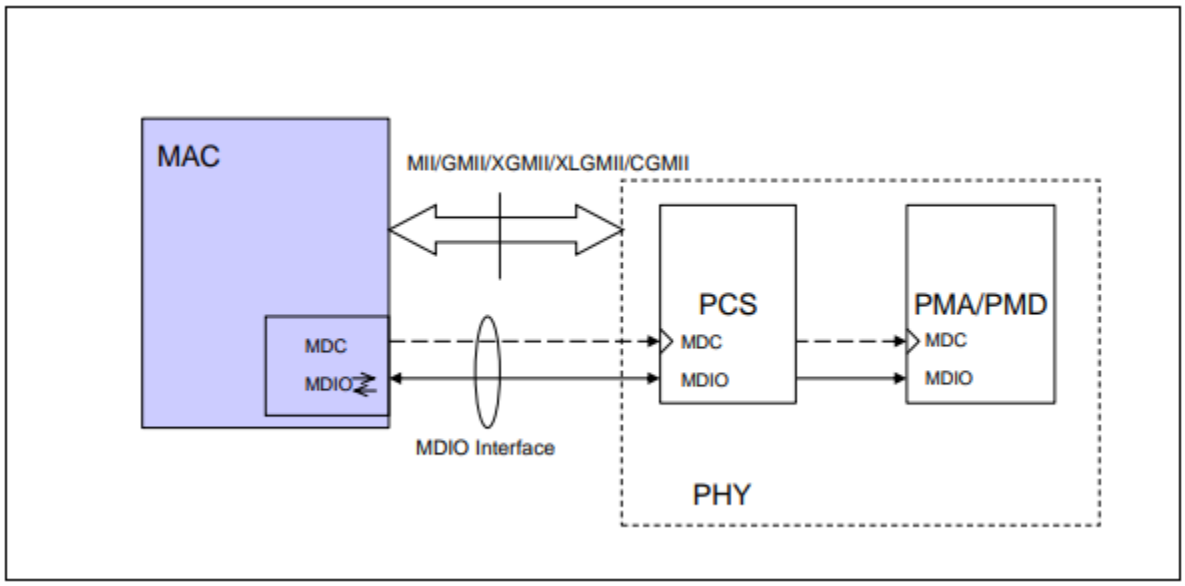
Hardware Concepts

By Jani Ahamed Habeeb Mohamed

Table of Contents

1. MDIO

Medium independent interface for all speeds ranging from 10Mbps to 100Gbps
Two wire management interface defined to connect MAC with PHY providing a standardized way to access internal registers of PHY.
Shared bus architecture that can connect upto 32 devices. All data is transferred clock synchronously to the Management Data Clock (MDC) which is provided by the MAC and sourced to all receiving devices.
Data line is driven by MAC for write and by PHY for read transactions.



802.3ae defines in Clause 45 an extension to Clause 22 MDIO management interface to enable more than 32 registers per PHY device.
MDIO (MDC) speed is 2.5 GHz.
MAC layer is the MDIO master

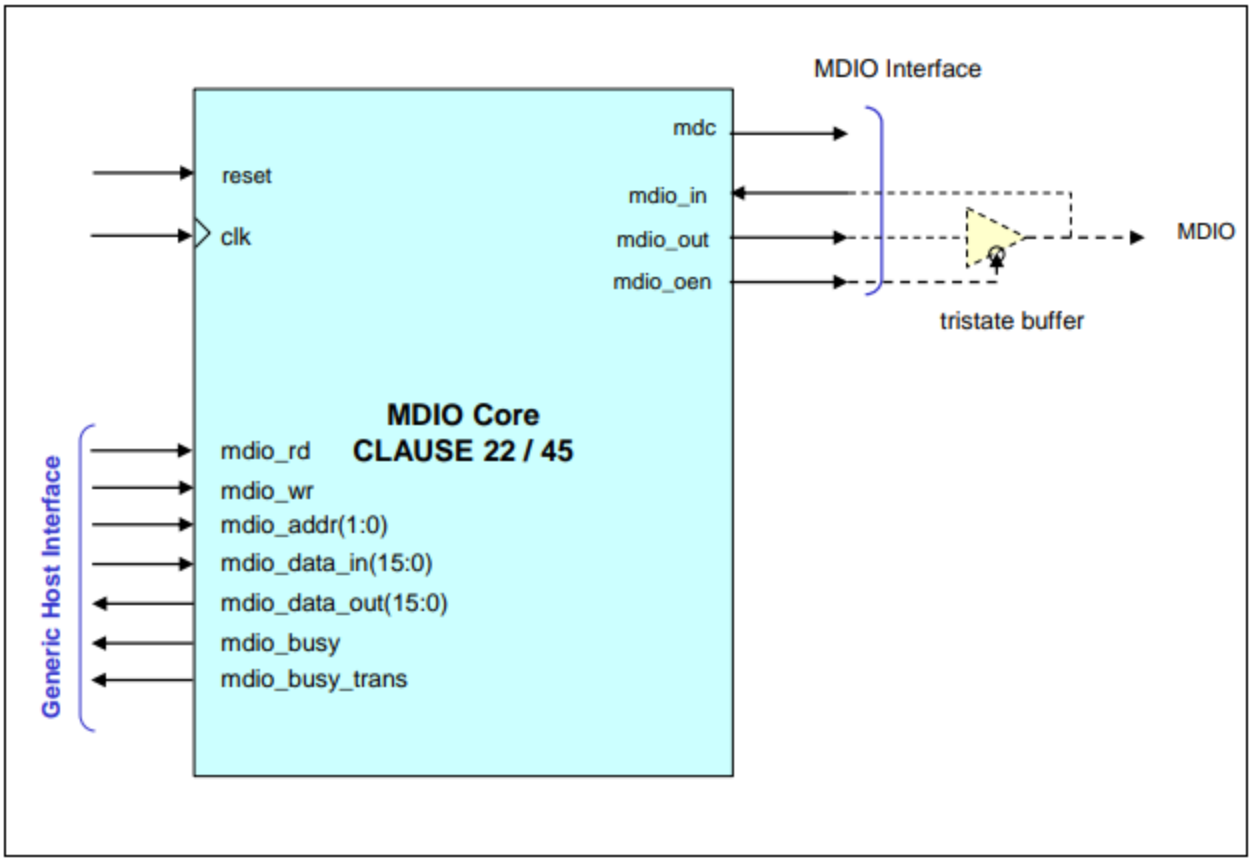


Figure 2: MDIO Core Pinout

3.1 Signals Description

Table 1: Signals Description

Signal Name	*Mode*	*Description*
		Global Signals
reset	In	Active High Reset for clk clock domain
		Application Interface Signals
clk	In	Register Access Reference Clock.
mdio_wr	In	Register Write Enable.
mdio_rd	In	Register Read Enable.
mdio_addr(1:0)	In	Register Address. Bit 0 is the least significant bit.
mdio_data_in(15:0)	In	Register Write Data. Bit 0 is the least significant bit.

mdio_data_out(15:0)	Out	Register Read Data. Bit 0 is the least significant bit.
mdio_busy	Out	Register interface busy. Asserted ('1') during register read or register write access. Set to '0' to indicate the completion of the current register access.
mdio_busy_trans	Out	MDIO transaction interface busy. Asserted ('1') if there is an ongoing transaction on PHY Management Interface. Set to '0' when there is no ongoing transaction on PHY Management Interface.
PHY Management Interface		
mdio_in	In	Serial data input. Serial data is shifted in on mdc rising edge.
mdio_out	Out	Serial data output. Serial data is shifted out on mdc rising edge.
mdio_oen	Out	Serial data output enable (active low). When asserted ('0') the mdio_out must drive the MDIO bus. When deasserted ('1') the output buffer must be tristated, therefore `mdio_out` not driving the MDIO bus.
mdc	Out	PHY Management interface clock. A data bit is shifted in/out on each rising edge of mdc.

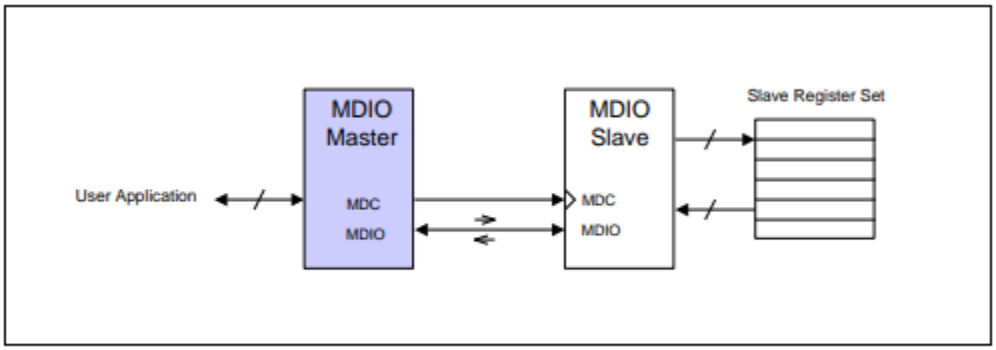


Figure 3: MDIO Cores Overview

Two wire management interface.
Clause 22 for PHY 1Gbps or less
Clause 45 for PHY above 1GBps
The PHY device distinguishes between Clause 22 and 45 via start sequence.
Clause 45 has a relative addressing scheme which offers to address upto 64K registers in the PHY.

4.1 Clause 22 MDIO Frame Format

A complete frame has a length of 64 bits (32 bit preamble, 14 bit command, 2 bit bus direction change, 16 bit data). Each bit is transferred with the rising edge of the MDIO clock (MDC signal).

