[UEFI Basics] EDK Network Framework (TCP4)

Category Column: UEFI Development Basics Article Tags: network uefi

摘要 This article introduces the TCP4 protocol in UEFI in detail, and explains its connection-oriented and high reliability characteristics compared with UDP4, and explains the header format, connection process, etc. It also summarizes the TCP4 code, including entry, port initialization, etc. It introduces related structures and protocols, such as SOCKET, EFI_TCP4_PROTOCOL, etc., and gives TCP code examples and running results.

The summary is generated in C Know , supported by DeepSeek-R1 full version, go to experience>

Expand

TCP4

TCP4 Protocol Description

Compared with UDP4 , TCP4 is a connection-oriented communication protocol and therefore has better reliability.

The TCP4 header format is as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```


The parameters are described as follows:

Fields	Length (bits)	meaning
Source Port	16	Source port, which identifies which application sent the message.
Destination Port	16	Destination port, which identifies which application receives the message.
Sequence Number	32	Sequence number field. Each byte in the data stream transmitted in the TCP link is assigned a sequence number. The value of the sequence number field refers to the sequence number of the first byte of the data sent in this segment.
Acknowledgment Number	32	Acknowledgement number. It is the sequence number of the first byte of the next message segment that is expected to be received from the other party. That is, the sequence number of the last successfully received data byte plus 1. This field is valid only when the ACK flag is 1.
Data Offset	4	Data offset, i.e. header length. Indicates how far the data start of a TCP segment is from the start of the TCP segment. It is calculated in units of 32 bits (4 bytes). The maximum header is 60 bytes. If there is no option field, it is normally 20 bytes.
Reserved	6	Reserved, must be filled with 0.
URG	1	The urgent pointer is valid. It tells the system that there is urgent data in this message segment and it should be transmitted as soon as possible (equivalent to high-priority data).
ACK	1	Confirmation number valid identifier. The confirmation number field is valid only when ACK=1. When ACK=0, the confirmation number is invalid.
PSH	1	The receiver should hand over the segment to the application layer as soon as possible. When a TCP segment with PSH = 1 is received, it should be delivered to the receiving application process as soon as possible, without waiting for the entire buffer to be filled before delivering it upward.
RST	1	Reestablish the connection flag. When RST=1, it indicates that a serious error has occurred in the TCP connection (such as due to a host crash or other reasons), and the connection must be released and then reestablished.
SYN	1	Synchronization sequence number identifier, used to initiate a connection. SYN=1 means this is a connection request or connection acceptance request.
FIN	1	The sender completes the sending task. It is used to release a connection. FIN=1 indicates that the sender has completed sending the data of this message segment and requires the connection to be released.
Window	16	Window: TCP flow control. The window starts at the value specified by the Acknowledgement Number field, which is the number of bytes the receiver expects to receive. The maximum window size is 65535 bytes.
Checksum	16	The checksum field, including the TCP header and TCP data, is a mandatory field that must be calculated and stored by the sender and verified by the receiver. When calculating the checksum, a 12-byte pseudo header is added to the front of the TCP segment.
Urgent Pointer	16	Urgent pointer is valid only when the URG flag is set to 1. TCP's urgent mode is a way for the sender to send urgent data to the other end. The urgent pointer indicates how many bytes of urgent data there are in this segment (the urgent data is placed at the front of the data in this segment).
Options	variable	Option field. The TCP protocol originally specified only one option, namely the maximum segment length (containing only the data field, excluding the TCP header), also known as MSS. MSS tells the other TCP "the maximum length of the data field of the segment that my cache can receive is MSS bytes".

