

Study Material

Course: Platform Validation



A unit of UNIVISION Technology Consulting Pvt. Ltd.
#473, SWATHI TOWERS, 3RD FLOOR, RBI LAYOUT, JP NAGAR
7TH PHASE, BANGALORE - 560078.
+91 7676651416, 080 4890 919
hello@etechprowess.com

INDEX

CHAPTER	Page No.
Chapter 1: Course Motivation and platform basic	2
Chapter 2: Intel Architecture	20
Chapter 3: Interrupts	52
Chapter 4: PC Components	98
Chapter 5 Peripheral Component Interconnect (PCI)	131
Chapter-6 Universal Serial Bus (USB)	148
Chapter 7: Memory devices	228
Chapter 8: On board Communication Protocols	249
Chapter 9: Introduction to BIOS	265
Chapter 10: Boot process	270
Chapter 11: Operating Systems (OS)	287
Chapter 12: ACPI and SMBIOS	326
Chapter 13: Operating system Installation and Troubleshooting	341
Chapter 14: OS Environmental Tools	362
Chapter 15: Testing Basics	366
Chapter 16: Hardware Lab Kit (HLK)	379
Chapter 17: Python Basics	387
Chapter 18: Introduction to JTAG	394
Chapter-19 Soldering Basics	401

CHAPTER: 1

Course Motivation and platform basic

1.1 Introduction to testing and validation

- The process of evaluating Product during the development process or at the end of the development process to determine whether it satisfies specified business requirements.
- Validation Testing ensures that the product actually meets the client's needs.

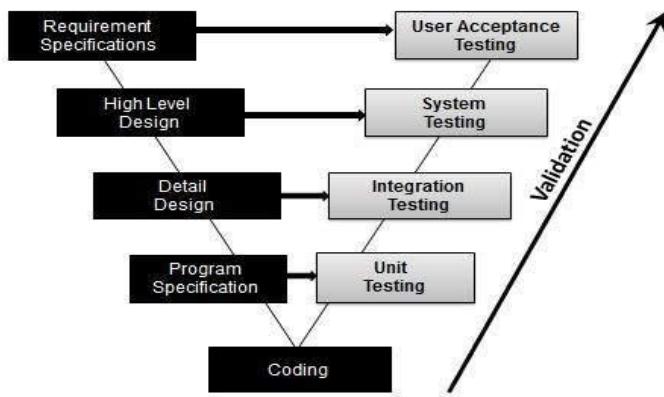


fig-1:V-model represents validation

1.2 Phases in Testing

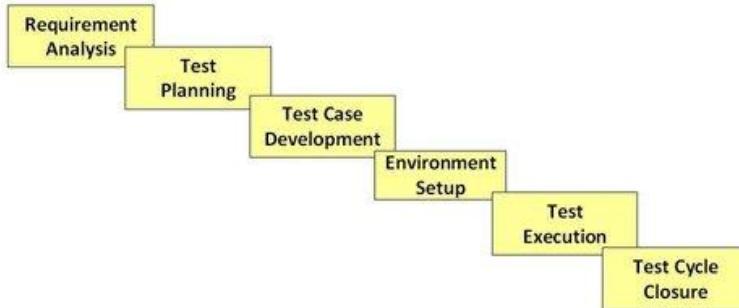


Fig-2: phases of Testing

1.2.1 Test Plan Types

One can have the following types of test plans:

- **Master Test Plan:** A single high-level test plan for a project/product that unifies all other test plans.
- **Testing Level Specific Test Plans:** Plans for each level of testing.
 - Unit Test Plan
 - Integration Test Plan

- System Test Plan
 - Acceptance Test Plan
- **Testing Type Specific Test Plans:** Plans for major types of testing like Performance Test Plan and Security Test Plan.

1.3 Below are the phases of STLC

- Requirement Analysis
- Test Planning
- Test case development
- Test Environment setup
- Test Execution
- Test Cycle closure

1.4 Importance of testing

Testing plays a vital role in development. In every company, testing is an important and valuable stage in the Development Life Cycle in order to:

Improve Your Software

The role of testing in software development begins with improved reliability, quality and performance of the software

Quality assurance

Quality plays a vital role in today's competitive world

Helps to avoid dangerous situations

Without proper testing, produced software can be dangerous to the users

2.0 CISC and RISC

2.1 What is The Difference Between RISC and CISC Architecture?

The architecture of the Central Processing Unit (CPU) operates the capacity to function from “Instruction Set Architecture” to where it was designed. The architectural design of the CPU is Reduced instruction set computing (RISC) and Complex instruction set computing (CISC). CISC has the capacity to perform multi-step operations or addressing modes within one instruction set. It is the CPU design where one instruction works several low-level acts. For instance, memory

storage, loading from memory, and an arithmetic operation. Reduced instruction set computing is a Central Processing Unit design strategy based on the vision that basic instruction set gives a great performance when combined with a microprocessor architecture which has the capacity to perform the instructions by using some microprocessor cycles per instruction. This article discusses the difference between the RISC and CISC architecture. The hardware part of the Intel is named as Complex Instruction Set Computer (CISC), and Apple hardware is Reduced Instruction Set Computer (RISC).

RISC	CISC
1. RISC stands for Reduced Instruction Set Computer.	1. CISC stands for Complex Instruction Set Computer.
2. RISC processors have simple instructions taking about one clock cycle. The average clock cycle per instruction (CPI) is 1.5	2. CSIC processor has complex instructions that take up multiple clocks for execution. The average clock cycle per instruction (CPI) is in the range of 2 and 15.
3. Performance is optimized with more focus on software	3. Performance is optimized with more focus on hardware.
4. It has no memory unit and uses a separate hardware to implement instructions..	4. It has a memory unit to implement complex instructions.
5. It has a hard-wired unit of programming.	5. It has a microprogramming unit.
6. The instruction set is reduced i.e. it has only a few instructions in the instruction set. Many of these instructions are very primitive.	6. The instruction set has a variety of different instructions that can be used for complex operations.
7. The instruction set has a variety of different instructions that can be used for complex operations.	7. CISC has many different addressing modes and can thus be used to represent higher-level programming language statements more efficiently.
8. Complex addressing modes are synthesized using the software.	8. CISC already supports complex addressing modes
9. Multiple register sets are present	9. Only has a single register set
10. RISC processors are highly pipelined	10. They are normally not pipelined or less pipelined
11. The complexity of RISC lies with the compiler that executes the program	11. The complexity lies in the microprogram
12. Execution time is very less	12. Execution time is very high
13. Code expansion can be a problem	13. Code expansion is not a problem
14. Decoding of instructions is simple.	14. Decoding of instructions is complex

15. It does not require external memory for calculations	15. It requires external memory for calculations
16. The most common RISC microprocessors are Alpha, ARC, ARM, AVR, MIPS, PA-RISC, PIC, Power Architecture, and SPARC.	16. Examples of CISC processors are the System/360, VAX, PDP-11, Motorola 68000 family, AMD and Intel x86 CPUs.

3.0 Intel Architecture (IA)

Intel processors are designed in reference to 3 main abstract models

- IA- 32 architecture
- Intel 64 architecture
- IA- 64 architecture

Intel compilers and libraries support three platforms: general combinations of processor architecture and operating system type. This section explains the terms that Intel uses to describe the platforms in its documentation, installation procedures and support site.

IA-32 Architecture refers to systems based on 32-bit processors generally compatible with the Intel Pentium® II processor, (for example, Intel Pentium® 4 processor or Intel Xeon® processor), or processors from other manufacturers supporting the same instruction set, running a 32-bit operating system.

Intel 64 Architecture refers to systems based on IA-32 architecture processors which have 64-bit architectural extensions, for example, Intel Core™2 processor family), running a 64-bit operating system such as Microsoft Windows XP* Professional x64 Edition or Microsoft Windows Vista* x64. If the system is running a 32-bit version of the Windows operating system, then IA-32 architecture applies instead. Systems based on AMD* processors running a 64-bit version of Windows are also supported by Intel compilers for Intel 64 architecture applications.

x-64 or IA32e is used as a short term for the 64 bit extensions of the "classical" x86 architecture; almost any "normal" PC produced in the last years have a processor based on such architecture.

IA 64 (also called **Intel Itanium architecture**) is the instruction_set_architecture (ISA) of the Itanium family of 64-bit Intel microprocessors. The IA-64 architecture receives the sigla EPIC, which means Explicit Parallel Instruction Computing. By using this sigla, Intel wants to say that the compiler will be the great responsible for determining and clearing the parallelism present in the instructions to be executed. This is a combination of concepts called speculation, predication and explicit parallelism. Next, we will briefly study each one of them.

3.1 INTEL 64 AND IA-32 PROCESSOR GENERATIONS

16-bit Processors and Segmentation (1978)

The Intel 286 processor introduced protected mode operation into the IA-32 architecture. Protected mode uses the segment register content as selectors or pointers into descriptor tables. Descriptors provide 24-bit base addresses with a physical memory size of up to 16 MBytes, support for virtual memory management on a segment swapping basis, and a number of protection mechanisms. These mechanisms include:

- Segment limit checking
- Read-only and execute-only segment options
- Four privilege levels

The Intel386 Processor (1985)

The Intel386 processor was the first 32-bit processor in the IA-32 architecture family. It introduced 32-bit registers for use both to hold operands and for addressing. The lower half of each 32-bit Intel386 register retains the properties of the 16-bit registers of earlier generations, permitting backward compatibility. The processor also provides a virtual-8086 mode that allows for even greater efficiency when executing programs created for 8086/8088 processors. In addition, the Intel386 processor has support for:

- A 32-bit address bus that supports up to 4-GBytes of physical memory

- A segmented-memory model and a flat memory model
- Paging, with a fixed 4-KByte page size providing a method for virtual memory management
- Support for parallel stages

The Intel Pentium Processor (1993)

The introduction of the Intel Pentium processor added a second execution pipeline to achieve superscalar performance (two pipelines, known as u and v, together can execute two instructions per clock). The on-chip first-level cache doubled, with 8 KBytes devoted to code and another 8 KBytes devoted to data. The data cache uses the MESI protocol to support more efficient write-back cache in addition to the write-through cache previously used by the Intel486 processor. Branch prediction with an on-chip branch table was added to increase performance in looping constructs. In addition, the processor added:

- Extensions to make the virtual-8086 mode more efficient and allow for 4-MByte as well as 4-KByte pages

- Internal data paths of 128 and 256 bits add speed to internal data transfers
- Burstable external data bus was increased to 64 bits
- An APIC to support systems with multiple processors
- A dual processor mode to support glueless two processor systems

A subsequent stepping of the Pentium family introduced Intel MMX technology (the Pentium Processor with MMX technology). Intel MMX technology uses the single-instruction, multiple-data (SIMD) execution model to perform parallel computations on packed integer data contained in 64-bit registers.

The P6 Family of Processors (1995-1999)

The P6 family of processors was based on a superscalar microarchitecture that set new performance standards; “P6 Family Microarchitecture.” One of the goals in the design of the P6 family microarchitecture was to exceed the performance of the Pentium processor significantly while using the same 0.6-micrometer, fourlayer, metal BICMOS manufacturing process. Members of this family include the following:

- **The Intel Pentium Pro** processor is three-way superscalar. Using parallel processing techniques, the processor is able on average to decode, dispatch, and complete execution of (retire) three instructions per clock cycle. The Pentium Pro introduced the dynamic execution (micro-data flow analysis, out-of-order execution, superior branch prediction, and speculative execution) in a superscalar implementation. The processor was further enhanced by its caches. It has the same two on-chip 8-KByte 1st-Level caches as the Pentium processor and an additional 256-KByte Level 2 cache in the same package as the processor.
- **The Intel Pentium II** processor added Intel MMX technology to the P6 family processors along with new packaging and several hardware enhancements. The processor core is packaged in the single edge contact cartridge (SECC). The Level I data and instruction caches were enlarged to 16 KBytes each, and Level 2 cache sizes of 256 KBytes, 512 KBytes, and 1 MByte are supported. A half-frequency backside bus connects the Level 2 cache to the processor. Multiple low-power states such as AutoHALT, Stop-Grant, Sleep, and Deep Sleep are supported to conserve power when idling.
- **The Pentium II Xeon** processor combined the premium characteristics of previous generations of Intel processors. This includes: 4-way, 8-way (and up) scalability and a 2 MByte 2nd-Level cache running on a full frequency backside bus.

- **The Intel Celeron processor** family focused on the value PC market segment. Its introduction offers an integrated 128 KBytes of Level 2 cache and a plastic pin grid array (P.P.G.A.) form factor to lower system design cost.
- **The Intel Pentium III** processor introduced the Streaming SIMD Extensions (SSE) to the IA-32 architecture. SSE extensions expand the SIMD execution model introduced with the Intel MMX technology by providing a Vol. 1 2-3 INTEL 64 AND IA-32 ARCHITECTURES new set of 128-bit registers and the ability to perform SIMD operations on packed single-precision floating point values.
- **The Pentium III Xeon** processor extended the performance levels of the IA-32 processors with the enhancement of a full-speed, on-die, and Advanced Transfer Cache.

The Intel Core Duo and Intel Core Solo Processors (2006-2007)

The Intel Core Duo processor offers power-efficient, dual-core performance with a low-power design that extends battery life. This family and the single-core Intel Core Solo processor offer microarchitectural enhancements over Pentium M processor family. Its enhanced microarchitecture includes

- : • Intel Smart Cache which allows for efficient data sharing between two processor cores
- Improved decoding and SIMD execution
- Intel Dynamic Power Coordination and Enhanced Intel Deeper Sleep to reduce power consumption
- Intel Advanced Thermal Manager which features digital thermal sensor interfaces
- Support for power-optimized 667 MHz bus The dual-core Intel Xeon processor LV is based on the same microarchitecture as Intel Core Duo processor, and supports IA-32 architecture

The Intel Atom Processor Family (2008)

The first generation of Intel AtomTM processors are built on 45 nm process technology. They are based on a new microarchitecture, Intel AtomTM microarchitecture, which is optimized for ultra low power devices. The Intel AtomTM microarchitecture features two in-order execution pipelines that minimize power consumption, increase battery life, and enable ultra-small form factors. The initial Intel Atom Processor family and subsequent generations including Intel Atom processor D2000, N2000, E2000, Z2000, C1000 series provide the following features:

- Enhanced Intel SpeedStep Technology
- Intel Hyper-Threading Technology
- Deep Power Down Technology with Dynamic Cache Sizing

- Support for instruction set extensions up to and including Supplemental Streaming SIMD Extensions 3 (SSSE3).
- Support for Intel Virtualization Technology
- Support for Intel 64 Architecture (excluding Intel Atom processor Z5xx Series)

2010 Intel Core Processor Family (2010)

2010 Intel Core processor family spans Intel Core i7, i5 and i3 processors. They are based on Intel microarchitecture code name Westmere using 32 nm process technology. The innovative features can include:

- Deliver smart performance using Intel Hyper-Threading Technology plus Intel Turbo Boost Technology.
- Enhanced Intel Smart Cache and integrated memory controller.
- Intelligent power gating.
- Repartitioned platform with on-die integration of 45 nm integrated graphics.
- Range of instruction set support up to AESNI, PCLMULQDQ, SSE4.2 and SSE4.1.

