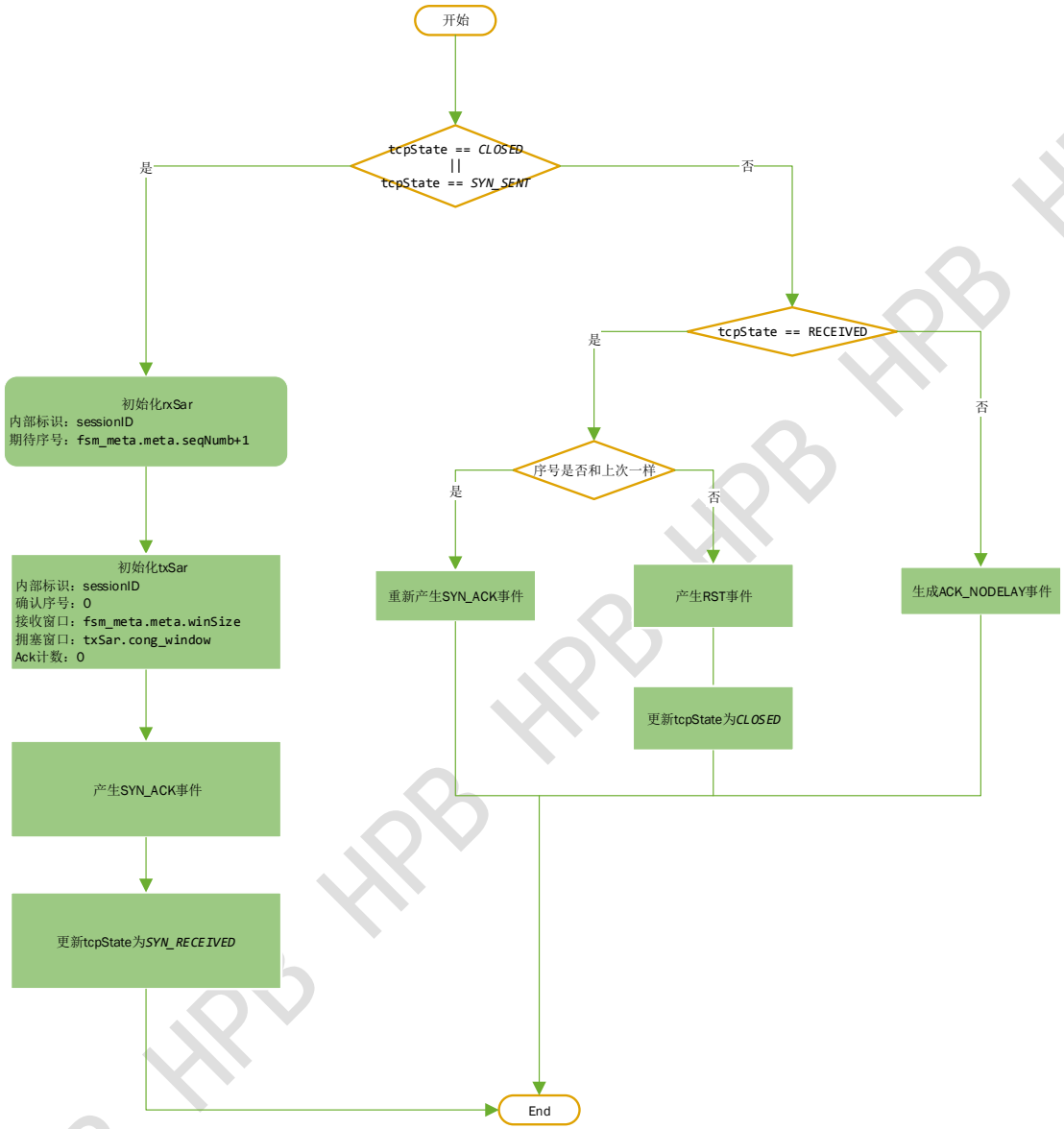
本模块包含了大部分核心状态机的内容，包括状态变化及相应处理。处理方式是根据收到数据包中的 `Flag` 字段进行分类处理。总体框图如下：