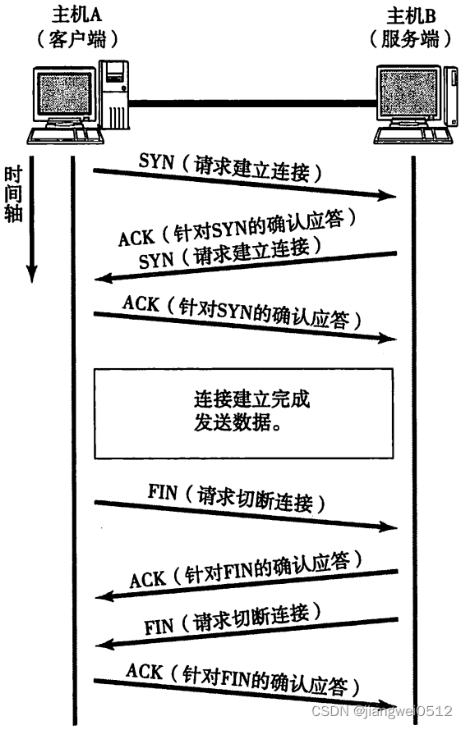
Fields	Length (bits)	meaning
		The new RFC specifies the following options: end of option table, no operation, maximum segment length, window expansion factor, timestamp. * End of option table. * No operation: no special meaning, generally used to fill the total length of the TCP option to an integer multiple of 4 bytes. * Maximum segment length: also known as MSS, contains only the data field, excluding the TCP header. * Window expansion factor: 3 bytes, one of which represents the offset value S. The new window value is equal to the number of window bits in the TCP header increased to (16+S), which is equivalent to shifting the window value to the left by S bits to obtain the actual window size. * Timestamp: 10 bytes, of which the most important fields are the timestamp value (4 bytes) and the timestamp echo reply field (4 bytes).
Padding	variable	The padding field is used to fill in the space so that the entire header length is an integer multiple of 4 bytes.
data	variable	TCP payload.

Corresponding code in UEFI :


```c
typedef UINT32  TCP_SEQNO;
typedef UINT16  TCP_PORTNO;

//
// TCP header definition
//
typedef struct {
  TCP_PORTNO    SrcPort;
  TCP_PORTNO    DstPort;
  TCP_SEQNO     Seq;
  TCP_SEQNO     Ack;
  UINT8         Res     : 4;
  UINT8         HeadLen : 4;
  UINT8         Flag;
  UINT16        Wnd;
  UINT16        Checksum;
  UINT16        Urg;
} TCP_HEAD;
```

The TCP connection process is roughly as follows:



TCP4 Code Overview

TCP4 is also a common network protocol . In fact, now it is NetworkPkg\TcpDxe\TcpDxe.inf. Here we first need to look at its entry:


```c
EFI_STATUS
EFIAPI
TcpDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  //
  // Install the TCP Driver Binding Protocol
  //
  Status = EfiLibInstallDriverBindingComponentName2 (
             ImageHandle,
             SystemTable,
             &gTcp4DriverBinding,
             ImageHandle,
             &gTcpComponentName,
             &gTcpComponentName2
             );

  //
  // Initialize ISS and random port.
  //
  Seed            = NetRandomInitSeed ();
  mTcpGlobalIss   = NET_RANDOM (Seed) % mTcpGlobalIss;
  mTcp4RandomPort = (UINT16)(TCP_PORT_KNOWN + (NET_RANDOM (Seed) % TCP_PORT_KNOWN));
}
```