The Second Generation Intel Core Processor Family (2011)

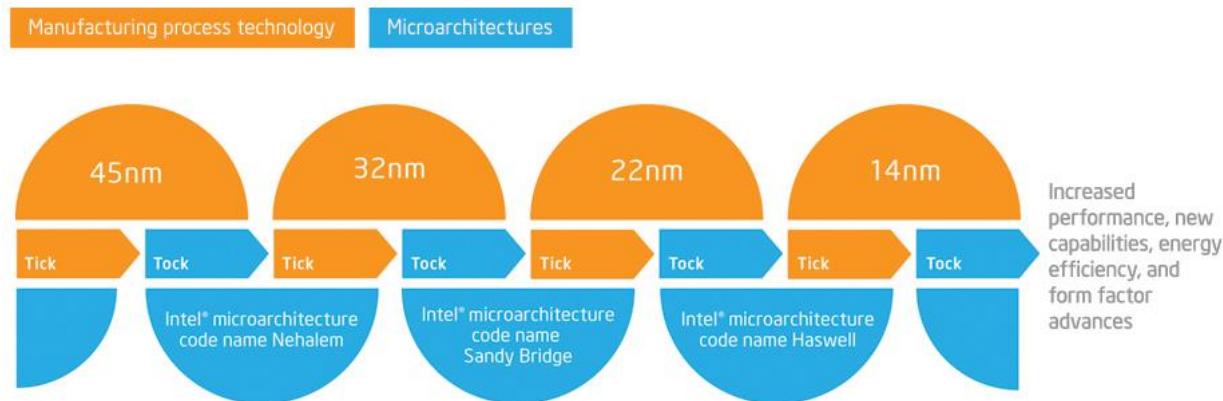
The Second Generation Intel Core processor family spans Intel Core i7, i5 and i3 processors based on the Sandy Bridge microarchitecture. They are built from 32 nm process technology and have innovative features including:

Intel Turbo Boost Technology for Intel Core i5 and i7 processors

- Intel Hyper-Threading Technology
- Enhanced Intel Smart Cache and integrated memory controller.
- Processor graphics and built-in visual features like Intel Quick Sync Video, Intel Insider™ etc.
- Range of instruction set support up to AVX, AESNI, PCLMULQDQ, SSE4.2 and SSE4.1

4.0 TICK_TOCK

The Tick-Tock model through the years



Tick-Tock was an aggressive development model introduced by Intel for their mainstream microprocessors in 2005 and phased out in 2016 whereby microarchitecture changes were in-sync with their process shrink. Under the tick-tock scheme roughly every 12-18 months the Intel alternated between "Tick" and "Tock". Intel no longer uses this model, instead they use the Process-Architecture-Optimization (PAO).

Under the Tick-Tock Model:

- **Tick** - With each tick, Intel advances their manufacturing process technology in line with Moore's Law. Each new process introduces higher transistor density and a generally a plethora of other advantages such as higher performance and lower power consumption. During a tick, Intel retrofits their previous microarchitecture to the new process which inherently yielded better performance and energy saving. At this phase, only lightweight features and improvements are introduced.
- **Tock** - With each tock, Intel uses the their latest manufacturing process technology from their "tick" to manufacture a newly designed microarchitecture. The new microarchitecture is designed with the new process in mind and typically introduces Intel's newest big features and functionalities. New instructions are often added during this cycle stage.

Intel Tick-Tock Roadmap			
Cycle	<u>Process</u>	Introduction	Microarchitecture
Tick	<u>65 nm</u>	2005	Pentium D
Tock	<u>65 nm</u>	2006	<u>Core</u>
Tick	<u>45 nm</u>	2007	<u>Penryn</u>
Tock	<u>45 nm</u>	2008	<u>Nehalem</u>
Tick	<u>32 nm</u>	2009	<u>Westmere</u>
Tock	<u>32 nm</u>	2010	<u>Sandy Bridge</u>
Tick	<u>22 nm</u>	2011	<u>Ivy Bridge</u>
Tock	<u>22 nm</u>	2013	<u>Haswell</u>
Tick	<u>14 nm</u>	2014	<u>Broadwell</u>
Tock	<u>14 nm</u>	2015	<u>Skylake</u>

5.0 PAO[Process Architecture optimization]

Process-Architecture-Optimization was a temporary development model introduced by Intel for their mainstream microprocessors in 2016 following the phase-out of their Tick-Tock model as a result of major delays and challenges involving their 10 nm process.

Under the Process-Architecture-Optimization Model:

- **Process** - With each process, Intel advances their manufacturing process technology in line with Moore's Law. Each new process introduces higher transistor density and generally a plethora of other advantages such as higher performance and lower power consumption. During a "process", Intel retrofits their previous microarchitecture to the new process which inherently yields better performance and energy saving. During a "process", usually, just a few features and improvements are introduced.
- **Architecture** - With each architecture, Intel uses the their latest manufacturing process technology from their "process" to manufacture a newly designed microarchitecture. The new microarchitecture is designed with the new process in mind and typically introduces Intel's newest big features and functionalities. New instructions are often added during this cycle stage.
- **Optimization** - With each optimization, Intel improves upon their previous microarchitecture by introducing incremental improvements and enhancements without introducing any large changes. Additionally, the process itself enjoys various refinements as it matures. (For example with Kaby Lake, an optimized process called "14 nm+" is used. The enhanced process had a number of transistor-level modifications done to it (e.g. taller fins) allowing for higher frequency at identical voltage levels.)

Intel PAO Roadmap			
Cycle	<u>Process</u>	<u>Introduction</u>	<u>Microarchitecture</u>
Process	<u>14 nm</u>	2014	<u>Broadwell</u>
Architecture	<u>14 nm</u>	2015	<u>Skylake (Client)</u>
Optimization	<u>14 nm+</u>	2016	<u>Kaby Lake</u>

Optimization	14 nm++	2017	Coffee Lake, Skylake (Server)
Optimization	14 nm++	2018	Whiskey Lake
Optimization	14 nm++	2019	Cascade Lake, Cooper Lake

6.0 PC AT/ATX

The **IBM Personal Computer AT**, more commonly known as the **IBM AT** and also sometimes called the **PC AT** or **PC/AT**, was [IBM's](#) second-generation [PC](#), designed around the 6 MHz [Intel 80286 microprocessor](#) and released in 1984 as System Unit **5170**. The name **AT** stood for "Advanced Technology," and was chosen because the AT offered various technologies that were then new in personal computers; one such advancement was that the 80286 processor supported [protected mode](#).

6.1 Power supply

The IBM PC AT came with a 192-watt [switching power supply](#). According to IBM's documentation, in order to function properly, the AT power supply needed a load of at least 7.0 amperes on the +5V line and a minimum of 2.5 amperes was on its +12V line. In practice, the AT power supply would randomly fail to start unless these minimum load requirements were met. Because the AT motherboard didn't provide much load on the +12V line, entry-level IBM AT models that didn't have a hard drive were shipped with a 5-ohm, 50-watt ([maximum power](#)) sandbar resistor connected on the +12V line of the hard disk power connector. In normal operation this resistor drew 2.4 amperes (28.8 watts), getting fairly hot.



Fig-3:IBM PC AT (System Unit 5170)

6.2 ATX

ATX (Advanced Technology eXtended) is a motherboard and power supply configuration specification developed by [Intel](#) in 1995 to improve on previous [*de facto standards*](#) like the [AT design](#). It was the first major change in [desktop computer enclosure](#), [motherboard](#) and [power supply](#) design in many years, improving standardization and interchangeability of parts. The specification defines the key mechanical dimensions, mounting point, I/O panel, power and connector interfaces between a [computer case](#), a [motherboard](#) and a [power supply](#).



fig-4: An ATX motherboard

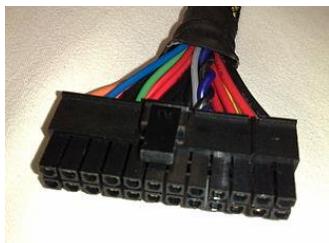
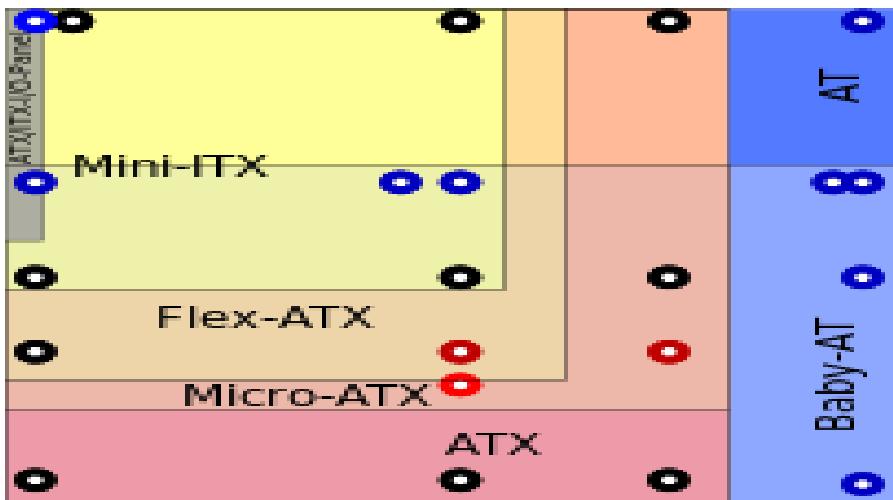


fig-5 ATX 1.3v



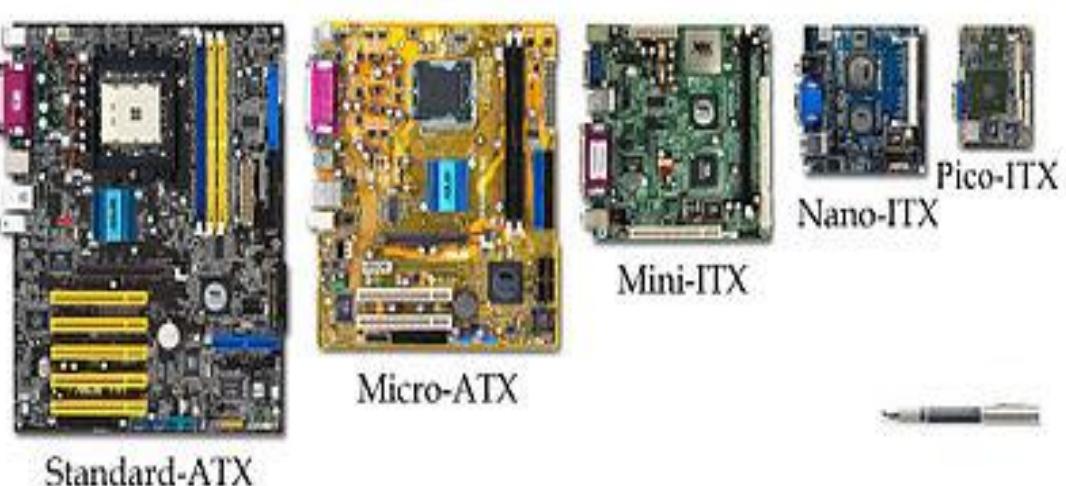


Fig-6:Different motherboard form factors

Size and Fit

Both AT and ATX motherboards have been produced in various sizes throughout the years, and the form factors fit different computer cases depending on their size

7.0 Desktop and Mobile Processors

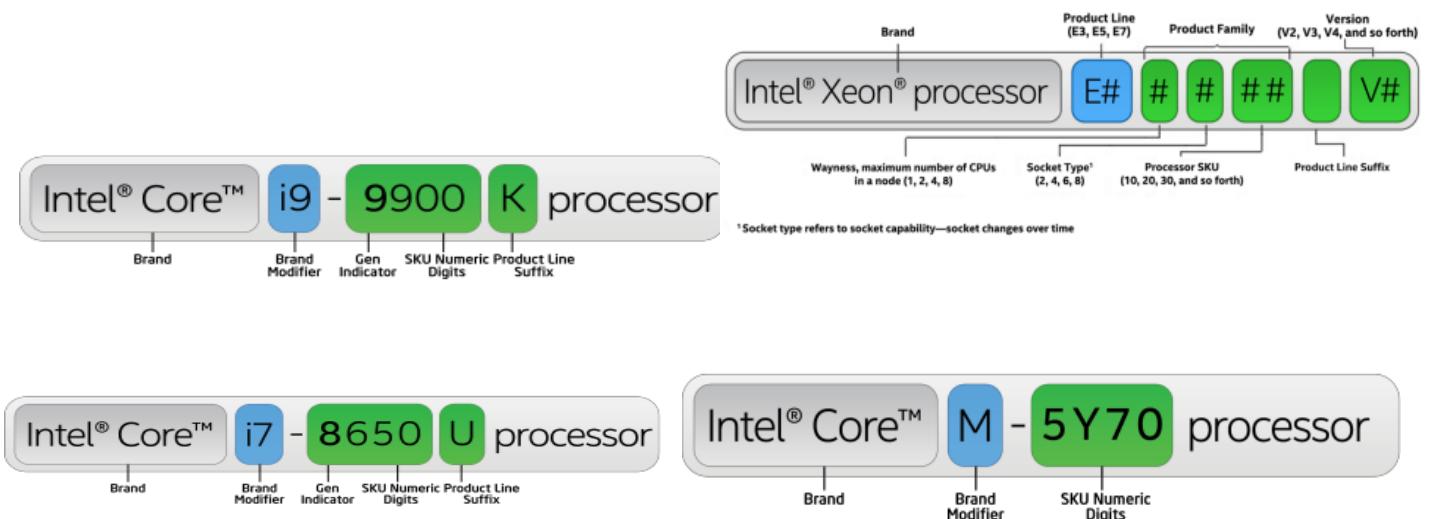


Fig-7:code names of desktop and processor

Alpha Suffix	Description	Example
Desktop		
K	Unlocked	8th Gen Intel Core i7-8700K processor
Mobile		
G	Includes discrete graphics on package	8th Gen Intel Core i7-8705G processor
U	Ultra-low power	8th Gen Intel Core i7-8650U processor

Alpha Suffix	Description	Example
Desktop		
K	Unlocked	Intel Core i9-9900K processor
F	Requires discrete graphics	Intel Core i9-9900KF processor

Alpha Suffix	Description	Example
H	High performance graphics	7th Gen Intel Core™ i3-7100H processor
HK	High performance graphics, unlocked	7th Gen Intel Core™ i7-7820HK processor
HQ	High performance graphics, quad core	7th Gen Intel Core i7-7920HQ processor
U	Ultra-low power	7th Gen Intel Core i7-7500U processor
Y	Extremely low power	7th Gen Intel Core i7-7Y75 processor

Comparing Mobile CPU vs Desktop CPU

Type	Processor	Model	Cores	Clock Rate	GPU	GPU Clock	TDP
Desktop	Intel Core i5	7500T	4	2.7 GHz	HD 620	1100 MHz	35W
Mobile	Intel Core i7	7500U	2	2.7 GHz	HD 620	1050 MHz	15W

A desktop version will have a **higher speed processing and better cache** than a mobile CPU which will feature a reduced speed. Compare the two models below: the mobile core i7 CPU has the same clock rate as the Desktop i5 core. The Desktop has twice more cores than the mobile core i7 and a small improvement on the GPU Clock rated speed.

