[UEFI Basics] EDK Network Framework (IP4)

jiangwei0512 Posted on 2024-01-13 16:28:04 Read 2.3k Collection 17 Likes 15

Category Column: UEFI Development Basics Article Tags: network uefi

摘要 This article introduces the IP4 protocol in detail, including its function modules (IP address, routing, packet sub-packeting and packet grouping), and related protocols (ICMP, IGMP). It explains the relationship of IP4 in the UEFI network protocol stack, and introduces IP4 services, interfaces, configuration, etc. It also gives IP4 code examples, such as the execution flow of the ping program.

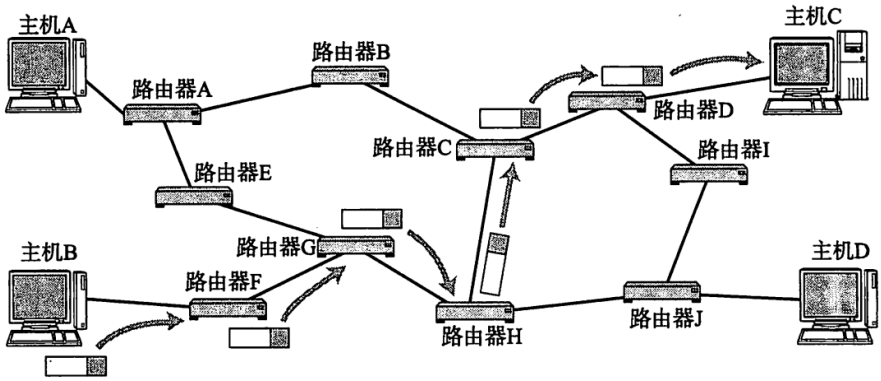
The summary is generated in C Know , supported by DeepSeek-R1 full version, go to experience>

IP4

IP4 Protocol Description

IP stands for **Internet Protocol** . It belongs to the network layer and packages various types of data link layers below it, so that the network layer can span different data links and realize data packet transmission between nodes at both ends even on different data links .

The main function of the IP layer is to "enable communication between terminal nodes":



IP的主要作用就是在复杂的网络环境中将数据包发给最终的目标地址。

The format of the IPv4 header 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Time to Live |    Protocol    |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Source Address                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Destination Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```


The description of each parameter is as follows:

Fields	Length (bits)	meaning
Version	4	4: indicates IPv4; 6: indicates IPv6.
IHL	4	The header length is 20 if there is no Option field. The maximum length is 60. This value limits the record routing options. Each unit is 4 bytes.
Type of Service	8	Service type. This field is effective only when QoS differentiated service is required.
Total Length	16	Total length, the length of the entire IP datagram, including the header and data, in bytes, with a maximum length of 65535. The total length must not exceed the maximum transmission unit MTU.
Identification	16	Identifier. Each time the host sends a message, it increases by 1. This field is used when fragments are reassembled.
Flags	3	Flag bit: Bit 0: Reserved bit, must be 0. Bit 1: DF (Don't Fragment), whether the message can be fragmented, 0 means it can be fragmented, 1 means it cannot be fragmented. Bit 2: MF (More Fragment), indicates whether the message is the last fragment, 0 means the last fragment, 1 means there are more fragments.
Fragment Offset	13	Fragment offset: This field is used when reassembling fragments. It indicates the relative position of a fragment in the original packet after the fragmentation of a longer packet. The offset unit is 8 bytes.
Time to Live	8	Lifetime: The maximum number of routes that a data packet can pass through, that is, the maximum number of routers that the data packet can pass through in the network.
Protocol	8	Protocol: The next layer protocol. Indicates which protocol is used by the data carried by this packet, so that the IP layer of the destination host can hand over the data to which process for processing. Common values: 0: Reserved 1: ICMP, Internet Control Message [RFC792] 2: IGMP, Internet Group Management [RFC1112] 3: GGP, Gateway-to-Gateway [RFC823] 4: IP in IP (encapsulation) [RFC2003] 6: TCP Transmission Control Protocol [RFC793] 17: UDP User Datagram Protocol [RFC768]

Fields	Length (bits)	meaning
		20: HMP Host Monitoring Protocol [RFC 869] 27: RDP Reliable Data Protocol [RFC908] 46: RSVP (Reservation Protocol) 47: GRE (General Routing Encapsulation) 50: ESP Encap Security Payload [RFC2406] 51: AH (Authentication Header) [RFC2402] 54: NARP (NBMA Address Resolution Protocol) [RFC1735] 58: IPv6-ICMP (ICMP for IPv6) [RFC1883] 59: IPv6-NoNxt (No Next Header for IPv6) [RFC1883] 60: IPv6-Opts (Destination Options for IPv6) [RFC1883] 89: OSPF (OSPF Version 2) [RFC 1583] 112: VRRP (Virtual Router Redundancy Protocol) [RFC3768] 115: L2TP (Layer Two Tunneling Protocol) 124: ISIS over IPv4 126: CRTP (Combat Radio Transport Protocol) 127: CRUDP (Combat Radio User Protocol) 132: SCTP (Stream Control Transmission Protocol) 136: UDPLite [RFC 3828] 137: MPLS-in-IP [RFC 4023]
Header Checksum	16	The header checksum only checks the header of the data packet, not the data part. Here, the CRC checksum is not used, but a simple calculation method is used.
Source Address	32	Source IP address.
Destination Address	32	Destination IP address.
Options	variable	The option field is used to support troubleshooting, measurement, and security measures, and has rich content. The option field length is variable, ranging from 1 byte to 40 bytes, depending on the function of the selected option.
Padding	variable	Fill the field with all 0s.

The preceding part constitutes the IP4 header in the UEFI code:

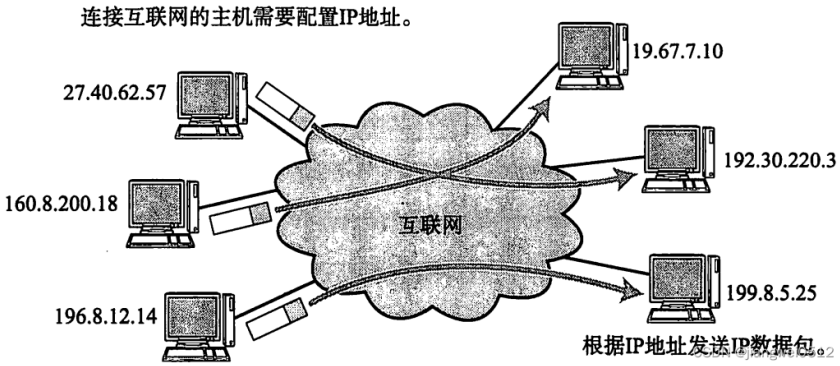
c AI generated projects 登录复制 run

```c
//
// The EFI_IP4_HEADER is hard to use because the source and
// destination address are defined as EFI_IPv4_ADDRESS, which
// is a structure. Two structures can't be compared or masked
// directly. This is why there is an internal representation.
//
typedef struct {
  UINT8       HeadLen : 4;
  UINT8       Ver     : 4;
  UINT8       Tos;
  UINT16      TotalLen;
  UINT16      Id;
  UINT16      Fragment;
  UINT8       Ttl;
  UINT8       Protocol;
  UINT16      Checksum;
  IP4_ADDR    Src;
  IP4_ADDR    Dst;
} IP4_HEAD;
```

IP is roughly divided into three major functional modules, namely IP address, routing (forwarding to the final node) and IP subpackaging and assembly.

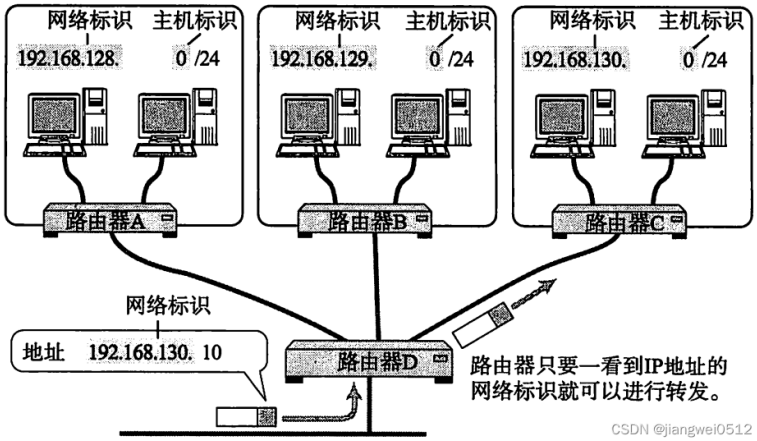
IP address

The IP address is used to "identify the destination address for communication among all hosts connected to the network." In TCP/IP communication, all hosts or routers must set their own IP addresses.



An IP address (only focusing on IPv4) is a 32-bit integer. For easy memorization, it is usually divided into 8-bit groups, divided into 4 groups, each group is separated by ".", and then each group of numbers is converted into decimal.

The IP address is also composed of two parts: the "network identifier" and the "host identifier". The network identifier must ensure that the addresses of each interconnected segment (network segment) are not repeated; the host identifier is not allowed to appear repeatedly in the same network segment. IP addresses with the same network identifier form the same network segment, and the specific bits of the network segment are determined by the subnet mask. There are two ways to identify the subnet mask. The first is to use a number, such as 24 in the figure below, which means counting from the beginning to the 24th bit (the value is 1) represents the subnet mask:

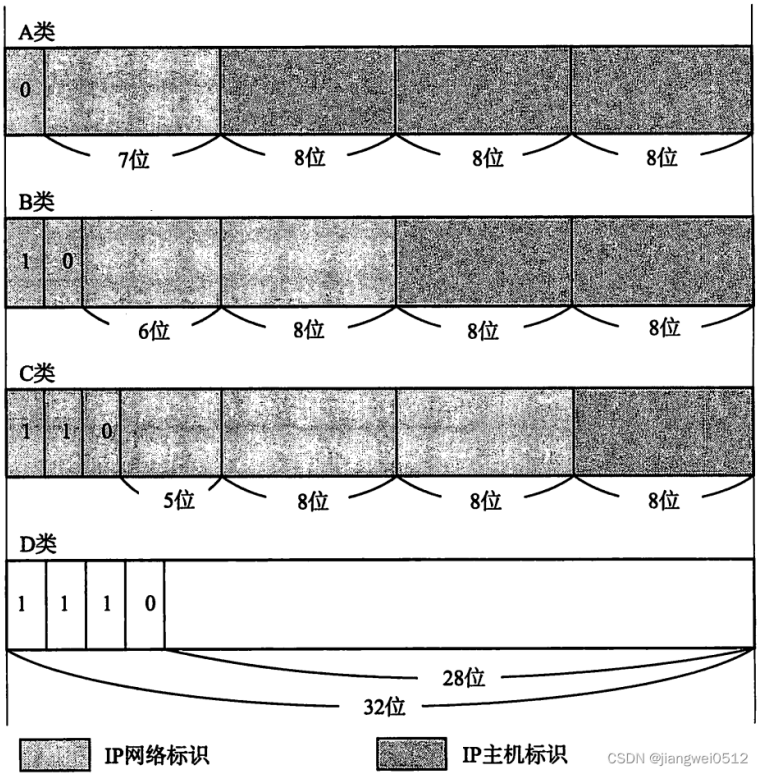


The second is to represent the subnet mask by the IP address, with the first 24 bits being 1 and the rest being 0, as shown here:


```bash
11111111 11111111 11111111 00000000
```

That is 255.255.255.0.

IP addresses can also be classified by level, namely Class A, Class B, Class C and Class D:



From here, we can see that the 192.168.128.x in the previous example is a Class C address. For this type, there are 254 addresses in the same network segment (0 and 1 are reserved addresses).

The broadcast address is used to send data packets between hosts connected to each other in the same link. If the host identification part is all 1, it indicates a broadcast address. Taking the previous example, 192.168.128.255 is a broadcast address. IP packets with broadcast addresses will be blocked by routers, so they will not reach other links except 192.168.128.0/24.

Multicast (also called groupcast) is used to send packets to all hosts in a specific group. Multicast uses class D addresses, so the multicast address is "1110" from the first to the fourth bit. Some common multicast addresses:

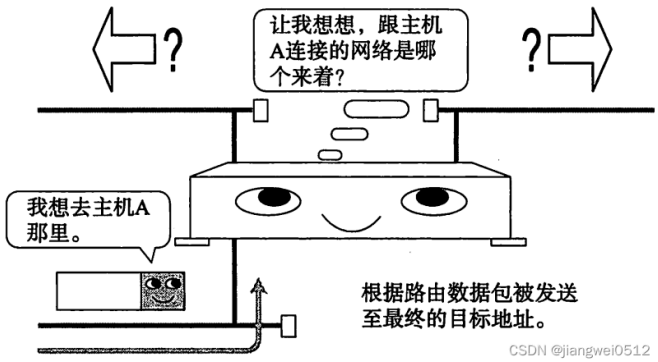
地址	内　　容
224. 0. 0. 0	（预定）
224. 0. 0. 1	子网内所有的系统
224. 0. 0. 2	子网内所有的路由器
224. 0. 0. 5	OSPF 路由器
224. 0. 0. 6	OSPF 指定路由器
224. 0. 0. 9	RIP2 路由器
224. 0. 0. 10	IGRP 路由器
224. 0. 0. 11	Mobile-Agents
224. 0. 0. 12	DHCP 服务器/中继器代理
224. 0. 0. 14	RSVP-ENCAPSULATION
224. 0. 1. 1	NTP Network Time Protocol
224. 0. 1. 8	SUN NIS+ Information Service
224. 0. 1. 22	Service Location （SVRLOC）
224. 0. 1. 33	RSVP-encap-1
224. 0. 1. 34	RSVP-encap-2
224. 0. 1. 35	Directory Agent Discovery （SVRLOC-DA）
224. 0. 2. 2	SUN RPC PMAPPROC CALLIT

At this point, IP communications can be divided into three categories:

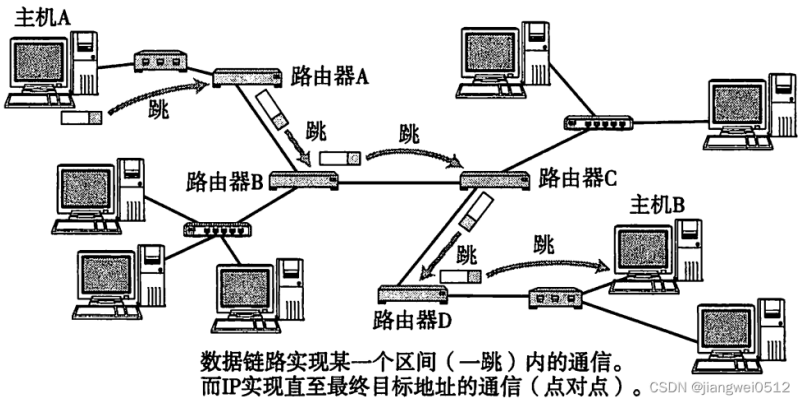
- Unicast: One-to-one communication.
- Broadcast: One-to-all communication.
- Multicast: One-to-group communication.

routing

Routing is the function of sending packet data to its final destination.