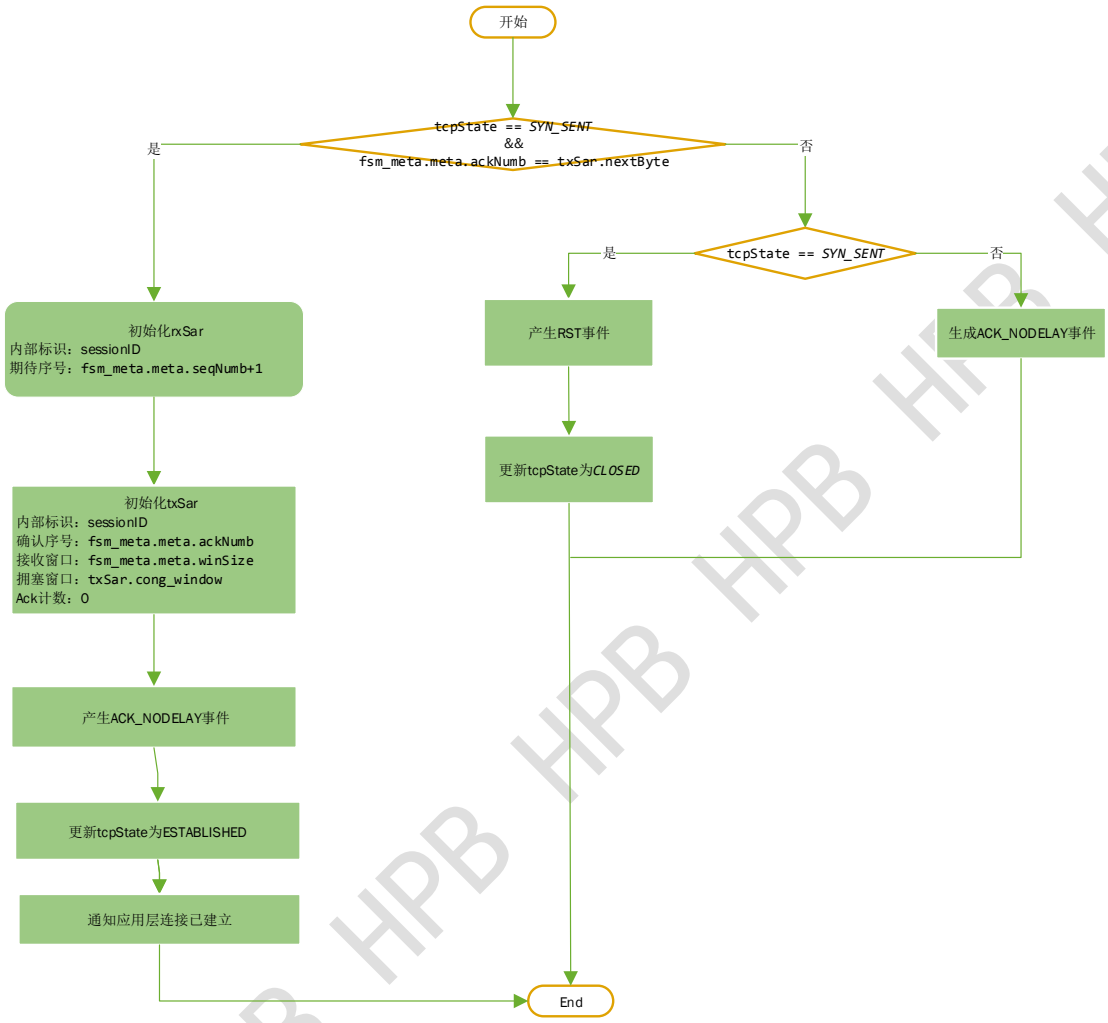
3.3.1. 接收帧 ack=1，其他=0



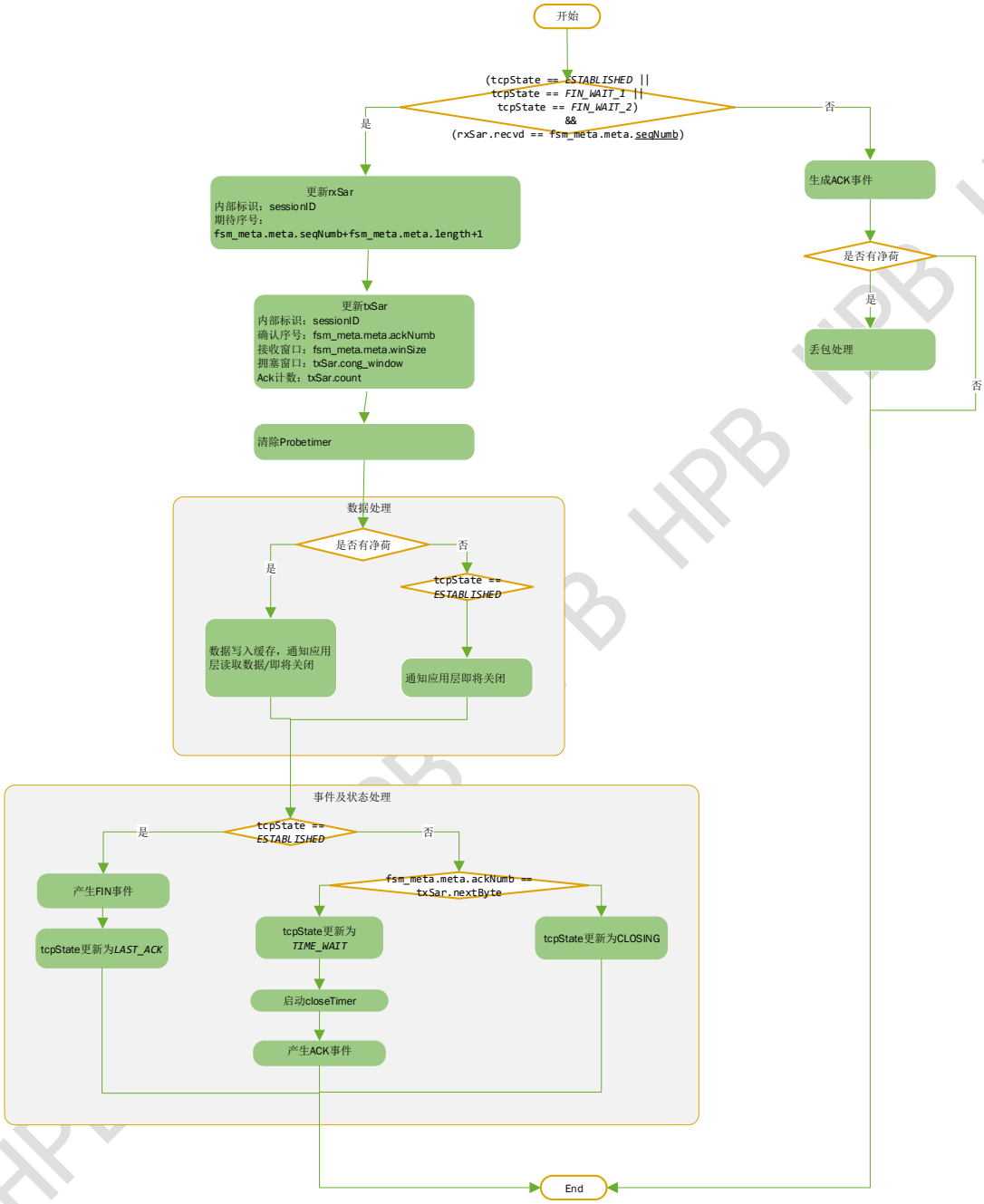
3.3.2. 接收帧 syn=1，其他=0



3.3.3. 接收帧 ack=1，syn=1，其他=0



3.3.4. 接收帧 ack=1，fin=1，其他=0



3.3.5. 接收帧其他 Flag 组合