Refer:: <https://www.intel.in/content/www/in/en/processors/processor-numbers.html>

8.0 Intel families and generation

						Arrandale	L	32 nm	January 2010
Core i3	Clarkdale	2	32 nm	January 2010	Sandy Bridge	2	32 nm	February 2011	
	Sandy Bridge	2	32 nm	February 2011	Ivy Bridge	2	22 nm	June 2012	
	Ivy Bridge	2	22 nm	September 2012	Haswell	2	22 nm	June 2013	
	Haswell	2	22 nm	September 2013	Broadwell	2	14 nm	January 2015	
	Skylake	2	14 nm	September 2015	Skylake	2	14 nm	September 2015	
	Kaby Lake	2	14 nm	January 2017	Skylake	2	14 nm	June 2016	
	Coffee Lake	4	14 nm	October 2017	Kaby Lake	2	14 nm	August 2016	
	Coffee Lake	4	14 nm	January 2019	Kaby Lake	2	14 nm	November 2016	
					Kaby Lake	2	14 nm	January 2017	
					Coffee Lake	2	14 nm	June 2017	
Core i5	Lynnfield	4	45 nm	September 2009	Coffee Lake	2	14 nm	April 2018	
	Clarkdale	2	32 nm	January 2010	Cannon Lake	2	10 nm	May 2018	
	Sandy Bridge	4	32 nm	January 2011	Coffee Lake	4	14 nm	July 2018	
	Sandy Bridge	2	32 nm	February 2011	Arrandale	2	32 nm	January 2010	
	Ivy Bridge	2-4	22 nm	April 2012	Sandy Bridge	2	32 nm	February 2011	
	Haswell	2-4	22 nm	June 2013	Ivy Bridge	2	22 nm	May 2012	
	Broadwell	4	14 nm	June 2015	Haswell	2	22 nm	June 2013	
	Skylake	4	14 nm	September 2015	Broadwell	2	14 nm	January 2015	
	Kaby Lake	4	14 nm	January 2017	Skylake	2-4	14 nm	September 2015	
	Coffee Lake	6	14 nm	October 2017	Kaby Lake	2	14 nm	August 2016	
	Coffee Lake	6	14 nm	Oct. 2018 - Jan. 2019	Kaby Lake	4	14 nm	January 2017	
					Kaby Lake-R	4	14 nm	October 2017	
					Coffee Lake	4	14 nm	April 2018	

10.0 Xeon code names

Code Name	Brand Name
Crystal Forest	Intel® Xeon®, Core™, Pentium®, Celeron® Processors
East Beach	Intel® Xeon®, Core™, Pentium®, Celeron® Processors
Bromolow (Sandy Bridge + Cougar Point)	Intel® Xeon®, 2nd Generation Intel® Core™, Intel® Pentium®, and Intel® Celeron® Processors with Intel® C206 Chipset
Purley	Intel® Xeon® Scalable Processors with Intel® C620 Series Chipsets
Brickland	Intel® Xeon® Processor E7 v4 Family
River Forest	Intel® Xeon® Processor E5-2600 v3 Product Family with Intel® Communications Chipset 89xx Series
Grantley	Intel® Xeon® Processor E5 v4 Product Family
Grantley	Intel® Xeon® Processor E5 v3 Product Family

Chapter-2

Intel Architecture

1.0 Computer Internal Architecture

1.1 CPU Basic Operation

The Four Primary Functions of the CPU

Fetch: The basic cycle of every CPU is to fetch the first instruction from memory

Decode: decode it to determine its type and operands

Execute: execute based the opcode

Store: store the end result to the desired location

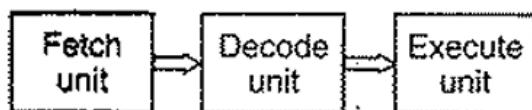


Figure 1. Instruction Cycle

The "brain" of the computer is the CPU. It fetches instructions from memory and executes them. The basic cycle of every CPU is to fetch the first instruction from memory, decode it to determine its type and operands, execute it, and then fetch, decode, and execute subsequent instructions. The cycle is repeated until the program finishes. In this way, programs are carried out

1.2 Prefetching

Prefetching is a technique for speeding up fetch operations by beginning a fetch operation whose result is expected to be needed soon. It reducing wait states and hides memory access latency.

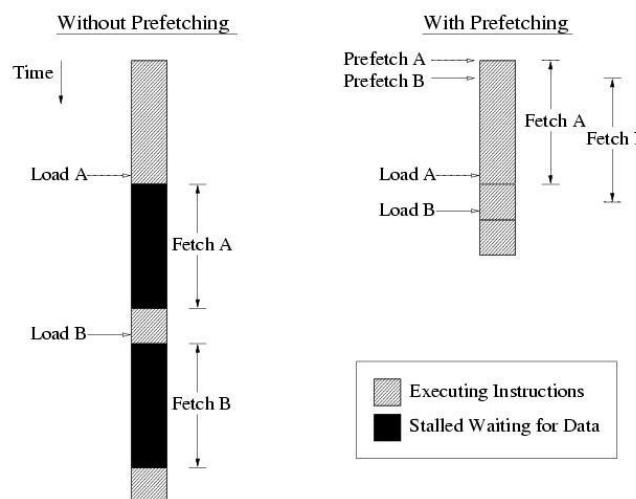


Figure 2. An example for Prefetching

1.3 Pipelining

All processors since about 1985 use pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called instruction-level parallelism (ILP). Pipeline is the continuous flow of instruction to the processor in an orderly manner to perform an operation. In Pipelining multiple instructions are overlapped during execution.

Pipeline is divided into stages which connected with one another to form a pipe like structure. Instructions enter from one end and exit from another end. Pipelining increases the overall instruction throughput.

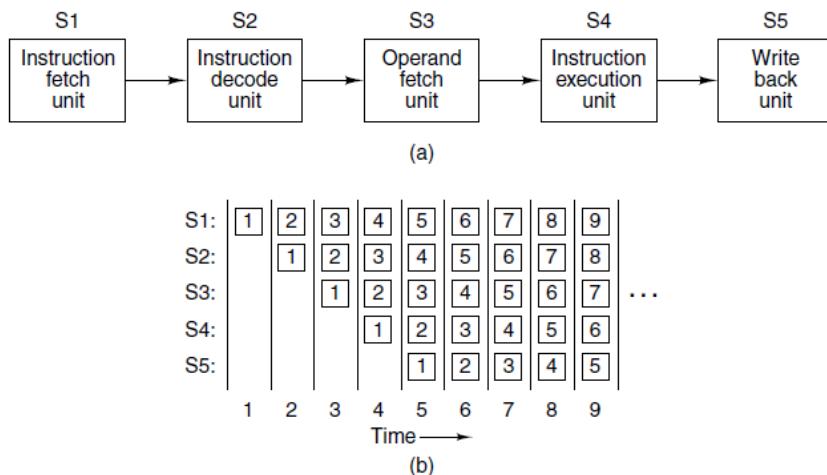


Figure 3. Pipelining

1.3.1 Types of Pipeline:

1.3.1.1 Arithmetic Pipeline:

- Arithmetic pipelines are usually found in most of the computers.
- They are used for floating point operations, multiplication of fixed point numbers etc.

1.3.1.2 Instruction Pipeline:

- A stream of instructions can be executed by overlapping fetch, decode and execute phases of an instruction cycle.
- It reads instruction from the memory while previous instructions are being executed in other segments of the pipeline.
- It increases the throughput of the computer system.

1.3.2 Advantages and disadvantages of pipeline

Advantages

- The cycle time/Machine cycle length of the processor is reduced. (Increase clock frequency)

- It increases the throughput of the system
- It makes the system reliable.

Disadvantages

- The design of pipelined processor is complex and costly to manufacture.
- The instruction latency is more.

1.4 Superscalar Architectures

It describes a processor design that issues multiple instructions—often four or six—in a single clock cycle. Of course, a superscalar CPU must have multiple functional units to handle all these instructions in a single clock cycle. Superscalar processors generally have one pipeline.

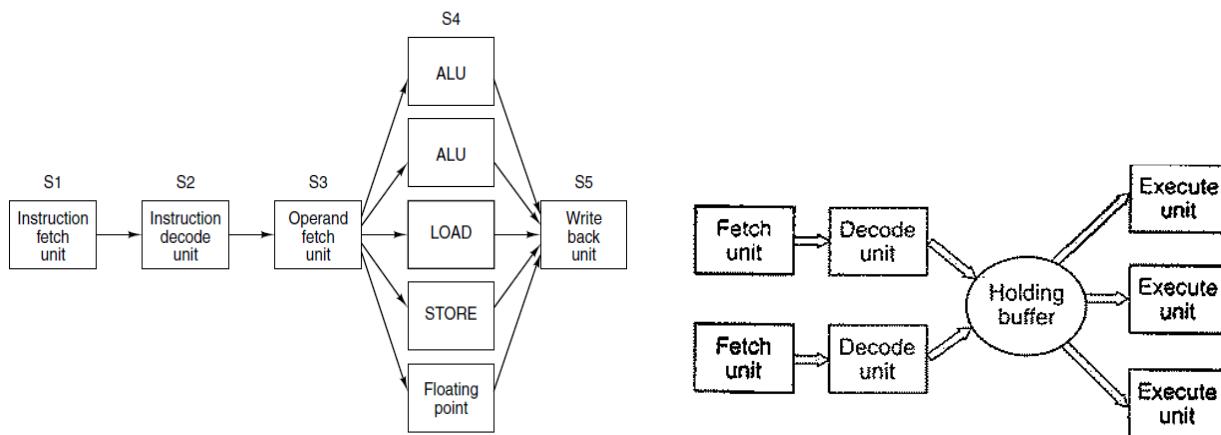


Figure 4. Superscalar architecture

1.4.1 Comparison between pipelining & superscalar

Pipelining

Divides an instruction into steps, and since each step is executed in a different part of the processor, multiple instructions can be in different "phases" each clock. In Pipelining multiple instructions are overlapped during execution. Sequencing unrelated activities such that they use different components at the same time.

Superscalar

Involves the processor being able to issue multiple instructions in a single clock with redundant facilities to execute an instruction within a single core. Multiple execution subunits able to do the same thing in parallel. Multiple sub-components capable of doing the same task simultaneously, but with the processor deciding how to do it.

1.5 Branch Prediction

Branch Predictor is a digital circuit that tries to guess which way a branch (e.g. an if-then-else structure) will go before this is known definitively. The purpose of the branch predictor is to improve the flow in the instruction pipeline because they must guess the address of the

next instruction to fetch before the current instruction has been executed. allows the processor to decode instructions beyond branches to keep the instruction pipeline full. The P6 processor family implements highly optimized branch prediction algorithms to predict the direction of the instruction.

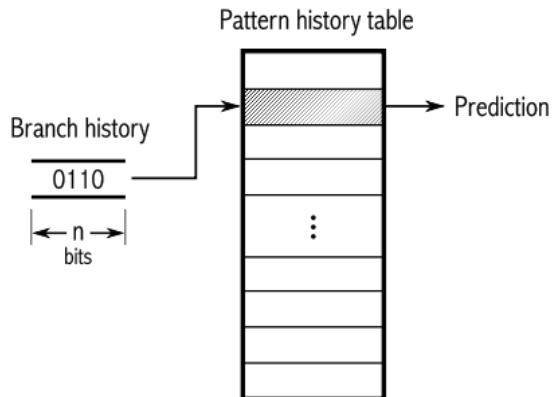


Figure 5. Branch Prediction

1.6 Out-of-order Execution

To improve the performance of the processor this can be done in two ways.

- By executing different sub-steps of sequential instructions simultaneously or even executing the instructions entirely simultaneously. Further improvement in the processor can be achieved through out-of-order.
- Out-of-order execution can be achieved by executing the instruction in a different order from the original order they appear.

The out-of-order execution core's ability to execute instructions out of order is a key factor in enabling parallelism. This feature enables the processor to reorder instructions so that if one micro-op is delayed; other micro-ops may proceed around it. The processor employs several buffers to smooth the flow of micro-ops. The core is designed to facilitate parallel execution. It can dispatch up to six micro-ops per cycle (this exceeds trace cache and retirement micro-op bandwidth). Most pipelines can start executing a new micro-op every cycle, so several instructions can be in flight at a time for each pipeline. A number of arithmetic logical unit (ALU) instructions can start at two per cycle; many floating-point instructions can start once every two cycles.

1.7 Speculative Execution

Speculative execution is an optimization technique where a computer system performs some task that may not be actually needed. The main idea is to do work before it is known whether that work will be needed at all, so as to prevent a delay that would have to be incurred by doing the work after it is known whether it is needed. If it turns out the work was not needed after all, any changes made by the work are reverted and the results are ignored.

It refers to the processor's ability to execute instructions that lie beyond a conditional branch that has not yet been resolved, and ultimately to commit the results in the order of the original instruction stream. To make speculative execution possible, the P6 processor microarchitecture decouples the dispatch and execution of instructions from the commitment of results. The processor's out-of-order execution core uses data-flow analysis to execute all available instructions in the instruction pool and store the results in temporary registers. The retirement unit then linearly searches the instruction pool for completed instructions that no longer have data dependencies with other instructions or unresolved branch predictions. When completed instructions are found, the retirement unit commits the results of these instructions to memory and/or the IA-32 registers (the processor's eight general-purpose registers and eight x87 FPU data registers) in the order they were originally issued and retires the instructions from the instruction pool.

1.8 P6 Family Microarchitecture

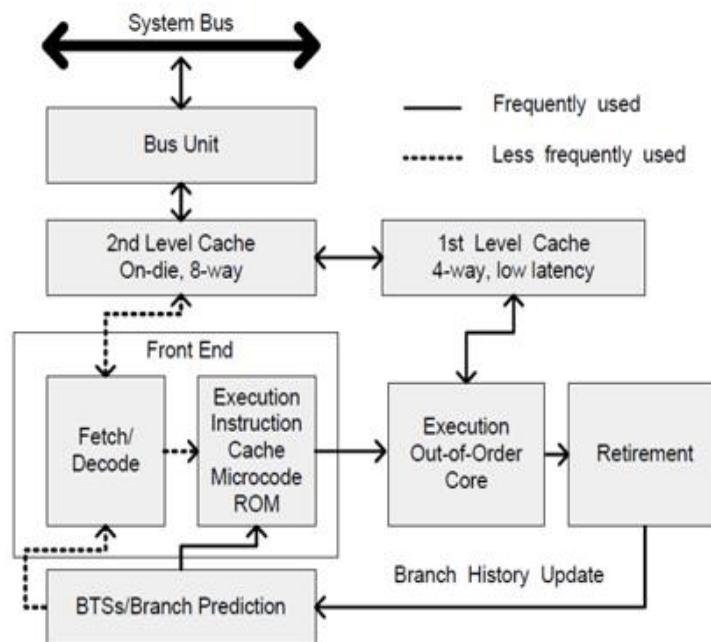


Figure 6. P6 Family Microarchitecture

- 2nd level cache, called advanced transfer cache.
- Three-way superscalar, pipelined architecture: - using parallel processing techniques, decode, dispatch and complete execution of (retire) up to three instructions per clock cycle.
- P6 processor family uses a decoupled, 12-stage super-pipeline that supports out-of-order instruction execution.
- The pipeline is divided into four sections (1st level and 2nd level caches, the front end, the out-of-order execution core, and the retire section). Instructions and data are supplied to these units through the bus interface unit.