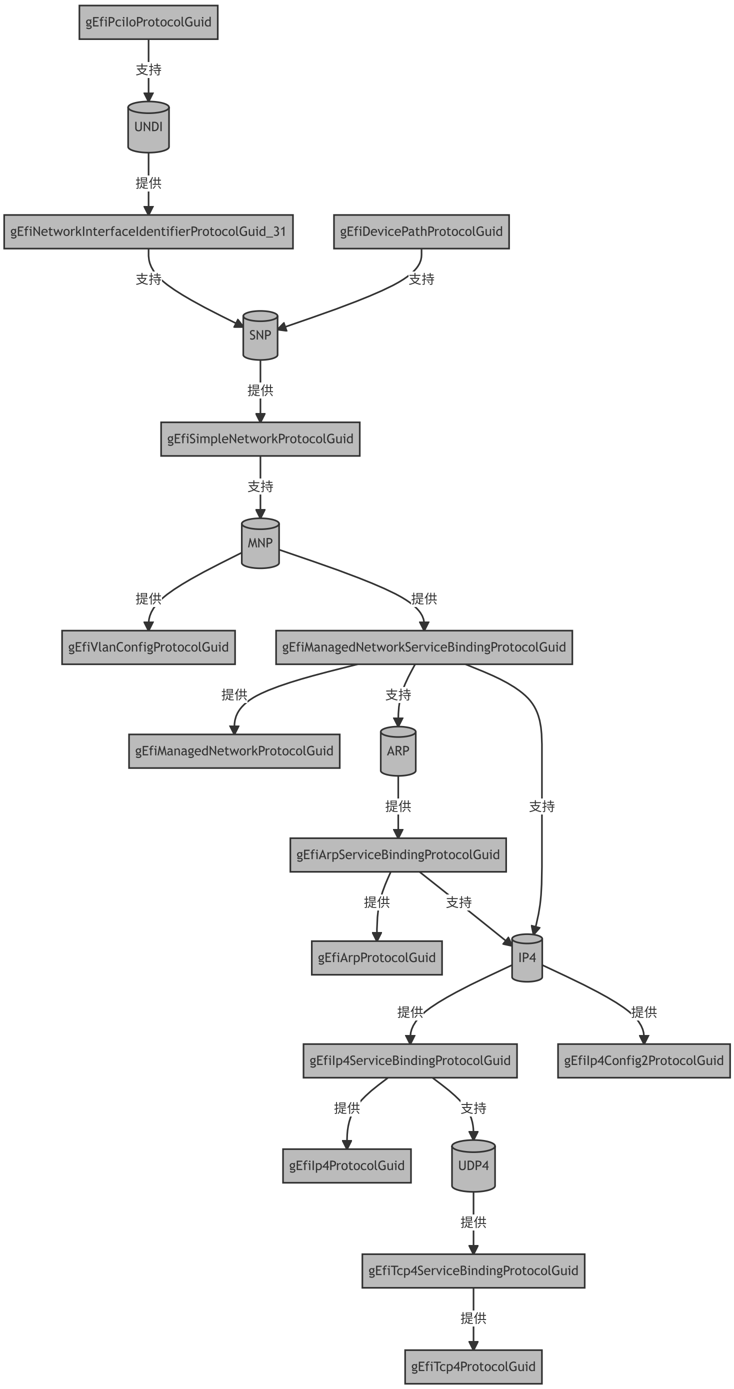
Because TCP4 is also a UEFI Driver Model, the first step is to install it `gTcp4DriverBinding` . The implementation is:

```c
EFI_DRIVER_BINDING_PROTOCOL  gTcp4DriverBinding = {
  Tcp4DriverBindingSupported,
  Tcp4DriverBindingStart,
  Tcp4DriverBindingStop,
  0xa,
  NULL,
  NULL
};
```

The second step is to initialize a random TCP port. According to the common network protocol, the TCP port occupies two bytes (i.e. 16 bits). It can be any port as long as it is not a recognized port in the range of 0-1023, and it does not matter if it is consistent with the UDP port.

Finally `mTcpGlobalIss` , ISS stands for **Initial S** ending Sequence. Its value itself is not very important. As the name suggests, it is used to specify the sequence number of the first packet sent by TCP. This is because TCP may send many packets at a time, so they need to be sorted.

Relationship diagram of UDP4 in UEFI network protocol stack:



Tcp4DriverBindingSupported

TCP4 depends on IP4 :


```c
EFI_STATUS
EFIAPI
Tcp4DriverBindingSupported (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath OPTIONAL
  )
{
  //
  // Test for the Ip4ServiceBinding Protocol
  //
  Status = gBS->OpenProtocol (
                  ControllerHandle,
```

```c
                   &gEfiIp4ServiceBindingProtocolGuid,
                   NULL,
                   This->DriverBindingHandle,
                   ControllerHandle,
                   EFI_OPEN_PROTOCOL_TEST_PROTOCOL
                   );
  return Status;
}
```

Tcp4DriverBindingStart

There is only one function in the Start function `TcpCreateService()` , and its function is to initialize `TCP_SERVICE_DATA` .