Table 2: MDIO Frame Formats (Read / Write)

Type		***Command***						
	PRE	**ST** MSB LSB	**OP** MSB LSB	**Addr1** MSB . LSB	**Addr2** MSB . LSB	**TA**	**Data** MSB LSB	**Idle**
Read	1 … 1	01	10	xxxxx	xxxxx	Z0	xxxxxxxxxxxxxxxx	Z
Write	1 … 1	01	01	xxxxx	xxxxx	10	xxxxxxxxxxxxxxxx	Z

Table 3: Clause 22 MDIO Frame Fields Description

Name	***Description***
PRE	Preamble: 32 Bits of logical '1' sent prior to every transaction
ST	Start indication: '01'
OP	The opcode defines whether a read or write operation is performed: • If '10' a read operation is performed. • If '01' a write operation is performed.
Addr1	The PHY device address (PHYAD). Up to 32 devices can be addressed.
Addr2	Register Address. Each PHY can implement up to 32 registers.
TA	Turnaround time. Two bit-times are reserved for read operations to switch the data bus from write to read for read operations. The PHY device will present its register contents in the data phase and drives the bus from the 2nd bit of the turnaround phase.
Data	16 bits of data written to the PHY or read from the PHY.
Idle	Between frames, the MDIO data signal is tri-stated.

4.2 Clause 45 MDIO Frame Format

The extended MDIO frame structure introduces indirect addressing. First, a write transaction to a address register is done, followed by a write or read transaction which will put the 16 bit data in the register or retrieve the register contents respectively.

The extended MDIO defines four transactions.

Table 4: Clause 45 MDIO Frame Structure

Type	PRE (32)	ST (2) MSB LSB	OP (2) MSB LSB	PRTAD (5) MSB . LSB	DEVAD (5) MSB . LSB	TA (2)	Data (16) MSB LSB	Idle (n)
Address	1 ... 1	00	00	xxxxx	xxxxx	10	16 bit address	Z
Write	1 ... 1	00	01	xxxxx	xxxxx	10	xxxxxxxxxxxxxxx x	Z
Read	1 ... 1	00	11	xxxxx	xxxxx	Z0	xxxxxxxxxxxxxxx x	Z
Read inc.	1 ... 1	00	10	xxxxx	xxxxx	Z0	xxxxxxxxxxxxxxx x	Z

All bits are transmitted from left to right (Preamble bits first) and all fields have their Most-Significant bit sent first (leftmost in above table). The complete frame has a length of 64 bits (32 bit preamble, 14 bit command, 2 bit bus direction change, 16 bit data). Each bit is transferred with the rising edge of the MDIO clock (MDC).

The fields and transactions are summarized in the following tables.

Table 5: MDIO Frame Fields

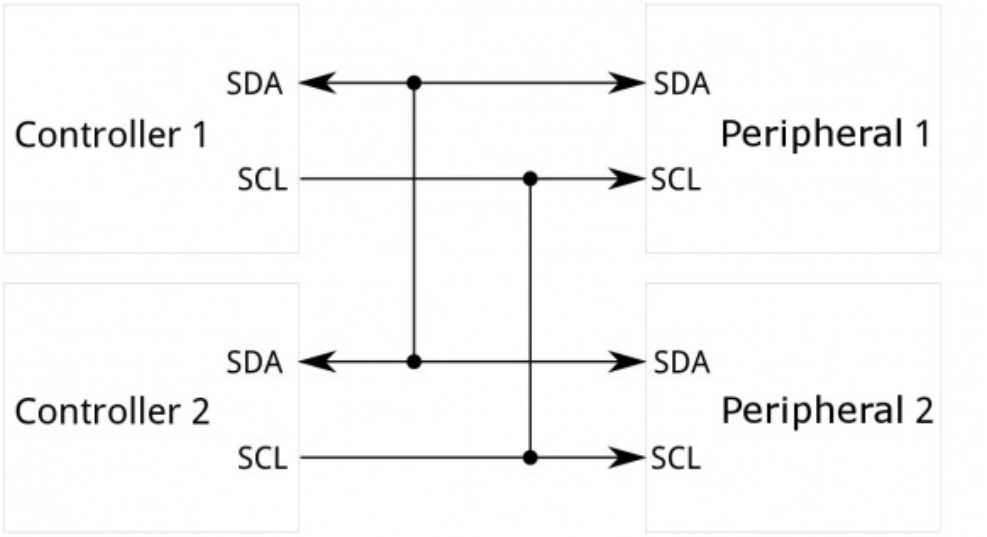
Name	Description
PRE	Preamble: 32 Bits of logical '1' sent prior to every transaction
ST	Start indication: indicates the end of the preamble and start of the frame.
OP	The Opcode defines whether a read or write operation is performed: • If bit 1 is set '1' a read operation is performed. • If bit 1 is set '0' a write operation is performed.
PRTAD	The port address specifies a MDIO port. Each Port can have up to 32 devices which

	each can have a separate set of registers.
DEVAD	Device address. Up to 32 devices can be addressed (within a port).
TA	Turnaround time. Two bit-times are reserved for read operations to switch the data bus from write to read. The PHY device will present its register contents in the data phase and drives the bus from the 2nd bit of the turnaround phase.
Data	16 bits of data written to the PHY or read from the PHY.
Idle	During idle, the MDIO data signal (MDIO) is tri-stated.

Table 6: MDIO Transactions

Transaction Type	Description
Address	A write transaction to the internal address register of the device/port. The data section of the frame contains the value to be stored in the device's internal address "pointer" register for further transactions.
Write	Data write to a register. The 16 bit data will be written to the register identified by the device-internal address.
Read	Data is read from the register identified by the device-internal address.
Read inc.	Read with address postincrement. The register identified by the device-internal address is read. After this, the device-internal address is incremented. If the address register is all '1' (0xFFFF) no increment is done (i.e. increment does not wrap around).

2. I2C



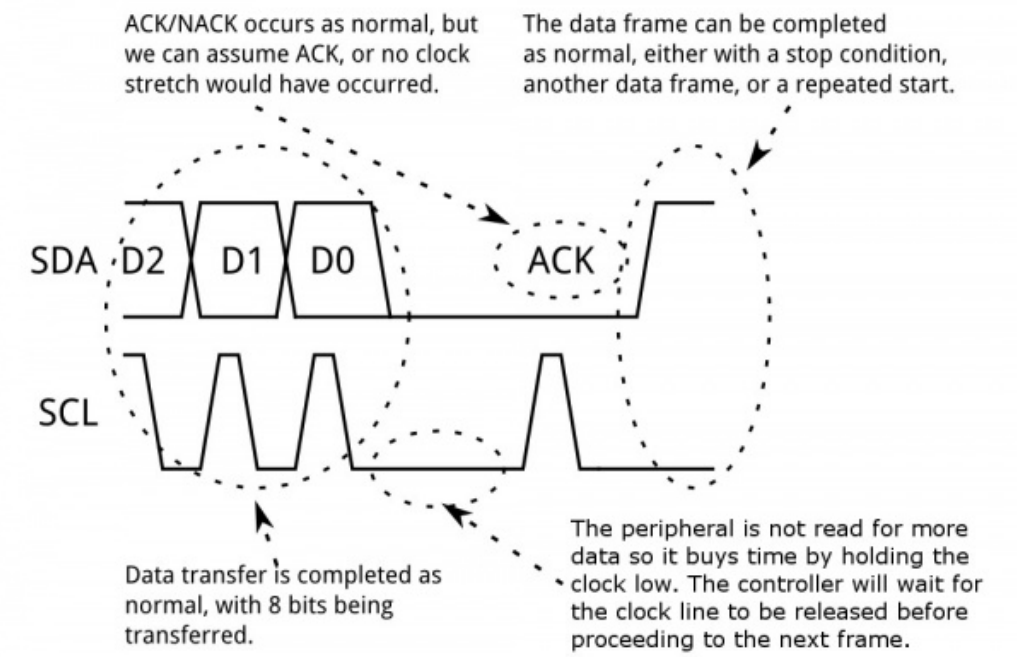
The Inter-Integrated Circuit (I^2C) Protocol is a protocol intended to allow multiple "peripheral" digital integrated circuits ("chips") to communicate with one or more "controller" chips. Like the Serial Peripheral Interface (SPI), it is only intended for short distance communications within a single device. Like Asynchronous Serial Interfaces (such as RS-232 or UARTs), it only requires two signal wires to exchange information.

I^2C requires a mere two wires, like asynchronous serial, but those two wires can support up to 1008 peripheral devices. Also, unlike SPI, I^2C can support a multi-controller system, allowing more than one controller [1] to communicate with all peripheral [1] devices on the bus (although the controller devices can't talk to each other over the bus and must take turns using the bus lines).

Data rates fall between asynchronous serial and SPI; most I^2C devices can communicate at 100kHz or 400kHz. There is some overhead with I^2C; for every 8 bits of data to be sent, one extra bit of meta data (the "ACK/NACK" bit, which we'll discuss later) must be transmitted.

Clock Stretching

At times, the controller's data rate will exceed the peripheral's ability to provide that data. This can be because the data isn't ready yet (for instance, the peripheral hasn't completed an analog-to-digital conversion yet) or because a previous operation hasn't yet completed (say, an EEPROM which hasn't completed writing to non-volatile memory yet and needs to finish that before it can service other requests).