1.9 Intel NetBurst® Microarchitecture

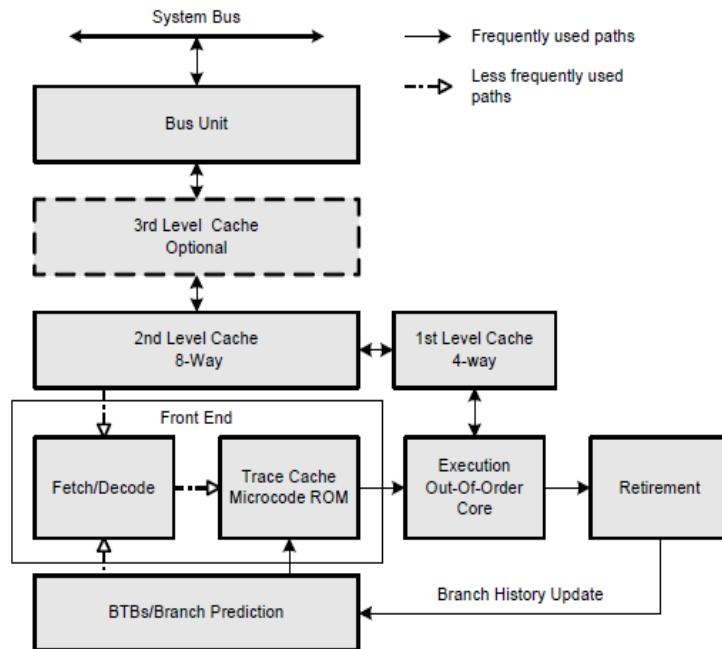


Figure 7. Intel Netburst Microarchitecture

Key features include

- The Rapid Execution Engine.
- Hyper-Pipelined Technology.
- Advanced Dynamic Execution.
- New cache subsystem.
- High-performance, quad-pumped bus interface to the Intel NetBurst microarchitecture system bus.
- Superscalar issue to enable parallelism.
- Expanded hardware registers with renaming to avoid register name space limitations.
- 64-byte cache line size (transfers data up to two lines per sector)

1.10 Intel® Core™ Microarchitecture

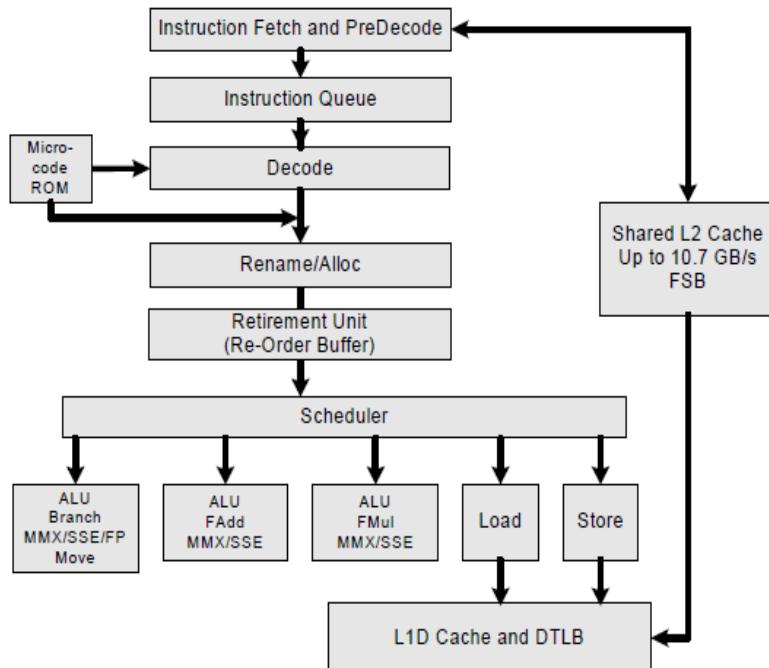


Figure 8. Intel Core Microarchitecture

Key features include

- Wide Dynamic Execution.
- Advanced Smart Cache.
- Advanced Smart Cache.
- Advanced Digital Media Boost.

1.11 Intel® Atom™ Microarchitecture

Key features include

- Advanced Micro-Ops Execution.
- Intel® Smart Cache.
- Efficient Memory Access.
- Intel® Digital Media Boost

2.0 Multiprocessors

The processing elements in an array processor are not independent CPUs, since there is only one control unit shared among all of them. Our first parallel system with multiple full-blown CPUs is the **multiprocessor**, a system with more than one CPU sharing a common memory, like a group of people in a room sharing a common blackboard. Since each CPU can read or write any part of memory, they must co-ordinate (in software) to avoid getting in each other's way. When two or more CPUs have the ability to interact closely, as is the case with multiprocessors, they are said to be tightly coupled.

2.1 Multi-Core Technology

A **multi-core processor** is a single computing component with two or more independent processing units called cores, which read and execute program instructions. The instructions are ordinary CPU instructions (such as add, move data, and branch) but the single processor can run multiple instructions on separate cores at the same time, increasing overall speed for programs amenable to parallel computing. Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP) or onto multiple dies in a single chip package.

2.2 Multiprocessors v/s multithreaded v/s Multicore processors

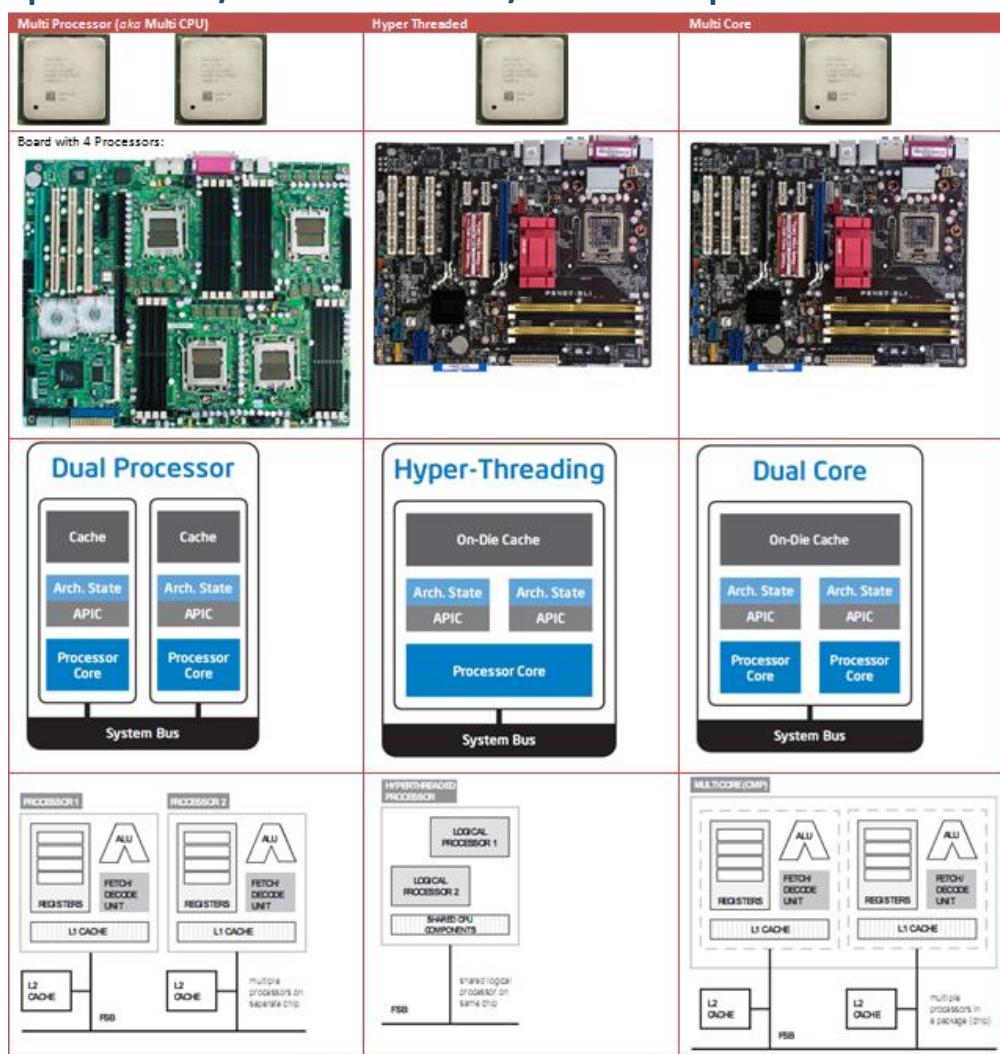


Figure 9. Difference between Dual Processor, Hyperthreading and Dual core processor

Features

- Enhanced performance
- Multiple applications
- Multiple users

- Multi-tasking inside an application
- High throughput and/or responsiveness
- Hardware sharing among CPU

2.3 Hyperthreading

Intel's implementation of Simultaneous Multithreading (SMT). Two threads executing concurrently in one processor. Intel® Hyper-Threading Technology uses processor resources more efficiently, enabling multiple threads to run on each core. Also increases processor throughput, improving overall performance. Improve productivity by doing more without slowing down

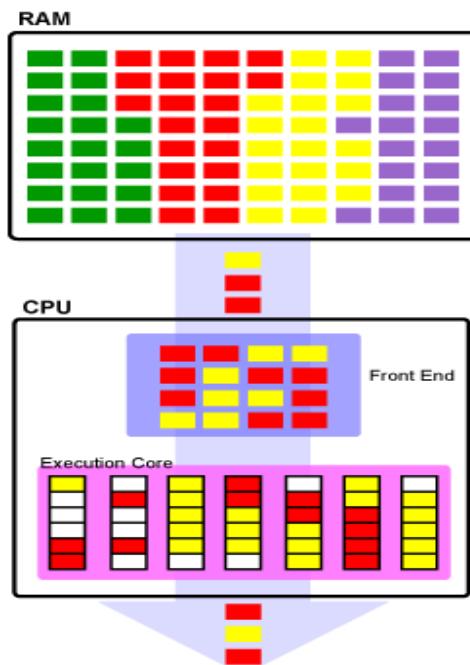


Figure 10. Single core processor with hyperthreading

3.0 Multiple Processor Initialization

A microprocessor performs a reset action for initializing the inside of the processor in response to assertion of a reset signal from the exterior upon power-on for execution of a user program, for example. The reset action includes a hardware reset action Wherein the reset signal asserted from the exterior of the microprocessor is applied to memory elements in the processor to reset the contents stored in the memory elements, and a software reset action Wherein a program for reset fetched from an external memory is executed in the processor to allow the user program to be ready for processing. Particularly, in the software reset action, When the reset signal is asserted, a reset vector entry is fetched from the external memory, and information contained in the entry is referred to, Whereby the program for reset is fetched.

IA-32 architecture defines a multiple-processor (MP) initialization protocol called the Multiprocessor Specification Version 1.4

- supports controlled booting of multiple processors without requiring dedicated system hardware.
- allows hardware to initiate the booting of a system without the need for a dedicated signal or a predefined boot processor
- allows all IA-32 processors to be booted in the same manner, including those supporting Intel Hyper-Threading Technology.
- MP initialization protocol also applies to MP systems using Intel 64 processors.

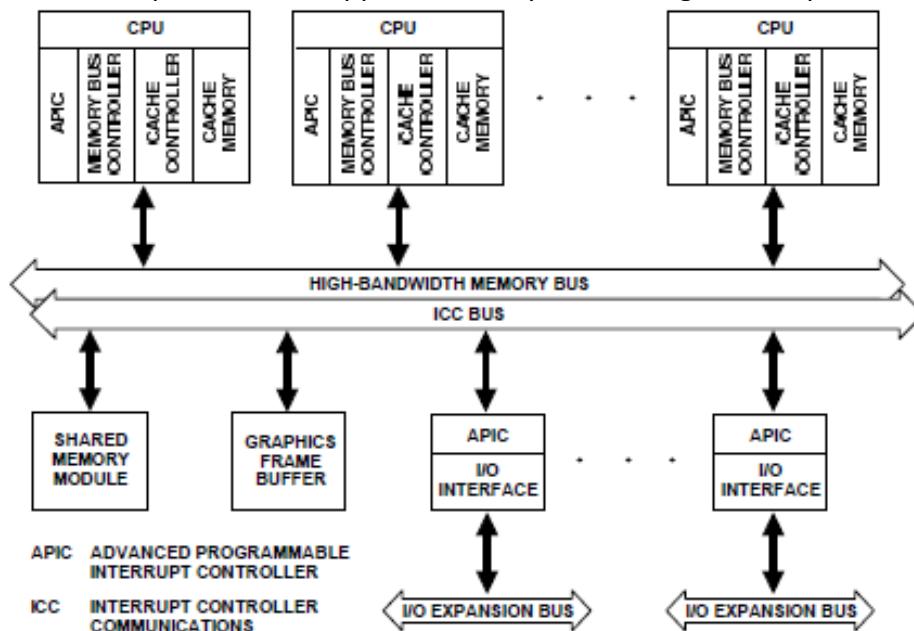


Figure 11. Relationship of Local APIC and I/O APIC

3.1 Bootstrap and Application Processors

MP initialization protocol defines two classes of processors

- bootstrap processor
- Application processors

Upon reset/Power ON system hardware dynamically selects one processor as BSP and rest as AP

Boot Strap Processor

- is responsible for initializing the system and for booting the operating system
- BSP flag is set in the IA32_APIC_BASE MSR
- executes the BIOS's boot-strap code to configure the APIC environment
- sets up system-wide data structures
- starts and initializes the Aps

- When the BSP and APs are initialized, the BSP then begins executing the operating-system initialization code

Application Processor

- are activated only after the operating system is up and running
- the APs complete a minimal self-configuration, then wait for a startup signal (a SIPI message) from the BSP processor.
- Upon receiving a SIPI message, an AP executes the BIOS AP configuration code, which ends with the AP being placed in halt state.