TCP_SERVICE_DATA

The structure itself is relatively simple:


```c
typedef struct _TCP_SERVICE_DATA {
  UINT32                        Signature;
  EFI_HANDLE                    ControllerHandle;
  EFI_HANDLE                    DriverBindingHandle;
  UINT8                         IpVersion;
  IP_IO                         *IpIo;
  EFI_SERVICE_BINDING_PROTOCOL  ServiceBinding;
  LIST_ENTRY                    SocketList;
} TCP_SERVICE_DATA;
```

The key point is actually one `SocketList` , and its corresponding list member is `SOCKET` .

SOCKET

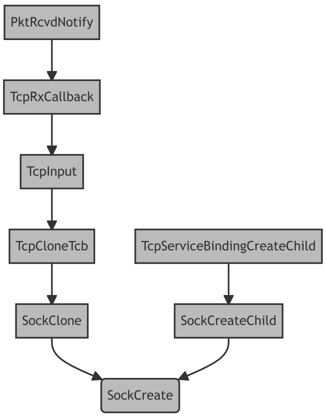
Socket is a sub-item of TCP. The structure is as follows:


```c
///
/// The socket structure representing a network service access point.
///
struct _TCP_SOCKET {
  //
  // Socket description information
  //
  UINT32                      Signature;     ///< Signature of the socket
  EFI_HANDLE                  SockHandle;    ///< The virtual handle of the socket
  EFI_HANDLE                  DriverBinding; ///< Socket's driver binding protocol
  EFI_DEVICE_PATH_PROTOCOL    *ParentDevicePath;
  EFI_DEVICE_PATH_PROTOCOL    *DevicePath;
  LIST_ENTRY                  Link;
  UINT8                       ConfigureState;
  SOCK_TYPE                   Type;
  UINT8                       State;
  UINT16                      Flag;
  EFI_LOCK                    Lock;         ///< The lock of socket
  SOCK_BUFFER                 SndBuffer;    ///< Send buffer of application's data
  SOCK_BUFFER                 RcvBuffer;    ///< Receive buffer of received data
  EFI_STATUS                  SockError;    ///< The error returned by low layer protocol
  BOOLEAN                     InDestroy;

  //
  // Fields used to manage the connection request
  //
  UINT32                      BackLog;        ///< the limit of connection to this socket
  UINT32                      ConnCnt;        ///< the current count of connections to it
  SOCKET                      *Parent;        ///< listening parent that accept the connection
  LIST_ENTRY                  ConnectionList; ///< the connections maintained by this socket
  //
  // The queue to buffer application's asynchronous token
  //
  LIST_ENTRY                  ListenTokenList;
  LIST_ENTRY                  RcvTokenList;
  LIST_ENTRY                  SndTokenList;
  LIST_ENTRY                  ProcessingSndTokenList;

  SOCK_COMPLETION_TOKEN       *ConnectionToken; ///< app's token to signal if connected
  SOCK_COMPLETION_TOKEN       *CloseToken;      ///< app's token to signal if closed
  //
  // Interface for low level protocol
  //
  SOCK_PROTO_HANDLER          ProtoHandler;                       ///< The request handler of protocol
  UINT8                       ProtoReserved[PROTO_RESERVED_LEN]; ///< Data fields reserved for protocol
  UINT8                       IpVersion;
  NET_PROTOCOL                NetProtocol;                       ///< TCP4 or TCP6 protocol socket used
  //
  // Callbacks after socket is created and before socket is to be destroyed.
  //
  SOCK_CREATE_CALLBACK        CreateCallback;  ///< Callback after created
  SOCK_DESTROY_CALLBACK       DestroyCallback; ///< Callback before destroyed
  VOID                        *Context;        ///< The context of the callback
};
```

The structure is created from `SockCreate()` the calling process:



The one on the left `PktRcvdNotify` is the callback function in IP4 , and the one on the right is the commonly used function for creating sub-items.