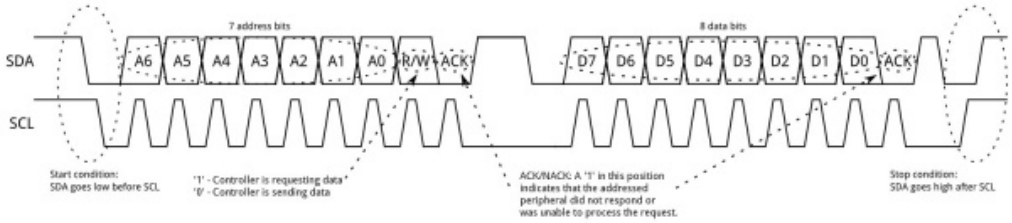
In this case, some peripheral devices will execute what is referred to as "clock stretching". Nominally, **all** clocking is driven by the controller — peripherals simply put data on the bus or take data off the bus in response to the controller's clock pulses. At any point in the data transfer process, an addressed peripheral can hold the SCL line low after the controller releases it. The controller is required to refrain from additional clock pulses or data transfer until such time as the peripheral releases the SCL line.

Protocol

Communication via I^2C is more complex than with a UART or SPI solution. The signalling must adhere to a certain protocol for the devices on the bus to recognize it as valid I^2C communications. Fortunately, most devices take care of all the fiddly details for you, allowing you to concentrate on the data you wish to exchange.

Basics

Messages are broken up into two types of frame: an address frame, where the controller indicates the peripheral to which the message is being sent, and one or more data frames, which are 8-bit data messages passed from controller to peripheral or vice versa. Data is placed on the SDA line after SCL goes low, and is sampled after the SCL line goes high. The time between clock edge and data read/write is defined by the devices on the bus and will vary from chip to chip.



Click on image for a closer view.

Start Condition

To initiate the address frame, the controller device leaves SCL high and pulls SDA low. This puts all peripheral devices on notice that a transmission is about to start. If two controllers wish to take ownership of the bus at one time, whichever device pulls SDA low first wins the race and gains control of the bus. It is possible to issue repeated starts, initiating a new communication sequence without relinquishing control of the bus to other controller(s); we'll talk about that later.

Address Frame

The address frame is always first in any new communication sequence. For a 7-bit address, the address is clocked out most significant bit (MSB) first, followed by a R/W bit indicating whether this is a read (1) or write (0) operation.

The 9th bit of the frame is the NACK/ACK bit. This is the case for all frames (data or address). Once the first 8 bits of the frame are sent, the receiving device is given control over SDA. If the receiving device does not pull the SDA line low before the 9th clock pulse, it can be inferred that the receiving device either did not receive the data or did not know how to parse the message. In that case, the exchange halts, and it's up to the controller of the system to decide how to proceed.

Data Frames

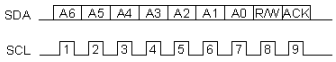
After the address frame has been sent, data can begin being transmitted. The controller will simply continue generating clock pulses at a regular interval, and the data will be placed on SDA by either the controller or the peripheral, depending on whether the R/W bit indicated a read or write operation. The number of data frames is arbitrary, and most peripheral devices will auto-increment the internal register, meaning that subsequent reads or writes will come from the next register in line.

Stop condition

Once all the data frames have been sent, the controller will generate a stop condition. Stop conditions are defined by a 0->1 (low to high) transition on SDA *after* a 0->1 transition on SCL, with SCL remaining high. During normal data writing operation, the value on SDA should **not** change when SCL is high, to avoid false stop conditions.

This is just two wires, called SCL and SDA. SCL is the clock line. It is used to synchronize all data transfers over the I2C bus. SDA is the data line. The SCL & SDA lines are connected to all devices on the I2C bus. There needs to be a third wire which is just the ground or 0 volts. There may also be a 5volt wire is power is being distributed to the devices. Both SCL and SDA lines are "open drain" drivers. What this means is that the chip can drive its output low, but it cannot drive it high. For the line to be able to go high you must provide pull-up resistors to the 5v supply. There should be a resistor from the SCL line to the 5v line and another from the SDA line to the 5v line. You only need one set of pull-up resistors for the whole I2C bus, not for each device, as illustrated below:

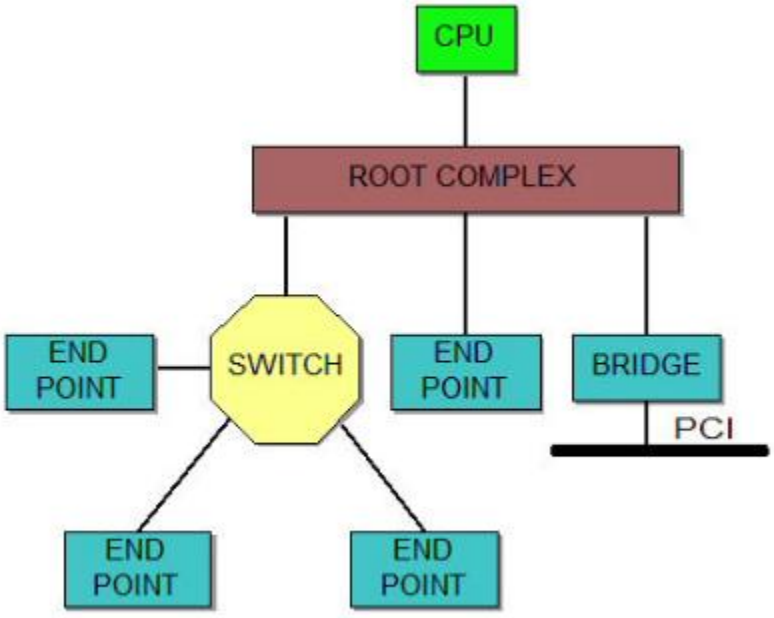
All I2C addresses are either 7 bits or 10 bits. The use of 10 bit addresses is rare and is not covered here. All of our modules and the common chips you will use will have 7 bit addresses. This means that you can have up to 128 devices on the I2C bus, since a 7bit number can be from 0 to 127. When sending out the 7 bit address, we still always send 8 bits. The extra bit is used to inform the slave if the master is writing to it or reading from it. If the bit is zero the master is writing to the slave. If the bit is 1 the master is reading from the slave. The 7 bit address is placed in the upper 7 bits of the byte and the Read/Write (R/W) bit is in the LSB (Least Significant Bit).



The placement of the 7 bit address in the upper 7 bits of the byte is a source of confusion for the newcomer. It means that to write to address 21, you must actually send out 42 which is 21 moved over by 1 bit. It is probably easier to think of the I2C bus addresses as 8 bit addresses, with even addresses as write only, and the odd addresses as the read address for the same device. To take our CMPS03 for example, this is at address 0xC0 ($C0). You would uses 0xC0 to write to the CMPS03 and 0xC1 to read from it. So the read/write bit just makes it an odd/even address.

3. PCIe

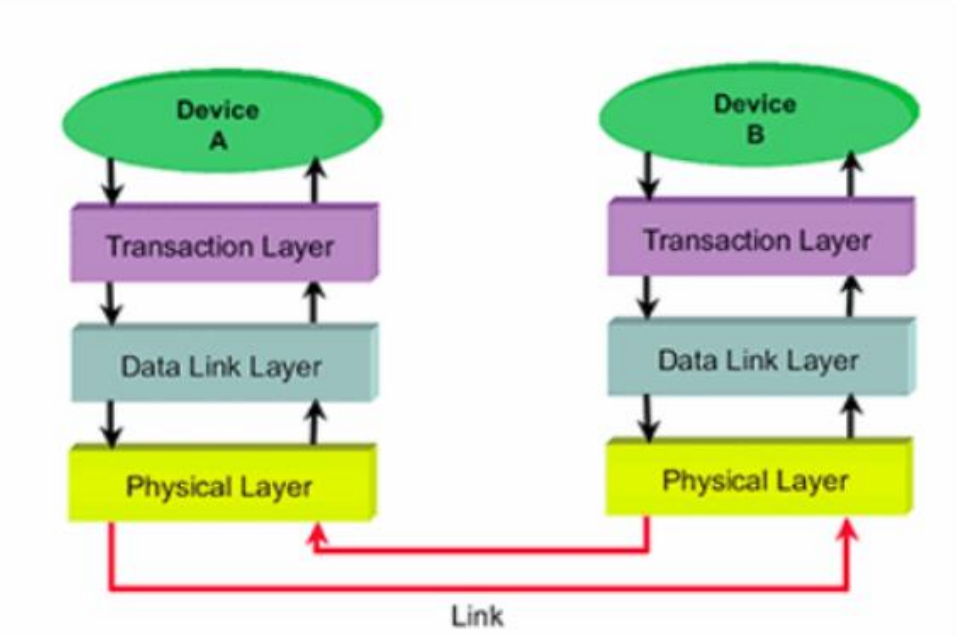
PCI express is a serial point to point link that operates at 2.5Gbps and higher in each direction. Point to point and there is no arbitration for resources on the link