3.2 Advanced Programmable Interrupt Controller (APIC)

- based on a distributed architecture in which interrupt control functions are distributed between two basic functional units, the local unit and the I/O unit.
- local and I/O units communicate through a bus called the Interrupt Controller Communications (ICC) bus
- The local APIC units also provide inter processor interrupts (IPIs), which allow any processor to interrupt any other processor or set of processors.
- Each local APIC has a Local Unit ID Register and each I/O APIC has an I/O Unit ID Register.
- The ID serves as a physical name for each APIC unit

3.3 MP Initialization Protocol Algorithm for MP Systems

1. Each logical processor is assigned a unique APIC ID, based on system topology. The unique ID is a 32-bit value if the processor supports CPUID leaf 0BH, otherwise the unique ID is an 8-bit value.
2. Each logical processor is assigned a unique arbitration priority based on its APIC ID.
3. Each logical processor executes its internal BIST simultaneously with the other logical processors in the system.
4. Upon completion of the BIST, the logical processors use a hardware-defined selection mechanism to select the BSP and the APs from the available logical processors on the system bus. The BSP selection mechanism differs depending on the family, model, and stepping IDs of the processors.
5. As part of the boot-strap code, the BSP creates an ACPI table and/or an MP table and adds its initial APIC ID to these tables as appropriate.
6. At the end of the boot-strap procedure, the BSP sets a processor counter to 1, then broadcasts a SIPI message to all the APs in the system. Here, the SIPI message contains a vector to the BIOS AP initialization code
7. The first action of the AP initialization code is to set up a race (among the APs) to a BIOS initialization semaphore. The first AP to the semaphore begins executing the initialization code. As part of the AP initialization procedure, the AP adds its APIC ID number to the ACPI and/or MP tables as appropriate and increments the processor counter by 1. At the

completion of the initialization procedure, the AP executes a CLI instruction and halts itself.

8. When each of the APs has gained access to the semaphore and executed the AP initialization code, the BSP establishes a count for the number of processors connected to the system bus, completes executing the BIOS boot-strap code, and then begins executing operating-system boot-strap and start-up code.
9. While the BSP is executing operating-system boot-strap and start-up code, the APs remain in the halted state. In this state they will respond only to INITs, NMIs, and SMIs. They will also respond to snoops and to assertions of the STPCLK# pin.

4.0 Basic Execution Environment

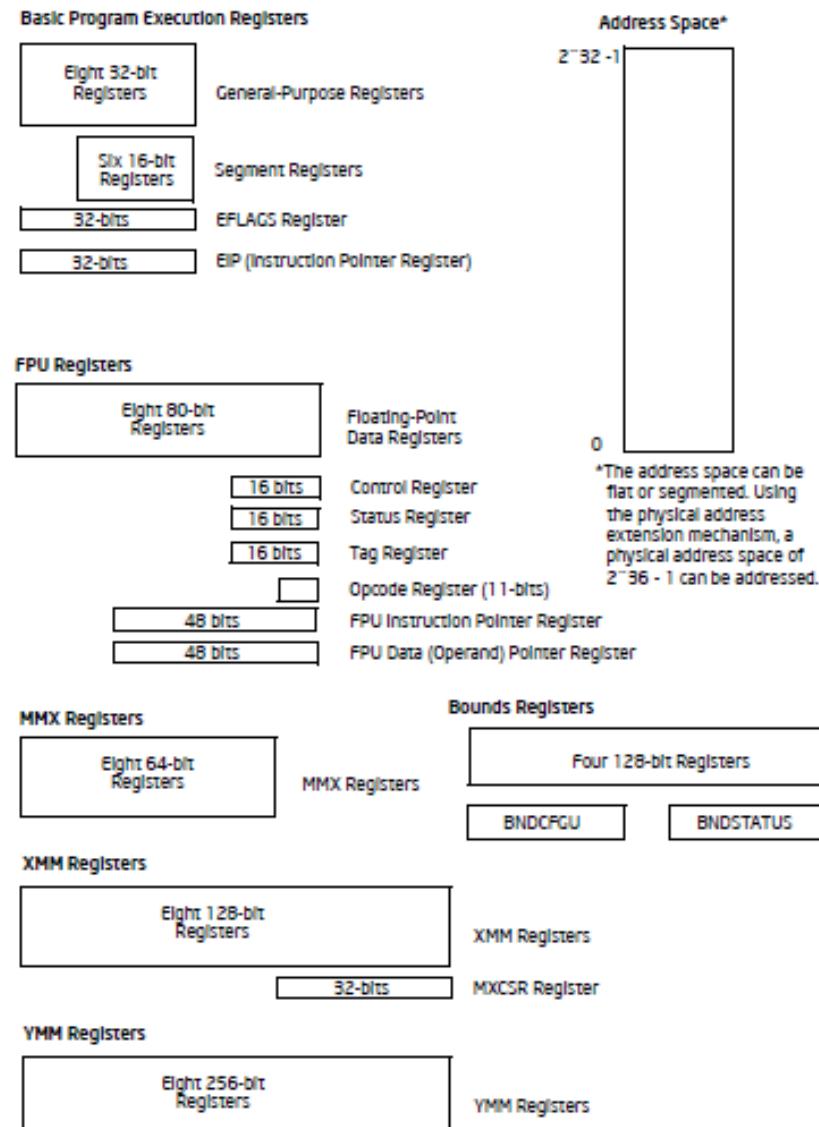


Figure 12. IA-32 Basic Execution Environment for Non-64-bit Modes

Any program running on a IA 32 processor is given a set of resources for executing instruction and for storing code, data and state information. These resources make up the basic execution environment for IA 32 architecture.

An Intel 64 processor supports the basic execution environment of an IA-32 processor, and a similar environment under IA-32e mode that can execute 64-bit programs (64-bit sub-mode) and 32-bit programs (compatibility submode).

The basic execution environment is used jointly by the application programs and the operating system or executive running on the processor.

- Address space — Any task or program running on an IA-32 processor can address a linear address space of up to 4 GBytes (2³² bytes) and a physical address space of up to 64 GBytes (2³⁶ bytes).
- Basic program execution registers — The eight general-purpose registers, the six segment registers, the EFLAGS register, and the EIP (instruction pointer) register comprise a basic execution environment in which to execute a set of general-purpose instructions. These instructions perform basic integer arithmetic on byte, word, and double word integers, handle program flow control, operate on bit and byte strings, and address memory.
- x87 FPU registers — The eight x87 FPU data registers, the x87 FPU control register, the status register, the x87 FPU instruction pointer register, the x87 FPU operand (data) pointer register, the x87 FPU tag register, and the x87 FPU opcode register provide an execution environment for operating on single-precision, double precision, and double extended-precision floating-point values, word integers, double word integers, quad word integers, and binary coded decimal (BCD) values.
- MMX registers — The eight MMX registers support execution of single-instruction, multiple-data (SIMD) operations on 64-bit packed byte, word, and double word integers.
- XMM registers — The eight XMM data registers and the MXCSR register support execution of SIMD operations on 128-bit packed single-precision and double-precision floating-point values and on 128-bit packed byte, word, double word, and quad word integers.
- YMM registers — The YMM data registers support execution of 256-bit SIMD operations on 256-bit packed single-precision and double-precision floating-point values and on 256-bit packed byte, word, double word, and quad word integers.
- Bounds registers — Each of the BND0-BND3 register stores the lower and upper bounds (64 bits each) associated with the pointer to a memory buffer. They support execution of the Intel MPX instructions.
- BNDCFGU and BNDSTATUS— BNDCFGU configures user mode MPX operations on bounds checking. BNDSTATUS provides additional information on the #BR caused by an MPX operation.

Stack — To support procedure or subroutine calls and the passing of parameters between procedures or subroutines, a stack and stack management resources are included in the execution environment.

4.1 Address space

Address a linear address space of up to 4GB (2^{32} bytes) and a physical space address up to 64GB (2^{36} bytes)

4.2 Basic program execution registers

4.2.1 General-Purpose Registers

The 32-bit general-purpose registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for holding the following items:

- Operands for logical and arithmetic operations
- Operands for address calculations
- Memory pointers

Although all of these registers are available for general storage of operands, results, and pointers, caution should be used when referencing the ESP register. The ESP register holds the stack pointer and as a general rule should not be used for another purpose. Many instructions assign specific registers to hold operands. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands. When using a segmented memory model, some instructions assume that pointers in certain registers are relative to specific segments. For instance, some instructions assume that a pointer in the EBX register points to a memory location in the DS segment.

- EAX Accumulator for operands and results data
Automatically used by multiplication and division instructions
- EBX Pointer to data in the DS segment
- ECX Counter for string and loop operations
- EDX I/O pointer
- ESI Pointer to data in the segment pointed to by the DS register; source pointer for string operations
- EDI Pointer to data (or destination) in the segment pointed to by the ES register;
destination pointer for string operations
- ESP Stack pointer (in the SS segment)
Used by PUSH and POP instructions, points to top of stack
- EBP Pointer to data on the stack (in the SS segment)
Used to reference parameters and local variables on the stack

General-Purpose Registers				16-bit	32-bit
31	16	15	8	7	0
		AH	AL	AX	EAX
		BH	BL	BX	EBX
		CH	CL	CX	ECX
		DH	DL	DX	EDX
			BP		EBP
			SI		ESI
			DI		EDI
			SP		ESP

Figure 13. General Purpose Register Names

If a 64-bit operand size is specified: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8-R15 are available. REX prefixes are used to generate 64-bit operand sizes or to reference registers R8-R15.

4.2.2 Segment registers

Each of the segment registers is associated with one of three types of storage: code, data, or stack. For example, the CS register contains the segment selector for the code segment, where the instructions being executed are stored. The processor fetches instructions from the code segment, using a logical address that consists of the segment selector in the CS register and the contents of the EIP register. The EIP register contains the offset within the code segment of the next instruction to be executed. The CS register cannot be loaded explicitly by an application program. Instead, it is loaded implicitly by instructions or internal processor operations that change program control (such as, procedure calls, interrupt handling, or task switching). The DS, ES, FS, and GS registers point to four data segments.

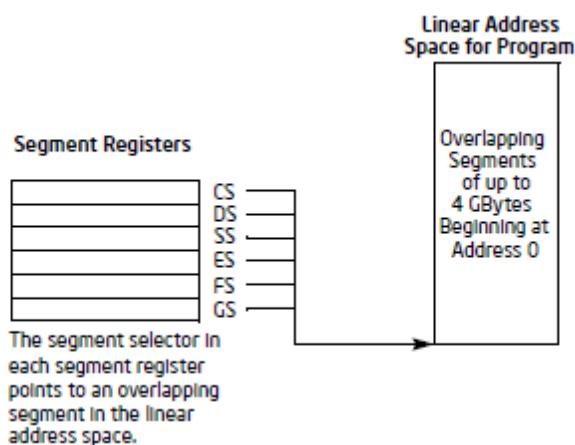


Figure 14. Use of Segment Registers for Flat Memory Model

The availability of four data segments permits efficient and secure access to different types of data structures. For example, four separate data segments might be created: one for the data structures of the current module, another for the data exported from a higher-level module, a third for a dynamically created data structure, and a fourth for data shared with another

program. To access additional data segments, the application program must load segment selectors for these segments into the DS, ES, FS, and GS registers, as needed. The SS register contains the segment selector for the stack segment, where the procedure stack is stored for the program, task, or handler currently being executed. All stack operations use the SS register to find the stack segment. Unlike the CS register, the SS register can be loaded explicitly, which permits application programs to set up multiple stacks and switch among them.

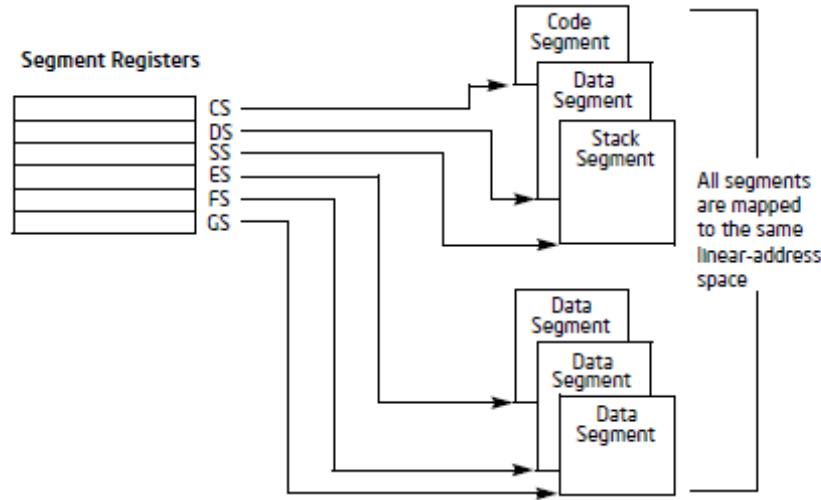


Figure 15. Use of Segment Registers in Segmented Memory Model

The four segment registers CS, DS, SS, and ES are the same as the segment registers found in the Intel 8086 and Intel 286 processors and the FS and GS registers were introduced into the IA-32 Architecture with the Intel386™ family of processors.

Segment Registers in 64-Bit Mode

In 64-bit mode: CS, DS, ES, SS are treated as if each segment base is 0, regardless of the value of the associated segment descriptor base. This creates a flat address space for code, data, and stack. FS and GS are exceptions. Both segment registers may be used as additional base registers in linear address calculations (in the addressing of local data and certain operating system data structures). Even though segmentation is generally disabled, segment register loads may cause the processor to perform segment access assists. During these activities, enabled processors will still perform most of the legacy checks on loaded values (even if the checks are not applicable in 64-bit mode). Such checks are needed because a segment register loaded in 64-bit mode may be used by an application running in compatibility mode. Limit checks for CS, DS, ES, SS, FS, and GS are disabled in 64-bit mode.

4.2.3 EFLAGS register

The 32-bit EFLAGS register contains a group of status flags, a control flag, and a group of system flags. Following initialization of the processor (either by asserting the RESET pin or the INIT pin),

the state of the EFLAGS register is 00000002H. Bits 1, 3, 5, 15, and 22 through 31 of this register are reserved. Software should not use or depend on the states of any of these bits.

Some of the flags in the EFLAGS register can be modified directly, using special-purpose instructions (described in the following sections). There are no instructions that allow the whole register to be examined or modified directly.

The following instructions can be used to move groups of flags to and from the procedure stack or the EAX register: LAHF, SAHF, PUSHF, PUSHFD, POPF, and POPFD. After the contents of the EFLAGS register have been transferred to the procedure stack or EAX register, the flags can be examined and modified using the processor's bit manipulation instructions (BT, BTS, BTR, and BTC).

When suspending a task (using the processor's multitasking facilities), the processor automatically saves the state of the EFLAGS register in the task state segment (TSS) for the task being suspended. When binding itself to a new task, the processor loads the EFLAGS register with data from the new task's TSS.

When a call is made to an interrupt or exception handler procedure, the processor automatically saves the state of the EFLAGS registers on the procedure stack. When an interrupt or exception is handled with a task switch, the state of the EFLAGS register is saved in the TSS for the task being suspended.