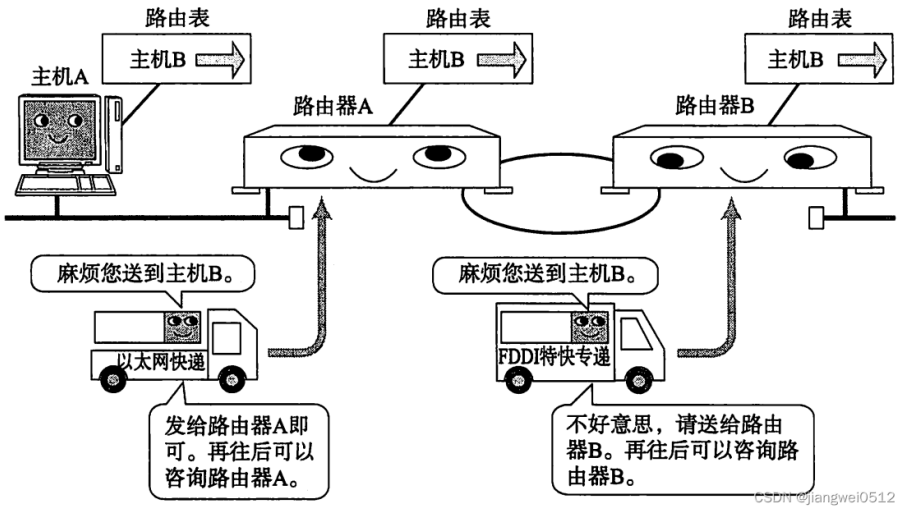


Sending data to the final destination address is achieved by "hopping" (Hop):



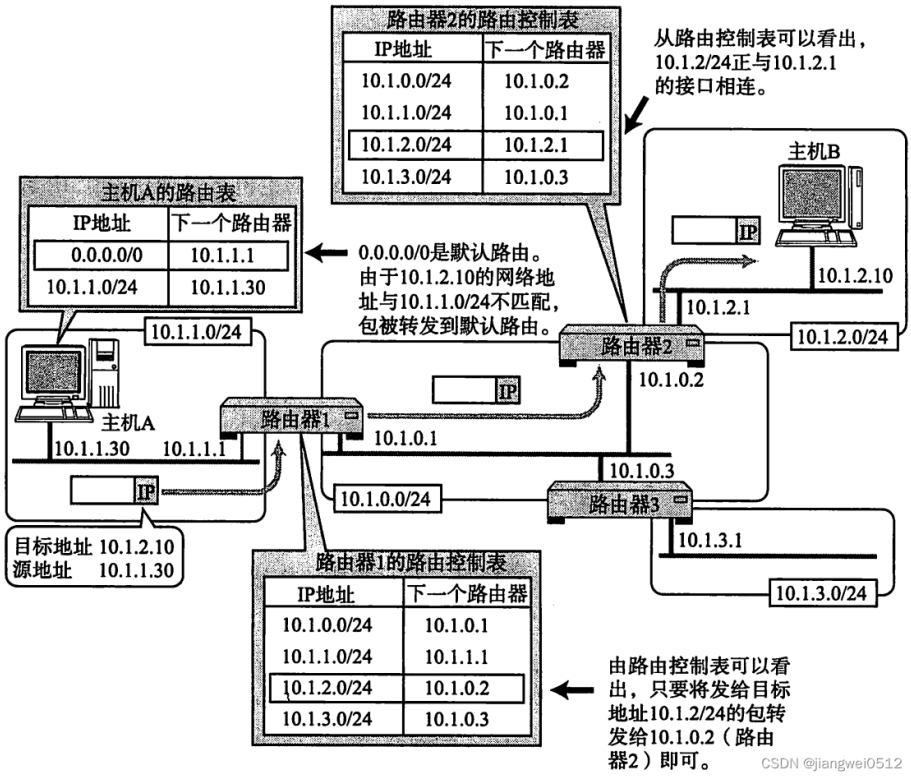
"Hop" refers to an interval in the network. IP packets are forwarded between "hops" in the network, so IP routing is also called multi-hop routing.

When forwarding an IP data packet, only the next router or host is specified, rather than all the paths to the final destination address. This is because each hop specifies the next hop operation when forwarding an IP data packet until the packet reaches the final destination address.



The address used when sending a data packet is the address of the network layer, that is, the IP address. However, the IP address alone is not enough to send the data packet to the destination address on the other end. In the process of data transmission, information such as "specifying the router or host" is also required in order to actually send it to the target address.

In order to send data packets to the target host, all hosts (including routers) maintain a routing table (routing control table). This table records which router the IP data should be sent to in the next step. The IP packet will be transmitted on each data link according to this routing table.



Subpackaging and grouping

It is necessary for any host to process IP packet fragmentation accordingly. Fragmentation is often processed when a large message cannot be sent out at once on the network. The default MTU (maximum transmission unit) of Ethernet is 1500 bytes.

IP-related protocols

In actual communication, IP protocol alone is not enough, and IP-related technologies are also needed to achieve it. Some of them will be further explained later, and ICMP and IGMP need to be introduced here. Their implementation is in the driver received in this section.

Both ICMP and IGMP are encapsulated in IP packets.

ICMP

ICMP stands for **Internet Control Message** Protocol . Its main functions include: confirming whether the IP packet is successfully delivered **to** the target address; notifying the specific reason why the IP packet is discarded during the sending process; improving network settings; etc. Its format is as follows:

```
+0------7-------15---------------31
|  Type | Code  |     Checksum     |
+----------------------------------+
|          Message Body            |
|        (Variable length)         |
+----------------------------------+
```


The description of each parameter is as follows:

Fields	Length (bytes)	meaning
Type	1	Message type, used to identify the message. The value and meaning of the Type field will be further described later.
Code	1	Code, which provides further information about the message type. The value and meaning of the Code field will be further explained later.
Checksum	2	Checksum, uses the same additive checksum algorithm as IP, but the ICMP checksum only covers ICMP messages.

Fields	Length (bytes)	meaning
Message Body	variable	The length and content of the field depends on the message type and code.

The ICMP message type code correspondence table is as follows:

Type	Code	describe
0	0	Echo reply (ping reply)
3	0	Network unreachable
3	1	Host unreachable
3	2	Protocol unreachable
3	3	Port Unreachable
3	4	Fragmentation is required but the no-fragmentation bit is set
3	5	Source station routing failure
3	6	Destination network unknown
3	7	The destination host is unknown
3	8	The source host is isolated (invalid)
3	9	The destination network is forcibly banned
3	10	The destination host is forcibly banned
3	11	Network unreachable due to TOS
3	12	Host unreachable due to TOS
3	13	Communication is forcibly blocked due to filtering
3	14	Host exceeds authority
3	15	Priority right ceases to take effect
4	0	The source is closed
5	0	Redirecting to the network
5	1	Redirecting to a host
5	2	For service types and network redirection
5	3	Redirection for service type and host
8	0	Request echo (ping request)
9	0	Router Advertisements
10	0	Router Solicitation
11	0	The lifetime during transmission is 0
11	1	The lifetime is 0 during datagram assembly.
12	0	Bad IP header
12	1	Missing required options
13	0	Timestamp request (deprecated)
14	0	Timestamp response (invalid)
15	0	Information Request (Void)
16	0	Information response (discarded)
17	0	Address Mask Request
18	0	Address Mask Reply

IGMP

IGMP stands for **Internet Group Management Protocol** . Its main functions include: establishing and maintaining the relationship between multicast members between the receiver host and the directly connected multicast router; and exchanging IGMP messages between the receiver host and the multicast router to implement group member management .

There are multiple versions of IGMP. Currently, IGMPv2 is maintained under BIOS. Its format is as follows:

The description of each parameter is as follows:

Fields	Length (bits)	describe
Type	8	Message type, there are the following types: 0x11: Membership Query, IGMP query message. 0x12: Version 1 Membership Report, IGMPv1 member report message. 0x16: Version 2 Membership Report, IGMPv2 member report message. 0x17: Leave Group, leave message.
Max Resp Time	8	The maximum time in 1/10 seconds before sending a response report. The default value is 10 seconds. The new Maximum Response Time (in 1/10 seconds) field allows a querying router to specify an exact query interval response time for its query messages. IGMP version 2 hosts use this as an upper limit when randomly selecting their response time values. This helps control bursts of responses when spacing query responses.
Checksum	16	IGMP message checksum. When transmitting a message, the checksum must be calculated and filled in this field; when receiving a message, the checksum must be checked before processing the message to determine whether an error occurred during the transmission of the IGMP message.
Group Address	32	Multicast group address (0.0.0.0 if it is a general query). This field has the same meaning as in IGMP version 1, except that it is set to 0.0.0.0 for a general query.

IP4 Code Overview

IP4 is a universal network protocol. IP protocol has v4 and v6 versions. This article only introduces v4. In fact, now NetworkPkg\Ip4Dxe\Ip4Dxe.inf, here we first need to look at its entry:

c AI generated projects 登录复制 run

```c
EFI_STATUS
EFIAPI
Ip4DriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  VOID  *Registration;

  EfiCreateProtocolNotifyEvent (
    &gEfiIpSec2ProtocolGuid,
    TPL_CALLBACK,
    IpSec2InstalledCallback,
    NULL,
    &Registration
    );

  return EfiLibInstallDriverBindingComponentName2 (
           ImageHandle,
           SystemTable,
           &gIp4DriverBinding,
           ImageHandle,
           &gIp4ComponentName,
           &gIp4ComponentName2
           );
}
```

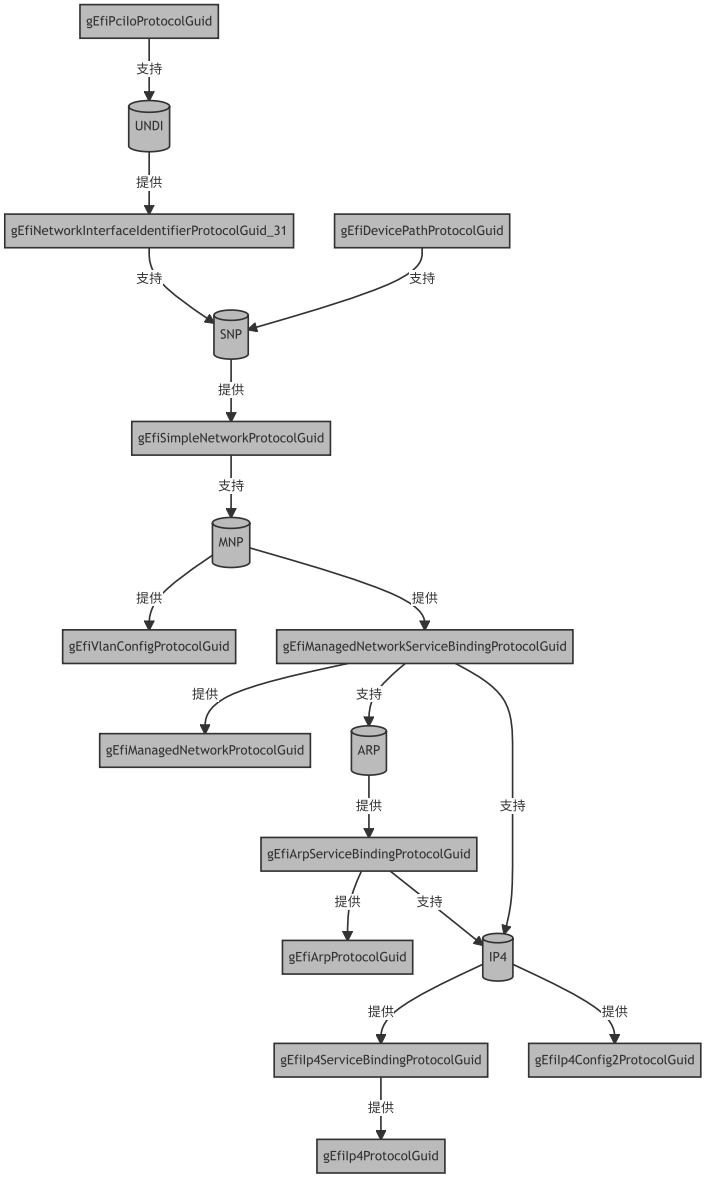
Installation `EFI_DRIVER_BINDING_PROTOCOL` is the basis, but before that there is a `gEfiIpSec2ProtocolGuid` callback, this protocol is for the security of the IP layer, but the current UEFI does not support the IPSec protocol, so there is actually no need to pay special attention to it. `gIp4DriverBinding` As follows:

c AI generated projects 登录复制 run

```c
EFI_DRIVER_BINDING_PROTOCOL  gIp4DriverBinding = {
  Ip4DriverBindingSupported,
  Ip4DriverBindingStart,
  Ip4DriverBindingStop,
  0xa,
  NULL,
  NULL
};
```

The IP4 driver code is much larger than other UEFI network protocol modules. The reasons include: 1) it needs to handle more protocols, such as ICMP, IGMP, etc.; 2) in addition to the transmission code, the IP configuration code is also added; 3) there is a lot of routing code. Due to the large amount of content, this article will not introduce all the content about ICMP, IGMP and routing.