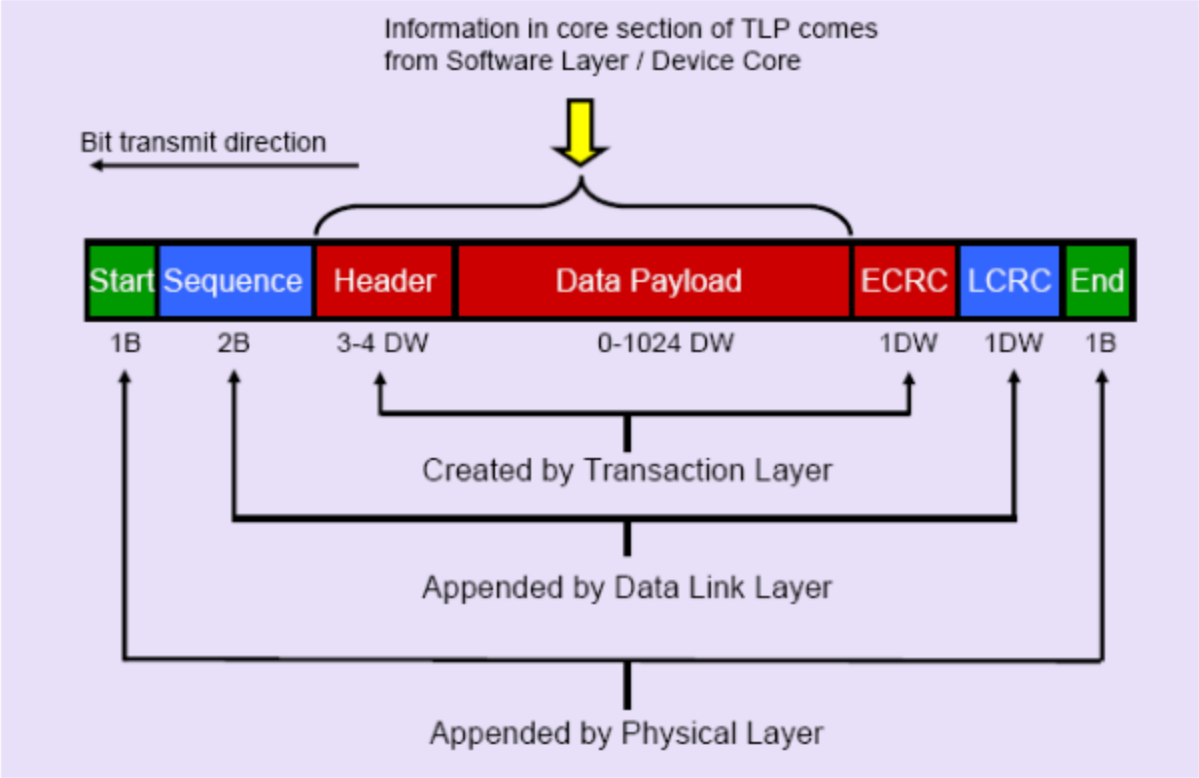
As can be seen in the figure below, a PCI Express fabric consists of three types of devices: the root complex, switches, and endpoints. The root complex is generally associated with the processor and is responsible for configuring the fabric at power-up. Since PCIe connections are point to point, switches are used to expand the fabric. PCIe endpoints are the I/O devices in the fabric - the sources of, and destinations for the data.

PCI Express Layers

PCIe is implemented in three of the OSI model layers: the transaction layer, the data link layer, and the physical layer. The following figure displays the layers as connected between two PCIe devices.



As can be seen in the figure, the user logic interfaces to the transaction layer. The user forms Transaction Layer Packets, or TLPs which contain a header, data payload, and optionally an end-to-end CRC, ECRC. The ECRC, if used, is generated by the user logic at the transmitter and checked by the user logic at the receiver. The data link layer is responsible for link management including error detection. In this layer, a CRC (called the Link CRC or LCRC) is appended and a sequence number is prepended to the Transaction Layer Packet. When a packet is transmitted from the data link layer, the receiver sends back an ACK (success) or NACK (failure) to the transmitter which will retransmit in the case of an error. These ACKs and NACKs are sent via special packets which originate from the data link layer called Data Link Layer Packets, or DLLPs. The physical layer consists of two differential pairs with 8B/10B encoded data allowing for a DC balance on the transmission media and for clock recovery at the destination. Framing information is added to the data link layer packet, and this is encoded and driven onto the link. The following diagram displays the encapsulation of packets in PCIe:


Information in core section of TLP comes from Software Layer / Device Core
Bit transmit direction
Start
Sequence
Header
Data Payload
ECRC
LCRC
End
1B
2B
3-4 DW
0-1024 DW
1DW
1DW
1B
Created by Transaction Layer
Appended by Data Link Layer
Appended by Physical Layer

4. DPDK

Linux network stack performance has become increasingly relevant over the past few years. This is perfectly understandable: the amount of data that can be transferred over a network and the corresponding workload has been growing not by the day, but by the hour.

Not even the widespread use of 10 GE network cards has resolved this issue; this is because a lot of bottlenecks that prevent packets from being quickly processed are found in the Linux kernel itself.

There have been many attempts to circumvent these bottlenecks with techniques called kernel bypasses (a short description can be found here). They let you process packets without involving the Linux network stack and make it so that the application running in the user space communicates directly with the networking device. We'd like to discuss one of these solutions, the Intel DPDK (Data Plane Development Kit), in today's article.

A lot of posts have already been published about the DPDK and in a variety of languages. Although many of these are fairly informative, they don't answer the most important questions: How does the DPDK process packets and what route does the packet take from the network device to the user?

Finding the answers to these questions was not easy; since we couldn't find everything we needed in the official documentation, we had to look through a myriad of additional materials and thoroughly review their sources. But first thing's first: before talking about the DPDK and the issues it can help resolve, we should review how packets are processed in Linux.

Processing Packets in Linux: Main Stages

When a network card first receives a packet, it sends it to a receive queue or RX. From there, it gets copied to the main memory via the DMA (Direct Memory Access) mechanism.

Afterwards, the system needs to be notified of the new packet and pass the data onto a specially allocated buffer (Linux allocates these buffers for every packet). To do this, Linux uses an interrupt mechanism: an interrupt is generated several times when a new packet enters the system. The packet then needs to be transferred to the userspace.

One bottleneck is already apparent: As more packets have to be processed, more resources are consumed, which negatively affects the overall system performance.

As we've already said, these packets are saved to specially allocated buffers — more specifically, the *sk_buff struct*. This struct is allocated for each packet and becomes free when a packet enters the userspace. This operation consumes a lot of bus cycles (i.e. cycles that transfer data from the CPU to the main memory).

S**ynopsis (struct sk_buff — socket buffer)**

```
struct sk_buff {
  union {unnamed_union};
```

```
  __u16 inner_transport_header;
  __u16 inner_network_header;
  __u16 inner_mac_header;
  __be16 protocol;
  __u16 transport_header;
  __u16 network_header;
  __u16 mac_header;
  sk_buff_data_t tail;
  sk_buff_data_t end;
  unsigned char * head;
  unsigned char * data;
  unsigned int truesize;
  atomic_t users;
};
```

There is another problem with the *sk_buff* struct: the Linux network stack was originally designed to be compatible with as many protocols as possible. As such, metadata for all of these protocols is included in the *sk_buff* struct, but that's simply **not necessary for processing specific packets**. Because of this overly complicated struct, the processing is slower than it could be.

Another factor that negatively affects performance is context switching. When an application in the user space needs to send or receive a packet, it executes a system call. The context is switched to kernel mode and then back to user mode. This consumes a significant amount of system resources.

To solve some of these problems, all Linux kernels since version 2.6 have included NAPI (New API), which combines interrupts with requests. Let's take a quick look at how this works.

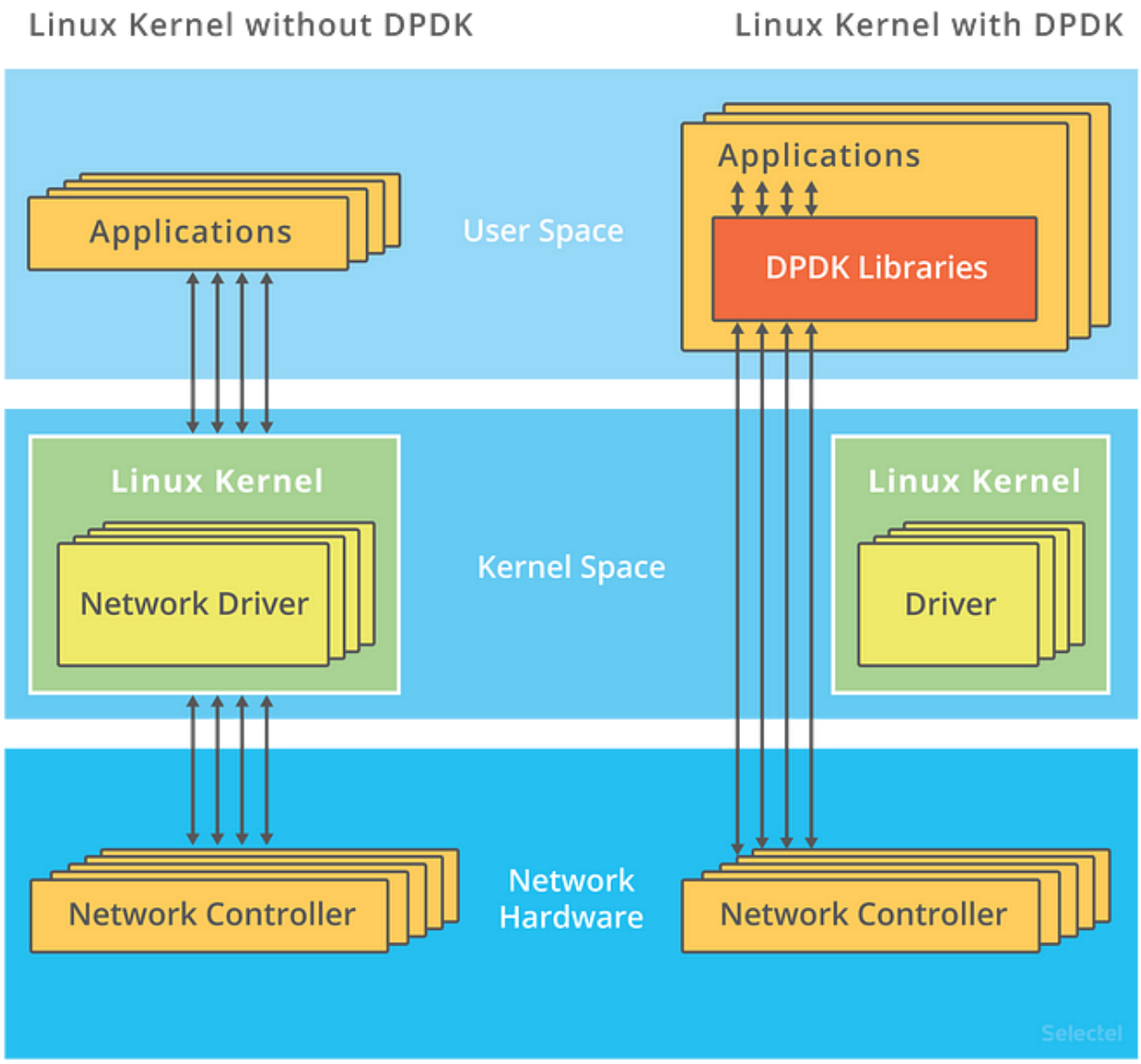
The network card first works in interrupt mode, but as soon as a packet enters the network interface, it registers itself in a poll queue and disables the interrupt. The system periodically checks the queue for new devices and gathers packets for further processing. As soon as the packets are processed, the card will be deleted from the queue and interrupts are again enabled.