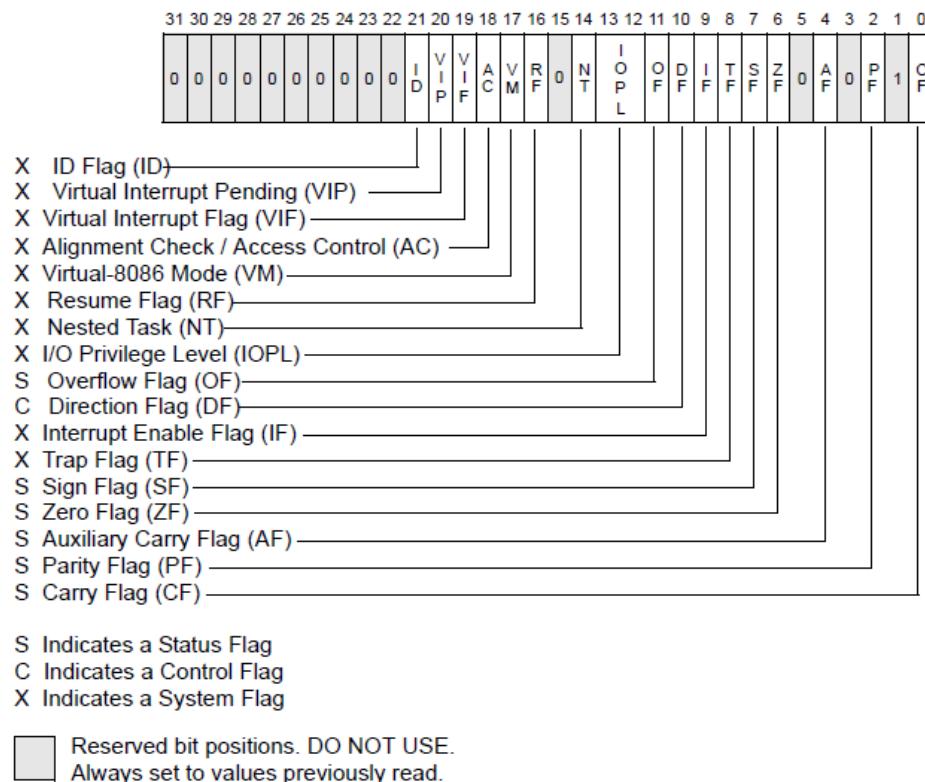


Figure 16. EFLAG Register

Status flags

- CF (bit 0) Carry flag — Set if an arithmetic operation generates a carry or a borrow out of the most significant bit of the result; cleared otherwise. indicates an overflow condition for unsigned-integer arithmetic; also used in multiple-precision arithmetic
- PF (bit 2) Parity flag — Set if the least-significant byte of the result contains an even number of 1 bit; cleared otherwise.
- AF (bit 4) Auxiliary Carry flag — Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise; used in binary-coded decimal (BCD) arithmetic.
- ZF (bit 6) Zero flag — Set if the result is zero; else cleared.
- SF (bit 7) Sign flag — Set equal to the MSB of the result, which is the sign bit of a signed integer. (0 for +ve value and 1 for -ve value.)
- OF (bit 11) Overflow flag — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise; indicates an overflow condition for signed-integer (two's complement) arithmetic.

Control flag

- DF (bit 10) Direction flag — control string instructions; if set, process string from high to low addresses (auto decrement),else vice versa

System Flags and IOPL Field

- TF (bit 8) Trap flag — Set to enable single-step mode for debugging; clear to disable single-step mode.
- IF (bit 9) Interrupt enable flag — Controls the response of the processor to maskable interrupt requests. Set to respond to maskable interrupts; cleared to inhibit maskable interrupts.
- IOPL (bits 12, 13) I/O privilege level field — Indicates the I/O privilege level of the currently running program or task. The current privilege level (CPL) of the currently running program or task must be less than or equal to the I/O privilege level to access the I/O address space.
- NT (bit 14) Nested task flag — Controls the chaining of interrupted and called tasks. Set when the current task is linked to the previously executed task; cleared when not linked.
- RF (bit 16) Resume flag — Controls the processor's response to debug exceptions.
- VM (bit 17) Virtual-8086 mode flag — Set to enable virtual-8086 mode; clear to return to protected mode without virtual-8086 mode semantics.

- AC (bit 18) Alignment check (or access control) flag — If the AM bit is set in the CR0 register, alignment checking of user-mode data accesses is enabled if and only if this flag is 1. If the SMAP bit is set in the CR4 register, explicit supervisor-mode data accesses to user-mode pages are allowed if and only if this bit is 1.
- VIF (bit 19) Virtual interrupt flag — Virtual image of the IF flag. Used in conjunction with the VIP flag. (To use this flag and the VIP flag the virtual mode extensions are enabled by setting the VME flag in control register CR4.)
- VIP (bit 20) Virtual interrupt pending flag — Set to indicate that an interrupt is pending; clear when no interrupt is pending. (Software sets and clears this flag; the processor only reads it.) Used in conjunction with the VIF flag.
- ID (bit 21) Identification flag — The ability of a program to set or clear this flag indicates support for the CPUID instruction.

4.2.4 EIP(Instruction Pointer)

EIP contains the offset in the current code segment for the next instruction to be executed. It is advanced from one instruction boundary to the next in straight-line code or it is moved ahead or backwards by a number of instructions when executing JMP, JCC, CALL, RET, and IRET instructions. The EIP register cannot be accessed directly by software

4.3 MMX registers

8 MMX registers

support execution of single-instruction, multiple-data (SIMD) operations on 64-bit packed byte, word, and double word integers

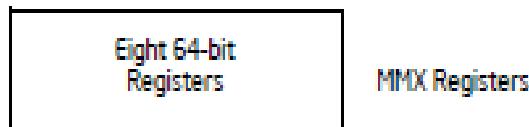


Figure 17. MMX Registers

4.4 XMM registers

8 XMM data registers

support execution of SIMD operations on 128-bit packed single-precision and double-precision floating-point values and on 128-bit packed byte, word, double word, and quad word integers

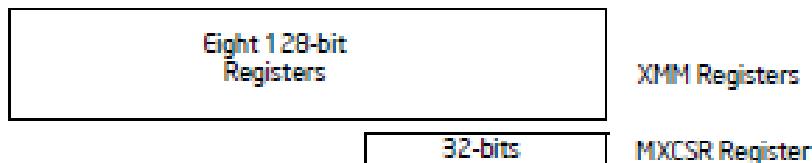


Figure 18. XMM Registers

4.5 YMM registers

support execution of 256-bit SIMD operations on 256-bit packed single-precision and double-precision floating-point values and on 256-bit packed byte, word, double word, and Quad word integers



Figure 18. YMM Registers

4.6 Bounds registers



Figure 19. Bounds Registers

Each of the BND0-BND3 register stores the lower and upper bounds (64 bits each) associated with the pointer to a memory buffer. They support execution of the Intel MPX instructions.

BNDCFGU configures user mode MPX operations on bounds checking. BNDSTATUS provides additional information on the #BR caused by an MPX operation.

Other architectural features apart from Basic Execution Environment include

I/O ports

transfers of data to and from input/output (I/O) ports

Control registers

5 control registers (CR0 through CR4) determine the operating mode of the processor and the characteristics of the currently executing task

Memory management registers

The GDTR, IDTR, task register, and LDTR specify the locations of data structures used in protected mode memory management.

Debug registers

debug registers (DR0 through DR7) control and allow monitoring of the processor's debugging operations

Memory type range registers (MTRRs)

The MTRRs are used to assign memory types to regions of memory

Machine specific registers (MSRs)

used to control and report on processor performance

Virtually all MSRs handle system related functions and are not accessible to an application program except the time-stamp counter

Machine check registers

consist of a set of control, status, and error reporting MSRs that are used to detect and report on hardware (machine) errors.

Performance monitoring counters

allow processor performance events to be monitored

5.0 Modes of operation

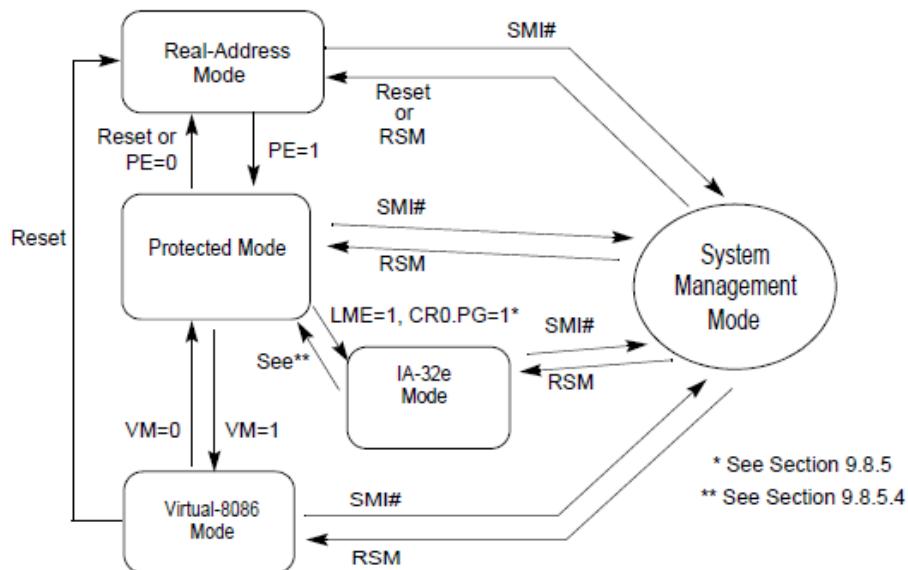


Figure 20. Transitions Among the Processor's Operating Modes

Real-address mode. This mode implements the programming environment of the Intel 8086 processor with a few extensions (such as the ability to switch to protected or system management mode). The processor is placed in real-address mode following power-up or a reset.

Protected mode: This mode is the native state of the processor. In this mode all instructions and architectural features are available, providing the highest performance and capability. This is the recommended mode for all new applications and operating systems. Among the capabilities of protected mode is the ability to directly execute “real-address mode” 8086 software in a protected, multi-tasking environment. This feature is called virtual-8086 mode, although it is not

actually a processor mode. Virtual-8086 mode is actually a protected mode attribute that can be enabled for any task.

Virtual-8086 mode

Hybrid of Protected: While in protected mode, the processor can directly execute real address mode software such as **MS-DOS program** in a safe multitasking environment. If an MS-DOS program crashes or attempts to write data into the system memory area, it will **not** affect other programs running at the same time. Each program has its own 8086 computer: **Windows XP** can execute **multiple** separate virtual-8086 sessions at the same time.

System management mode: This mode provides an operating system or executive with a transparent mechanism for implementing platform-specific functions such as power management and system security. The processor enters SMM when the external SMM interrupt pin (SMI#) is activated or an SMI is received from the advanced programmable interrupt controller (APIC). In SMM, the processor switches to a separate address space while saving the entire context of the currently running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the system management interrupt. SMM was introduced with the Intel386 TM SL and Intel486TM SL processors and became a standard IA-32 feature with the Pentium processor family.

Intel 64 adds IA-32e mode with 2 sub modes

Compatibility mode (sub-mode of IA-32e mode) — Compatibility mode permits most legacy 16-bit and 32-bit applications to run without recompilation under a 64-bit operating system. For brevity, the compatibility sub-mode is referred to as compatibility mode in IA-32 architecture. The execution environment of compatibility mode is the same as described in Section 3.2. Compatibility mode also supports all of the privilege levels that are supported in 64-bit and protected modes. Legacy applications that run in Virtual 8086 mode or use hardware task management will not work in this mode.

Compatibility mode is enabled by the operating system (OS) on a code segment basis. This means that a single 64-bit OS can support 64-bit applications running in 64-bit mode and support legacy 32-bit applications (not recompiled for 64-bits) running in compatibility mode.

Compatibility mode is similar to 32-bit protected mode. Applications access only the first 4 G Byte of linear address space. Compatibility mode uses 16-bit and 32-bit address and operand sizes. Like protected mode, this mode allows applications to access physical memory greater than 4 GByte using PAE (Physical Address Extensions).

64-bit mode (sub-mode of IA-32e mode) — This mode enables a 64-bit operating system to run applications written to access 64-bit linear address space. For brevity, the 64-bit sub-mode is referred to as 64-bit mode in IA-32 architecture. 64-bit mode extends the number of general purpose registers and SIMD extension registers from 8 to 16. General purpose registers are

widened to 64 bits. The mode also introduces a new opcode prefix (REX) to access the register extensions. See Section 3.2.1 for a detailed description.

64-bit mode is enabled by the operating system on a code-segment basis. Its default address size is 64 bits and its default operand size is 32 bits. The default operand size can be overridden on an instruction-by instruction basis using a REX opcode prefix in conjunction with an operand size override prefix.

REX prefixes allow a 64-bit operand to be specified when operating in 64-bit mode. By using this mechanism, many existing instructions have been promoted to allow the use of 64-bit registers and 64-bit addresses

5.1 Memory models

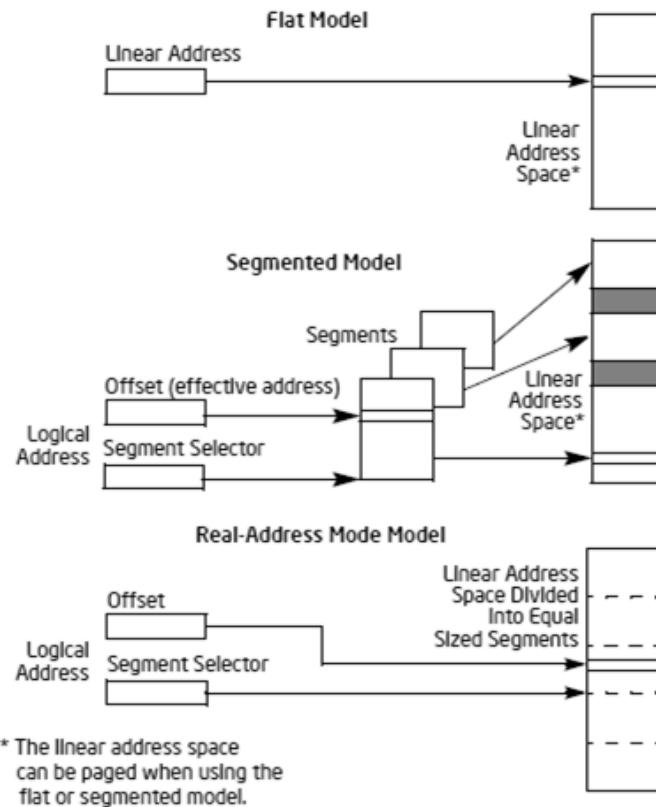


Figure 21. Three Memory Management Models

Flat memory model — Memory appears to a program as a single, continuous address space (Figure 3-3). This space is called a **linear address space**. Code, data, and stacks are all contained in this address space. Linear address space is byte addressable, with addresses running contiguously from 0 to $2^{32}-1$ (if not in 64-bit mode). An address for any byte in linear address space is called a **linear address**.