`SOCKET` The main members are described as follows:

- `SockHandle` , `NetProtocol` : These two parameters need to be viewed together, and their initialization is located `SockCreate()` in the function:

c

```c
  Status = gBS->InstallMultipleProtocolInterfaces (
                  &Sock->SockHandle,
                  TcpProtocolGuid,
                  &Sock->NetProtocol,
                  NULL
                  );
```

In fact, a Protocol is installed. The corresponding GUID is `TcpProtocolGuid` , which actually has two options, v4 and v6, corresponding to `NetProtocol` two versions:

c

```c
  if (SockInitData->IpVersion == IP_VERSION_4) {
    TcpProtocolGuid = &gEfiTcp4ProtocolGuid;
    ProtocolLength  = sizeof (EFI_TCP4_PROTOCOL);
  } else {
    TcpProtocolGuid = &gEfiTcp6ProtocolGuid;
    ProtocolLength  = sizeof (EFI_TCP6_PROTOCOL);
  }
```

What we need to focus on are `gEfiTcp4ProtocolGuid` and `EFI_TCP4_PROTOCOL` , which correspond to the structure:

c

```c
struct _EFI_TCP4_PROTOCOL {
  EFI_TCP4_GET_MODE_DATA    GetModeData;
  EFI_TCP4_CONFIGURE        Configure;
  EFI_TCP4_ROUTES           Routes;
  EFI_TCP4_CONNECT          Connect;
  EFI_TCP4_ACCEPT           Accept;
  EFI_TCP4_TRANSMIT         Transmit;
  EFI_TCP4_RECEIVE          Receive;
  EFI_TCP4_CLOSE            Close;
  EFI_TCP4_CANCEL           Cancel;
  EFI_TCP4_POLL             Poll;
};
```

It is the TCP interface actually used to send and receive data.

- `DriverBinding` : This value comes `SOCK_INIT_DATA` from `DriverBinding` :

c

```c
Sock->DriverBinding       = SockInitData->DriverBinding;
```

And the latter has `TCP_SERVICE_DATA` from `DriverBindingHandle` :

c

```c
mTcpDefaultSockData.DriverBinding = TcpServiceData->DriverBindingHandle;
```

So in the final analysis `SOCKET` , what is right `DriverBinding` is `TCP_SERVICE_DATA` right `DriverBindingHandle` , and in the end it is `EFI_DRIVER_BINDING_PROTOCOL` right `DriverBindingHandle` .

- `ParentDevicePath` : It is related to the previous parameter:

c

```c
  Status = gBS->OpenProtocol (
                  TcpServiceData->ControllerHandle,
                  &gEfiDevicePathProtocolGuid,
                  (VOID **)&This->ParentDevicePath,
                  TcpServiceData->DriverBindingHandle,
                  This->SockHandle,
                  EFI_OPEN_PROTOCOL_GET_PROTOCOL
                  );
```

In fact, it represents the device path of the network card. Its value is expressed as a string as follows:

bash

```bash
PciRoot(0x0)/Pci(0x2,0x0)/MAC(525400123456,0x1)
```

You don't need to pay attention to the PCI path, the focus is on reaching MAC.

- `DevicePath` : It is the result of `ParentDevicePath` adding on top `IPv4_DEVICE_PATH` :

c

```c
  Sock->DevicePath = AppendDevicePathNode (Sock->ParentDevicePath, DevicePath);
  Status = gBS->InstallProtocolInterface (
                  &Sock->SockHandle,
                  &gEfiDevicePathProtocolGuid,
                  EFI_NATIVE_INTERFACE,
```

```
                  Sock->DevicePath
                  );
```

Its value is represented as a string like this:


```bash
PciRoot(0x0)/Pci(0x2,0x0)/MAC(525400123456,0x1)/IPv4(0.0.0.0)
```

- `Link TCP_SERVICE_DATA` : The connection between this value and `SocketList` .
- `ConfigureState` : Indicates the configuration status of the Socket, with the following values:


```c
///
/// Socket configure state
///
#define SO_UNCONFIGURED        0
#define SO_CONFIGURED_ACTIVE   1
#define SO_CONFIGURED_PASSIVE  2
#define SO_NO_MAPPING          3
```