This has been just a cursory description of how packets are processed. A more detailed look at this process can be found in an article series from Private Internet Access. However, even a quick glance is enough to see the problems slowing down packet processing. In the next section, we'll describe how these problems are solved using DPDK.

DPDK: How It Works

General Features

Let's look at the following illustration:



[source: https://blog.selectel.com/wp-content/uploads/2016/11/PR-3303.png]

On the left, you see the traditional way packets are processed, and on the right–with DPDK. As we can see, the kernel in the second example doesn't step in at all: interactions with the network card are performed via special drivers and libraries.

If you've already read about DPDK or have ever used it, then you know that the ports receiving incoming traffic on network cards need to be unbound from Linux (the kernel driver). This is done using the dpdk_nic_bind (or dpdk-devbind) command, or ./dpdk_nic_bind.py in earlier versions.

How are ports then managed by DPDK? Every driver in Linux has bind and unbind files. That includes network card drivers:

```
ls /sys/bus/pci/drivers/ixgbe
bind  module  new_id  remove_id  uevent  unbind
```

To unbind a device from a driver, the device's bus number needs to be written to the unbind file. Similarly, to bind a device to another driver, the bus number needs to be written to its bind file. More detailed information about this can be found here.

The DPDK installation instructions tell us that our ports need to be managed by the vfio_pci, igb_uio, or uio_pci_generic driver. (We won't be getting into details here, but we suggested interested readers look at the following articles on kernel.org: 1 and 2.)

These drivers make it possible to interact with devices in the user space. Of course they include a kernel module, but that's just to initialize devices and assign the PCI interface.

All further communication between the application and network card is organized by the DPDK poll mode driver (PMD) DPDK has poll mode drivers for all supported network cards and virtual devices.

The DPDK also requires hugepages to be configured. This is required for allocating large chunks of memory and writing data to them. We can say that hugepages does the same job in DPDK that DMA does in traditional packet processing.

We'll discuss all of its nuances in more detail, but for now, let's go over the main stages of packet processing with the DPDK:

1. Incoming packets go to a ring buffer (we'll look at its setup in the next section). The application periodically checks this buffer for new packets.
2. If the buffer contains new packet descriptors, the application will refer to the DPDK packet buffers in the specially allocated memory pool using the pointers in the packet descriptors.
3. If the ring buffer does not contain any packets, the application will queue the network devices under the DPDK and then refer to the ring again.

Let's take a closer look at the DPDK's internal structure.

EAL: Environment Abstraction

The EAL, or Environment Abstraction Layer, is the main concept behind the DPDK.

The EAL is a set of programming tools that let the DPDK work in a specific hardware environment and under a specific operating system. In the official DPDK repository, libraries and drivers that are part of the EAL are saved in the rte_eal directory.

Drivers and libraries for Linux and the BSD system are saved in this directory. It also contains a set of header files for various processor architectures: ARM, x86, TILE64, and PPC64.

We access software in the EAL when we compile the DPDK from the source code:

```
make config T=x86_64-native-linuxapp-gcc
```

One can guess that this command will compile DPDK for Linux in an x86_64 architecture.

The EAL is what binds the DPDK to applications. All of the applications that use the DPDK (see here for examples) must include the EAL's header files.

The most commonly of these include:

- rte_lcore.h — manages processor cores and sockets;
- rte_memory.h — manages memory;
- rte_pci.h — provides the interface access to PCI address space;
- rte_debug.h — provides trace and debug functions (logging, dump_stack, and more);
- rte_interrupts.h — processes interrupts.

More details on this structure and EAL functions can be found in the official documentation.

Managing Queues: rte_ring

As we've already said, packets received by the network card are sent to a ring buffer, which acts as a receiving queue. Packets received in the DPDK are also sent to a queue implemented on the rte_ring library. The library's description below comes from information gathered from the developer's guide and comments in the source code.

The rte_ring was developed from the FreeBSD ring buffer. If you look at the source code, you'll see the following comment: Derived from FreeBSD's bufring.c.

The queue is a lockless ring buffer built on the FIFO (First In, First Out) principle. The ring buffer is a table of pointers for objects that can be saved to the memory. Pointers can be divided into four categories: prod_tail, prod_head, cons_tail, cons_head.

Prods is short for producer, and cons for consumer. The **producer** is the process that writes data to the buffer at a given time, and the consumer is the process that removes data from the buffer.

The **tail** is where writing takes place on the ring buffer. The place the buffer is read from at a given time is called the **head**.

The idea behind the process for adding and removing elements from the queue is as follows: when a new object is added to the queue, the ring->prod_tail indicator should end up pointing to the location where ring->prod_head previously pointed to.

This is just a brief description; a more detailed account of how the ring buffer scripts work can be found in the developer's manual on the DPDK site.

This approach has a number of advantages. Firstly, data is written to the buffer extremely quickly. Secondly, when adding or removing a large number of objects from the queue, cache misses occur much less frequently since pointers are saved in a table.

The drawback to DPDK's ring buffer is its fixed size, which cannot be increased on the fly. Additionally, much more memory is spent working with the ring structure than in a linked queue since the buffer always uses the maximum number of pointers.

Memory Management: rte_mempool

We mentioned above that DPDK requires hugepages. The installation instructions recommend creating 2MB hugepages.

These pages are combined into segments, which are then divided into zones. Objects that are created by applications or other libraries, like queues and packet buffers, are placed in these zones.

These objects include memory pools, which are created by the rte_mempool library. These are fixed size object pools that use rte_ring for storing free objects and can be identified by a unique name.

Memory alignment techniques can be implemented to improve performance.

Even though access to free objects is designed on a lockless ring buffer, consumption of system resources may still be very high. As multiple cores have access to the ring, a compare-and-set (CAS) operation usually has to be performed each time it is accessed.

To prevent bottlenecking, every core is given an additional local cache in the memory pool. Using the locking mechanism, cores can fully access the free object cache. When the cache is full or entirely empty, the memory pool exchanges data with the ring buffer. This gives the core access to frequently used objects.

Buffer Management: rte_mbuf

In the Linux network stack, all network packets are represented by the the sk_buff data structure. In DPDK, this is done using the rte_mbuf struct, which is described in the rte_mbuf.h header file.

The buffer management approach in DPDK is reminiscent of the approach used in FreeBSD: instead of one big sk_buff struct, there are many smaller rte_mbuf buffers. The buffers are created before the DPDK application is launched and are saved in memory pools (memory is allocated by rte_mempool).

In addition to its own packet data, each buffer contains metadata (message type, length, data segment starting address). The buffer also contains pointers for the next buffer. This is needed when handling packets with large amounts of data. In cases like these, packets can be combined (as is done in FreeBSD; more detailed information about this can be found here).

Other Libraries: General Overview

In previous sections, we talked about the most basic DPDK libraries. There's a great deal of other libraries, but one article isn't enough to describe them all. Thus, we'll be limiting ourselves to just a brief overview.

With the LPM library, DPDK runs the Longest Prefix Match (LPM) algorithm, which can be used to forward packets based on their IPv4 address. The primary function of this library is to add and delete IP addresses as well as to search for new addresses using the LPM algorithm.

A similar function can be performed for IPv6 addresses using the LPM6 library.

Other libraries offer similar functionality based on hash functions. With rte_hash, you can search through a large record set using a unique key. This library can be used for classifying and distributing packets, for example.

The rte_timer library lets you execute functions asynchronously. The timer can run once or periodically.

Conclusion

In this article, we went over the internal device and principles of DPDK. This is far from comprehensive though; the subject is too complex and extensive to fit into one article. So sit tight, we will continue this topic in a future article, where we'll discuss the practical aspects of using DPDK.

We'd be happy to answer your questions in the comments below. And if you've had any experience using DPDK, we'd love to hear your thoughts and impressions.

For anyone interested in learning more, please visit the following links:

- http://dpdk.org/doc/guides/prog_guide/ — a detailed (but confusing in some places) description of all the DPDK libraries;
- https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2014-08-1/NET-2014-08-1_15.pdf — a brief overview of DPDK's capabilities and comparison with other frameworks (netmap and PF_RING);
- http://www.slideshare.net/garyachy/dpdk-44585840 — an introductory presentation to DPDK for beginners;
- http://www.it-sobytie.ru/system/attachments/files/000/001/102/original/LinuxPiter-DPDK-2015.pdf — a presentation explaining the DPDK structure.

