**Network**

Kernel.cpp

PCIController.SelectDrivers(&drvManager, &interrupts);

Pci.cpp

PeripheralComponentInterconnectController::SelectDrivers()

This iterates over the PCi buses , devices and functions and then it calls :

Driver\* driver = GetDriver(dev, interrupts);

Which calls :

Driver\* PeripheralComponentInterconnectController::GetDriver()

This switches over the device id and device vendor and when it finds the amd79c973 it allocates memory for it and then calls :

new (driver) amd\_am79c973(&dev, interrupts);

amd\_am79c973.cpp

This then calls the constructor of amd79c973 which sets the ports , mac address , initialization block , send & receive descriptors and buffers .

Pci.cpp

After calling amd79c973 constructor and allocating memory for it we return the driver and return from GetDriver() .

Back To SelectDrivers():

if(driver != 0)

driverManager->AddDriver(driver);

driver.cpp

DriverManager::AddDriver();

This add the amd79c973 driver to the drivers array and increments the number of drivers . Note that we previously added the keyboard driver then the mouse driver . So the amd79c973 driver is member 2 of the drivers array .

Pci.cpp

We now print PCI Bus , device , Function , Vendor id , Device id and loop through SelectDrivers() till it finishes and return from it back to kernel.cpp .

Kernel.cpp

drvManager.ActivateAll();

This loops through the drivers array and call Activate() for each driver . So amd79c973 calls it own Activate() .

amd\_am79c973.cpp

void amd\_am79c973::Activate();

It activates the amd79c973 card .

Then back to kernel.cpp

Kernel.cpp

We define our ip address in virtual box and convert it to BigEndian : ip\_be

and define the gateway ip in VirtualBox and convert it also to BigEndian : gip\_be .

amd\_am79c973\* eth0 = (amd\_am79c973\*)(drvManager.drivers[2]);

Then we define a pointer to our amd79c973 driver and call it eth0 .

eth0->SetIPAddress(ip\_be);

We set our ip address so we jump to :

amd\_am79c973.cpp

void amd\_am79c973::SetIPAddress(uint32\_t ip);

This sets the LogicalAddress field in the initialization Block to our ip address .

Back to :

kernel.cpp

EtherFrameProvider etherframe(eth0);

We create an EtherFrameProvider object and pass to it the pointer to the driver of amd79c973 which is the backend of EtherFrameProvider . This calls EtherFrameProvider constructor so we jump to it .

Etherframe.cpp

EtherFrameProvider::EtherFrameProvider();

This initializes the RawDataHandler constructor by : RawDataHandler(backend) so we jump to the constructor in amd\_am79c973.cpp .

amd\_am79c973.cpp

RawDataHandler::RawDataHandler(amd\_am79c973\* backend);

This sets the amd\_am79c973\* backend protected member of RawDataHandler to eth0 .

backend->SetHandler(this);

This calls amd\_am79c973::SetHandler(RawDataHandler\* handler); which sets the RawDataHandler\* handler member of amd\_am79c973 object to this which is the “ etherframe “ object we created because EtherFrameProvider is inherited from RawDataHandler . Then we return back to EtherFrameProvider() constructor .

Etherframe.cpp

In the EtherFrameProvider() constructor we loop through the handlers array and set every member to zero . This array is array of EtherFrameHandler\* .

Then we return back to kernel.cpp

Kernel.cpp

AddressResolutionProtocol arp(&etherframe);

We create an AddressResolutionProtocol object and this calls its constructor and we pass it by reference etherframe which is its backend . We jump to arp.cpp

Arp.cpp

AddressResolutionProtocol::AddressResolutionProtocol(EtherFrameProvider\* backend)

: EtherFrameHandler(backend, 0x806)

This calls the EtherFrameHandler so we jump to it in etherframe.cpp and we pass it &etherframe and 0x806 which is ether type for ARP .

Etherframe.cpp

EtherFrameHandler::EtherFrameHandler(EtherFrameProvider\* backend, uint16\_t etherType);

This sets the ethertype but in BigEndian , sets the protected member EtherFrameProvider\* backend to &etherframe and sets handlers[0608] to our “ arp “ object .

Then we return back to arp.cpp

Arp.cpp

We set the number of cache entries to 0 then return back to kernel.cpp

Kernel.cpp

After activating interrupts we call arp.Resolve(gip\_be); So we jump to arp.cpp .