Relationship diagram of IP4 in UEFI network protocol stack:



Ip4DriverBindingSupported

IP4 depends on MNP and ARP :

c AI generated projects 登录复制 run

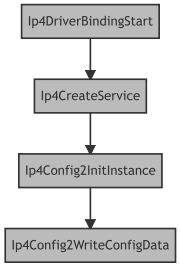
```c
EFI_STATUS
EFIAPI
Ip4DriverBindingSupported (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath OPTIONAL
  )
{
  //
  // Test for the MNP service binding Protocol
  //
  Status = gBS->OpenProtocol (
                  ControllerHandle,
                  &gEfiManagedNetworkServiceBindingProtocolGuid,
                  NULL,
                  This->DriverBindingHandle,
                  ControllerHandle,
                  EFI_OPEN_PROTOCOL_TEST_PROTOCOL
                  );

  //
  // Test for the Arp service binding Protocol
  //
  Status = gBS->OpenProtocol (
                  ControllerHandle,
                  &gEfiArpServiceBindingProtocolGuid,
                  NULL,
                  This->DriverBindingHandle,
                  ControllerHandle,
                  EFI_OPEN_PROTOCOL_TEST_PROTOCOL
                  );
}
```

Ip4DriverBindingStart

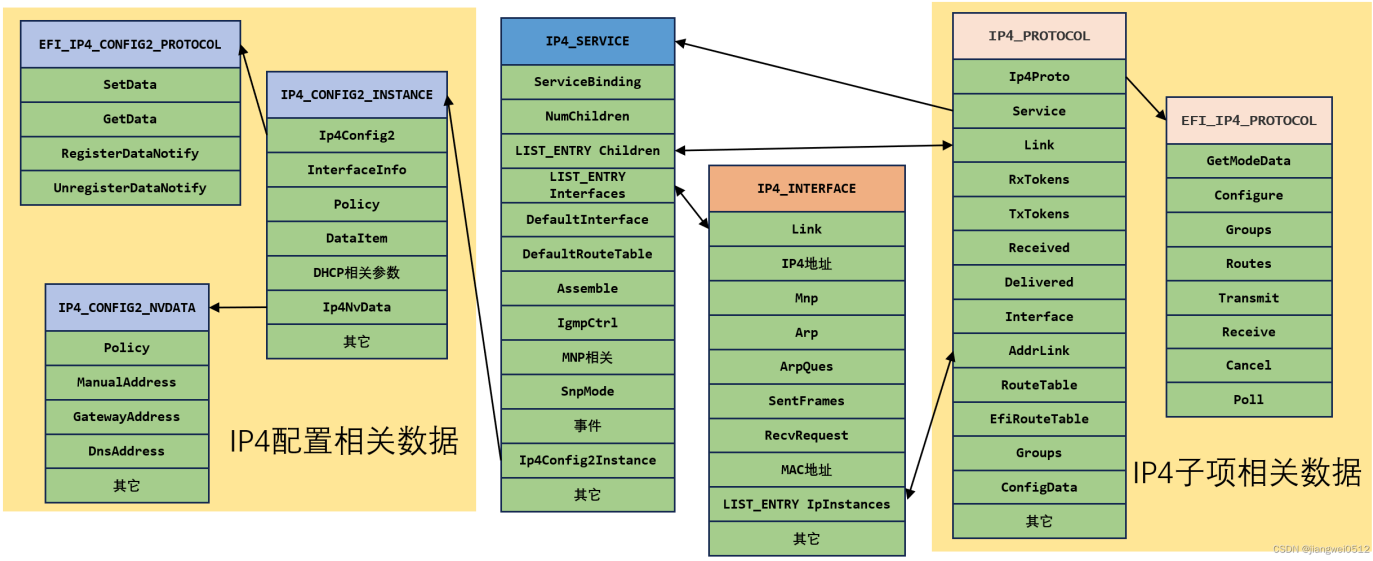
The flow of the Start function is as follows:

1. The IP4 service is created through `Ip4CreateService()` the function, and the main implementation is to initialize `IP4_SERVICE` the structure.
2. Installation `gEfiIp4ServiceBindingProtocolGuid` and `gEfiIp4Config2ProtocolGuid` . The former is similar to other network drivers, which is a service interface; the latter is used to configure IP.
3. Get the IP4 configuration. These configurations are stored in variables. The name is the string corresponding to the MAC address. The structure corresponding to the configuration is `IP4_CONFIG2_INSTANCE` . `Ip4CreateService()` There is a default initialization in. The process is roughly as follows:



4. There are different types of IP4 configurations, and the interface called is processed according to the actual `EFI_IP4_CONFIG2_PROTOCOL` type `SetData()` .
5. Start receiving data, and the timer will be started at this time. The timer is also `Ip4CreateService()` created in .

Therefore, the main operations are still in `IP4_SERVICE` the structures related to it, and their relationship is as follows:



IP4_SERVICE

IP4 services correspond to MNP and ARP services, so there can be multiple IP4 services, each corresponding to such a structure. Like ARP , since it does not involve hardware, it starts with a SERVICE structure, and there is no hardware-related structure. `IP4_SERVICE` Initialization is completed `Ip4DriverBindingStart()` by `Ip4CreateService()` :

c AI generated projects 登录复制 run

```c
EFI_STATUS
EFIAPI
Ip4DriverBindingStart (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath OPTIONAL
  )
{
  Status = Ip4CreateService (ControllerHandle, This->DriverBindingHandle, &IpSb);
}
```

`IP4_SERVICE` Located in NetworkPkg\Ip4Dxe\Ip4Impl.h, its members are as follows:

c AI generated projects 登录复制 run

```c
struct _IP4_SERVICE {
  UINT32                          Signature;
  EFI_SERVICE_BINDING_PROTOCOL    ServiceBinding;
  INTN                            State;

  //
  // List of all the IP instances and interfaces, and default
  // interface and route table and caches.
  //
  UINTN                           NumChildren;
  LIST_ENTRY                      Children;

  LIST_ENTRY                      Interfaces;

  IP4_INTERFACE                   *DefaultInterface;
  IP4_ROUTE_TABLE                 *DefaultRouteTable;

  //
  // Ip reassemble utilities, and IGMP data
  //
  IP4_ASSEMBLE_TABLE              Assemble;
  IGMP_SERVICE_DATA               IgmpCtrl;

  //
  // Low level protocol used by this service instance
  //
  EFI_HANDLE                      Image;
  EFI_HANDLE                      Controller;

  EFI_HANDLE                      MnpChildHandle;
  EFI_MANAGED_NETWORK_PROTOCOL    *Mnp;

  EFI_MANAGED_NETWORK_CONFIG_DATA MnpConfigData;
  EFI_SIMPLE_NETWORK_MODE         SnpMode;

  EFI_EVENT                       Timer;
  EFI_EVENT                       ReconfigCheckTimer;
  EFI_EVENT                       ReconfigEvent;

  BOOLEAN                         Reconfig;

```

```
  //
  // Underlying media present status.
  //
  BOOLEAN                          MediaPresent;

  //
  // IPv4 Configuration II Protocol instance
  //
  IP4_CONFIG2_INSTANCE             Ip4Config2Instance;

  CHAR16                           *MacString;

  UINT32                           MaxPacketSize;
  UINT32                           OldMaxPacketSize; ///< The MTU before IPsec enable.
};
```

Here are the more important members:

- `ServiceBinding` : Corresponding IP4 service interface:

c AI generated projects 登录复制 run

```
  IpSb->ServiceBinding.CreateChild  = Ip4ServiceBindingCreateChild;
  IpSb->ServiceBinding.DestroyChild = Ip4ServiceBindingDestroyChild;
```

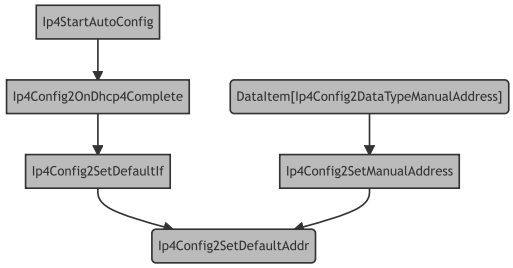
- `State` ： IP4 service status, there are different types, different types have different operations, the current status is:

c AI generated projects 登录复制 run

```
//
// The state of IP4 service. It starts from UNSTARTED. It transits
// to STARTED if autoconfigure is started. If default address is
// configured, it becomes CONFIGED. and if partly destroyed, it goes
// to DESTROY.
//
#define IP4_SERVICE_UNSTARTED  0
#define IP4_SERVICE_STARTED    1
#define IP4_SERVICE_CONFIGED   2
#define IP4_SERVICE_DESTROY    3
```

`IP4_SERVICE_STARTED` It will be `Ip4StartAutoConfig()` set in and will be further introduced later.

`IP4_SERVICE_CONFIGED` It will be `Ip4Config2SetDefaultAddr()` set in the following process:



This is back to the beginning `Ip4StartAutoConfig()` .

These parameters are all related to IP configuration, which will be further introduced later.

- `NumChildren` , `Children` : The sub-item of IP4, the linked list corresponds to `IP4_PROTOCOL` , it is created through the IP4 service `CreateChild()` , and is the unit of IP4 communication.
- `Interfaces` : corresponds to `IP4_INTERFACE` a list of , which will be further introduced later.
- `DefaultInterface` : Default `IP4_INTERFACE` , by `Ip4CreateInterface()` creation.
- `DefaultRouteTable` : The default routing table.
- `Assemble` : corresponds to `IP4_ASSEMBLE_ENTRY` the list of , which will be further introduced later.
- `IgmpCtrl` IGMP stands for **Internet Group** Management **Protocol** , which is a protocol in the TCP/IP protocol suite responsible for IPv4 multicast member management. This member is a structure used to **control** IGMP . `IGMP_SERVICE_DATA`
- `MnpChildHandle` 、 `Mnp` : IP4 corresponding to `EFI_MANAGED_NETWORK_PROTOCOL` its Handle.
- `MnpConfigData` : MNP configuration data, its value is also fixed:

c AI generated projects 登录复制 run

```
  IpSb->MnpConfigData.ReceivedQueueTimeoutValue = 0;
  IpSb->MnpConfigData.TransmitQueueTimeoutValue = 0;
  IpSb->MnpConfigData.ProtocolTypeFilter        = IP4_ETHER_PROTO;
  IpSb->MnpConfigData.EnableUnicastReceive      = TRUE;
  IpSb->MnpConfigData.EnableMulticastReceive    = TRUE;
  IpSb->MnpConfigData.EnableBroadcastReceive    = TRUE;
  IpSb->MnpConfigData.EnablePromiscuousReceive  = FALSE;
  IpSb->MnpConfigData.FlushQueuesOnReset        = TRUE;
  IpSb->MnpConfigData.EnableReceiveTimestamps   = FALSE;
  IpSb->MnpConfigData.DisableBackgroundPolling  = FALSE;
```

- `SnpMode` ： SNP mode.
- `Timer` , `ReconfigCheckTimer` , `Ip4AutoReconfigCallBack` : Events required by IP4, which will be further introduced later.
- `Reconfig` : A flag indicating whether reconfiguration is required.
- `MediaPresent` : Indicates whether the network cable is connected.
- `Ip4Config2Instance` ： The corresponding structure is IP4_CONFIG2_INSTANCE . IP4 also has different configurations, such as static and DHCP. This structure is used to handle these configurations, which will be further introduced later. Its initialization:

c AI generated projects 登录复制 run

```
EFI_STATUS
Ip4CreateService (
  IN  EFI_HANDLE   Controller,
  IN  EFI_HANDLE   ImageHandle,
  OUT IP4_SERVICE  **Service
  )
{
  ZeroMem (&IpSb->Ip4Config2Instance, sizeof (IP4_CONFIG2_INSTANCE));

  Status = Ip4Config2InitInstance (&IpSb->Ip4Config2Instance);
```

IP4_INTERFACE

Different from the network protocols introduced above, IP4 has a header between the service data and sub-item data `IP4_INTERFACE` . This is because the main function of the IP layer is communication between terminal nodes, and in order to represent a node, it must be described by an IP address, MAC, etc. `IP4_INTERFACE` The main function of the header is to describe the node.

`IP4_INTERFACE` Created by `Ip4CreateInterface()` :

c AI generated projects 登录复制 run

```c
IP4_INTERFACE *
Ip4CreateInterface (
  IN  EFI_MANAGED_NETWORK_PROTOCOL  *Mnp,
  IN  EFI_HANDLE                    Controller,
  IN  EFI_HANDLE                    ImageHandle
  )
{
  IP4_INTERFACE            *Interface;
  EFI_SIMPLE_NETWORK_MODE  SnpMode;

  Interface = AllocatePool (sizeof (IP4_INTERFACE));

  Interface->Signature = IP4_INTERFACE_SIGNATURE;
  InitializeListHead (&Interface->Link);
  Interface->RefCnt = 1;

  Interface->Ip         = IP4_ALLZERO_ADDRESS;
  Interface->SubnetMask = IP4_ALLZERO_ADDRESS;
  Interface->Configured = FALSE;

  Interface->Controller = Controller;
  Interface->Image      = ImageHandle;
  Interface->Mnp        = Mnp;
  Interface->Arp        = NULL;
  Interface->ArpHandle  = NULL;

  InitializeListHead (&Interface->ArpQues);
  InitializeListHead (&Interface->SentFrames);

  Interface->RecvRequest = NULL;

  //
  // Get the interface's Mac address and broadcast mac address from SNP
  //
  if (EFI_ERROR (Mnp->GetModeData (Mnp, NULL, &SnpMode))) {
    FreePool (Interface);
    return NULL;
  }

  CopyMem (&Interface->Mac, &SnpMode.CurrentAddress, sizeof (Interface->Mac));
  CopyMem (&Interface->BroadcastMac, &SnpMode.BroadcastAddress, sizeof (Interface->BroadcastMac));
  Interface->HwaddrLen = SnpMode.HwAddressSize;

  InitializeListHead (&Interface->IpInstances);
  Interface->PromiscRecv = FALSE;
}
```

The calling process consists of two parts:

1. First, `Ip4CreateService()` it is used when creating a service, but the one created in this process will be assigned to `IP4_INTERFACE` after initialization : `IP4_SERVICE DefaultInterface`

c AI generated projects 登录复制 run

```c
EFI_STATUS
Ip4CreateService (
  IN  EFI_HANDLE   Controller,
  IN  EFI_HANDLE   ImageHandle,
  OUT IP4_SERVICE  **Service
  )
{
  IpSb->DefaultInterface = Ip4CreateInterface (IpSb->Mnp, Controller, ImageHandle);
}
```

So this is just a default version.

2. `Ip4CreateInterface()` Secondly, it will be called to create when configuring IP `IP4_INTERFACE` .