The term "RETA" in the context of DPDK (Data Plane Development Kit) stands for "Redirection Table." The Redirection Table is a key component of the Receive Side Scaling (RSS) feature in DPDK. RSS is a mechanism used to distribute incoming network traffic among multiple processing cores in a symmetric manner, improving the parallelism and scalability of packet processing.

The Redirection Table (RETA) in DPDK is used to determine the core to which a received packet should be directed based on certain criteria. This criteria might include fields in the packet header, such as the source and destination IP addresses or port numbers. The Redirection Table is a configurable data structure that allows you to define how the incoming packets should be distributed across the available processing cores.

Here's a brief overview of how it works:

Initialization:

During DPDK initialization, the application can configure the Redirection Table.
RSS Hash Calculation:

When a packet is received, the RSS hash function calculates a hash value based on the specified criteria (e.g., packet header fields).
Lookup in Redirection Table:

The calculated hash value is then used as an index into the Redirection Table.
Core Assignment:

The value stored in the Redirection Table at the calculated index determines the target processing core to which the packet should be directed.
By configuring the Redirection Table, you can control how the hash values map to processing cores, influencing the distribution of network traffic across the available cores.

It's worth noting that DPDK evolves, and features and functionalities may be updated or extended in newer versions. For the most accurate and detailed information, it's recommended to refer to the official DPDK documentation or the specific documentation for the version of DPDK you are working with.

User
what is the purpose of rte_eth_dev_filter_ctrl
ChatGPT
In DPDK (Data Plane Development Kit), rte_eth_dev_filter_ctrl is a function used to control or configure specific filtering operations on a network interface controller (NIC). This function provides a generic interface for managing different types of filters on a given Ethernet device.

Here's a brief explanation of its purpose:

Purpose:
Filtering Operations:

DPDK supports various filtering operations to control the flow of packets. This includes functionalities such as adding or removing MAC address filters, configuring VLAN filters, and other types of filters.
Device Abstraction:

rte_eth_dev_filter_ctrl provides a consistent and abstracted way to interact with different types of filters across various NICs. This abstraction helps in writing DPDK applications that can work seamlessly with different types of network devices.
Dynamic Configuration:

Some filtering operations may require dynamic configuration based on runtime conditions. This function allows an application to dynamically control and adjust the filter settings on an Ethernet device.
Function Signature:
c
Copy code
```
int rte_eth_dev_filter_ctrl(uint16_t port_id, enum rte_filter_type filter_type,
                            enum rte_filter_op filter_op, void *arg);
```
Parameters:
port_id: The identifier of the Ethernet device.
filter_type: Type of filter to be controlled (e.g., MAC address, VLAN, etc.).
filter_op: Operation to be performed on the filter (e.g., add, delete).

arg: A pointer to a structure or data associated with the filter operation.
Example Usage:
Here's a hypothetical example to illustrate how rte_eth_dev_filter_ctrl might be used to add or remove a MAC address filter:

c
Copy code
```
struct rte_eth_dev_info dev_info;
struct rte_eth_dev_addr mac_addr;

// Get device information
rte_eth_dev_info_get(port_id, &dev_info);

// Assume 'mac_addr' contains the MAC address to be added or removed
int ret = rte_eth_dev_filter_ctrl(port_id, RTE_ETH_FILTER_MACVLAN,
                                  RTE_ETH_FILTER_ADD, &mac_addr);
if (ret < 0) {
    // Handle error
}
```
This example assumes the use of MAC address filtering (RTE_ETH_FILTER_MACVLAN), and it attempts to add the specified MAC address to the device's filter list.

Keep in mind that the specific usage may vary based on the type of filter and the requirements of your DPDK application. Always refer to the DPDK documentation for the version you are using for the most accurate and up-to-date information.

User
give example for reta
ChatGPT
To provide a practical example of using the Receive Side Scaling (RSS) feature and the Redirection Table (RETA) in DPDK, we'll walk through a simplified code snippet. This example demonstrates how to configure RSS with a specific hash function and populate the RETA.

c
Copy code
```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <inttypes.h>
#include <rte_ethdev.h>
#include <rte_hash.h>

// Assume this is a constant for the number of processing cores.
#define NUM_RX_CORES 4

// Function to initialize RSS and RETA
void init_rss_and_reta(uint16_t port_id) {
    struct rte_eth_conf port_conf = { 0 };
    struct rte_eth_rss_conf rss_conf = { 0 };

    // Initialize Ethernet port with RSS and RETA configurations.
    rte_eth_dev_configure(port_id, NUM_RX_CORES, NUM_RX_CORES, &port_conf);

    // Configure RSS hash function.
    rss_conf.rss_key_len = 40;
```

```
    rte_eth_dev_rss_hash_conf_get(port_id, &rss_conf);
    rte_eth_dev_rss_hash_function_set(port_id, &rss_conf);

    // Configure RETA (Redirection Table).
    struct rte_eth_rss_reta_entry64 reta_conf[NUM_RX_CORES];
    memset(reta_conf, 0, sizeof(reta_conf));

    // Assume we want to evenly distribute traffic among NUM_RX_CORES cores.
    for (uint16_t i = 0; i < NUM_RX_CORES; ++i) {
        uint16_t queue_idx = i % RTE_MAX_QUEUES_PER_PORT;
        reta_conf[i / RTE_RETA_GROUP_SIZE].mask |= 1ULL << (i % RTE_RETA_GROUP_SIZE);
        reta_conf[i / RTE_RETA_GROUP_SIZE].reta[i % RTE_RETA_GROUP_SIZE] = queue_idx;
    }

    // Set the RETA configuration.
    rte_eth_dev_rss_reta_update(port_id, reta_conf, NUM_RX_CORES);
}

int main() {
    uint16_t port_id = 0;  // Assume we are working with port 0.

    // Initialize DPDK and Ethernet device.
    rte_eal_init(argc, argv);
    rte_eth_dev_init();

    // Initialize RSS and RETA for the specified port.
    init_rss_and_reta(port_id);

    // Start receiving and processing packets on the specified port and cores.

    return 0;
}
```

Please note that this is a simplified example, and the actual initialization process may vary based on your specific requirements and DPDK version. Additionally, you might need to handle error checking and other configurations based on your application's needs.

In this example, we use rte_eth_dev_rss_hash_function_set to configure the hash function for RSS and rte_eth_dev_rss_reta_update to configure the RETA. The RETA configuration evenly distributes traffic among the specified number of processing cores. Make sure to refer to the DPDK documentation for the specific version you are using for more detailed information and options.

5. IGB

Overview
========
This driver supports kernel versions 2.6.30 or newer. However, some features may require a newer kernel version. The associated Virtual Function (VF) driver for this driver is igbvf.

Driver information can be obtained using ethtool, lspci, and ip. Instructions on updating ethtool can be found in the section Additional Configurations later in this document.

This driver is only supported as a loadable module at this time. Intel is not supplying patches against the kernel source to allow for static linking of the drivers.

For questions related to hardware requirements, refer to the documentation supplied with your Intel adapter. All hardware requirements listed apply to use with Linux.

The igb driver supports IEEE 1588 time stamping for kernels 2.6.30 and newer.

The igb driver supports 2.5 Gbps operating speed on 2500BASE-KX only for I354-based network connections.

To manually build the driver
----
1. Move the base driver tar file to the directory of your choice.
   For example, use '/home/username/igb' or '/usr/local/src/igb'.

2. Untar/unzip the archive, where <x.x.x> is the version number for the
   driver tar file:

   # tar zxf igb-<x.x.x>.tar.gz

3. Change to the driver src directory, where <x.x.x> is the version number
   for the driver tar:

   # cd igb-<x.x.x>/src/

4. Compile the driver module:

   # make install

   The binary will be installed as:
   /lib/modules/<KERNEL VER>/updates/drivers/net/ethernet/intel/igb/igb.ko

   The install location listed above is the default location. This may differ
   for various Linux distributions.

5. Load the module using the modprobe command.

   To check the version of the driver and then load it:

   # modinfo igb
   # modprobe igb [parameter=port1_value,port2_value]

   Alternately, make sure that any older igb drivers are removed from the

kernel before loading the new module:

rmmod igb; modprobe igb

6. Assign an IP address to the interface by entering the following,
where <ethX> is the interface name that was shown in dmesg after modprobe:

ip address add <IP_address>/<netmask bits> dev <ethX>

7. Verify that the interface works. Enter the following, where IP_address
is the IP address for another machine on the same subnet as the interface
that is being tested:

ping <IP_address>

Note: For certain distributions like (but not limited to) Red Hat Enterprise
Linux 7 and Ubuntu, once the driver is installed, you may need to update the
initrd/initramfs file to prevent the OS loading old versions of the igb driver.
For Red Hat distributions:
dracut --force

For Ubuntu:
update-initramfs -u

InterruptThrottleRate

Valid Range:
0=off
1=dynamic
3=dynamic conservative
<min_ITR>-<max_ITR>
Interrupt Throttle Rate controls the number of interrupts each interrupt
vector can generate per second. Increasing ITR lowers latency at the cost of
increased CPU utilization, though it may help throughput in some circumstances.
0 = Setting InterruptThrottleRate to 0 turns off any interrupt moderation
and may improve small packet latency. However, this is generally not
suitable for bulk throughput traffic due to the increased CPU utilization
of the higher interrupt rate.
1 = Setting InterruptThrottleRate to Dynamic mode attempts to moderate
interrupts per vector while maintaining very low latency. This can
sometimes cause extra CPU utilization. If planning on deploying igb
in a latency sensitive environment, this parameter should be considered.
<min_ITR>-<max_ITR> = 100-100000
Setting InterruptThrottleRate to a value greater or equal to <min_ITR>
will program the adapter to send at most that many interrupts
per second, even if more packets have come in. This reduces interrupt load
on the system and can lower CPU utilization under heavy load, but will
increase latency as packets are not processed as quickly.