- `Type` : There are two types of Socket, namely stream format socket and datagram format socket, corresponding to the following codes:


```c
///
///  The socket type.
///
typedef enum {
  SockDgram, ///< This socket providing datagram service
  SockStream ///< This socket providing stream service
} SOCK_TYPE;
```

Stream format sockets are also called "connection-oriented sockets" and have the following characteristics:

1. Data will not disappear during transmission;
2. Data is transmitted in sequence;
3. The sending and receiving of data are not synchronous (some tutorials also say that "there are no data boundaries").

Datagram format sockets are also called "connectionless sockets" and have the following characteristics:

1. Emphasis on rapid transfer rather than sequential transfer;
2. Transmitted data may be lost or corrupted;
3. Limit the size of data transferred each time;
4. The sending and receiving of data are synchronous (some tutorials also call it "the existence of data boundaries").

- `State` : Indicates the status of the Socket itself, with the following values:

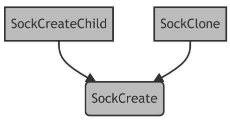

```c
///
/// Socket state
///
#define SO_CLOSED         0
#define SO_LISTENING      1
#define SO_CONNECTING     2
#define SO_CONNECTED      3
#define SO_DISCONNECTING  4
```

- `Flag` : Indicates the identifier in the TCP header, with the following values:


```c
///
/// Flags in the TCP header
///
#define TCP_FLG_FIN  0x01
#define TCP_FLG_SYN  0x02
#define TCP_FLG_RST  0x04
#define TCP_FLG_PSH  0x08
#define TCP_FLG_ACK  0x10
#define TCP_FLG_URG  0x20
```

Their description can be found in the TCP4 Protocol Description .

- `SndBuffer` , `RcvBuffer` : Cache used for sending and receiving data.
- `SockError` : The status of the socket.
- `BackLog` : Indicates the upper limit of the number of socket connections.
- `ConnCnt` : Indicates the current number of Socket connections.
- `Parent` : Its type is also `SOCKET` , from which we can see that there is also a parent-child relationship between Sockets. From the previous call flow, we can see that Socket can `SockCreate()` be created by a function, which in turn is called by two functions:



Their corresponding input parameters are different, `SockCreateChild()` the input parameters are `mTcpDefaultSockData` :


```c
SOCK_INIT_DATA  mTcpDefaultSockData = {
  SockStream,
  SO_CLOSED,
  NULL, // Parent
  TCP_BACKLOG,
  TCP_SND_BUF_SIZE,
  TCP_RCV_BUF_SIZE,
  IP_VERSION_4,
  NULL,
  TcpCreateSocketCallback,
  TcpDestroySocketCallback,
  NULL,

```

```
  NULL,
  0,
  TcpDispatcher,
  NULL,
};
```

`SockClone()` Implementation:


```c
SOCKET *
SockClone (
  IN SOCKET  *Sock
  )
{
  SOCKET          *ClonedSock;
  SOCK_INIT_DATA  InitData;

  InitData.BackLog         = Sock->BackLog;
  InitData.Parent          = Sock;  // 注意这里的Parent
  InitData.State           = Sock->State;
  InitData.ProtoHandler    = Sock->ProtoHandler;
  InitData.Type            = Sock->Type;
  InitData.RcvBufferSize   = Sock->RcvBuffer.HighWater;
  InitData.SndBufferSize   = Sock->SndBuffer.HighWater;
  InitData.DriverBinding   = Sock->DriverBinding;
  InitData.IpVersion       = Sock->IpVersion;
  InitData.Protocol        = &(Sock->NetProtocol);
  InitData.CreateCallback  = Sock->CreateCallback;
  InitData.DestroyCallback = Sock->DestroyCallback;
  InitData.Context         = Sock->Context;
  InitData.ProtoData       = Sock->ProtoReserved;
  InitData.DataSize        = sizeof (Sock->ProtoReserved);

  ClonedSock = SockCreate (&InitData);
```

This brings up a new parent-child relationship. In actual testing, it is found that `SockCreateChild()` it will be executed during startup and `Parent` the value is 0, while `SockClone()` when a Socket is created using TCP, it `Parent` is a valid value.

