UEFI Development Exploration 50 – UEFI and Network 2

原创 luobing4365 Posted on 2020-03-26 21:12:47 Read 2.7k Collection 4 Likes copyright
Category Column: UEFI Development Article Tags: UEFI Application Development UEFI Programming Low-level programming Network Programming UEFI BIOS

(Please keep it-> Author: Luo Bing https://blog.csdn.net/luobing4365)

3 Using Networking in VirtualBox

In the 48th blog of the UEFI development exploration series, I introduced how to build a UEFI Shell in VirtualBox. This section builds on this blog and enables the UEFI Shell in VirtualBox to access the network.

1) Install network driver and network protocol driver

I am using VirtualBox6.1.4, and its virtual network card is Intel Pro/1000MT Desktop. The network card driver download address is:

https://downloadcenter.intel.com/download/27539/Ethernet-Intel-Ethernet-Connections-Boot-Utility-Preboot-Images-and-EFI-Drivers

Download the 22.10 version of PREBOOT.exe and double-click to install. Copy the driver E3522X2.EFI in the directory /APPS/EIF/EFIx64 to the virtual machine 's hard disk. At the same time, also copy the previously compiled x64 Ipv4 network protocol driver (MdeModulePkg) to the virtual machine's hard disk.

Start the virtual machine, enter the UEFI Shell, and execute the following command to load the network card driver and network protocol driver:

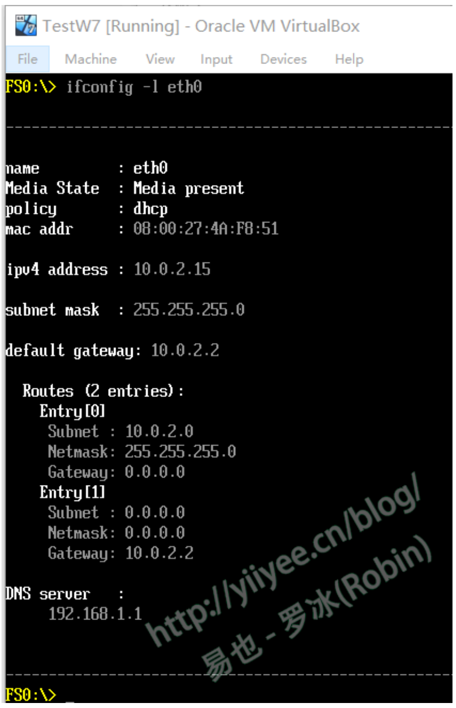
```
Shell>fs0:
fs0:>load E3522X2.EFI
fs0:> load SnpDxe.efi MnpDxe.efi ArpDxe.efi Ip4Dxe.efi VlanConfigDxe.efi Udp4Dxe.efi Dhcp4Dxe.efi Mtftp4Dxe.efi Tcp4Dxe.efi
```

2) Configure the network card.

Use ifconfig command to set:

```
fs0:\>ifconfig -s eth0 dhcp
```

Check the network configuration:

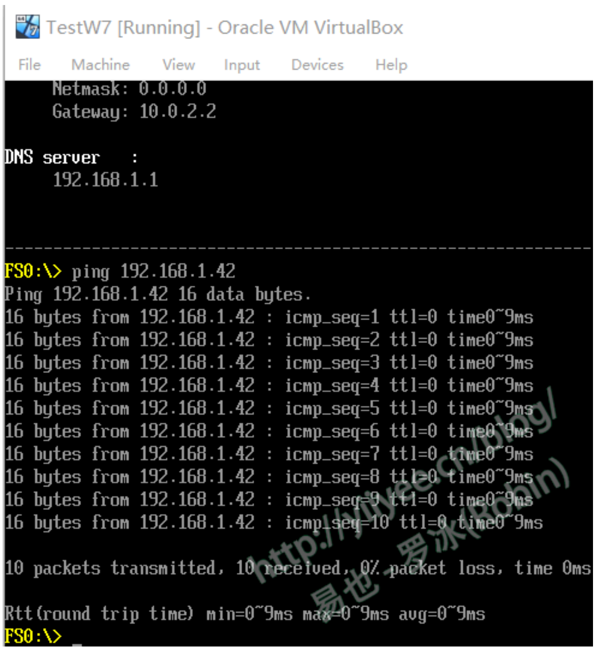

TestW7 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

```
FS0:\> ifconfig -l eth0

-----------------------------------------------

name         : eth0
Media State  : Media present
policy       : dhcp
mac addr     : 08:00:27:4A:F8:51

ipv4 address : 10.0.2.15

subnet mask  : 255.255.255.0

default gateway: 10.0.2.2

  Routes (2 entries):
    Entry[0]
     Subnet : 10.0.2.0
     Netmask: 255.255.255.0
     Gateway: 0.0.0.0
    Entry[1]
     Subnet : 0.0.0.0
     Netmask: 0.0.0.0
     Gateway: 10.0.2.2

DNS server    :
     192.168.1.1

-----------------------------------------------
FS0:\> _
```

Figure 1 Viewing IP in VirtualBox's UEFI Shell

3) Test network connection

My host IP address is 192.168.1.42, and the VirtualBox virtual machine is connected to the host via NAT. The test results are as follows:


TestW7 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

```
     Netmask: 0.0.0.0
     Gateway: 10.0.2.2

DNS server    :
     192.168.1.1


-------------------------------------------------------------
FS0:\> ping 192.168.1.42
Ping 192.168.1.42 16 data bytes.
16 bytes from 192.168.1.42 : icmp_seq=1 ttl=0 time0~9ms
16 bytes from 192.168.1.42 : icmp_seq=2 ttl=0 time0~9ms
16 bytes from 192.168.1.42 : icmp_seq=3 ttl=0 time0~9ms
16 bytes from 192.168.1.42 : icmp_seq=4 ttl=0 time0~9ms
16 bytes from 192.168.1.42 : icmp_seq=5 ttl=0 time0~9ms
16 bytes from 192.168.1.42 : icmp_seq=6 ttl=0 time0~9ms
16 bytes from 192.168.1.42 : icmp_seq=7 ttl=0 time0~9ms
16 bytes from 192.168.1.42 : icmp_seq=8 ttl=0 time0~9ms
16 bytes from 192.168.1.42 : icmp_seq=9 ttl=0 time0~9ms
16 bytes from 192.168.1.42 : icmp_seq=10 ttl=0 time0~9ms

10 packets transmitted, 10 received, 0% packet loss, time 0ms

Rtt(round trip time) min=0~9ms max=0~9ms avg=0~9ms
FS0:\> _
```

Figure 2 Network connection test in VirtualBox UEFI shell

4 Using Network in Qemu

In the following experiments, the OVMF image was compiled under VS2015+UDK2018, and the Qemu network experiment was performed under Ubuntu16.04 LTS.

In the official documentation, OVMF experiments have always been based on Qemu. Qemu is very powerful and flexible, but the problem is that its configuration is relatively complex. Therefore, I rarely use it except when I need to debug at the source level.

To build a network test environment in Qemu, follow the steps below:

1) Compile the OVMF image.

Download E3522X2.EFI from Intel website, which is the driver of network card E1000. The download address is given in the previous section. Enter the UEFI compilation directory, create a new directory Intel3.5/EFIX64, and copy E3522X2.EFI to this directory.

Compile the OVMF image. The compile command is:

C:\Myworkspace> build -p OvmfPkg/OvmfPkgX64.dsc -a X64 -D E1000_ENABLE -D DEBUG_ON_SERIAL_PORT

At the same time, compile the 64-bit IPv4 network protocol under UEFI according to the previous method, copy the compiled image OVMF.fd and network protocol driver, and prepare for the following steps.

2) Install Qemu and necessary network tools

Use apt-get to install, the command is:

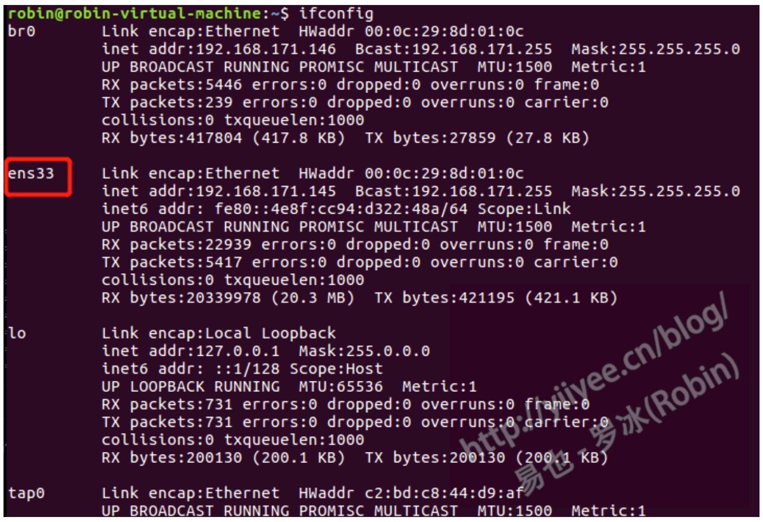
$ sudo apt-get install qemu
$ sudo apt-get install bridge-utils #Virtual bridge setting tool
$ sudo apt-get install uml-utilities #UML (User-mod linux) tool

3) Build Qemu 's network channel

In the previous compilation, the network card driver has been included in the UEFI shell started by the VOMF image. If no network settings are specified, Qemu will use user mode networking with a built-in DHCP server. When the virtual machine is running, it can be set to DHCP mode, and it can access the physical host's network through the IP disguised by QEMU.

However, in this case, only TCP and UDP protocols can be used for communication, so ICMP protocols (including ping) will not work. Therefore, we are going to use tap mode and bridge to enable the virtual machine to communicate with the outside world.

First, use the ifconfig command to obtain the network interface of this machine:



```
robin@robin-virtual-machine:~$ ifconfig
br0       Link encap:Ethernet  HWaddr 00:0c:29:8d:01:0c
          inet addr:192.168.171.146  Bcast:192.168.171.255  Mask:255.255.255.0
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
          RX packets:5446 errors:0 dropped:0 overruns:0 frame:0
          TX packets:239 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:417804 (417.8 KB)  TX bytes:27859 (27.8 KB)

ens33     Link encap:Ethernet  HWaddr 00:0c:29:8d:01:0c
          inet addr:192.168.171.145  Bcast:192.168.171.255  Mask:255.255.255.0
          inet6 addr: fe80::4e8f:cc94:d322:48a/64 Scope:Link
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
          RX packets:22939 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5417 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:20339978 (20.3 MB)  TX bytes:421195 (421.1 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:731 errors:0 dropped:0 overruns:0 frame:0
          TX packets:731 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:200130 (200.1 KB)  TX bytes:200130 (200.1 KB)

tap0      Link encap:Ethernet  HWaddr c2:bd:c8:44:d9:af
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
```

Figure 3 Obtaining the network interface

Then follow the steps below. Of course, you can also write a sh file for batch processing :

$sudo ifconfig ens33 down # Shut down the ens33 interface first
$sudo brctl addbr br0 # Add a virtual bridge br0
$sudo brctl addif br0 ens33 # Add an interface ens33 in br0

$sudo brctl stp br0 off # There is only one bridge, so turn off the spanning tree protocol
$sudo brctl setfd br0 1 # Set the forwarding delay of br0
$sudo brctl sethello br0 1 # Set the hello time of br0
$sudo ifconfig br0 0.0.0.0 promisc up # Open the br0 interface
$sudo ifconfig ens33 0.0.0.0 promisc up # Open the ens33 interface
$sudo dhclient br0 # Get the IP address of br0 from the DHCP server

$sudo tunctl -t tap0 -u root # Create a tap0 interface and only allow root user to access
$sudo brctl addif br0 tap0 # Add a tap0 interface to the virtual bridge
$sudo ifconfig tap0 0.0.0.0 promisc up # Open the tap0 interface

Its function is to create a virtual bridge br0 and a virtual network card interface tap0, and use tap0 and the host's network interface (ens33) as two interfaces of the bridge. In this way, the host's interface is used as a bridge interface to connect to the external network; the TAP device is used as another interface of the bridge to connect to the Vlan in the Qemu virtual machine.

4Configure the UEFI network card in the virtual machine

Copy the previous Ipv4 64-bit driver file to hda.img. The copying method has been described in detail in the previous blog "UEFI Development Exploration 39".

