THE ETHERSIA LIBRARY

How it manages the Ethernet/IP/TCP datagrams'buffer ?

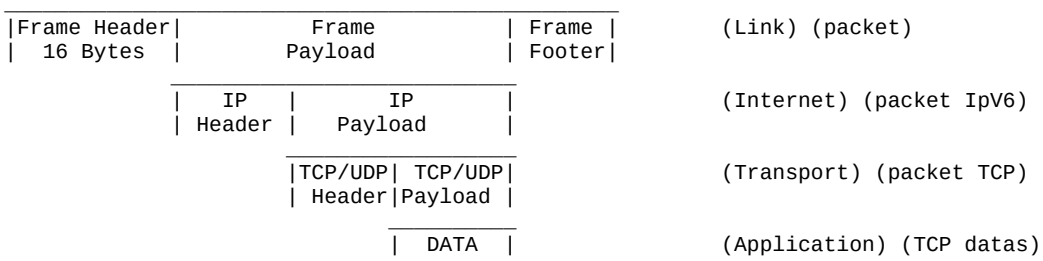
Device : Nano or Uno with an ENC28J60 ethernet controller

Example of the TCP Client mode

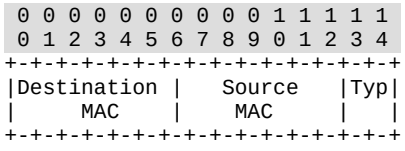
Frames and datagrams

A Frame generally consists of a header plus a payload of data

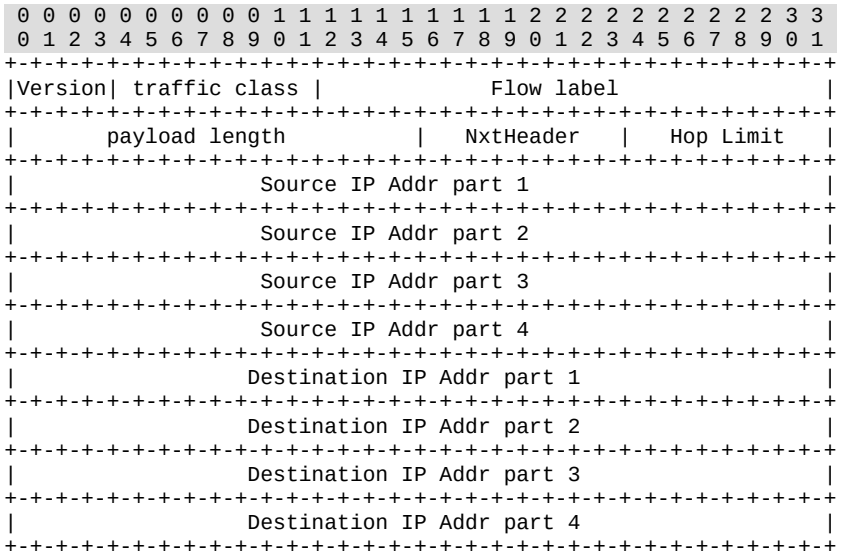
Structure of a Frame



Frame Header or Ethernet Header (14 bytes)

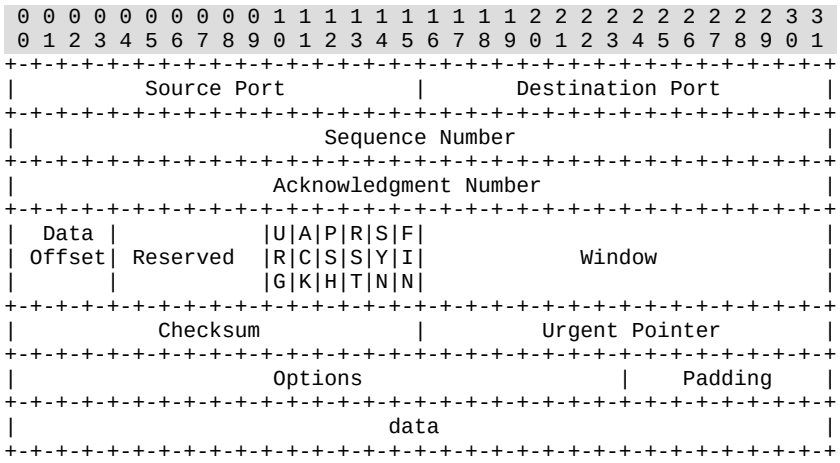


IPv6 Header (40 bytes)



Hop Limit replaces the TTL parameter of the previous Ipv4 Header

TCPIP Header (20 or 24 bytes)



The data Offset represents the number of 32 bits words in the TCP header, ie 5 or 6