Segmented memory model — Memory appears to a program as a group of independent address spaces called segments. Code, data, and stacks are typically contained in separate segments. To

address a byte in a segment, a program issues a logical address. This consists of a segment selector and an offset (logical addresses are often referred to as far pointers). The segment selector identifies the segment to be accessed and the offset identifies a byte in the address space of the segment. Programs running on an IA-32 processor can address up to 16,383 segments of different sizes and types, and each segment can be as large as 232 bytes. Internally, all the segments that are defined for a system are mapped into the processor's linear address space. To access a memory location, the processor thus translates each logical address into a linear address. This translation is transparent to the application -program.

The primary reason for using segmented memory is to increase the reliability of programs and systems. For example, placing a program's stack in a separate segment prevents the stack from growing into the code or data space and overwriting instructions or data, respectively.

Real-address mode memory model — This is the memory model for the Intel 8086 processor. It is supported to provide compatibility with existing programs written to run on the Intel 8086 processor. The real address mode uses a specific implementation of segmented memory in which the linear address space for the program and the operating system/executive consists of an array of segments of up to 64 KBytes in size each. The maximum size of the linear address space in real-address mode is 220 bytes.

With the flat or the segmented memory model, linear address space is mapped into the processor's physical address space either directly or through paging. When using direct mapping (paging disabled), each linear address has a one-to-one correspondence with a physical address. Linear addresses are sent out on the processor's address lines without translation. When using the IA-32 architecture's paging mechanism (paging enabled), linear address space is divided into pages which are mapped to virtual memory. The pages of virtual memory are then mapped as needed into physical memory. When an operating system or executive uses paging, the paging mechanism is transparent to an application program. All that the application sees is linear address space.

5.2 Modes of Operation vs. Memory Model

Protected mode- any model

- Can use any model,virtual-8086 mode generally uses real addressing mode memory model
- Model used depends on the design of the OS
- In multitasking, individual tasks can use different models

Real address mode

- Only supports real address memory model
- System management mode
- Processor switches to separate address space (system management RAM)

- The memory model used to address bytes in this address space is similar to the real-address mode model

Compatibility mode

- The effect of segmentation is the same as it is in 32-bit protected mode semantics.

64-bit mode

- Segmentation is generally disabled
- Segmented and real address mode are not available in 64 bit mode

6.0 Memory Management

6.1 Basic Flat Model

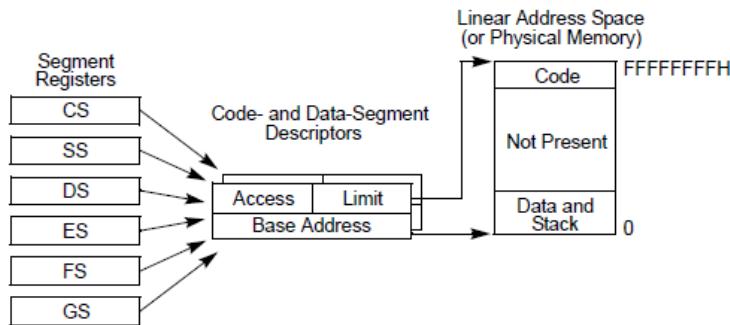


Figure 22. Basic Flat Model

The simplest memory model for a system is the basic “flat model,” in which the operating system and application programs have access to a continuous, unsegmented address space. To the greatest extent possible, this basic flat model hides the segmentation mechanism of the architecture from both the system designer and the application programmer.

To implement a basic flat memory model with the IA-32 architecture, at least two segment descriptors must be created, one for referencing a code segment and one for referencing a data segment. Both of these segments, however, are mapped to the entire linear address space: that is, both segment descriptors have the same base address value of 0 and the same segment limit of 4 GBytes. By setting the segment limit to 4 GBytes, the segmentation mechanism is kept from generating exceptions for out of limit memory references, even if no physical memory resides at a particular address. ROM (EPROM) is generally located at the top of the physical address space, because the processor begins execution at FFFF_FFFOH. RAM (DRAM) is placed at the bottom of the address space because the initial base address for the DS data segment after reset initialization is 0.

6.2 Protected Flat Model

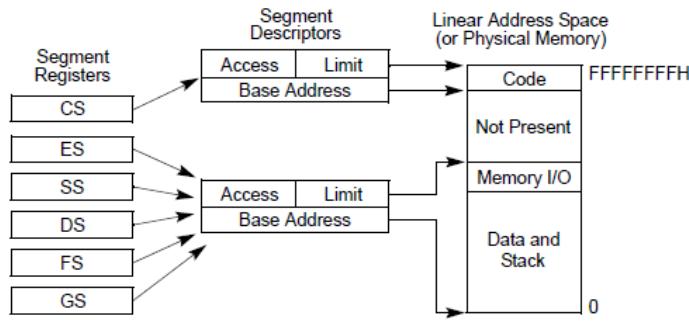


Figure 23. Protected Flat Model

The protected flat model is similar to the basic flat model, except the segment limits are set to include only the range of addresses for which physical memory actually exists. A general-protection exception (#GP) is then generated on any attempt to access nonexistent memory. This model provides a minimum level of hardware protection against some kinds of program bugs.

More complexity can be added to this protected flat model to provide more protection. For example, for the paging mechanism to provide isolation between user and supervisor code and data, four segments need to be defined: code and data segments at privilege level 3 for the user, and code and data segments at privilege level 0 for the supervisor. Usually these segments all overlay each other and start at address 0 in the linear address space. This flat segmentation model along with a simple paging structure can protect the operating system from applications, and by adding a separate paging structure for each task or process, it can also protect applications from each other. Similar designs are used by several popular multitasking operating systems.

6.3 Multi-Segment Model

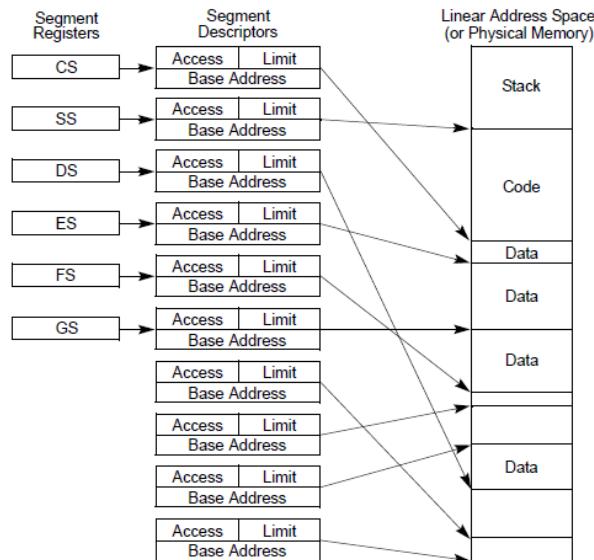


Figure 24. Multi Segment Model

A multi-segment model uses the full capabilities of the segmentation mechanism to provide hardware enforced protection of code, data structures, and programs and tasks. Here, each program (or task) is given its own table of segment descriptors and its own segments. The segments can be completely private to their assigned programs or shared among programs. Access to all segments and to the execution environments of individual programs running on the system is controlled by hardware.

Access checks can be used to protect not only against referencing an address outside the limit of a segment, but also against performing disallowed operations in certain segments. For example, since code segments are designated as read-only segments, hardware can be used to prevent writes into code segments. The access rights information created for segments can also be used to set up protection rings or levels. Protection levels can be used to protect operating-system procedures from unauthorized access by application programs.

6.4 Logical Address to Linear Address Translation

At the system-architecture level in protected mode, the processor uses two stages of address translation to arrive at a physical address: logical-address translation and linear address space paging.

Even with the minimum use of segments, every byte in the processor's address space is accessed with a logical address. A logical address consists of a 16-bit segment selector and a 32-bit offset. The segment selector identifies the segment the byte is located in and the offset specifies the location of the byte in the segment relative to the base address of the segment.

The processor translates every logical address into a linear address. A linear address is a 32-bit address in the processor's linear address space. Like the physical address space, the linear address space is a flat (unsegmented), 2³²-byte address space, with addresses ranging from 0 to FFFFFFFFH. The linear address space contains all the segments and system tables defined for a system.

To translate a logical address into a linear address, the processor does the following:

1. Uses the offset in the segment selector to locate the segment descriptor for the segment in the GDT or LDT and reads it into the processor. (This step is needed only when a new segment selector is loaded into a segment register.)
2. Examines the segment descriptor to check the access rights and range of the segment to insure that the segment is accessible and that the offset is within the limits of the segment.
3. Adds the base address of the segment from the segment descriptor to the offset to form a linear address.

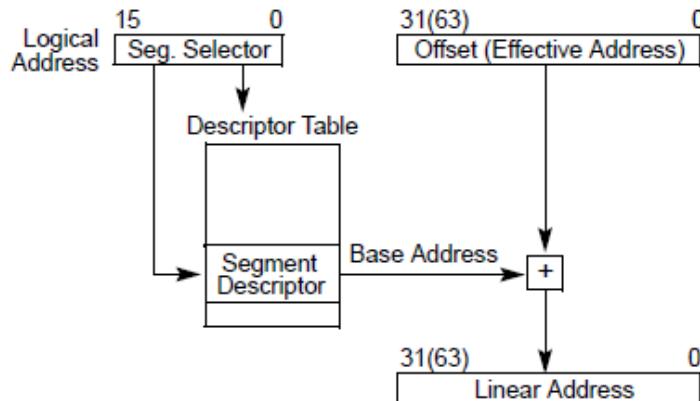


Figure 25. Logical Address to Linear Address Translation

If paging is not used, the processor maps the linear address directly to a physical address (that is, the linear address goes out on the processor's address bus). If the linear address space is paged, a second level of address translation is used to translate the linear address into a physical address.

6.4.1 Segment Selectors

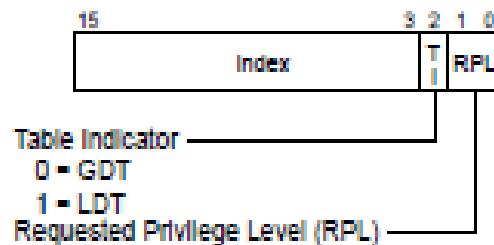


Figure 26. Segment Selector

A segment selector is a 16-bit identifier for a segment (see Figure 3-6). It does not point directly to the segment, but instead points to the segment descriptor that defines the segment. A segment selector contains the following items:

Index (Bits 3 through 15) — Selects one of 8192 descriptors in the GDT or LDT. The processor multiplies the index value by 8 (the number of bytes in a segment descriptor) and adds the result to the base address of the GDT or LDT (from the GDTR or LDTR register, respectively).

TI (table indicator) flag (Bit 2) — Specifies the descriptor table to use: clearing this flag selects the GDT; setting this flag selects the current LDT.

Requested Privilege Level (RPL) (Bits 0 and 1) — Specifies the privilege level of the selector. The privilege level can range from 0 to 3, with 0 being the most privileged level.

6.4.2 Segment Registers

Visible Part	Hidden Part
Segment Selector	Base Address, Limit, Access Information
	CS
	SS
	DS
	ES
	FS
	GS

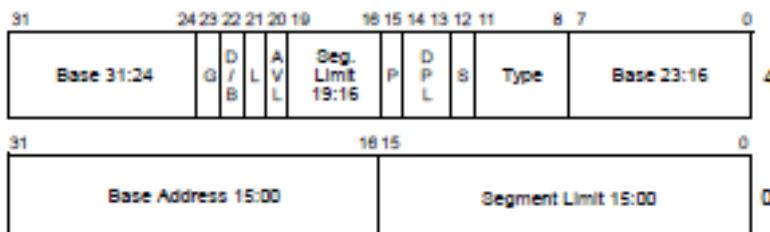
Figure 27. Segment Registers

When a segment selector is loaded into the visible part of a segment register, the processor also loads the hidden part of the segment register with the base address, segment limit, and access control information from the segment descriptor

In systems in which multiple processors have access to the same descriptor tables, it is the responsibility of software to reload the segment registers when the descriptor tables are modified

6.4.3 Segment Descriptors

It is a data structure in a GDT or LDT that provides the processor with the size and location of a segment, as well as access control and status information.



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure 28. Segment Descriptor

6.4.4 Segment Descriptor Tables

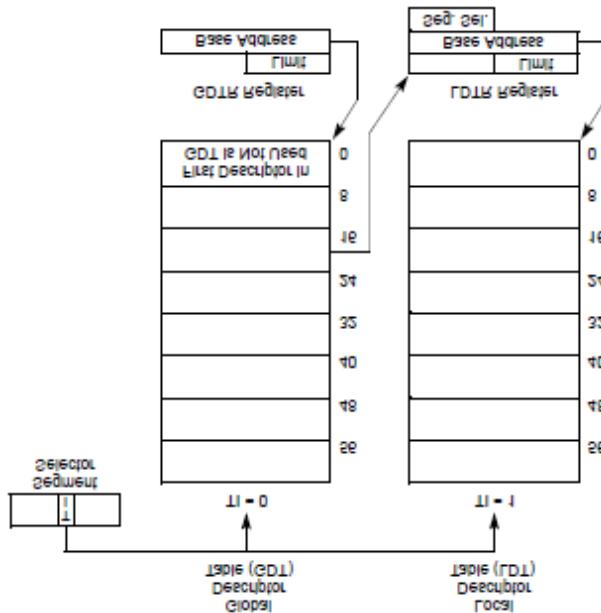


Figure 29. Global and Local Descriptor Tables

2 types- global descriptor table (GDT) and local descriptor tables (LDT)

GDT- which may be used for all programs and tasks in the system

LDT- can be defined for each separate task being run, or some or all tasks can share the same LDT

Global descriptor table (GDT)

- Only one GDT table is provided by the operating system
- GDT table contains segment descriptors for all programs
- Also used by the operating system itself
- Table is initialized during boot up
- GDT table address is stored in the GDTR register
- Modern operating systems (Windows-XP) use one GDT table

Local descriptor table (LDT)

- Another choice is to have a unique LDT table for each program
- LDT table contains segment descriptors for only one program
- LDT table address is stored in the LDTR register

memory management facilities of the IA-32 architecture are divided into two parts: segmentation and paging

Segmentation

provides a mechanism of isolating individual code, data, and stack modules so that multiple programs (or tasks) can run on the same processor without interfering with one another

Paging

provides a mechanism for implementing a conventional demand-paged, virtual-memory system where sections of a program's execution environment are mapped into physical memory as needed can also be used to provide isolation between multiple tasks

6.4.5 Overview

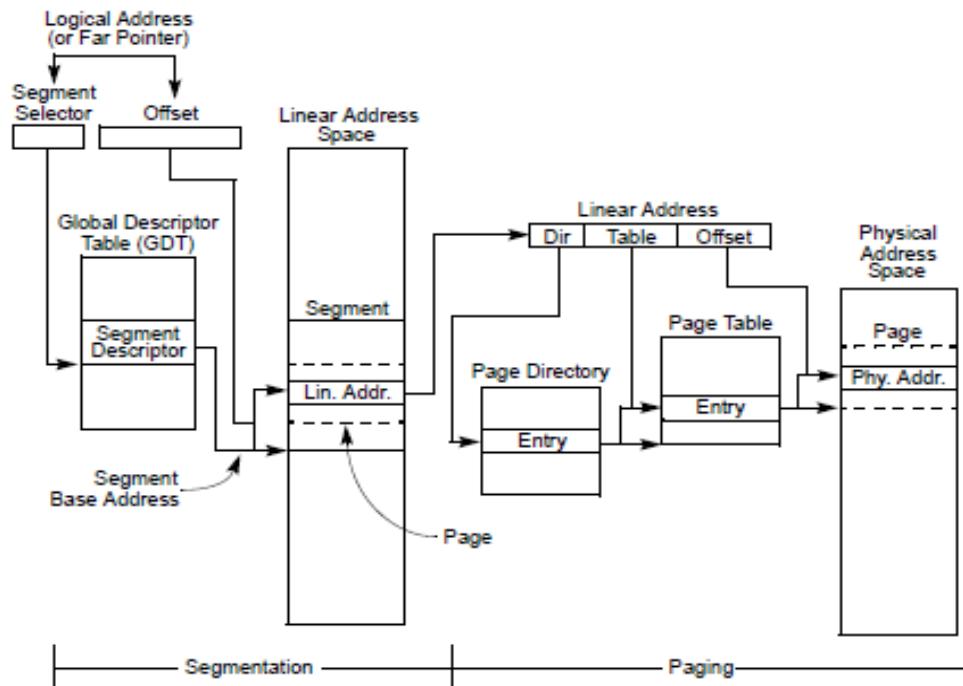


Figure 30. Segmentation and Paging

- processor's addressable memory space (called the **linear address space**) is divided into smaller protected address spaces called **segments** to hold the code, data, and stack for a program or to hold system data structures
- If more than one program (or task) is running on a processor, each program can be assigned its own set of segments.
- Processor ensures that programs don't write on each other's segments
- A logical address consists of a **segment selector** (provides base address) and an **offset**
- The base address plus the offset thus forms a **linear address** in the processor's linear address space.
- If paging is not used, the linear address space of the processor is mapped directly into the physical address space of processor.