- `ConnectionList` : The connection maintained by the current Socket.
- `ListenTokenList` , `RcvTokenList` , `SndTokenList` , `ProcessingSndTokenList` : Token list for processing sending and receiving data.
- `ConnectionToken` : Token called after the Socket is connected.
- `CloseToken` : Token called when the Socket is closed.
- `ProtoHandler` 、 `ProtoReserved` ： Socket request callback function and corresponding input parameters. The callback function will `TcpDispatcher()` perform different operations according to the input parameters:


```c
EFI_STATUS
TcpDispatcher (
  IN SOCKET  *Sock,
  IN UINT8   Request,
  IN VOID    *Data    OPTIONAL
  )
{
  switch (Request) {
    case SOCK_POLL:
    case SOCK_CONSUMED:
    case SOCK_SND:
    case SOCK_CLOSE:
    case SOCK_ABORT:
    case SOCK_SNDPUSH:
    case SOCK_SNDURG:
    case SOCK_CONNECT:
    case SOCK_ATTACH:
    case SOCK_FLUSH:
    case SOCK_DETACH:
    case SOCK_CONFIGURE:
    case SOCK_MODE:
    case SOCK_ROUTE:
    default:
  }
}
```

- `IpVersion` ： This is it `IP_VERSION_4` .
- `CreateCallback` , `DestroyCallback` , `Context` : corresponding `mTcpDefaultSockData` functions and their input parameters.

EFI_TCP4_PROTOCOL

The structure of the Protocol is as follows:


```c
///
/// The EFI_TCP4_PROTOCOL defines the EFI TCPv4 Protocol child to be used by
/// any network drivers or applications to send or receive data stream.
/// It can either listen on a specified port as a service or actively connected
/// to remote peer as a client. Each instance has its own independent settings,
/// such as the routing table.
///
struct _EFI_TCP4_PROTOCOL {
  EFI_TCP4_GET_MODE_DATA    GetModeData;
  EFI_TCP4_CONFIGURE        Configure;
  EFI_TCP4_ROUTES           Routes;
  EFI_TCP4_CONNECT          Connect;
  EFI_TCP4_ACCEPT           Accept;
  EFI_TCP4_TRANSMIT         Transmit;
  EFI_TCP4_RECEIVE          Receive;
  EFI_TCP4_CLOSE            Close;
  EFI_TCP4_CANCEL           Cancel;
  EFI_TCP4_POLL             Poll;
};
```

The corresponding implementation is in NetworkPkg\TcpDxe\TcpDriver.c:


```c
EFI_TCP4_PROTOCOL  gTcp4ProtocolTemplate = {
  Tcp4GetModeData,
  Tcp4Configure,
  Tcp4Routes,
  Tcp4Connect,
  Tcp4Accept,
```

```
  Tcp4Transmit,
  Tcp4Receive,
  Tcp4Close,
  Tcp4Cancel,
  Tcp4Poll
};
```

Compared with other network protocols, this one is slightly different. It includes common TCP operations such as Connect, Accept, and Close.

The implementation of these functions will be introduced later.

Tcp4.Connect

The corresponding implementation is `Tcp4Connect` that its implementation is the connection of Socket:

c AI generated projects 登录复制 run

```c
EFI_STATUS
EFIAPI
Tcp4Connect (
  IN EFI_TCP4_PROTOCOL          *This,
  IN EFI_TCP4_CONNECTION_TOKEN  *ConnectionToken
  )
{
  return SockConnect (Sock, ConnectionToken);
}
```