Start the virtual machine:

$sudo qemu-system-x86_64 -bios OVMF.fd -hdd hda.img -net nic -net tap,ifname=tap0 -serial stdio

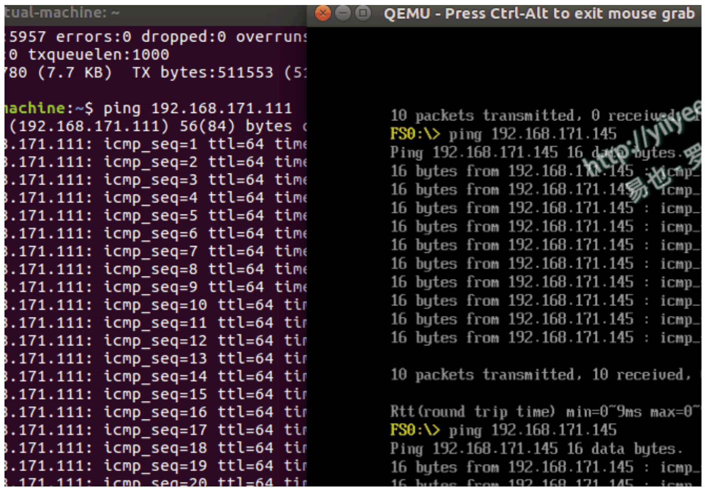
After entering the UEFI shell, set the IP address of the network card, making sure it is in the same network segment as the host's network card address.

FS0:\> ifconfig -s eth0 static 192.168.171.111 255.255.255.0 192.168.171.141

At this point, all configuration processes are completed, and the virtual machine's network card is connected to the host's network card. You can follow the method described above to load the IPv4-related network protocol driver for subsequent experiments.

5) Test network connection

Ping the virtual machine from the host machine, and ping the host machine from the virtual machine, as follows:



```
tual-machine: ~
:5957 errors:0 dropped:0 overruns
:0 txqueuelen:1000
780 (7.7 KB)  TX bytes:511553 (51

achine:~$ ping 192.168.171.111
 (192.168.171.111) 56(84) bytes c
3.171.111: icmp_seq=1 ttl=64 time
3.171.111: icmp_seq=2 ttl=64 time
3.171.111: icmp_seq=3 ttl=64 time
3.171.111: icmp_seq=4 ttl=64 time
3.171.111: icmp_seq=5 ttl=64 time
3.171.111: icmp_seq=6 ttl=64 time
3.171.111: icmp_seq=7 ttl=64 time
3.171.111: icmp_seq=8 ttl=64 time
3.171.111: icmp_seq=9 ttl=64 time
3.171.111: icmp_seq=10 ttl=64 tir
3.171.111: icmp_seq=11 ttl=64 tir
3.171.111: icmp_seq=12 ttl=64 tir
3.171.111: icmp_seq=13 ttl=64 tir
3.171.111: icmp_seq=14 ttl=64 tir
3.171.111: icmp_seq=15 ttl=64 tir
3.171.111: icmp_seq=16 ttl=64 tir
3.171.111: icmp_seq=17 ttl=64 tir
3.171.111: icmp_seq=18 ttl=64 tir
3.171.111: icmp_seq=19 ttl=64 tir
3.171.111: icmp_seq=20 ttl=64 tir
```

```
QEMU - Press Ctrl-Alt to exit mouse grab

10 packets transmitted, 0 received
FS0:\> ping 192.168.171.145
Ping 192.168.171.145 16 data bytes.
16 bytes from 192.168.171.145 : icmp_
16 bytes from 192.168.171.145 : icmp_
16 bytes from 192.168.171.145 : icmp_
16 bytes from 192.168.171.145 : icmp_
16 bytes from 192.168.171.145 : icmp_
16 bytes from 192.168.171.145 : icmp_
16 bytes from 192.168.171.145 : icmp_
16 bytes from 192.168.171.145 : icmp_
16 bytes from 192.168.171.145 : icmp_
16 bytes from 192.168.171.145 : icmp_

10 packets transmitted, 10 received,

Rtt(round trip time) min=0~9ms max=0~
FS0:\> ping 192.168.171.145
Ping 192.168.171.145 16 data bytes.
16 bytes from 192.168.171.145 : icmp_
16 bytes from 192.168.171.145 : icmp
```

Figure 4 Network connection test between Qemu virtual machine and host machine

I have tried three different methods of setting up UEFI networks in two blog posts. For normal programming, the most commonly used one is probably the Nt32 simulation environment.

Starting from the next article, try TCP and UDP programming under UEFI.