IPv6Packet objects

An object from the IPv6Packet Class in EtherSia consists of the ethernet header (14 bytes) plus the IPv6 Header (40 bytes). Its sizes is 54 bytes...and nothing more : application data are in the EtherSia buffer

In the EtherSia vocabulary, a 'packet' usually refers to an Ipv6Packet

An Ipv6Packet object contains the following attributes :

			Size in byte
Ethernet Header	_etherDestination	Ethernet frame destination MAC Address	6
	_etherSource	Ethernet frame source MAC Address	6
	_etherType	Ethernet Type (0x86dd for IPv6)	2

IP Header	_ver_tc	Version + 4 bytes of traffic class	1
	_tc_fl	4 remaining bytes of traffic class plus 4 bytes of flow label	1
	_flowlabel	Remaining 16 bytes of flow label	2
	_length	Payload length, not including the IP Header	2
	_protocol	Type header immediately following (0x06 for TCP)	1
	_hopLimit	Hop Limit	1
	_source	Ipv6 packet source address	16
	_destination	Ipv6 packet destination address	16
		TOTAL :	54

packet.init() usually fills _etherType, _ver_tc, _tc_fl, _flowlabel and _hopLimit

Several methods permit to access to the full IPv6 payload

- packet.payload() returns a pointer to the start of the IPv6 payload of data
- packet.length() returns total length of the packet including ethernet header, IPv6 Header, TCP Header and application data
- packet.payloadLength() returns the length of the IPv6 payload, ie TCP Header plus application data.
- packet.setPayloadLength(uint16_t) permits to set the Ipv6 payload length

Ethersia's buffer

```
EtherSia_ENC28J60 ether;
```

An EtherSia_ENC28J60 object manages the ethernet interface.

It contains a buffer and a bunch of attributes specific to the device :

_linkLocalAddress	3 Ipv6Address of 16 bytes
_globalAddress	
_dnsServerAddress	

_localMAC	2 MACAddress of 6 bytes
_routerMAC	

_bufferContainsReceived	2 flag booleans
_autoConfigurationEnabled	

The buffer is an array of bytes. It is the same for receiving and sending operations.

The buffer is declared as an union :

```
union {
        uint8_t _buffer[ETHERSIA_MAX_PACKET_SIZE];
        void* _ptr;
    };
```

In the EtherSia class, a specific method packet() returns a reference to the buffer as a reference to an IPv6Packet object :

```
inline IPv6Packet& packet()
    {
         return (IPv6Packet&)_ptr;
    }
```

Sockets and client objects

```
TCPClient tcp(ether);
```

A TCPClient object is a socket attached to the EtherSia_ENC28J60 object

A socket inherits from the Print class and contains some specific "socket" attributes :

_ether	The address of the ethernet interface the socket is attached to

_remoteAddress	Identification parameters of the server we want to connect the device to : Ipv6Address, MAC and port
_remoteMAC	
_remotePort	

_localPort
_writePos

The TCPClient object, in addition to the parameters inherited from the Socket class, contains some "client" specific attributes :

_state
_remoteSeqNum
_localSeqNum

TCPClient objects (and TCPServer objects) can access to the TCP Header via a specific structure tcp_header which is defined in the tcp.h file. It contains all the parameters of a TCP header :

sourcePort
destinationPort
sequenceNum
acknowledgementNum
dataOffset
flags
window
checksum
urgentPointer
mssOptionKind
mssOptionLen
mssOptionValue

If you want to access or to update the headers'parameters within a TCPClient method, you have to write the following instructions :

```
IPv6Packet& packet = _ether.packet();
struct tcp_header *tcpHeader = (struct tcp_header*)(packet.payload());

 or

IPv6Packet& packet = _ether.packet();
struct tcp_header *tcpHeader = TCP_HEADER_PTR;
```

TCP_HEADER_PTR is a pointer to the start of the Ipv6 payload of data, of type tcp_header.
TCP_HEADER_PTR is supposed to be private but is still public right now...

The Ethernet or IP parameters can be accessed in the following way :
- packet.etherDestination()
- parket.etherSource()
- packet.destination()
- parket.source()

etc etc

The TCP parameters can be accessed in the following way :
- tcpHeader->acknowledgementNum
- tcpHeader->sequenceNum

etc etc

Filling the buffer with application data

Application data is filled in the buffer via the TCPClient socket, via the print function

A pointer _writePos supervises the filling
_writePos is initialized with a -1 value, which means no application data have been written, and that we are just at the end of the TCP transmit Header (24 bytes). When data is written, _writePos is incremented
The pointer gets back its -1 value when TCPClient send() method is called

The TCPClient transmitPayload() method returns a pointer to the start of the next TCP payload of application data to be transmitted