Arp.cpp

uint64\_t AddressResolutionProtocol::Resolve(uint32\_t IP\_BE)

{

uint64\_t result = GetMACFromCache(IP\_BE);

This calls :

uint64\_t AddressResolutionProtocol::GetMACFromCache(uint32\_t IP\_BE);

which loops through the IPcache and if it finds the requested IP address it will return its MAC address , but other than that it will return the broadcast address 0xFFFFFFFFFFFF . then we return to Resolve()

if(result == 0xFFFFFFFFFFFF)

RequestMACAddress(IP\_BE);

And this calls :

void AddressResolutionProtocol::RequestMACAddress(uint32\_t IP\_BE)

{

…

…

this->Send(arp.dstMAC, (uint8\_t\*)&arp, sizeof(AddressResolutionProtocolMessage));

}

“this” refers to “arp” object so we call AddressResolutionProtocol::Send() . But wait a minute there is no function like that . So we call EtherFrameHandler::Send() because AddressResolutionProtocol is inherited from EtherFrameHandler . so we jump to etherframe.cpp

Etherframe.cpp

void EtherFrameHandler::Send(common::uint64\_t dstMAC\_BE, common::uint8\_t\* data, common::uint32\_t size)

{

backend->Send(dstMAC\_BE, etherType\_BE, data, size);

}

“backend” here is “ EtherFrameProvider\* backend ” so we call EtherFrameProvider::Send()

Which allocates memory for a new EtherFrameHeader which initializes it and sets the payload to the arp message and then calls backend->Send(buffer2, size + sizeof(EtherFrameHeader));

“backend” here is “ RawDataHandler:: amd\_am79c973\* backend “ so we call amd\_am79c973::Send();

We jump to :

amd\_am79c973.cpp

We call amd\_am79c973::Send() which sets the send Descriptor and then prints “ Sending : “ and prints the message we want to send and then it triggers an interrupt by the send command 0x48 .

So now an Interrupt occurs and amd\_am79c973::HandleInterrupt is called which prints that an interrupt occurred from amd79c973 and then prints that data is sent .

Then we now wait for the gateway to respond with its mac address because remember we asked for “gip” mac address .

So when the gateway responds and we receive data an Interrupt occurs so amd\_am79c973::HandleInterrupt is called and it prints that an Interrupt occurred and calls

amd\_am79c973::Receive() which prints that Data was received and then calls :

if(handler != 0)

if(handler->OnRawDataReceived(buffer, size))

“ handler “ here is “ etherframe “ object so EtherFrameProvider::OnRawDataReceived() is called

So we jump to :

Etherframe.cpp

bool EtherFrameProvider::OnRawDataReceived(common::uint8\_t\* buffer, common::uint32\_t size)

So when the gateway responds it sends the data to a specific Mac address so we check if the Mac address specified is ours and then execute :

if(handlers[frame->etherType\_BE] != 0)

sendBack = handlers[frame->etherType\_BE]->OnEtherFrameReceived(

buffer + sizeof(EtherFrameHeader), size - sizeof(EtherFrameHeader));

“ handlers “ here are the “ EtherFrameHandler\* handlers[] “ so we check to see if we have a handler to the frame we received . And we already have set a handler for the ARP messages which is defined at “ handlers[0608] “ which is our “ arp “ object so now we jump to :

Arp.cpp

And call AddressResolutionProtocol::OnEtherFrameReceived() which switches to see if the arp->command is a request or a response . We have already sent a request to the gateway so now we have received a response so the command in BigEndian is 0x0200 and now we will enter the IP and MAC addresses sent by the gatway in our IP and MAC caches . And then we will return false because this is a response and we will not send data again for now .

So back to :

Etherframe.cpp

“ Sendback “ is set to false and we return false from OnRawDataReceived() function .

So back to :

amd\_am79c973.cpp

if(handler != 0)

if(handler->OnRawDataReceived(buffer, size))

Send(buffer, size);

The Send() function won’t be executed because we returned false from OnRawDataReceived() .And now the Receive() function prints the data we received . So now we return to our Resolve() function and get our result from the MAC cache if it was entered in the cache . if not we will enter in an infinite loop .