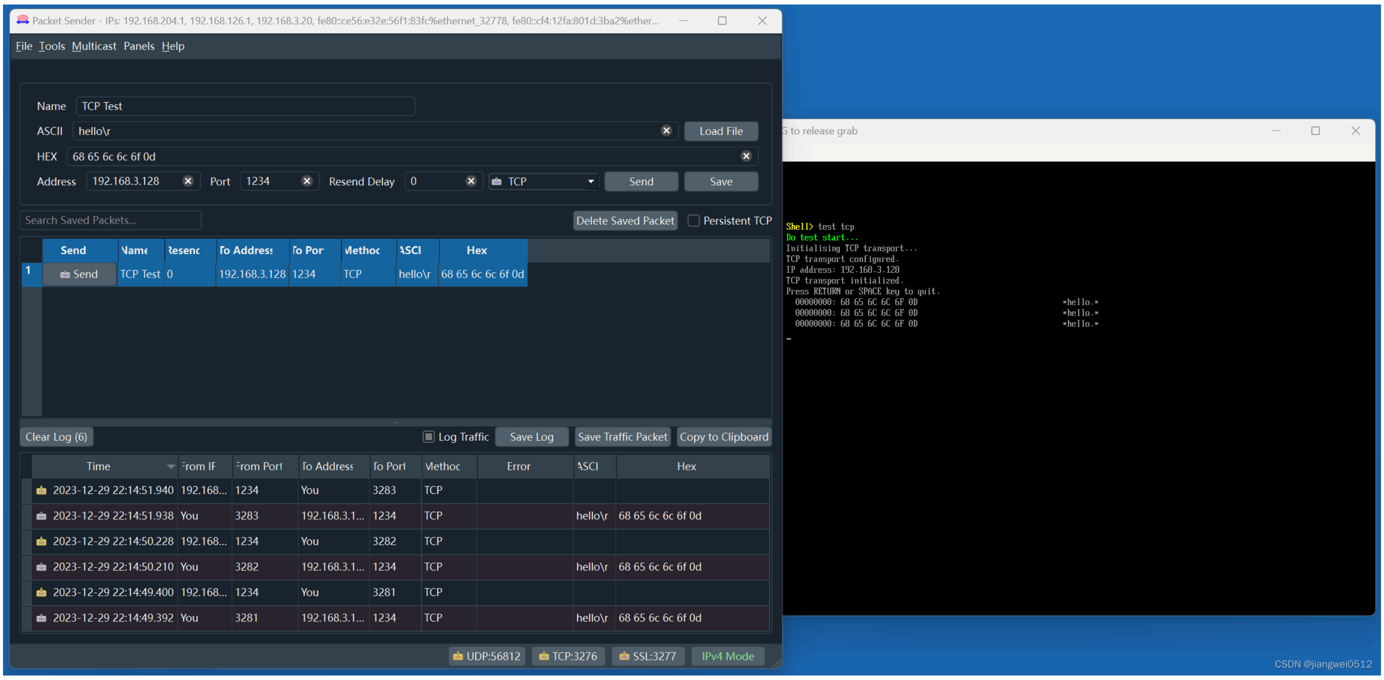
Other operations such as Tcp4Accept, Tcp4Transmit, Tcp4Receive, and Tcp4Close are also Socket operations.

TCP Code Examples

The TCP code example can be found in beni/BeniPkg/DynamicCommand/TestDynamicCommand/TestTcp.c · jiangwei/edk2-beni - Code Cloud - Open Source China (gitee.com) . It actually comes from EmbeddedPkg\Drivers\AndroidFastbootTransportTcpDxe\FastbootTransportTcpDxe.inf, which is an open source module. However, an error will be reported during the compilation process, so it is transplanted here, corresponding to BeniPkg\Dxe\TransportTcpDxe\TcpTransportDxe.inf, and the TestTcp.c module calls this module.

It will start a TCP server, which can be interacted with by a TCP client (here we use Packet Sender, from Packet Sender - Free utility to for sending / receiving of network packets. TCP, UDP, SSL.).

The results are as follows:



The data is received and printed here, so you can see the data passed by the program on the left under the right Shell.

Note that the Address (192.168.3.128) and Port (1234) here are hard-coded and need to be modified according to the actual situation.