Once the buffer is written, it has to be sent before any cal to the receivePacket() method of the EtherSia class

Headers preparation and data sending

The EtherSia prepareSend() method :
- sets _bufferContainsReceived to false
- launches packet.init()
- fixes source MAC and IP via packet.setSource() and packet.setEtherSource()

Please note that destination MAC and IP are to be fixed later via the TCPClient send() method

The EtherSia prepareReply() method is a similar preparation method devoted to response. It swaps Ethernet source and destination MAC and IP. Additionnaly, as it does not call packet.init(), it has to fix a fresh hop limit via packet.setHopLimit()

The TCPClient send() method :

- calls the EtherSia method prepareReply()
- or calls the EtherSia method prepareSend() and fixes destination MAC and IP via the packet.setDestination() and packet.setEtherDestination()

The TCPClient send() method then fixes, via its sendInternal() method, all TCP headers parameters except flags which are fixed by the connect(), disconnect() and sendAck() methods from the TCPClient class

Still via the SendInternal() method, The TCPClient send() method also has to fix the nxtHeader and IPv6 payload length (including its TCP header and data application) via the packet.setProtocol() and packet.setPayloadLength()

The process reaches its end stage when the TCPClient sendInternal() method invokes the EtherSia send()/sendFrame() methods

Reading data

The receivePacket() method from the EtherSia class feeds the buffer of the EtherSia_ENC28J60 object with received network datagrams

The havePacket() method from the TCPClient class will check if there is a valid packet for the TCPClient socket attached to the EtherSia_ENC28J60

The TCPClient payload() method returns a pointer to the start of the last received TCP payload of application data

The TCPClient payloadLength() returns the length of the last received TCP payload of application data – should not be used of transmitted payload of data

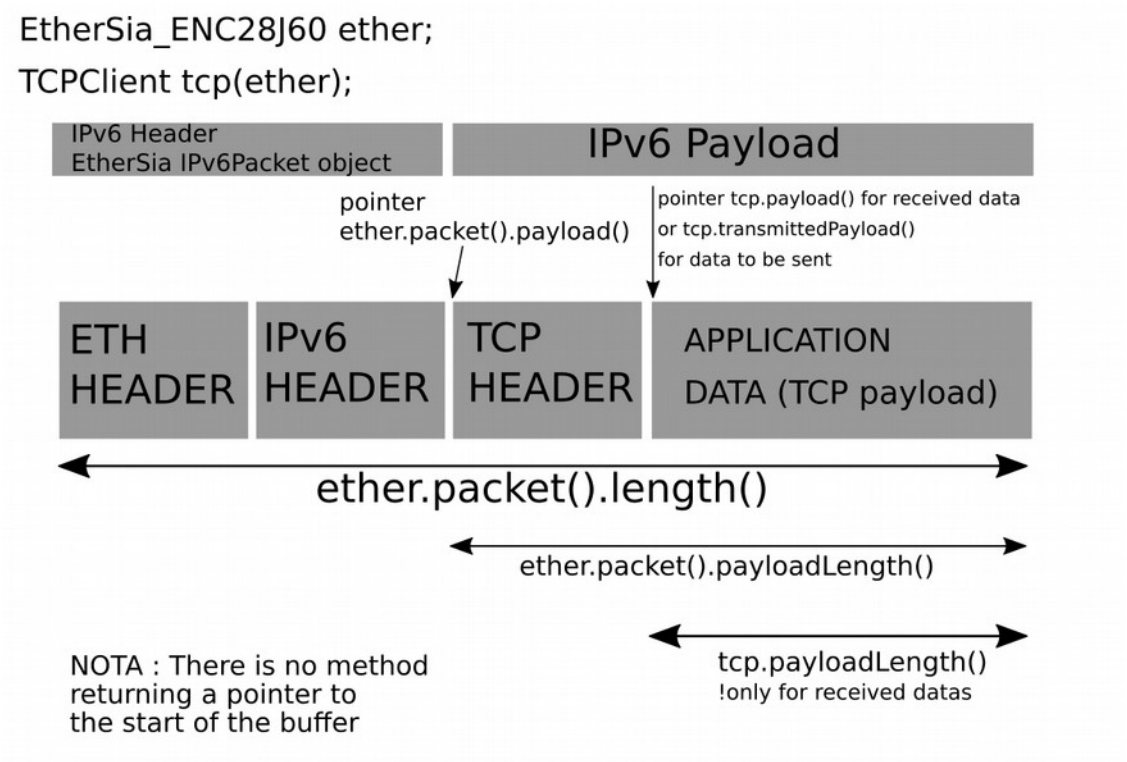


Illustration 1 : synthesis : data management in EtherSia

havePAcket() and TCP finite state machines

havePacket() is doing most of the communication job, implementing the TCP finite state machine, in the way it has been done in the 2000s by Adam DUNKEL in UIP

The havePacket from the TCPClient class doesn't implement a listen mode as it is devoted to a pure client for sending/receiving datas

As UIP, havePacket() from the TCPCLient class uses the goto statement.