Another case scenario is what if we were sent a request from another device that wants to know our MAC address . An Interrupt will occur so amd\_am79c973::HandleInterrupt is called and it prints that an Interrupt occurred and calls

amd\_am79c973::Receive() which prints that Data was received and then calls :

if(handler != 0)

if(handler->OnRawDataReceived(buffer, size))

“ handler “ here is “ etherframe “ object so EtherFrameProvider::OnRawDataReceived() is called

So we jump to :

Etherframe.cpp

bool EtherFrameProvider::OnRawDataReceived(common::uint8\_t\* buffer, common::uint32\_t size)

So we check if the Mac address specified is ours and then execute :

if(handlers[frame->etherType\_BE] != 0)

sendBack = handlers[frame->etherType\_BE]->OnEtherFrameReceived(

buffer + sizeof(EtherFrameHeader), size - sizeof(EtherFrameHeader));

“ handlers “ here are the “ EtherFrameHandler\* handlers[] “ so we check to see if we have a handler to the frame we received . And we already have set a handler for the ARP messages which is defined at “ handlers[0608] “ which is our “ arp “ object so now we jump to :

Arp.cpp

And call AddressResolutionProtocol::OnEtherFrameReceived() which switches to see if the arp->command is a request or a response . It is now a request so the command in BigEndian is 0x0100 .

Then we will set in the ARP message our IP and MAC addresses and respond back to the requesting device . A true is returned .

Back to :

Etherframe.cpp

Sendback is now set to true so now we return with true .

Back to :

amd\_am79c973.cpp

if(handler != 0)

if(handler->OnRawDataReceived(buffer, size))

Send(buffer, size);

Now Send() is executed and a Sending interrupt will be triggered .

Note that when we send or receive frame packets , what is printed is the following :

Ether Frame header + payload ( ARP or IPv4 ) .

Kernel.cpp

InternetProtocolProvider ipv4(&etherframe, &arp, gip\_be, subnet\_be);

This calls the constructor in ipv4.cpp so we jump to it .

Ipv4.cpp

InternetProtocolProvid er::InternetProtocolProvider(EtherFrameProvider\* backend,

AddressResolutionProtocol\* arp,

uint32\_t gatewayIP, uint32\_t subnetMask)

: EtherFrameHandler(backend, 0x800)

This calls the EtherFrameHandler constructor in etherframe.cpp so we jump to it .

Etherframe.cpp

EtherFrameHandler::EtherFrameHandler(EtherFrameProvider\* backend, uint16\_t etherType)

{

..

..

backend->handlers[etherType\_BE] = this;

}

This sets the handlers[0x800] defined in EtherFrameProvider class to “ipv4” object defined in kernel.cpp

After that we return to the rest of the InternetProtocolProvider constructor and then return back to kernel.cpp

Kernel.cpp

ipv4.Send(gip\_be, 0x0008, (uint8\_t\*) "foobar", 6);

This calls InternetProtocolProvider::Send() in ipv4.cpp

Ipv4.cpp

void InternetProtocolProvider::Send(uint32\_t dstIP\_BE, uint8\_t protocol, uint8\_t\* data, uint32\_t size)

This allocates memory for the ipv4 message that we want to send then initializes the ipv4 Header .

Then it copies the data that we want to send into the ipv4 message’s payload .

Then it resolves the the ip address of the destination and you can see from previous summary how arp.resolve() works . Then it calls :

backend->Send(arp->Resolve(route), this->etherType\_BE, buffer, sizeof(InternetProtocolV4Message) + size);

“backend” is the EtherFrameProvider object “etherframe” so we jump to etherframe.cpp

Etherframe.cpp

void EtherFrameProvider::Send(common::uint64\_t dstMAC\_BE, common::uint16\_t etherType\_BE, common::uint8\_t\* buffer, common::uint32\_t size)

This allocates memory for the etherframe message that we want to send . Then it initializes the etherframe header and then copies the data from the buffer to the etherframe payload .

Then it calls backend->Send(buffer2, size + sizeof(EtherFrameHeader)); which calls

void amd\_am79c973::Send(uint8\_t\* buffer, int size) which puts the data we want to send in the current Send Descriptor and then triggers an Interrupt .

The Final data that we send now is the Ether Frame Packet which consists of the Ether Frame Header and Ether Frame Payload . The Payload consist of the ipv4 message which consists of the ipv4 Header and the ipv4 payload which is our “foobar” string .

Then we free the memory allocated in EtherFrameProvider::Send() then we return back and free the memory allocated in InternetProtocolProvider::Send() .