`IP4_INTERFACE` The structure is located in NetworkPkg\Ip4Dxe\Ip4Common.h and contains the following members:

c AI generated projects 登录复制 run

```c
//
// Each IP4 instance has its own station address. All the instances
// with the same station address share a single interface structure.
// Each interface has its own ARP child, and shares one MNP child.
// Notice the special cases that DHCP can configure the interface
// with 0.0.0.0/0.0.0.0.
//
struct _IP4_INTERFACE {
  UINT32                        Signature;
  LIST_ENTRY                    Link;
  INTN                          RefCnt;

  //
  // IP address and subnet mask of the interface. It also contains
  // the subnet/net broadcast address for quick access. The fields
  // are invalid if (Configured == FALSE)
  //
  IP4_ADDR                      Ip;
  IP4_ADDR                      SubnetMask;
  IP4_ADDR                      SubnetBrdcast;
  IP4_ADDR                      NetBrdcast;
  BOOLEAN                       Configured;

  //
  // Handle used to create/destroy ARP child. All the IP children
  // share one MNP which is owned by IP service binding.
  //
  EFI_HANDLE                    Controller;
  EFI_HANDLE                    Image;

  EFI_MANAGED_NETWORK_PROTOCOL  *Mnp;
  EFI_ARP_PROTOCOL              *Arp;
  EFI_HANDLE                    ArpHandle;

  //
  // Queues to keep the frames sent and waiting ARP request.
  //
```

```c
  LIST_ENTRY                    ArpQues;
  LIST_ENTRY                    SentFrames;
  IP4_LINK_RX_TOKEN             *RecvRequest;

  //
  // The interface's MAC and broadcast MAC address.
  //
  EFI_MAC_ADDRESS               Mac;
  EFI_MAC_ADDRESS               BroadcastMac;
  UINT32                        HwaddrLen;

  //
  // All the IP instances that have the same IP/SubnetMask are linked
  // together through IpInstances. If any of the instance enables
  // promiscuous receive, PromiscRecv is true.
  //
  LIST_ENTRY                    IpInstances;
  BOOLEAN                       PromiscRecv;
};
```

Its members include several parts:

- `Ip` , `SubnetMask` , `SubnetBrdcast` , `NetBrdcast` : common IP addresses.
- `Configured` : `Ip4SetAddres()` Becomes TRUE after executing Set IP.
- `Mnp` , `Arp` : The underlying network interface.
- `Mac` , `BroadcastMac` , `HwaddrLen` : MAC address data, all of which come from SNP :

```c
  CopyMem (&Interface->Mac, &SnpMode.CurrentAddress, sizeof (Interface->Mac));
  CopyMem (&Interface->BroadcastMac, &SnpMode.BroadcastAddress, sizeof (Interface->BroadcastMac));
  Interface->HwaddrLen = SnpMode.HwAddressSize;
```

- `ArpQues` : The corresponding `IP4_ARP_QUE` linked list is used to process ARP data.
- `SentFrames` : The corresponding `IP4_LINK_TX_TOKEN` linked list will be used when processing the sent data.
- `RecvRequest` : A Token, which is used by IP4 to process the received data. Its initialization code is:

```c
EFI_STATUS
Ip4ReceiveFrame (
  IN  IP4_INTERFACE       *Interface,
  IN  IP4_PROTOCOL        *IpInstance       OPTIONAL,
  IN  IP4_FRAME_CALLBACK  CallBack,
  IN  VOID                *Context
  )
{
  Token = Ip4CreateLinkRxToken (Interface, IpInstance, CallBack, Context);  // 构建Token

  Interface->RecvRequest = Token;
  Status                 = Interface->Mnp->Receive (Interface->Mnp, &Token->MnpToken);
}
```

`Ip4CreateLinkRxToken()` Implementation:

```c
IP4_LINK_RX_TOKEN *
Ip4CreateLinkRxToken (
  IN IP4_INTERFACE       *Interface,
  IN IP4_PROTOCOL        *IpInstance,
  IN IP4_FRAME_CALLBACK  CallBack,
  IN VOID                *Context
  )
{
  Token = AllocatePool (sizeof (IP4_LINK_RX_TOKEN));

  Token->Signature  = IP4_FRAME_RX_SIGNATURE;
  Token->Interface  = Interface;
  Token->IpInstance = IpInstance;
  Token->CallBack   = CallBack;
  Token->Context    = Context;

  MnpToken         = &Token->MnpToken;
  MnpToken->Status = EFI_NOT_READY;

  Status = gBS->CreateEvent (
                  EVT_NOTIFY_SIGNAL,
                  TPL_NOTIFY,
                  Ip4OnFrameReceived,
                  Token,
                  &MnpToken->Event
                  );

  MnpToken->Packet.RxData = NULL;
}
```

`Ip4OnFrameReceived()` It is the callback function for receiving data, and its implementation is also a DPC process:

```c
VOID
EFIAPI
Ip4OnFrameReceived (
  IN EFI_EVENT  Event,
  IN VOID       *Context
  )
{
  //
  // Request Ip4OnFrameReceivedDpc as a DPC at TPL_CALLBACK
  //
  QueueDpc (TPL_CALLBACK, Ip4OnFrameReceivedDpc, Context);
}
```

`Ip4OnFrameReceivedDpc()` The most important part of the processing code is that `Token->CallBack` it comes from code similar to the following:

```c
Ip4ReceiveFrame (IpIf, NULL, Ip4AccpetFrame, IpSb);
```

This will be further described in Ip4AccpetFrame .

- `IpInstances`： The corresponding `IP4_PROTOCOL` linked list. `IP4_INTERFACE` After initialization, it is just a network interface containing IP addresses, but it does not have the ability to send and receive data. It needs cooperation `IP4_PROTOCOL` to complete the sending and receiving of data. At this time, the data contains `IP4_INTERFACE` information such as the IP address in the network, forming an IP packet.

`IP4_INTERFACE` It is equivalent to an external network interface of an IP4 sub-item, through which IP packets can be sent and received.

IP4_CONFIG2_INSTANCE

`IP4_SERVICE` The type of a member in `Ip4Config2Instance` is `IP4_CONFIG2_INSTANCE` one `IP4_SERVICE` -to-one `IP4_CONFIG2_INSTANCE` , and its main function is to configure information such as IP address. It includes the real processing interface `EFI_IP4_CONFIG2_PROTOCOL` and related data. Therefore, in order to complete the real IP configuration, it is necessary to rely on this structure.

`IP4_SERVICE` The function will be executed during initialization `Ip4CreateService()` , which initializes `IP4_CONFIG2_INSTANCE` the structure:

c AI generated projects 登录复制 run

```c
EFI_STATUS
Ip4CreateService (
  IN  EFI_HANDLE   Controller,
  IN  EFI_HANDLE   ImageHandle,
  OUT IP4_SERVICE  **Service
  )
{
  ZeroMem (&IpSb->Ip4Config2Instance, sizeof (IP4_CONFIG2_INSTANCE));

  Status = Ip4Config2InitInstance (&IpSb->Ip4Config2Instance);
```

`IP4_CONFIG2_INSTANCE` The structure is located in NetworkPkg\Ip4Dxe\Ip4Config2Impl.h, and its members are as follows:

c AI generated projects 登录复制 run

```c
struct _IP4_CONFIG2_INSTANCE {
  UINT32                          Signature;
  BOOLEAN                         Configured;
  LIST_ENTRY                      Link;
  UINT16                          IfIndex;

  EFI_IP4_CONFIG2_PROTOCOL        Ip4Config2;

  EFI_IP4_CONFIG2_INTERFACE_INFO  InterfaceInfo;
  EFI_IP4_CONFIG2_POLICY          Policy;
  IP4_CONFIG2_DATA_ITEM           DataItem[Ip4Config2DataTypeMaximum];

  EFI_EVENT                       Dhcp4SbNotifyEvent;
  VOID                            *Registration;
  EFI_HANDLE                      Dhcp4Handle;
  EFI_DHCP4_PROTOCOL              *Dhcp4;
  BOOLEAN                         DhcpSuccess;
  BOOLEAN                         OtherInfoOnly;
  EFI_EVENT                       Dhcp4Event;
  UINT32                          FailedIaAddressCount;
twen EFI_IPv4_ADDRESS             *DeclineAddress;
twen UINT32                       DeclineAddressCount;

twen IP4_FORM_CALLBACK_INFO       CallbackInfo;

  IP4_CONFIG2_NVDATA              Ip4NvData;
};
```

There is a lot of content, so here we will only explain the more important ones:

- `Configured Ip4Config2InitInstance()` : After executing the initialization function, it becomes `TRUE` .

c AI generated projects 登录复制 run

```c
EFI_STATUS
Ip4Config2InitInstance (
  OUT IP4_CONFIG2_INSTANCE  *Instance
  )
{
  Instance->Configured = TRUE;
}
```

- `Link` , `IfIndex` : A network card may have multiple network ports, so there are `IfIndex` , and `Link` what is connected is `IP4_CONFIG2_INSTANCE` the instance, it will `IfIndex` have multiple, these instances `mIp4Config2InstanceList` are connected through. From the current implementation, there should be only 1, so `IfIndex` the value of is always 0.
- `Ip4Config2`： `EFI_IP4_CONFIG2_PROTOCOL` Example, which implements:

c AI generated projects 登录复制 run

```c
  Instance->Ip4Config2.SetData             = EfiIp4Config2SetData;
  Instance->Ip4Config2.GetData             = EfiIp4Config2GetData;
  Instance->Ip4Config2.RegisterDataNotify  = EfiIp4Config2RegisterDataNotify;
  Instance->Ip4Config2.UnregisterDataNotify = EfiIp4Config2UnregisterDataNotify;
```

- `InterfaceInfo` : Network configuration parameters:

c AI generated projects 登录复制 run

```c
///
/// EFI_IP4_CONFIG2_INTERFACE_INFO
///
typedef struct {
  ///
  /// The name of the interface. It is a NULL-terminated Unicode string.
  ///
  CHAR16              Name[EFI_IP4_CONFIG2_INTERFACE_INFO_NAME_SIZE];
  ///
  /// The interface type of the network interface. See RFC 1700,
  /// section "Number Hardware Type".
  ///
  UINT8               IfType;
  ///
  /// The size, in bytes, of the network interface's hardware address.
  ///
  UINT32              HwAddressSize;
  ///
  /// The hardware address for the network interface.
  ///
twen EFI_MAC_ADDRESS     HwAddress;
twen ///
twen /// The station IPv4 address of this EFI IPv4 network stack.
twen ///
  EFI_IPv4_ADDRESS    StationAddress;
  ///
```

```c
  /// The subnet address mask that is associated with the station address.
  ///
  EFI_IPv4_ADDRESS    SubnetMask;
  ///
  /// Size of the following RouteTable, in bytes. May be zero.
  ///
  UINT32              RouteTableSize;
  ///
  /// The route table of the IPv4 network stack runs on this interface.
  /// Set to NULL if RouteTableSize is zero. Type EFI_IP4_ROUTE_TABLE is defined in
  /// EFI_IP4_PROTOCOL.GetModeData().
  ///
  EFI_IP4_ROUTE_TABLE   *RouteTable     OPTIONAL;
} EFI_IP4_CONFIG2_INTERFACE_INFO;
```

- `Policy` : Network configuration methods include static and dynamic:

```c
///
/// EFI_IP4_CONFIG2_POLICY
///
typedef enum {
  ///
  /// Under this policy, the Ip4Config2DataTypeManualAddress,
  /// Ip4Config2DataTypeGateway and Ip4Config2DataTypeDnsServer configuration
  /// data are required to be set manually. The EFI IPv4 Protocol will get all
  /// required configuration such as IPv4 address, subnet mask and
  /// gateway settings from the EFI IPv4 Configuration II protocol.
  ///
  Ip4Config2PolicyStatic,
  ///
  /// Under this policy, the Ip4Config2DataTypeManualAddress,
  /// Ip4Config2DataTypeGateway and Ip4Config2DataTypeDnsServer configuration data are
  /// not allowed to set via SetData(). All of these configurations are retrieved from DHCP
  /// server or other auto-configuration mechanism.
  ///
  Ip4Config2PolicyDhcp,
  Ip4Config2PolicyMax
} EFI_IP4_CONFIG2_POLICY;
```

- `DataItem` : An array containing network parameter operations. Each array member represents a set of network parameter operations. The operations are:

```c
///
/// EFI_IP4_CONFIG2_DATA_TYPE
///
typedef enum {
  ///
  /// The interface information of the communication device this EFI
  /// IPv4 Configuration II Protocol instance manages. This type of
  /// data is read only. The corresponding Data is of type
  /// EFI_IP4_CONFIG2_INTERFACE_INFO.
  ///
  Ip4Config2DataTypeInterfaceInfo,
  ///
  /// The general configuration policy for the EFI IPv4 network stack
  /// running on the communication device this EFI IPv4
  /// Configuration II Protocol instance manages. The policy will
  /// affect other configuration settings. The corresponding Data is of
  /// type EFI_IP4_CONFIG2_POLICY.
  ///
  Ip4Config2DataTypePolicy,
  ///
  /// The station addresses set manually for the EFI IPv4 network
  /// stack. It is only configurable when the policy is
  /// Ip4Config2PolicyStatic. The corresponding Data is of
  /// type EFI_IP4_CONFIG2_MANUAL_ADDRESS. When DataSize
  /// is 0 and Data is NULL, the existing configuration is cleared
  /// from the EFI IPv4 Configuration II Protocol instance.
  ///
  Ip4Config2DataTypeManualAddress,
  ///
  /// The gateway addresses set manually for the EFI IPv4 network
  /// stack running on the communication device this EFI IPv4
  /// Configuration II Protocol manages. It is not configurable when
  /// the policy is Ip4Config2PolicyDhcp. The gateway
  /// addresses must be unicast IPv4 addresses. The corresponding
  /// Data is a pointer to an array of EFI_IPv4_ADDRESS instances.
  /// When DataSize is 0 and Data is NULL, the existing configuration
  /// is cleared from the EFI IPv4 Configuration II Protocol instance.
  ///
  Ip4Config2DataTypeGateway,
  ///
  /// The DNS server list for the EFI IPv4 network stack running on
  /// the communication device this EFI IPv4 Configuration II
  /// Protocol manages. It is not configurable when the policy is
  /// Ip4Config2PolicyDhcp. The DNS server addresses must be
  /// unicast IPv4 addresses. The corresponding Data is a pointer to
  /// an array of EFI_IPv4_ADDRESS instances. When DataSize
  /// is 0 and Data is NULL, the existing configuration is cleared
  /// from the EFI IPv4 Configuration II Protocol instance.
  ///
  Ip4Config2DataTypeDnsServer,
  Ip4Config2DataTypeMaximum
} EFI_IP4_CONFIG2_DATA_TYPE;
```

So it is actually an array `DataItem[Ip4Config2DataTypeMaximum]` that contains most of the operation functions needed for configuration.