The general process follows the following diagram, extracted from
http://tcpipguide.com/free/t_TCPOperationalOverviewandtheTCPFiniteStateMachineF-2.htm

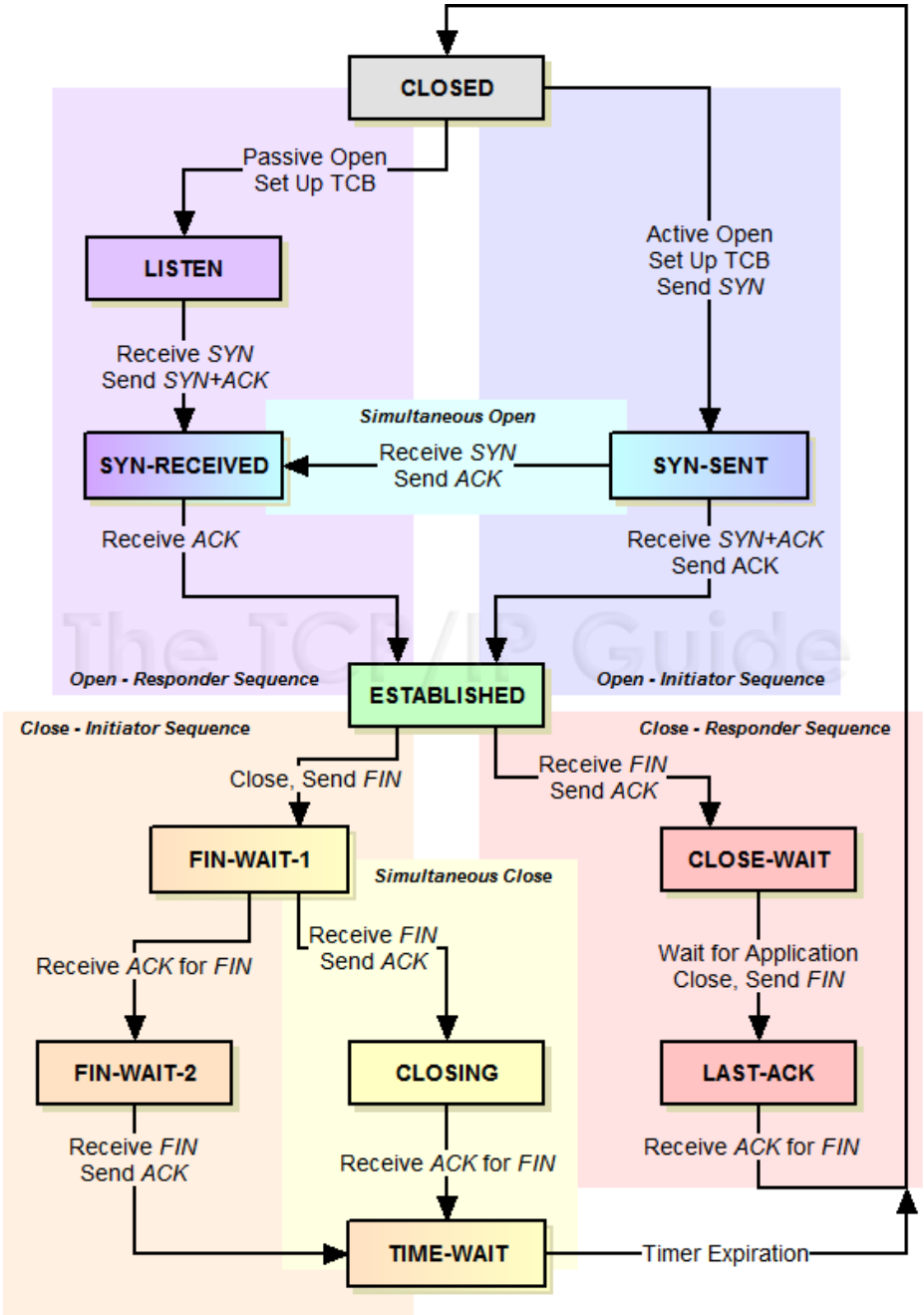


Illustration 2: Finite state TCP machine - extrated from the TCPguide in http://tcpipguide.com/

Creating examples

you will need to create two main objects :

- an EtherSia_ENC28J60 object, for the interface management
- a TCPClient or HTTPServer object, for the socket management