NOTE: InterruptThrottleRate is NOT supported by 82542, 82543, or 82544-based
adapters.

LLI (Low Latency Interrupts)

LLI allows for immediate generation of an interrupt upon processing receive
packets that match certain criteria as set by the parameters described below.

LLI parameters are not enabled when Legacy interrupts are used. You must be using MSI or MSI-X (see cat /proc/interrupts) to successfully use LLI.

LLIPort

Valid Range: 0-65535
LLI is configured with the LLIPort command-line parameter, which specifies which TCP port should generate Low Latency Interrupts.
For example, using LLIPort=80 would cause the board to generate an immediate interrupt upon receipt of any packet sent to TCP port 80 on the local machine.
WARNING: Enabling LLI can result in an excessive number of interrupts/second that may cause problems with the system and in some cases may cause a kernel panic.

LLIPush

Valid Range: 0-1
LLIPush can be set to be enabled or disabled (default). It is most effective in an environment with many small transactions.
NOTE: Enabling LLIPush may allow a denial of service attack.

LLISize

Valid Range: 0-1500
LLISize causes an immediate interrupt if the board receives a packet smaller than the specified size.

IntMode

Valid Range: 0-2 (0 = Legacy Int, 1 = MSI and 2 = MSI-X)
IntMode controls the allowed load time control over the type of interrupt registered for by the driver. MSI-X is required for multiple queue support, and some kernels and combinations of kernel .config options will force a lower level of interrupt support.
'cat /proc/interrupts' will show different values for each type of interrupt.

RSS

Valid Range: 0-8
0 = Assign up to the lesser value of the number of CPUs or the number of queues
X = Assign X queues, where X is less than or equal to the maximum number of queues (8 queues).

The maximum number of queues allowed are:
- I350-based adapters: 8 queues
- 82575-based adapters: 4 queues
- 82576-based and newer adapters: 8 queues
- I210-based adapters: 4 queues
- I211-based adapters: 2 queues

This parameter is also affected by the VMDq parameter in that it will limit the queues more. For example, if you set an 82575 device to VMDQ Mode 2, you will only be able to set 3 RSS queues. See the following table.
Model VMDQ Mode

Number0 1 2 3+
82575 4 4 3 1
82576 8 2 2 2
82580 8 1 1 1

VMDQ

Valid Range: 0-4 on 82575-based adapters; 0-8 for 82576/82580-based adapters
Supports enabling VMDq pools as this is needed to support SR-IOV.
0 = Disabled
1 = Sets the netdev as pool 0
2+ = Add additional queues but they currently are not used
This parameter is forced to 1 or more if the max_vfs module parameter is used.
In addition, the number of queues available for RSS is limited if this is set
to 1 or greater.

NOTE: When either SR-IOV mode or VMDq mode is enabled, hardware VLAN filtering
and VLAN tag stripping/insertion will remain enabled.

max_vfs

This parameter adds support for SR-IOV. It causes the driver to spawn up to
max_vfs worth of virtual functions.
Valid Range: 0-7
If the value is greater than 0 it will also force the VMDq parameter to be 1 or
more.

The parameters for the driver are referenced by position. Thus, if you have a
dual port adapter, or more than one adapter in your system, and want N virtual
functions per port, you must specify a number for each port with each parameter
separated by a comma. For example:

modprobe igb max_vfs=4

This will spawn 4 VFs on the first port.

modprobe igb max_vfs=2,4

This will spawn 2 VFs on the first port and 4 VFs on the second port.

NOTE: Caution must be used in loading the driver with these parameters.
Depending on your system configuration, number of slots, etc., it is impossible
to predict in all cases where the positions would be on the command line.

NOTE: Neither the device nor the driver control how VFs are mapped into config
space. Bus layout will vary by operating system. On operating systems that
support it, you can check sysfs to find the mapping.

NOTE: When either SR-IOV mode or VMDq mode is enabled, hardware VLAN filtering
and VLAN tag stripping/insertion will remain enabled. Please remove the old
VLAN filter before the new VLAN filter is added. For example:

ip link set eth0 vf 0 vlan 100   // set vlan 100 for VF 0
ip link set eth0 vf 0 vlan 0   // Delete vlan 100
ip link set eth0 vf 0 vlan 200   // set a new vlan 200 for VF 0

QueuePairs

Valid Range: 0-1
If set to 0, when MSI-X is enabled, the Tx and Rx will attempt to occupy separate vectors.
This option can be overridden to 1 if there are not sufficient interrupts available. This can occur if any combination of RSS, VMDQ, and max_vfs results in more than 4 queues being used.

Node

Valid Range: 0-n
0 - n: where n is the number of the NUMA node that should be used to allocate memory for this adapter port.
-1: uses the driver default of allocating memory on whichever processor is running modprobe.
The Node parameter allows you to choose which NUMA node you want to have the adapter allocate memory from. All driver structures, in-memory queues, and receive buffers will be allocated on the node specified. This parameter is only useful when interrupt affinity is specified; otherwise, part of the interrupt time could run on a different core than where the memory is allocated causing slower memory access and impacting throughput, CPU, or both.

EEE (Energy Efficient Ethernet)

Valid Range: 0-1
0 = Disables EEE
1 = Enables EEE

A link between two EEE-compliant devices will result in periodic bursts of data followed by periods where the link is in an idle state. This Low Power Idle (LPI) state is supported at 1 Gbps and 100 Mbps link speeds.

NOTES:
- EEE support requires auto-negotiation.
- Both link partners must support EEE.
- EEE is not supported on all Intel(R) Ethernet Network devices or at all link speeds.

Example:

ethtool --show-eee <ethX>
ethtool --set-eee <ethX> [eee on|off]

DMAC

Valid Range: 0, 1, 250, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000
This parameter enables or disables DMA Coalescing feature. Values are in microseconds and set the internal DMA Coalescing internal timer.
DMA (Direct Memory Access) allows the network device to move packet data directly to the system's memory, reducing CPU utilization. However, the frequency and random intervals at which packets arrive do not allow the system to enter a lower power state. DMA Coalescing allows the adapter to collect

packets before it initiates a DMA event. This may increase network latency but also increases the chances that the system will enter a lower power state. Turning on DMA Coalescing may save energy with kernel 2.6.32 and newer. DMA Coalescing must be enabled across all active ports in order to save platform power.

MDD (Malicious Driver Detection)

Valid Range: 0-1
0 = Disabled
1 = Enabled
This parameter is only relevant for I350 devices operating in SR-IOV mode. When this parameter is set, the driver detects malicious VF driver and disables its Tx/Rx queues until a VF driver reset occurs.

Jumbo Frames

Jumbo Frames support is enabled by changing the Maximum Transmission Unit (MTU) to a value larger than the default value of 1500.

Use the ip command to increase the MTU size. For example, enter the following where <ethX> is the interface number:

ip link set mtu 9000 dev <ethX>
ip link set up dev <ethX>

This setting is not saved across reboots.

Add 'MTU=9000' to the following file to make the setting change permanent:
/etc/sysconfig/network-scripts/ifcfg-<ethX> for RHEL
or
/etc/sysconfig/network/<config_file> for SLES

NOTE: The maximum MTU setting for jumbo frames is 9216. This corresponds to the maximum jumbo frame size of 9234 bytes.

NOTE: Using jumbo frames at 10 or 100 Mbps is not supported and may result in poor performance or loss of link.

NOTE: Packet loss may have a greater impact on throughput when you use jumbo frames. If you observe a drop in performance after enabling jumbo frames, enabling flow control may mitigate the issue.

Speed and Duplex Configuration

In addressing speed and duplex configuration issues, you need to distinguish between copper-based adapters and fiber-based adapters.

In the default mode, an Intel(R) Ethernet Network Adapter using copper connections will attempt to auto-negotiate with its link partner to determine the best setting. If the adapter cannot establish link with the link partner using auto-negotiation, you may need to manually configure the adapter and link partner to identical settings to establish link and pass packets. This should only be needed when attempting to link with an older switch that does not support auto-negotiation or one that has been forced to a specific speed or duplex mode. Your link partner must match the setting you choose. 1 Gbps speeds

and higher cannot be forced. Use the autonegotiation advertising setting to manually set devices for 1 Gbps and higher.

Speed, duplex, and autonegotiation advertising are configured through the ethtool utility.

To see the speed configurations your device supports, run the following:

ethtool <ethX>

Caution: Only experienced network administrators should force speed and duplex or change autonegotiation advertising manually. The settings at the switch must always match the adapter settings. Adapter performance may suffer or your adapter may not operate if you configure the adapter differently from your switch.

An Intel(R) Ethernet Network Adapter using fiber-based connections, however, will not attempt to auto-negotiate with its link partner since those adapters operate only in full duplex and only at their native speed.