- `Dhcp4SbNotifyEvent` , `Registration` , `Dhcp4Handle` , `Dhcp4` , `Dhcp4Event` : DHCP4 related parameters. DHCP4 has not been installed at this time, so this is implemented through callback.
- `DhcpSuccess` : After DHCP4 `TRUE` is initialized successfully, it is set to , from the function `Ip4Config2OnDhcp4Complete()` , which is `Dhcp4Event` the callback function of the event.
- `OtherInfoOnly` , `FailedIaAddressCount` , `DeclineAddress` , `DeclineAddressCount` : Currently unused, they have an IPv6 version that will be used.
- `CallbackInfo` : Callback function for UI operations.
- `Ip4NvData` : Network parameters:

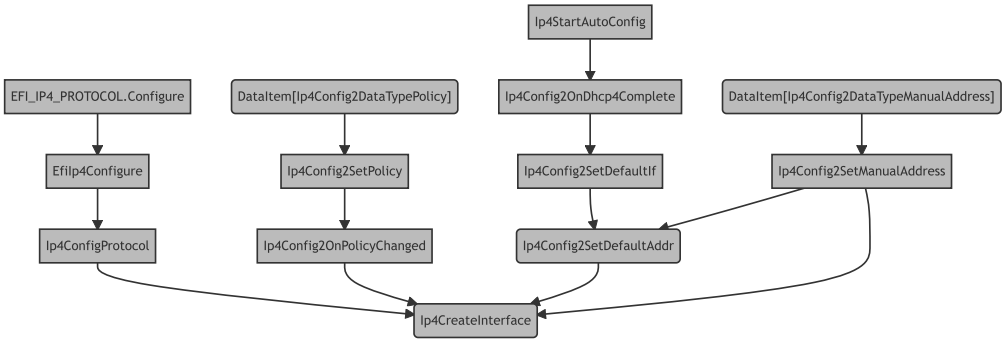
```c
typedef struct {
  EFI_IP4_CONFIG2_POLICY           Policy;                 ///< manual or automatic
  EFI_IP4_CONFIG2_MANUAL_ADDRESS   *ManualAddress;         ///< IP addresses
  UINT32                           ManualAddressCount;     ///< IP addresses count
```

```
  EFI_IPv4_ADDRESS                *GatewayAddress;            ///< Gateway address
  UINT32                          GatewayAddressCount;        ///< Gateway address count
  EFI_IPv4_ADDRESS                *DnsAddress;                ///< DNS server address
  UINT32                          DnsAddressCount;            ///< DNS server address count
} IP4_CONFIG2_NVDATA;
```

IP Configuration

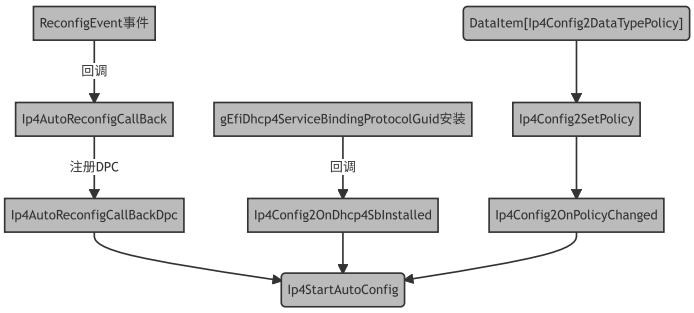
IP configuration involves several important functions, which have been mentioned before and will be further explained here.

First `Ip4CreateInterface()`, it creates `IP4_INTERFACE` the structure, which is the main body of the configuration IP. Its calling process is as follows:



The above-mentioned calling `Ip4CreateInterface()` codes can be divided into three categories, corresponding to:

- `Configure` The member functions of EFI_IP4_PROTOCOL will be further introduced later.
- DHCP configuration includes usage `DataItem[Ip4Config2DataTypePolicy]` and `Ip4StartAutoConfig()` functions. The former actually includes the latter, which can be seen from the latter's call:



The first call here, `ReconfigEvent` event, will be created in Ip4TimerReconfigChecking, which is a timer event used to detect whether the network connection has changed. If so, and the configuration policy happens to be DHCP, IP reconfiguration may be required.

The second call here determines whether the DHCP4 module is installed. However, this is actually in `Ip4StartAutoConfig()` the function. This is because the relationship between the execution of IP4 and DHCP4 is uncertain. There is a situation where the IP4 module has been executed but the DHCP4 module has not been executed. Therefore, this is the reason for the following processing:

c AI generated projects 登录复制 run

```c
VOID
EFIAPI
Ip4Config2OnDhcp4SbInstalled (
  IN EFI_EVENT  Event,
  IN VOID       *Context
  )
{
  IP4_CONFIG2_INSTANCE  *Instance;

  Instance = (IP4_CONFIG2_INSTANCE *)Context;

  if ((Instance->Dhcp4Handle != NULL) || (Instance->Policy != Ip4Config2PolicyDhcp)) {
    //
    // The DHCP4 child is already created or the policy is no longer DHCP.
    //
    return;
  }

  Ip4StartAutoConfig (Instance);
}
twen
twen EFI_STATUS
twen Ip4StartAutoConfig (
twen   IN IP4_CONFIG2_INSTANCE  *Instance
  )
{
  Status = NetLibCreateServiceChild (
             IpSb->Controller,
             IpSb->Image,
             &gEfiDhcp4ServiceBindingProtocolGuid,
             &Instance->Dhcp4Handle
             );

  if (Status == EFI_UNSUPPORTED) {
    //
    // No DHCPv4 Service Binding protocol, register a notify.
    //
    if (Instance->Dhcp4SbNotifyEvent == NULL) {
      Instance->Dhcp4SbNotifyEvent = EfiCreateProtocolNotifyEvent (
                                       &gEfiDhcp4ServiceBindingProtocolGuid,
                                       TPL_CALLBACK,
                                       Ip4Config2OnDhcp4SbInstalled, // 该函数中还是执行了Ip4StartAutoConfig
                                       (VOID *)Instance,
                                       &Instance->Registration
                                       );
    }
  }
}
```

The third call here uses IP4_CONFIG2_INSTANCE, `DataItem` which has appeared before. `DataItem[Ip4Config2DataTypePolicy]` The corresponding function `Ip4Config2SetPolicy()` contains the following code:

```c
  //
  // Start the dhcp configuration.
  //
  if (NewPolicy == Ip4Config2PolicyDhcp) {
    Ip4StartAutoConfig (&IpSb->Ip4Config2Instance);
  }
```

That is, if the IP configuration policy becomes HDCP, `Ip4StartAutoConfig()` the function needs to be executed.

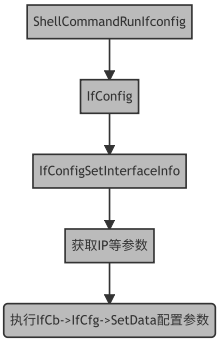
- Manual configuration, use `DataItem[Ip4Config2DataTypeManualAddress]` .

IP Configuration Example

IP needs to be configured before it can take effect. For example, use ifconfig to configure it in Shell:

```bash
ifconfig -s eth0 static 192.168.3.128 255.255.255.0 192.168.3.1
```

It will execute the code in ShellPkg\Library\UefiShellNetwork1CommandsLib\Ifconfig.c, and the process is as follows:



The `IfCfg` corresponding member function here is : `EFI_IP4_CONFIG2_PROTOCOL SetData EfiIp4Config2SetData()`

```c
EFI_STATUS
EFIAPI
EfiIp4Config2SetData (
  IN EFI_IP4_CONFIG2_PROTOCOL   *This,
  IN EFI_IP4_CONFIG2_DATA_TYPE  DataType,
  IN UINTN                      DataSize,
  IN VOID                       *Data
  )
{
  Status = Instance->DataItem[DataType].Status;
  if (Status != EFI_NOT_READY) {
    if (Instance->DataItem[DataType].SetData == NULL) {
      //
      // This type of data is readonly.
      //
      Status = EFI_WRITE_PROTECTED;
    } else {
      Status = Instance->DataItem[DataType].SetData (Instance, DataSize, Data);
      if (!EFI_ERROR (Status)) {
        //
        // Fire up the events registered with this type of data.
        //
        NetMapIterate (&Instance->DataItem[DataType].EventMap, Ip4Config2SignalEvent, NULL);
        Ip4Config2WriteConfigData (IpSb->MacString, Instance);
      } else if (Status == EFI_ABORTED) {
        //
        // The SetData is aborted because the data to set is the same with
        // the one maintained.
        //
        Status = EFI_SUCCESS;
        NetMapIterate (&Instance->DataItem[DataType].EventMap, Ip4Config2SignalEvent, NULL);
      }
    }
  } else {
    //
    // Another asynchronous process is on the way.
    //
    Status = EFI_ACCESS_DENIED;
  }
}
```

IP4_CONFIG2_INSTANCE is used here `DataItem` , whether it is manual or DHCP.

IP4_PROTOCOL

`IP4_PROTOCOL` It is the data structure used by the IP4 sub-item. Note the difference with EFI_IP4_PROTOCOL , that is an interface, while this is data. The two are actually used together, `IP4_PROTOCOL` including `EFI_IP4_PROTOCOL` this member. It is `Ip4ServiceBindingCreateChild()` initialized in, which is basically the same as the service interface of other UEFI network protocols:

```c
EFI_STATUS
EFIAPI
Ip4ServiceBindingCreateChild (
  IN EFI_SERVICE_BINDING_PROTOCOL  *This,
  IN OUT EFI_HANDLE                *ChildHandle
  )
{
  IpSb       = IP4_SERVICE_FROM_PROTOCOL (This);
  IpInstance = AllocatePool (sizeof (IP4_PROTOCOL));

  Ip4InitProtocol (IpSb, IpInstance);

  //
  // Install Ip4 onto ChildHandle
  //
  Status = gBS->InstallMultipleProtocolInterfaces (
                  ChildHandle,
                  &gEfiIp4ProtocolGuid,
                  &IpInstance->Ip4Proto,
                  NULL
```

```
                );

  IpInstance->Handle = *ChildHandle;

  //
  // Open the Managed Network protocol BY_CHILD.
  //
  Status = gBS->OpenProtocol (
                IpSb->MnpChildHandle,
                &gEfiManagedNetworkProtocolGuid,
                (VOID **)&Mnp,
                gIp4DriverBinding.DriverBindingHandle,
                IpInstance->Handle,
                EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER
                );

  InsertTailList (&IpSb->Children, &IpInstance->Link);
  IpSb->NumChildren++;
}
```

The structure is located in NetworkPkg\Ip4Dxe\Ip4Impl.h:

c AI generated projects 登录复制 run

```c
struct _IP4_PROTOCOL {
  UINT32                Signature;

  EFI_IP4_PROTOCOL      Ip4Proto;
  EFI_HANDLE            Handle;
  INTN                  State;

  BOOLEAN               InDestroy;

  IP4_SERVICE           *Service;
  LIST_ENTRY            Link;          // Link to all the IP protocol from the service

  //
  // User's transmit/receive tokens, and received/delivered packets
  //
  NET_MAP               RxTokens;
  NET_MAP               TxTokens;      // map between (User's Token, IP4_TXTOKE_WRAP)
  LIST_ENTRY            Received;      // Received but not delivered packet
  LIST_ENTRY            Delivered;     // Delivered and to be recycled packets
  EFI_LOCK              RecycleLock;

  //
  // Instance's address and route tables. There are two route tables.
  // RouteTable is used by the IP4 driver to route packet. EfiRouteTable
  // is used to communicate the current route info to the upper layer.
  //
  IP4_INTERFACE         *Interface;
  LIST_ENTRY            AddrLink;      // Ip instances with the same IP address.
  IP4_ROUTE_TABLE       *RouteTable;

  EFI_IP4_ROUTE_TABLE   *EfiRouteTable;
  UINT32                EfiRouteCount;

  //
  // IGMP data for this instance
  //
  IP4_ADDR              *Groups;       // stored in network byte order
  UINT32                GroupCount;

  EFI_IP4_CONFIG_DATA   ConfigData;
};
```

Here are the more important values:

- `Ip4Proto` : Corresponding `EFI_IP4_PROTOCOL` interface.
- `Handle` `EFI_IP4_PROTOCOL` : The corresponding Handle of IP4 sub-item installation .
- `InDestroy` : Mainly `Ip4ServiceBindingDestroyChild()` used in to prevent reentry, because data may arrive at any time, if data appears when it is about an IP4 sub-item, problems may occur, so it is needed to control it.
- `Service` : Point to `IP4_SERVICE` .
- `Link` : After the structure `Ip4ServiceBindingCreateChild()` is created , it will be placed in the linked list through this member. `IP4_PROTOCOL` `IP4_SERVICE` `Children`
- `RxTokens` , `TxTokens` , `Received` , `Delivered` : structures for data processing.
- `Interface` : Points to `IP4_INTERFACE` an instance.
- `AddrLink` : A linked list with the same IP configuration `IP4_INTERFACE` .
- `RouteTable` , `EfiRouteTable` : Parameters related to the routing table.
- `Groups` : Point to `IP4_ADDR` .
- `ConfigData` : Configuration parameters, its structure is as follows:

c AI generated projects 登录复制 run

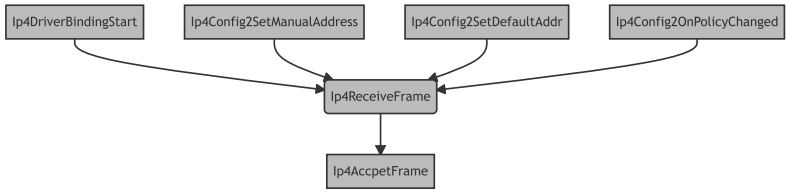
```c
typedef struct {
  ///
  /// The default IPv4 protocol packets to send and receive. Ignored
  /// when AcceptPromiscuous is TRUE.
  ///
  UINT8               DefaultProtocol;
  ///
  /// Set to TRUE to receive all IPv4 packets that get through the receive filters.
  /// Set to FALSE to receive only the DefaultProtocol IPv4
  /// packets that get through the receive filters.
  ///
  BOOLEAN             AcceptAnyProtocol;
  ///
  /// Set to TRUE to receive ICMP error report packets. Ignored when
  /// AcceptPromiscuous or AcceptAnyProtocol is TRUE.
  ///
  BOOLEAN             AcceptIcmpErrors;
  ///
  /// Set to TRUE to receive broadcast IPv4 packets. Ignored when
  /// AcceptPromiscuous is TRUE.
  /// Set to FALSE to stop receiving broadcast IPv4 packets.
  ///
  BOOLEAN             AcceptBroadcast;
  ///
  /// Set to TRUE to receive all IPv4 packets that are sent to any
  /// hardware address or any protocol address.
  /// Set to FALSE to stop receiving all promiscuous IPv4 packets
  ///
```

```
  BOOLEAN             AcceptPromiscuous;
  ///
  /// Set to TRUE to use the default IPv4 address and default routing table.
  ///
  BOOLEAN             UseDefaultAddress;
  ///
  /// The station IPv4 address that will be assigned to this EFI IPv4Protocol instance.
  ///
  EFI_IPv4_ADDRESS    StationAddress;
  ///
  /// The subnet address mask that is associated with the station address.
  ///
  EFI_IPv4_ADDRESS    SubnetMask;
  ///
  /// TypeOfService field in transmitted IPv4 packets.
  ///
  UINT8               TypeOfService;
  ///
  /// TimeToLive field in transmitted IPv4 packets.
  ///
  UINT8               TimeToLive;
  ///
  /// State of the DoNotFragment bit in transmitted IPv4 packets.
  ///
  BOOLEAN             DoNotFragment;
  ///
  /// Set to TRUE to send and receive unformatted packets. The other
  /// IPv4 receive filters are still applied. Fragmentation is disabled for RawData mode.
  ///
  BOOLEAN             RawData;
  ///
  /// The timer timeout value (number of microseconds) for the
  /// receive timeout event to be associated with each assembled
  /// packet. Zero means do not drop assembled packets.
  ///
  UINT32              ReceiveTimeout;
  ///
  /// The timer timeout value (number of microseconds) for the
  /// transmit timeout event to be associated with each outgoing
  /// packet. Zero means do not drop outgoing packets.
  ///
  UINT32              TransmitTimeout;
} EFI_IP4_CONFIG_DATA;
```

These are parameters related to IP4 data transmission and reception. These parameters are configured by members `EFI_IP4_PROTOCOL.Configure`

Ip4AccpetFrame

`Ip4AccpetFrame()` It is `Ip4ReceiveFrame()` used together with , and the calling process is as follows:



`Ip4ReceiveFrame()` `Ip4AccpetFrame()` There are several situations where it is used in the code:

c AI generated projects 登录复制 run

```c
// 位于Ip4Config2OnPolicyChanged()/Ip4Config2SetDefaultAddr()/Ip4Config2SetManualAddress():
Ip4ReceiveFrame (IpIf, NULL, Ip4AccpetFrame, IpSb);

//
// 位于Ip4DriverBindingStart()（Ip4DriverBindingStop()中的不关注）：
// Ready to go: start the receiving and timer.
// Ip4Config2SetPolicy maybe call Ip4ReceiveFrame() to set the default interface's RecvRequest first after
// Ip4Config2 instance is initialized. So, EFI_ALREADY_STARTED is the allowed return status.
//
Status = Ip4ReceiveFrame (IpSb->DefaultInterface, NULL, Ip4AccpetFrame, IpSb);
```

First of all, we need to introduce `Ip4ReceiveFrame()` that this function is located in NetworkPkg\Ip4Dxe\Ip4If.c:

c AI generated projects 登录复制 run

```c
EFI_STATUS
Ip4ReceiveFrame (
  IN  IP4_INTERFACE       *Interface,
  IN  IP4_PROTOCOL        *IpInstance       OPTIONAL,
  IN  IP4_FRAME_CALLBACK  CallBack,
  IN  VOID                *Context
  )
{
  Token = Ip4CreateLinkRxToken (Interface, IpInstance, CallBack, Context);

  Interface->RecvRequest = Token;
  Status                 = Interface->Mnp->Receive (Interface->Mnp, &Token->MnpToken);
}
```

Here `CallBack` is the member `Ip4AccpetFrame()` corresponding to `IP4_LINK_RX_TOKEN` this Token `CallBack`. Token comes from the function `Ip4CreateLinkRxToken()`:

c AI generated projects 登录复制 run

```c
IP4_LINK_RX_TOKEN *
Ip4CreateLinkRxToken (
  IN IP4_INTERFACE       *Interface,
  IN IP4_PROTOCOL        *IpInstance,
  IN IP4_FRAME_CALLBACK  CallBack,
  IN VOID                *Context
  )
{
  Token = AllocatePool (sizeof (IP4_LINK_RX_TOKEN));

  Token->Signature  = IP4_FRAME_RX_SIGNATURE;
  Token->Interface  = Interface;
  Token->IpInstance = IpInstance;
  Token->CallBack   = CallBack;
  Token->Context    = Context;

  MnpToken         = &Token->MnpToken;
  MnpToken->Status = EFI_NOT_READY;

```

```c
  Status = gBS->CreateEvent (
                  EVT_NOTIFY_SIGNAL,
                  TPL_NOTIFY,
                  Ip4OnFrameReceived,
                  Token,
                  &MnpToken->Event
                  );

  MnpToken->Packet.RxData = NULL;
}
```

So when MNP receives the data, it will execute `Ip4OnFrameReceived()`:

c AI generated projects 登录复制 run

```c
VOID
EFIAPI
Ip4OnFrameReceived (
  IN EFI_EVENT  Event,
  IN VOID       *Context
  )
{
  //
  // Request Ip4OnFrameReceivedDpc as a DPC at TPL_CALLBACK
  //
  QueueDpc (TPL_CALLBACK, Ip4OnFrameReceivedDpc, Context);
}
```

It registers the DPC function `Ip4OnFrameReceivedDpc()`:

c AI generated projects 登录复制 run

```c
VOID
EFIAPI
Ip4OnFrameReceivedDpc (
  IN VOID  *Context
  )
{
  EFI_MANAGED_NETWORK_COMPLETION_TOKEN  *MnpToken;
  EFI_MANAGED_NETWORK_RECEIVE_DATA      *MnpRxData;
  IP4_LINK_RX_TOKEN                     *Token;
  NET_FRAGMENT                          Netfrag;
  NET_BUF                               *Packet;
  UINT32                                Flag;

  Token = (IP4_LINK_RX_TOKEN *)Context;
  NET_CHECK_SIGNATURE (Token, IP4_FRAME_RX_SIGNATURE);

  //
  // First clear the interface's receive request in case the
  // caller wants to call Ip4ReceiveFrame in the callback.
  //
  Token->Interface->RecvRequest = NULL;

  MnpToken  = &Token->MnpToken;
  MnpRxData = MnpToken->Packet.RxData;

  if (EFI_ERROR (MnpToken->Status) || (MnpRxData == NULL)) {
    Token->CallBack (Token->IpInstance, NULL, MnpToken->Status, 0, Token->Context);
    Ip4FreeFrameRxToken (Token);

    return;
  }

  //
  // Wrap the frame in a net buffer then deliver it to IP input.
  // IP will reassemble the packet, and deliver it to upper layer
  //
  Netfrag.Len  = MnpRxData->DataLength;
  Netfrag.Bulk = MnpRxData->PacketData;

  Packet = NetbufFromExt (&Netfrag, 1, 0, IP4_MAX_HEADLEN, Ip4RecycleFrame, Token);

  if (Packet == NULL) {
    gBS->SignalEvent (MnpRxData->RecycleEvent);

    Token->CallBack (Token->IpInstance, NULL, EFI_OUT_OF_RESOURCES, 0, Token->Context);
    Ip4FreeFrameRxToken (Token);

    return;
  }

  Flag  = (MnpRxData->BroadcastFlag ? IP4_LINK_BROADCAST : 0);
  Flag |= (MnpRxData->MulticastFlag ? IP4_LINK_MULTICAST : 0);
  Flag |= (MnpRxData->PromiscuousFlag ? IP4_LINK_PROMISC : 0);

  Token->CallBack (Token->IpInstance, Packet, EFI_SUCCESS, Flag, Token->Context);
}
```

The final call `Token->CallBack()` is actually a call `Ip4AccpetFrame()`. The implementation of this function is in NetworkPkg\Ip4Dxe\Ip4Input.c. The code is as follows:

c AI generated projects 登录复制 run

```c
VOID
Ip4AccpetFrame (
  IN IP4_PROTOCOL  *Ip4Instance,
  IN NET_BUF       *Packet,
  IN EFI_STATUS    IoStatus,
  IN UINT32        Flag,
  IN VOID          *Context
  )
{
  // 参数判断
  if (EFI_ERROR (IoStatus) || (IpSb->State == IP4_SERVICE_DESTROY)) {
    goto DROP;
  }
  if (!Ip4IsValidPacketLength (Packet)) {
    goto RESTART;
  }

  Head = (IP4_HEAD *)NetbufGetByte (Packet, 0, NULL);
  ASSERT (Head != NULL);
  OptionLen = (Head->HeadLen << 2) - IP4_MIN_HEADLEN;
  if (OptionLen > 0) {
    Option = (UINT8 *)(Head + 1);
```

```
  }

  //
  // Validate packet format and reassemble packet if it is necessary.
  //
  Status = Ip4PreProcessPacket (
             IpSb,
             &Packet,
             Head,
             Option,
             OptionLen,
             Flag
             );

  if (EFI_ERROR (Status)) {
    goto RESTART;
  }

  //
  // After trim off, the packet is a esp/ah/udp/tcp/icmp6 net buffer,
  // and no need consider any other ahead ext headers.
  //
  Status = Ip4IpSecProcessPacket (
             IpSb,
             &Head,
             &Packet,
             &Option,
             &OptionLen,
             EfiIPsecInBound,
             NULL
             );

  if (EFI_ERROR (Status)) {
    goto RESTART;
  }

  //
  // If the packet is protected by tunnel mode, parse the inner Ip Packet.
  //
  ZeroMem (&ZeroHead, sizeof (IP4_HEAD));
  if (0 == CompareMem (Head, &ZeroHead, sizeof (IP4_HEAD))) {
    // Packet may have been changed. Head, HeadLen, TotalLen, and
    // info must be reloaded before use. The ownership of the packet
    // is transferred to the packet process logic.
    //
    if (!Ip4IsValidPacketLength (Packet)) {
      goto RESTART;
    }

    Head = (IP4_HEAD *)NetbufGetByte (Packet, 0, NULL);
    ASSERT (Head != NULL);
    Status = Ip4PreProcessPacket (
               IpSb,
               &Packet,
               Head,
               Option,
               OptionLen,
               Flag
               );
    if (EFI_ERROR (Status)) {
      goto RESTART;
    }
  }

  ASSERT (Packet != NULL);
  Head                              = Packet->Ip.Ip4;
  IP4_GET_CLIP_INFO (Packet)->Status = EFI_SUCCESS;

  switch (Head->Protocol) {
    case EFI_IP_PROTO_ICMP:
      Ip4IcmpHandle (IpSb, Head, Packet);
      break;

    case IP4_PROTO_IGMP:
      Ip4IgmpHandle (IpSb, Head, Packet);
      break;

    default:
      Ip4Demultiplex (IpSb, Head, Packet, Option, OptionLen);
  }

  Packet = NULL;

  //
  // Dispatch the DPCs queued by the NotifyFunction of the rx token's events
  // which are signaled with received data.
  //
  DispatchDpc ();

RESTART:
  Ip4ReceiveFrame (IpSb->DefaultInterface, NULL, Ip4AccpetFrame, IpSb);

DROP:
  if (Packet != NULL) {
    NetbufFree (Packet);
  }

  return;
}
```

Input parameters `Packet` are the parameters that need to be processed, and data processing mainly goes through `Ip4PreProcessPacket()` and `Ip4IpSecProcessPacket()` after processing, it enters the actual processing stage:

c AI generated projects 登录复制 run

```c
  switch (Head->Protocol) {
    case EFI_IP_PROTO_ICMP:
      Ip4IcmpHandle (IpSb, Head, Packet);
      break;

    case IP4_PROTO_IGMP:
      Ip4IgmpHandle (IpSb, Head, Packet);
      break;

    default:


```

```
      Ip4Demultiplex (IpSb, Head, Packet, Option, OptionLen);
    }
```

It will handle:

- ICMP, Internet Control Message Protocol, Internet Control Message Protocol.
- IGMP, Internet Group Management Protocol, Internet Group Management Protocol.
- Other processing.

Note that in the code `RESTART`, execution starts again `Ip4ReceiveFrame`, so this is a cyclic process, indicating that IP4 starts processing the received data.

IP4 Events

Several major events of IP4 are `Ip4CreateService()` created in:

c AI generated projects 登录复制 run

```c
  Status = gBS->CreateEvent (
                 EVT_NOTIFY_SIGNAL | EVT_TIMER,
                 TPL_CALLBACK,
                 Ip4TimerTicking,
                 IpSb,
                 &IpSb->Timer
                 );
  Status = gBS->CreateEvent (
                 EVT_NOTIFY_SIGNAL | EVT_TIMER,
                 TPL_CALLBACK,
                 Ip4TimerReconfigChecking,
                 IpSb,
                 &IpSb->ReconfigCheckTimer
                 );
  Status = gBS->CreateEvent (
                 EVT_NOTIFY_SIGNAL,
                 TPL_NOTIFY,
                 Ip4AutoReconfigCallBack,
                 IpSb,
                 &IpSb->ReconfigEvent
twen             );
```

Timer

This is a timed event. From the comments, we can see that it is a heartbeat event. The corresponding callback function is `Ip4TimerTicking()`, which handles two parts, which can be seen from its implementation (located in NetworkPkg\Ip4Dxe\Ip4Impl.c):

c AI generated projects 登录复制 run

```c
/**
  This heart beat timer of IP4 service instance times out all of its IP4 children's
  received-but-not-delivered and transmitted-but-not-recycle packets, and provides
  time input for its IGMP protocol.

  @param[in]  Event                 The IP4 service instance's heart beat timer.
  @param[in]  Context               The IP4 service instance.

**/
VOID
EFIAPI
Ip4TimerTicking (
  IN EFI_EVENT  Event,
  IN VOID       *Context
  )
{
  Ip4PacketTimerTicking (IpSb);
  Ip4IgmpTicking (IpSb);
}
```

The function has been explained in the comments.