IGB 5.9,3 code has been modified to suit our purpose

Set the link mode as sgmii for internal connection between i210 and bcm macsec phy
Read E1000_CTRL_EXT
Read link_mode on E1000_CTRL_EXT_LINK_MODE_MASK (23/24th bit)
reg_data |= E1000_CTRL_EXT_LINK_MODE_SGMII;
E1000_WRITE_REG(hw, E1000_CTRL_EXT, reg_data);

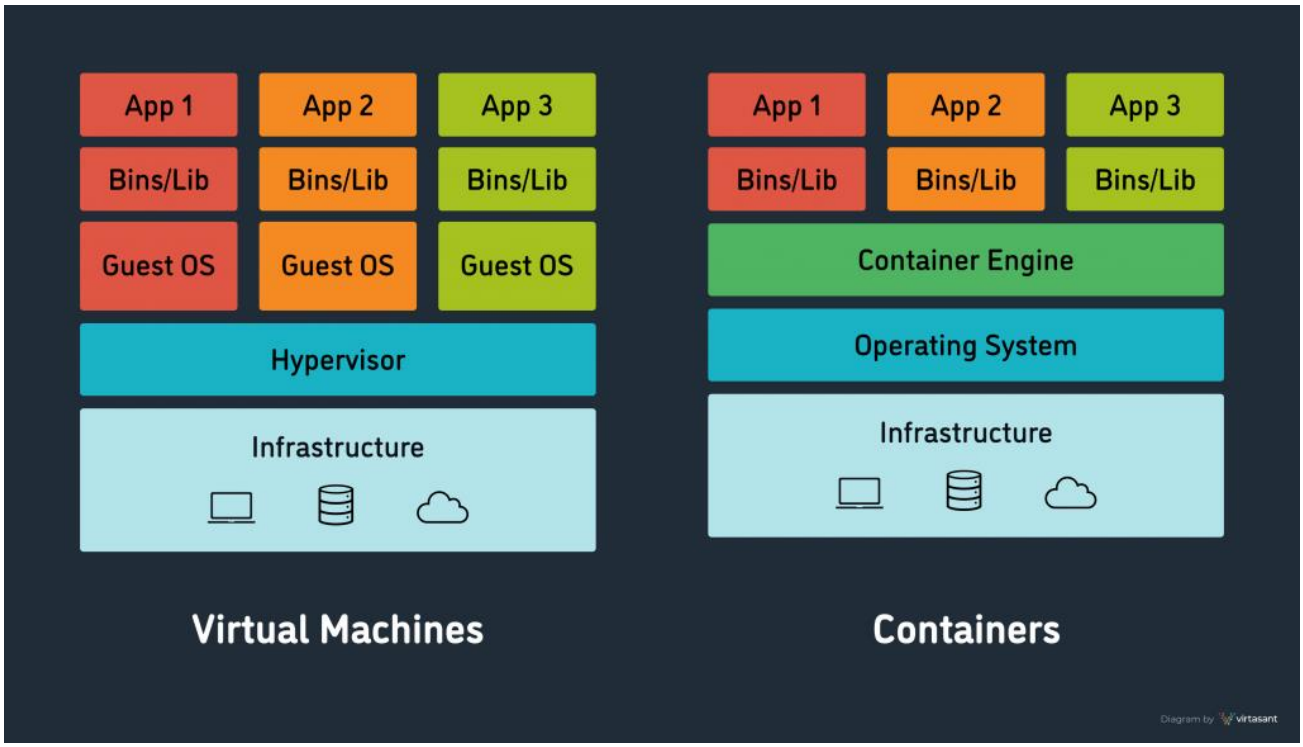
b)Set the destination bit as ext_mdio in mdicnfg register, also write phy address as 0
Then, write to MDICNF register as ext_mdio
reg |= E1000_MDICNFG_EXT_MDIO;
u32 phy_addr = 0x0ULL << E1000_MDICNFG_PHY_SHIFT;
reg |= phy_addr;
E1000_WRITE_REG(hw, E1000_MDICNFG, reg);

c)Read/write of ioctl calls should be done by programming the phy_id from the mii data recevied in mdicnfg register.
Basically set the destination phy addres from where the register's have to be set/got into mdicnfg register before trying to do actual read/write via mdic
and then mdic read/write will be done.

```
reg = E1000_READ_REG(hw, E1000_MDICNFG);
phy_addr = data->phy_id << E1000_MDICNFG_PHY_SHIFT;
reg &= ~E1000_MDICNFG_PHY_MASK;
reg |= phy_addr;
E1000_WRITE_REG(hw, E1000_MDICNFG, reg);
```

6. Virtualization

A hypervisor is a form of virtualization software used in Cloud hosting to divide and allocate the resources on various pieces of hardware. The program which provides partitioning, isolation, or abstraction is called a virtualization hypervisor. The hypervisor is a hardware virtualization technique that allows multiple guest operating systems (OS) to run on a single host system at the same time. A hypervisor is sometimes also called a virtual machine manager(VMM).



Types of Hypervisor –
TYPE-1 Hypervisor:
The hypervisor runs directly on the underlying host system. It is also known as a “Native Hypervisor” or “Bare metal hypervisor”. It does not require any base server operating system. It has direct access to hardware resources. Examples of Type 1 hypervisors include VMware ESXi, Citrix XenServer, and Microsoft Hyper-V hypervisor.

Pros & Cons of Type-1 Hypervisor:

Learn more
Pros: Such kinds of hypervisors are very efficient because they have direct access to the physical hardware resources(like Cpu, Memory, Network, and Physical storage). This causes the empowerment of the security because there is nothing any kind of the third party resource so that attacker couldn’t compromise with anything.

Cons: One problem with Type-1 hypervisors is that they usually need a dedicated separate machine to perform their operation and to instruct different VMs and control the host hardware resources.

TYPE-2 Hypervisor:
A Host operating system runs on the underlying host system. It is also known as ‘Hosted Hypervisor”. Such kind of hypervisors doesn’t run directly over the underlying hardware rather they run as an application in a

Host system(physical machine). Basically, the software is installed on an operating system. Hypervisor asks the operating system to make hardware calls. An example of a Type 2 hypervisor includes VMware Player or Parallels Desktop. Hosted hypervisors are often found on endpoints like PCs. The type-2 hypervisor is very useful for engineers, and security analysts (for checking malware, or malicious source code and newly developed applications).

Learn more
Pros & Cons of Type-2 Hypervisor:

Pros: Such kind of hypervisors allows quick and easy access to a guest Operating System alongside the host machine running. These hypervisors usually come with additional useful features for guest machines. Such tools enhance the coordination between the host machine and the guest machine.

Cons: Here there is no direct access to the physical hardware resources so the efficiency of these hypervisors lags in performance as compared to the type-1 hypervisors, and potential security risks are also there an attacker can compromise the security weakness if there is access to the host operating system so he can also access the guest operating system.

Choosing the right hypervisor :

Type 1 hypervisors offer much better performance than Type 2 ones because there's no middle layer, making them the logical choice for mission-critical applications and workloads. But that's not to say that hosted hypervisors don't have their place – they're much simpler to set up, so they're a good bet if, say, you need to deploy a test environment quickly. One of the best ways to determine which hypervisor meets your needs is to compare their performance metrics. These include CPU overhead, the amount of maximum host and guest memory, and support for virtual processors. The following factors should be examined before choosing a suitable hypervisor:

1. Understand your needs: The company and its applications are the reason for the data center (and your job). Besides your company's needs, you (and your co-workers in IT) also have your own needs. Needs for a virtualization hypervisor are:

a. Flexibility
b. Scalability
c. Usability
d. Availability
e. Reliability
f. Efficiency
g. Reliable support

2. The cost of a hypervisor: For many buyers, the toughest part of choosing a hypervisor is striking the right balance between cost and functionality. While a number of entry-level solutions are free, or practically free, the prices at the opposite end of the market can be staggering. Licensing frameworks also vary, so it's important to be aware of exactly what you're getting for your money.

3. Virtual machine performance: Virtual systems should meet or exceed the performance of their physical counterparts, at least in relation to the applications within each server. Everything beyond meeting this benchmark is profit.

4. Ecosystem: It's tempting to overlook the role of a hypervisor's ecosystem – that is, the availability of documentation, support, training, third-party developers and consultancies, and so on – in determining whether or not a solution is cost-effective in the long term.

5. Test for yourself: You can gain basic experience from your existing desktop or laptop. You can run both VMware vSphere and Microsoft Hyper-V in either VMware Workstation or VMware Fusion to create a nice virtual learning and testing environment.

HYPERVISOR REFERENCE MODEL :
There are 3 main modules coordinates in order to emulate the underlying hardware:

DISPATCHER:
The dispatcher behaves like the entry point of the monitor and reroutes the instructions of the virtual machine instance to one of the other two modules.

ALLOCATOR:
The allocator is responsible for deciding the system resources to be provided to the virtual machine instance. It means whenever a virtual machine tries to execute an instruction that results in changing the machine resources associated with the virtual machine, the allocator is invoked by the dispatcher.

INTERPRETER:
The interpreter module consists of interpreter routines. These are executed, whenever a virtual machine executes a privileged instruction.

Hypervisors allow you to divide a single computer's hardware resources between multiple VMs. Containers allow you to split a single computer into segregated logical namespaces. Containers are all about isolation, not virtualization. From an application development point of view, they look like the same thing, but they work in different layers.

Popular containerization tools, like Docker, can create and run multiple containers on the host's Linux kernel. Every container has its specific network stack and its own process space, including all of the underlying dependencies required to run the application. Containers do not contain the operating system, so they are very compact, and start up in milliseconds. Containers provide an excellent platform for building and sharing packaged, ready-to-run applications, and micro-services.

Containers run closer to the application layer, while hypervisors run closer to the hardware layer. In most cases, hypervisors run the VMs, and containers run on those VMs.