Ip4TimerReconfigChecking

This is a timed event, which is used to check whether the network connection is normal. The corresponding callback function is `Ip4TimerReconfigChecking()`:

c AI generated projects 登录复制 run

```c
/**
  This dedicated timer is used to poll underlying network media status. In case
  of cable swap or wireless network switch, a new round auto configuration will
  be initiated. The timer will signal the IP4 to run DHCP configuration again.
  IP4 driver will free old IP address related resource, such as route table and
  Interface, then initiate a DHCP process to acquire new IP, eventually create
  route table for new IP address.

  @param[in]  Event                 The IP4 service instance's heart beat timer.
  @param[in]  Context               The IP4 service instance.

**/
VOID
EFIAPI
Ip4TimerReconfigChecking (
  IN EFI_EVENT  Event,
  IN VOID       *Context
  )
{
  OldMediaPresent = IpSb->MediaPresent;
twen
twen  //
twen  // Get fresh mode data from MNP, since underlying media status may change.
twen  // Here, it needs to mention that the MediaPresent can also be checked even if
  // EFI_NOT_STARTED returned while this MNP child driver instance isn't configured.
  //
  Status = IpSb->Mnp->GetModeData (IpSb->Mnp, NULL, &SnpModeData);

  IpSb->MediaPresent = SnpModeData.MediaPresent;
  //
  // Media transimit Unpresent to Present means new link movement is detected.
  //
  if (!OldMediaPresent && IpSb->MediaPresent && (IpSb->Ip4Config2Instance.Policy == Ip4Config2PolicyDhcp)) {
    //
    // Signal the IP4 to run the dhcp configuration again. IP4 driver will free
    // old IP address related resource, such as route table and Interface, then
    // initiate a DHCP round to acquire new IP, eventually
    // create route table for new IP address.
    //
    if (IpSb->ReconfigEvent != NULL) {
      Status = gBS->SignalEvent (IpSb->ReconfigEvent);
```

```
      DispatchDpc ();
    }
  }
}
```

The key point is to determine whether DHCP needs to be re-activated based on the underlying layer `MediaPresent` (the premise is that the IP configuration strategy is DHCP, and static does not need to be concerned). Re-activating DHCP will also use events `ReconfigEvent`, which correspond to callback functions `Ip4AutoReconfigCallBack()`. This is the function that actually does DHCP, and is ultimately called `Ip4StartAutoConfig()`. This part is introduced in detail in IP configuration.

EFI_IP4_PROTOCOL

The structure of the Protocol is as follows:

c AI generated projects 登录复制 run

```c
///
/// The EFI IPv4 Protocol implements a simple packet-oriented interface that can be
/// used by drivers, daemons, and applications to transmit and receive network packets.
///
struct _EFI_IP4_PROTOCOL {
  EFI_IP4_GET_MODE_DATA    GetModeData;
  EFI_IP4_CONFIGURE        Configure;
  EFI_IP4_GROUPS           Groups;
  EFI_IP4_ROUTES           Routes;
  EFI_IP4_TRANSMIT         Transmit;
  EFI_IP4_RECEIVE          Receive;
  EFI_IP4_CANCEL           Cancel;
  EFI_IP4_POLL             Poll;
};
```

Its implementation is located in NetworkPkg\Ip4Dxe\Ip4Impl.c:

c AI generated projects 登录复制 run

```c
EFI_IP4_PROTOCOL mEfiIp4ProtocolTemplete = {
  EfiIp4GetModeData,
  EfiIp4Configure,
  EfiIp4Groups,
  EfiIp4Routes,
  EfiIp4Transmit,
  EfiIp4Receive,
  EfiIp4Cancel,
  EfiIp4Poll
};
```

The implementation of these functions will be introduced later.

Ip4.GetModeData

The corresponding implementation is that `EfiIp4GetModeData()` it not only obtains IP4 data, but also obtains MNP and SNP data, and both are optional:

c AI generated projects 登录复制 run

```c
EFI_STATUS
EFIAPI
EfiIp4GetModeData (
  IN  CONST EFI_IP4_PROTOCOL                *This,
  OUT       EFI_IP4_MODE_DATA               *Ip4ModeData     OPTIONAL,
  OUT       EFI_MANAGED_NETWORK_CONFIG_DATA  *MnpConfigData   OPTIONAL,
  OUT       EFI_SIMPLE_NETWORK_MODE          *SnpModeData     OPTIONAL
  )
{
  // 获取IP4数据，它们主要来自IP4_PROTOCOL
  if (Ip4ModeData != NULL) {
    //
    // IsStarted is "whether the EfiIp4Configure has been called".
    // IsConfigured is "whether the station address has been configured"
    //
    Ip4ModeData->IsStarted = (BOOLEAN)(IpInstance->State == IP4_STATE_CONFIGED);
    CopyMem (&Ip4ModeData->ConfigData, &IpInstance->ConfigData, sizeof (Ip4ModeData->ConfigData));
    Ip4ModeData->IsConfigured = FALSE;

    Ip4ModeData->GroupCount = IpInstance->GroupCount;
    Ip4ModeData->GroupTable = (EFI_IPv4_ADDRESS *)IpInstance->Groups;

    Ip4ModeData->IcmpTypeCount = 23;
    Ip4ModeData->IcmpTypeList  = mIp4SupportedIcmp;

    Ip4ModeData->RouteTable = NULL;
    Ip4ModeData->RouteCount = 0;

    Ip4ModeData->MaxPacketSize = IpSb->MaxPacketSize;

    //
    // return the current station address for this IP child. So,
    // the user can get the default address through this. Some
    // application wants to know it station address even it is
    // using the default one, such as a ftp server.
    //
    if (Ip4ModeData->IsStarted) {
      Config = &Ip4ModeData->ConfigData;

      Ip = HTONL (IpInstance->Interface->Ip);
      CopyMem (&Config->StationAddress, &Ip, sizeof (EFI_IPv4_ADDRESS));

      Ip = HTONL (IpInstance->Interface->SubnetMask);
      CopyMem (&Config->SubnetMask, &Ip, sizeof (EFI_IPv4_ADDRESS));

      Ip4ModeData->IsConfigured = IpInstance->Interface->Configured;

      //
      // Build a EFI route table for user from the internal route table.
      //
      Status = Ip4BuildEfiRouteTable (IpInstance);

      Ip4ModeData->RouteTable = IpInstance->EfiRouteTable;
      Ip4ModeData->RouteCount = IpInstance->EfiRouteCount;
    }
  }

  // 获取MNP和SNP的数据
  //
  // Get fresh mode data from MNP, since underlying media status may change
  //
  Status = IpSb->Mnp->GetModeData (IpSb->Mnp, MnpConfigData, SnpModeData);
}
```

Ip4.Configure

The corresponding implementation is `EfiIp4Configure()` :

c AI generated projects 登录复制 run

```c
EFI_STATUS
EFIAPI
EfiIp4Configure (
  IN EFI_IP4_PROTOCOL     *This,
  IN EFI_IP4_CONFIG_DATA  *IpConfigData       OPTIONAL
  )
{
  //
  // Validate the configuration first.
  //
  if (IpConfigData != NULL) {
    // 获取IP地址和掩码
    CopyMem (&IpAddress, &IpConfigData->StationAddress, sizeof (IP4_ADDR));
    CopyMem (&SubnetMask, &IpConfigData->SubnetMask, sizeof (IP4_ADDR));

    IpAddress  = NTOHL (IpAddress);
    SubnetMask = NTOHL (SubnetMask);

    //
    // Check whether the station address is a valid unicast address
    //
    if (!IpConfigData->UseDefaultAddress) {
      // 不适用默认的IP，表示需要用新IP，则需要验证新IP是否有效
      AddrOk = Ip4StationAddressValid (IpAddress, SubnetMask);

      if (!AddrOk) {
        Status = EFI_INVALID_PARAMETER;
        goto ON_EXIT;
      }
    }

    //
    // User can only update packet filters when already configured.
    // If it wants to change the station address, it must configure(NULL)
    // the instance first.
    //
    if (IpInstance->State == IP4_STATE_CONFIGED) {
      Current = &IpInstance->ConfigData;

      if (Current->UseDefaultAddress != IpConfigData->UseDefaultAddress) {
        Status = EFI_ALREADY_STARTED;
        goto ON_EXIT;
      }

      if (!Current->UseDefaultAddress &&
          (!EFI_IP4_EQUAL (&Current->StationAddress, &IpConfigData->StationAddress) ||
           !EFI_IP4_EQUAL (&Current->SubnetMask, &IpConfigData->SubnetMask)))
      {
        Status = EFI_ALREADY_STARTED;
        goto ON_EXIT;
      }

      if (Current->UseDefaultAddress && IP4_NO_MAPPING (IpInstance)) {
        Status = EFI_NO_MAPPING;
        goto ON_EXIT;
      }
    }
  }

  //
  // Configure the instance or clean it up.
  //
  if (IpConfigData != NULL) {
    Status = Ip4ConfigProtocol (IpInstance, IpConfigData);
  } else {
    Status = Ip4CleanProtocol (IpInstance);

    //
    // Consider the following valid sequence: Mnp is unloaded-->Ip Stopped-->Udp Stopped,
    // Configure (ThisIp, NULL). If the state is changed to UNCONFIGED,
    // the unload fails miserably.
    //
    if (IpInstance->State == IP4_STATE_CONFIGED) {
      IpInstance->State = IP4_STATE_UNCONFIGED;
    }
  }

  //
  // Update the MNP's configure data. Ip4ServiceConfigMnp will check
  // whether it is necessary to reconfigure the MNP.
  //
  Ip4ServiceConfigMnp (IpInstance->Service, FALSE);
}
```

`IpConfigData` It can be NULL, which means clearing the IP configuration. The configuration function finally called is `Ip4ConfigProtocol()` :

c AI generated projects 登录复制 run

```c
EFI_STATUS
Ip4ConfigProtocol (
  IN OUT IP4_PROTOCOL         *IpInstance,
  IN     EFI_IP4_CONFIG_DATA  *Config
  )
{
  // 如果已经配置过了，则只需要更新数据即可
  //
  // User is changing packet filters. It must be stopped
  // before the station address can be changed.
  //
  if (IpInstance->State == IP4_STATE_CONFIGED) {
    //
    // Cancel all the pending transmit/receive from upper layer
    //
    Status = Ip4Cancel (IpInstance, NULL);

    if (EFI_ERROR (Status)) {
      return EFI_DEVICE_ERROR;
    }

    CopyMem (&IpInstance->ConfigData, Config, sizeof (IpInstance->ConfigData));
    return EFI_SUCCESS;
```

```
twen  }
  25
  26  //
  27  // Configure a fresh IP4 protocol instance. Create a route table.
  28  // Each IP child has its own route table, which may point to the
  29  // default table if it is using default address.
  30  //
  31  Status                = EFI_OUT_OF_RESOURCES;
  32  IpInstance->RouteTable = Ip4CreateRouteTable ();
  33
  34  //
  35  // Set up the interface.
  36  //
  37  CopyMem (&Ip, &Config->StationAddress, sizeof (IP4_ADDR));
  38  CopyMem (&Netmask, &Config->SubnetMask, sizeof (IP4_ADDR));
  39
  40  Ip      = NTOHL (Ip);
  41  Netmask = NTOHL (Netmask);
  42
  43  if (!Config->UseDefaultAddress) {
  44    //
  45    // Find whether there is already an interface with the same
  46    // station address. All the instances with the same station
  47    // address shares one interface.
  48    //
  49    IpIf = Ip4FindStationAddress (IpSb, Ip, Netmask);
  50    if (IpIf != NULL) {
  51      NET_GET_REF (IpIf);
  52    } else {
  53      IpIf = Ip4CreateInterface (IpSb->Mnp, IpSb->Controller, IpSb->Image);
  54      Status = Ip4SetAddress (IpIf, Ip, Netmask);
  55      InsertTailList (&IpSb->Interfaces, &IpIf->Link);
  56    }
  57    //
  58    // Add a route to this connected network in the instance route table.
  59    //
  60    Ip4AddRoute (
  61      IpInstance->RouteTable,
  62      Ip & Netmask,
  63      Netmask,
  64      IP4_ALLZERO_ADDRESS
  65      );
  66  } else {
  67    //
  68    // Use the default address. Check the state.
  69    //
  70    if (IpSb->State == IP4_SERVICE_UNSTARTED) {
  71      //
  72      // Trigger the EFI_IP4_CONFIG2_PROTOCOL to retrieve the
  73      // default IPv4 address if it is not available yet.
  74      //
  75      Policy = IpSb->Ip4Config2Instance.Policy;
  76      if (Policy != Ip4Config2PolicyDhcp) {
  77        Ip4Config2 = &IpSb->Ip4Config2Instance.Ip4Config2;
  78        Policy     = Ip4Config2PolicyDhcp;
  79        Status     = Ip4Config2->SetData (
  80                                    Ip4Config2,
  81                                    Ip4Config2DataTypePolicy,
  82                                    sizeof (EFI_IP4_CONFIG2_POLICY),
  83                                    &Policy
  84                                    );
  85      }
  86    }
  87
  88    IpIf = IpSb->DefaultInterface;
  89    NET_GET_REF (IpSb->DefaultInterface);
  90
  91    //
  92    // If default address is used, so is the default route table.
  93    // Any route set by the instance has the precedence over the
  94    // routes in the default route table. Link the default table
  95    // after the instance's table. Routing will search the local
  96    // table first.
  97    //
  98    NET_GET_REF (IpSb->DefaultRouteTable);
  99    IpInstance->RouteTable->Next = IpSb->DefaultRouteTable;
 100  }
 101
 102  IpInstance->Interface = IpIf;
 103  if (IpIf->Arp != NULL) {
 104    Arp    = NULL;
 105    Status = gBS->OpenProtocol (
 106                    IpIf->ArpHandle,
 107                    &gEfiArpProtocolGuid,
 108                    (VOID **)&Arp,
 109                    gIp4DriverBinding.DriverBindingHandle,
 110                    IpInstance->Handle,
 111                    EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER
 112                    );
 113  }
 114
 115  InsertTailList (&IpIf->IpInstances, &IpInstance->AddrLink);
 116
 117  CopyMem (&IpInstance->ConfigData, Config, sizeof (IpInstance->ConfigData));
 118  IpInstance->State = IP4_STATE_CONFIGED;
 119 }
```

The code focuses on creating `IP4_INTERFACE` and updating `IP4_PROTOCOL` data.

Ip4.Transmit

The corresponding implementation is that `EfiIp4Transmit()` IP4 data transmission is more complicated:

1. The first is the construction of the IP packet. Depending on `RawData` the parameters in the configuration, there will be different processes:

```
c                                                                AI generated projects   登录复制   run
  1  //
  2  // Build the IP header, need to fill in the Tos, TotalLen, Id,
  3  // fragment, Ttl, protocol, Src, and Dst.
  4  //
  5  TxData = Token->Packet.TxData;
  6  if (Config->RawData) {
  7    //
  8    // When RawData is TRUE, first buffer in FragmentTable points to a raw
  9
```

```
    // IPv4 fragment including IPv4 header and options.
    //
  } else {
  }
```

2. The data is then further packaged:

```c
  //
  // OK, it survives all the validation check. Wrap the token in
  // a IP4_TXTOKEN_WRAP and the data in a netbuf
  //
```

3. Finally, perform the output operation:

```c
  Status = Ip4Output (
             IpSb,
             IpInstance,
             Wrap->Packet,
             &Head,
             OptionsBuffer,
             OptionsLength,
             GateWay,
             Ip4OnPacketSent,
             Wrap
             );
```

Of course, the output operation is not that simple. If there is a security protocol (currently there is not), the data needs to be processed securely. And because of the existence of routing, the address of the next hop node needs to be found, and finally the data is sent out through MNP .

Ip4.Receive

The corresponding implementation is that `EfiIp4Receive()` , like other network protocols under UEFI, Receive usually does not mean the reception of data, but only the processing function of registering data:

```c
EFI_STATUS
EFIAPI
EfiIp4Receive (
  IN EFI_IP4_PROTOCOL          *This,
  IN EFI_IP4_COMPLETION_TOKEN  *Token
  )
{
  //
  // Check whether the toke is already on the receive queue.
  //
  Status = NetMapIterate (&IpInstance->RxTokens, Ip4TokenExist, Token);

  //
  // Queue the token then check whether there is pending received packet.
  //
  Status = NetMapInsertTail (&IpInstance->RxTokens, Token, NULL);

  Status = Ip4InstanceDeliverPacket (IpInstance);

  //
  // Dispatch the DPC queued by the NotifyFunction of this instane's receive
  // event.
  //
  DispatchDpc ();
}
```

Ip4.Poll

The corresponding implementation is that `EfiIp4Poll()` this function simply calls the member function of MNP `Poll` :

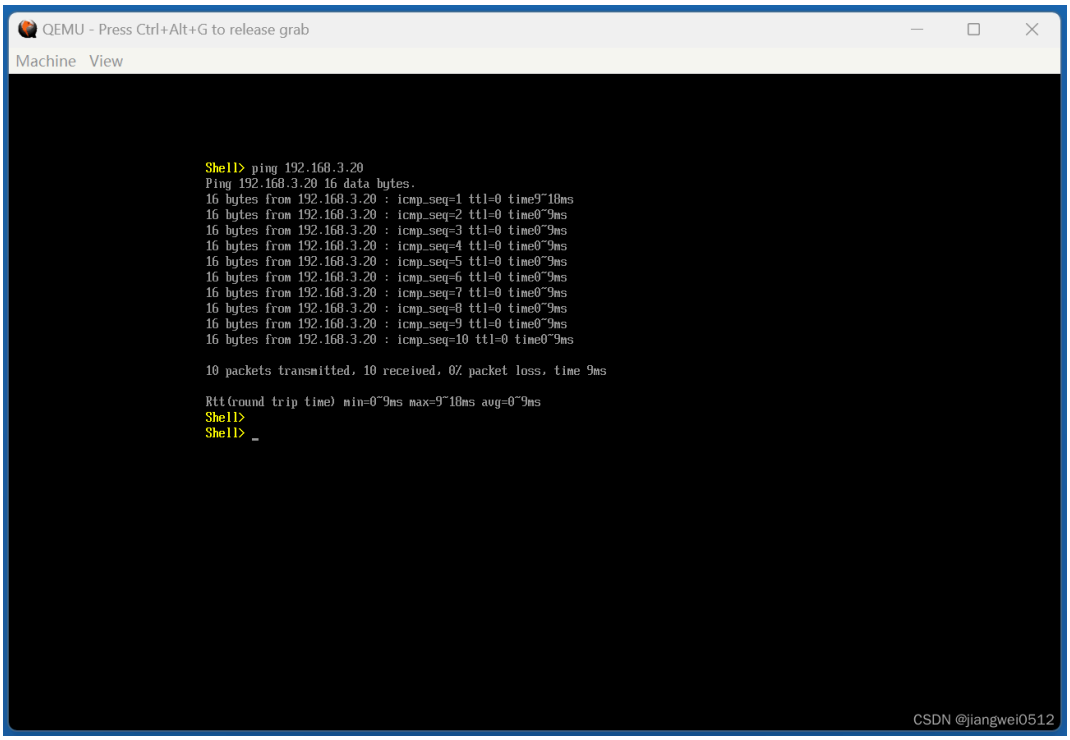
```c
EFI_STATUS
EFIAPI
EfiIp4Poll (
  IN EFI_IP4_PROTOCOL  *This
  )
{
  Mnp = IpInstance->Service->Mnp;

  //
  // Don't lock the Poll function to enable the deliver of
  // the packet polled up.
  //
  return Mnp->Poll (Mnp);
}
```

The member function of MNP `Poll` will receive network data and execute the callback interface registered by the upper-layer network protocol to process the data, so the registration interface of IP4 will also be called.

IP4 Code Examples

A good code example of IP4 is the ping program, which, when executed, displays:



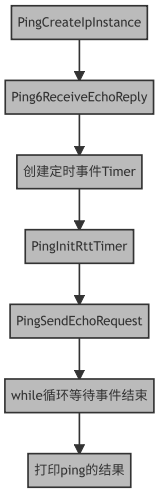
The corresponding code file is ShellPkg\Library\UefiShellNetwork1CommandsLib\Ping.c, which is part of ShellPKg. This section mainly analyzes the content of this part. First is the entry function:

```c
//
// Enter into ping process.
//
ShellStatus = ShellPing (
                (UINT32)SendNumber,
                (UINT32)BufferSize,
                &SrcAddress,
                &DstAddress,
                IpChoice
                );
```

Description of parameters:

- `SendNumber` : Indicates the number of pings, which `-n` is specified by parameters. If not specified, the default is 10 times.
- `BufferSize` : Indicates the size of the message, `-l` which is specified by parameters. If not specified, the default is 16 bytes.
- `SrcAddress` : Source IP address.
- `DstAddress` : Target IP address.
- `IpChoice` : Choose IPv4 or IPv6. Here we only focus on IPv4.

The execution flow of this function is:



There are several key functions that need to be explained here.

- First of all `Ping6ReceiveEchoReply()` , although there is a 6 here, it is not really IPv6, it is just a name, which is not very important. The focus of this function is the initialization of a Token:

```c
Status = gBS->CreateEvent (
                EVT_NOTIFY_SIGNAL,
                TPL_CALLBACK,
                Ping6OnEchoReplyReceived,
                Private,
                &Private->RxToken.Event
                );
Private->RxToken.Status = EFI_NOT_READY;
Status = Private->ProtocolPointers.Receive (Private->IpProtocol, &Private->RxToken);
```

One is used here `PING_IPX_PROTOCOL` , and its structure is:

```c
///
/// A set of pointers to either IPv6 or IPv4 functions.
/// Unknown which one to the ping command.
///
typedef struct {
  PING_IPX_TRANSMIT    Transmit;
  PING_IPX_RECEIVE     Receive;
  PING_IPX_CANCEL      Cancel;
  PING_IPX_POLL        Poll;
} PING_IPX_PROTOCOL;
```

It is actually just a wrapper for `EFI_IP4_PROTOCOL` or `EFI_IP6_PROTOCOL`, and its initialization is mentioned above `PingCreateIpInstance()`, taking the IPv4 version as an example:

```c
    Private->ProtocolPointers.Transmit = (PING_IPX_TRANSMIT)((EFI_IP4_PROTOCOL *)Private->IpProtocol)->Transmit;
    Private->ProtocolPointers.Receive  = (PING_IPX_RECEIVE)((EFI_IP4_PROTOCOL *)Private->IpProtocol)->Receive;
    Private->ProtocolPointers.Cancel   = (PING_IPX_CANCEL)((EFI_IP4_PROTOCOL *)Private->IpProtocol)->Cancel;
    Private->ProtocolPointers.Poll     = (PING_IPX_POLL)((EFI_IP4_PROTOCOL *)Private->IpProtocol)->Poll;
```

So the function that is actually called in the end `EFI_IP4_PROTOCOL` is `Receive()`

Therefore, the ultimate purpose of this function is to create a Token to process the data at the IP layer, and the processing function is the callback function here `Ping6OnEchoReplyReceived()`.

- Then introduce the callback function `Ping6OnEchoReplyReceived()`, its implementation is to process the received message:

```c
    Reply   = ((EFI_IP4_RECEIVE_DATA *)Private->RxToken.Packet.RxData)->FragmentTable[0].FragmentBuffer;
    PayLoad = ((EFI_IP4_RECEIVE_DATA *)Private->RxToken.Packet.RxData)->DataLength;
    if (!IP4_IS_MULTICAST (EFI_IP4 (*(EFI_IPv4_ADDRESS *)Private->DstAddress)) &&
        !EFI_IP4_EQUAL (&((EFI_IP4_RECEIVE_DATA *)Private->RxToken.Packet.RxData)->Header->SourceAddress, (EFI_IPv4_ADDRESS *)&Private->DstAddress))
    {
      goto ON_EXIT;
    }

    if ((Reply->Type != ICMP_V4_ECHO_REPLY) || (Reply->Code != 0)) {
      goto ON_EXIT;
    }
  }

  if (PayLoad != Private->BufferSize) {
    goto ON_EXIT;
  }

  //
  // Check whether the reply matches the sent request before.
  //
twen Status = Ping6MatchEchoReply (Private, Reply);
twen if (EFI_ERROR (Status)) {
twen   goto ON_EXIT;
twen }

  //
  // Display statistics on this icmp6 echo reply packet.
  //
  Rtt = CalculateTick (Private, Reply->TimeStamp, ReadTime (Private));

  Private->RttSum += Rtt;
  Private->RttMin  = Private->RttMin > Rtt ? Rtt : Private->RttMin;
  Private->RttMax  = Private->RttMax < Rtt ? Rtt : Private->RttMax;

  ShellPrintHiiEx (
    -1,
    -1,
    NULL,
    STRING_TOKEN (STR_PING_REPLY_INFO),
    gShellNetwork1HiiHandle,
    PayLoad,
    mDstString,
    Reply->SequenceNum,
    Private->IpChoice == PING_IP_CHOICE_IP6 ? ((EFI_IP6_RECEIVE_DATA *)Private->RxToken.Packet.RxData)->Header->HopLimit : 0,
    Rtt,
    Rtt + Private->TimerPeriod
    );
```

There is not much to introduce about this, it is just ordinary protocol layer processing. It will eventually print the results of each ping, and when the number of times reaches the requirement of the command line parameter, it will exit. The corresponding code is:

```c
  if (Private->RxCount < Private->SendNum) {
    //
    // Continue to receive icmp echo reply packets.
    //
    Private->RxToken.Status = EFI_ABORTED;

    Status = Private->ProtocolPointers.Receive (Private->IpProtocol, &Private->RxToken);

    if (EFI_ERROR (Status)) {
      ShellPrintHiiEx (-1, -1, NULL, STRING_TOKEN (STR_PING_RECEIVE), gShellNetwork1HiiHandle, Status);
      Private->Status = EFI_ABORTED;
    }
  } else {
    //
    // All reply have already been received from the dest host.
    //
    Private->Status = EFI_SUCCESS;
  }
```

This is just a preparatory work, because no data has actually been sent yet, so here we are just preparing to receive it.

- Later, we will introduce that the full name of RTT here is Round Trip `PingInitRttTimer()` Time **, which is** a performance indicator of network delay, indicating the total delay from the start of sending data to the sender receiving confirmation from the receiver (the receiver sends confirmation immediately after receiving the data). The code uses a timer to count:

```c
  Private->RttTimerTick = 0;
  Status                = gBS->CreateEvent (
                                  EVT_TIMER | EVT_NOTIFY_SIGNAL,
                                  TPL_NOTIFY,
                                  RttTimerTickRoutine,
                                  &Private->RttTimerTick,
```

```
                              &Private->RttTimer
                              );
  Status = gBS->SetTimer (
                 Private->RttTimer,
                 TimerPeriodic,
                 TICKS_PER_MS
                 );
```

Its timing interval is 1ms, so in the previous running process you can see that the results obtained are all in milliseconds.

- Next is to send IP packets, using functions `PingSendEchoRequest()` , the most important code is to send data:

c AI generated projects 登录复制 run

```
  Status = Private->ProtocolPointers.Transmit (Private->IpProtocol, TxInfo->Token);
```

The data sent `PingGenerateToken()` is created by the function, which is also wrapped in a Token, and its content is:

c AI generated projects 登录复制 run

```
  //
  // Assembly echo request packet.
  //
  Request->Type        = (UINT8)(Private->IpChoice == PING_IP_CHOICE_IP6 ? ICMP_V6_ECHO_REQUEST : ICMP_V4_ECHO_REQUEST);
  Request->Code        = 0;
  Request->SequenceNum = SequenceNum;
  Request->Identifier  = 0;
  Request->Checksum    = 0;

  ((EFI_IP4_TRANSMIT_DATA *)TxData)->FragmentTable[0].FragmentBuffer = (VOID *)Request;
  ((EFI_IP4_TRANSMIT_DATA *)TxData)->FragmentTable[0].FragmentLength = Private->BufferSize;
```

Ping is implemented through ICMP messages, `Request` which correspond to a structure here:

c AI generated projects 登录复制 run

```
typedef struct _ICMPX_ECHO_REQUEST_REPLY {
  UINT8     Type;
  UINT8     Code;
  UINT16    Checksum;
  UINT16    Identifier;
  UINT16    SequenceNum;
  UINT32    TimeStamp;
  UINT8     Data[1];
} ICMPX_ECHO_REQUEST_REPLY;
```

Corresponding to ICMP .

- Next comes the process of waiting for a response or key press:

c AI generated projects 登录复制 run

```
  //
  // Control the ping6 process by two factors:
  // 1. Hot key
  // 2. Private->Status
  //   2.1. success means all icmp6 echo request packets get reply packets.
  //   2.2. timeout means the last icmp6 echo reply request timeout to get reply.
  //   2.3. noready means ping6 process is on-the-go.
  //
  while (Private->Status == EFI_NOT_READY) {
    Status = Private->ProtocolPointers.Poll (Private->IpProtocol);
    if (ShellGetExecutionBreakFlag ()) {
      Private->Status = EFI_ABORTED;
      goto ON_STAT;
    }
  }
```

Here's the initialization `EFI_NOT_READY` from `Ping6ReceiveEchoReply()` :

c AI generated projects 登录复制 run

```
Private->RxToken.Status = EFI_NOT_READY;
```

After that, when the number of pings exceeds the command line parameter, it will be set to `EFI_ABORTED` :

c AI generated projects 登录复制 run

```
  if (Private->RxCount < Private->SendNum) {
    //
    // Continue to receive icmp echo reply packets.
    //
    Private->RxToken.Status = EFI_ABORTED;
```

This exits the loop.

- Finally, the ping result is printed, which is the last two sentences in the previous figure.