- Paging supports a “virtual memory” environment where a large linear address space is simulated with a small amount of physical memory (RAM and ROM) and some disk storage
- each segment is divided into pages (typically 4 KBytes each in size), which are stored either in physical memory or on the disk.
- If implemented properly, the swapping of pages between physical memory and the disk is transparent to the correct execution of a program.

Chapter-3

Interrupts

1.0 Introduction

Exceptions are so important in embedded systems, without exception the development of systems would be a very complex task. With exceptions we can detect bugs in the application, errors in memory access and finally debug it by placing breakpoints and building the program with debugging information.

Interrupts which are kinds of exceptions are essential in embedded systems. It enables the system to deal with external events by receiving interrupt signals telling the CPU that there is something to be done instead of the alternative way of doing the same operation by the polling mechanism which wastes the CPU time in looping forever checking some flags to know that the event occurred.

Due to the fact that systems are going more complex day after day, we have nowadays systems with more than one interrupt source. That is why an interrupt handling scheme is needed to define how different cases will be handled. We may need priorities to be assigned to different interrupts and in some other cases we may need nested handling capabilities.

1.1 Definition:

An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. Whenever an interrupt occurs, the controller completes the execution of the current instruction and starts the execution of an **Interrupt Service Routine (ISR)** or **Interrupt Handler**. ISR tells the processor or controller what to do when the interrupt occurs. The interrupts can be either hardware interrupts or software interrupts.

1.1.1 Hardware Interrupt

A hardware interrupt is an electronic alerting signal sent to the processor from an external device, like a disk controller or an external peripheral. For example, when we press a key on the keyboard or move the mouse, they trigger hardware interrupts which cause the processor to read the keystroke or mouse position.

1.1.2 Software Interrupt

A software interrupt is caused either by an exceptional condition or a special instruction in the instruction set which causes an interrupt when it is executed by the processor. For example, if the processor's arithmetic logic unit runs a command to divide a number by zero, to cause a divide-by-zero exception, thus causing the computer to abandon the calculation or display an error message. Software interrupt instructions work similar to subroutine calls.

2.0 Need of interrupts:

An **interrupt** is a condition that causes the **microprocessor** to temporarily work on a different task, and then later return to its previous task. **Interrupts** can be internal or external. ... Notice that when the **interrupt** (Int) occurs, the program stops executing and the microcontroller begins to execute the ISR.

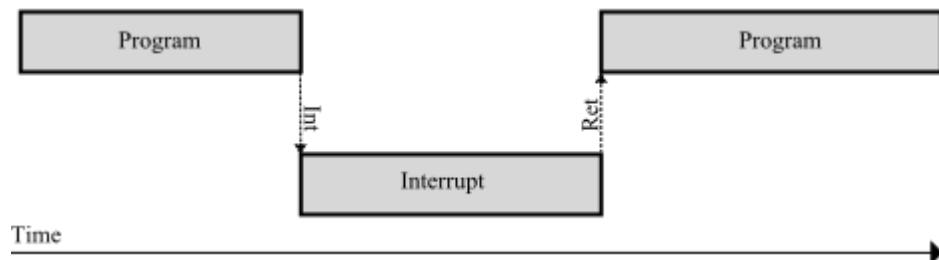


Fig 1 microprocessor design/interrupt

3.0 Polling Vs Interrupts:

The state of continuous monitoring is known as **polling**. The microcontroller keeps checking the status of other devices; and while doing so, it does no other operation and consumes all its processing time for monitoring. This problem can be addressed by using interrupts.

In the interrupt method, the controller responds only when an interruption occurs. Thus, the controller is not required to regularly monitor the status (flags, signals etc.) of interfaced and inbuilt devices.

3.1 Interrupts v/s Polling

Here is an analogy that differentiates an interrupt from polling –

Interrupt	Polling
An interrupt is like a shopkeeper . If one needs a service or product, he goes to him and apprises him of his needs. In case of interrupts, when the flags or signals are received, they notify the controller that they need to be serviced.	The polling method is like a salesperson . The salesman goes from door to door while requesting to buy a product or service. Similarly, the controller keeps monitoring the flags or signals one by one for all devices and provides service to whichever component that needs its service.

4.0 Types of Interrupts

Interrupts can be broadly classified as

- Hardware Interrupts

These are interrupts caused by the connected devices.

- Software Interrupts

These are interrupts deliberately introduced by software instructions to generate user defined Exceptions.

- Trap Interrupts

These are interrupts used by the processor alone to detect any exception such as divide by zero.

Depending on the service the interrupts also can be classified as:

- Fixed interrupt

- Address of the ISR built into microprocessor, cannot be changed
- Either ISR stored at address or a jump to actual ISR stored if not enough bytes available

- Vectored interrupt

- Peripheral must provide the address of the ISR
- Common when microprocessor has multiple peripherals connected by a system bus

- **Compromise between fixed and vectored interrupts**

- One interrupt pin
- Table in memory holding ISR addresses (maybe 256 words)
- Peripheral doesn't provide ISR address, but rather index into table
 - Fewer bits are sent by the peripheral
 - Can move ISR location without changing peripheral

- Maskable vs. Non-maskable interrupts

- Maskable: programmer can set bit that causes processor to ignore interrupt
 - This is important when the processor is executing a time-critical code

- Non-maskable: a separate interrupt pin that can't be masked

- Typically reserved for drastic situations, like power failure requiring immediate backup of data to non-volatile memory

5.0 Interrupt Vector Table:

An "**interrupt vector table**" (IVT) is a data structure that associates a list of interrupt handlers with a list of interrupt requests in a table of interrupt vectors. Each entry of the interrupt vector table, called an interrupt vector, is the address of an interrupt handler. While the concept is common across processor architectures, IVTs may be implemented in architecture-

specific fashions. For example, a dispatch table is one method of implementing an interrupt vector table.

Background

Most processors have an interrupt vector table, including chips from Intel, AMD, Infineon, Microchip, Atmel, NXP, etc.

An interrupt vector table is used in the three most popular methods of finding the starting address of the interrupt service routine:

1. "Predefined"

The "predefined" method loads the program counter (PC) directly with the address of some entry inside the interrupt vector table. The jump table itself contains executable code. While in principle an extremely short interrupt handler could be stored entirely inside the interrupt vector table, in practice the code at each entry is a single jump instruction that jumps to the full interrupt service routine (ISR) for that interrupt. The Intel 8080, Atmel AVR and all 8051 and Microchip microcontrollers use the predefined approach.

2. "Fetch"

The "fetch" method loads the PC indirectly, using the address of some entry inside the interrupt vector table to pull an address out of that table, and then loading the PC with that address. Each and every entry of the IVT is the address of an interrupt service routine. All Motorola/Freescale microcontrollers use the fetch method.

3. "Interrupt acknowledge"

For the "interrupt acknowledge" method, the external device gives the CPU an interrupt handler number. The interrupt acknowledge method is used by the Intel Pentium and many older microprocessors.

When the CPU is affected by an interrupt, it looks up the handler in the interrupt vector table, and transfers control to it.

6.0 Interrupt Descriptor Table IDT:

The Interrupt Descriptor Table (IDT) is a data structure used by the x86 architecture to implement an interrupt vector table. The IDT is used by the processor to determine the correct response to interrupts and exceptions.

The details in the description below apply specifically to the x86 architecture and the AMD64 architecture. Other architectures have similar data structures, but may behave differently.

Use of the IDT is triggered by three types of events: hardware interrupts, software interrupts, and processor exceptions, which together are referred to as interrupts. The IDT consists of 256 interrupt vectors—the first 32 (0–31 or 0x00–0x1F) of which are reserved for processor exceptions.

6.1 Real Mode

In the 8086 processor, the interrupt table is called IVT (interrupt vector table). The IVT always resides at the same location in memory, ranging from 0x0000 to 0x03ff, and consists of 256 four-byte real mode far pointers ($256 \times 4 = 1024$ bytes of memory).

A real mode pointer is defined as a 16-bit segment and a 16-bit offset into that segment. The segment is expanded internally by the processor to 20 bits by shifting it 4 bits to the left, thus limiting real mode interrupt handlers to the first 1 megabyte of memory. The first 32 vectors are reserved for the processor's internal exceptions, and hardware interrupts may be mapped to any of the vectors by way of a programmable interrupt controller.

On the 80286 and later, the size and locations of the IVT can be changed in the same way as it is done with the IDT in protected mode (i.e., via the LIDT instruction) though it does not change the format of it. The 80286 also introduced the high memory area, which raises the address limit in real mode by 65,520 bytes.

A commonly used x86 real mode interrupt is INT 10, the Video BIOS code to handle primitive screen drawing functions such as pixel drawing and changing the screen resolution.

6.2 Protected Mode

In protected mode, the IDT is an array of 8-byte descriptors stored consecutively in memory and indexed by an interrupt vector. These descriptors may be either interrupt gates, trap gates or task gates. Interrupt and trap gates point to a memory location containing code to execute by specifying both a segment (present in either the GDT or LDT) and an offset within that segment. The only difference between these two is that an interrupt gate will disable further processor handling of hardware interrupts, making it especially suitable to service hardware interrupts, while a trap gate will leave hardware interrupts enabled and is thus mainly used for handling software interrupts and exceptions. Finally, a task gate will cause the currently active task-state segment to be switched, using the hardware task switch mechanism to effectively hand over use of the processor to another program, thread or process.

The protected mode IDT may reside anywhere in physical memory. The processor has a special register (IDTR) to store both the physical base address and the length in bytes of the IDT. When an interrupt occurs, the processor multiplies the interrupt vector by 8 and adds the result to the IDT base address. With help of the IDT length, the resulting memory address is then verified to

be within the table; if it is too large, an exception is generated. If everything is okay, the 8-byte descriptor stored at the calculated memory location is loaded and actions are taken according to the descriptor's type and contents.

A fully populated IDT is 2 KB (256 entries of 8 bytes each) in length. It is not necessary to use all of the possible entries: it is sufficient to populate the IDT up to the highest interrupt vector used, and set the IDT length portion of the IDTR accordingly.

Vectors 0–31 are reserved by Intel for processor generated exceptions (general protection fault, page fault, etc.). Though currently only vectors 0–20 are used by the processor, future processors may create incompatibilities for broken software which use these vectors for other purposes.

7.0 Interrupt Service Routine

Stands for "Interrupt Service Routine." An ISR (also called an interrupt handler) is a software process invoked by an interrupt request from a hardware device. It handles the request and sends it to the CPU, interrupting the active process. When the ISR is complete, the process is resumed.

A basic example of an ISR is a routine that handles keyboard events, such as pressing or releasing a key. Each time a key is pressed, the ISR processes the input. For example, if you press and hold the right arrow key in a text file, the ISR will signal to the CPU that the right arrow key is depressed. The CPU sends this information to the active word processor or text editing program, which will move the cursor to the right. When you let go of the key, the ISR handles the "key up" event. This interrupts the previous "key down" state, which signals to the program to stop moving the cursor.

Similar to Newton's law of inertia (an object in motion tends to stay in motion), computer processes continue to run unless interrupted. Without an interrupt request, a computer will remain in its current state. Each input signal causes an interrupt, forcing the CPU to process the corresponding event.

Many types of hardware devices, including internal components and external peripherals can send interrupts to the CPU. Examples include keyboards, mice, sound cards, and hard drives. A device driver enables communication between each of these devices and the CPU. ISRs prioritize interrupt requests based on the IRQ setting of the device (or port). Typically the keyboard is at the top of the IRQ list, while devices like hard drives are further down.

8.0 Interrupt Latency

It is the interval of time between from an external interrupt signal being raised to the first

fetch of an instruction of the ISR of the raised interrupt signal. System architects must balance between two things, first is to handle multiple interrupts simultaneously, second is to minimize the interrupt latency.

Minimization of the interrupt latency is achieved by software handlers by two main methods, the first one is to allow nested interrupt handling so the system can respond to new interrupts during handling an older interrupt. This is achieved by enabling interrupts immediately after the interrupt source has been serviced but before finishing the interrupt handling. The second one is the possibility to give priorities to different interrupt sources; this is achieved by programming the interrupt controller to ignore interrupts of the same or lower priority than the interrupt being handled if there is one.

9.0 Nested Interrupt

In this handling scheme handling more than one interrupt at a time is possible. This is achieved by re-enabling interrupts before the handler has fully served the current interrupt. This feature increases the complexity of the system but improves the latency. The scheme should be designed carefully to protect the context saving and restoration from being interrupted. The designer should balance between efficiency and safety by using defensive coding style that assumes problems will occur.

The goal of nested handling is to respond to interrupts quickly and to execute periodic tasks without any delays. Re-enabling interrupts requires switching out of the IRQ mode to user mode to protect link register from being corrupted. Also performing context switch requires emptying the IRQ stack because the handler will not perform switching if there is data on the IRQ stack, so all registers saved on the IRQ stack have to be transferred to task stack. The part of the task stack used in this process is called **stack frame**.

The main disadvantage of this interrupt handling scheme is that it doesn't differ between interrupts by priorities, so lower priority interrupt can block higher priority interrupts.

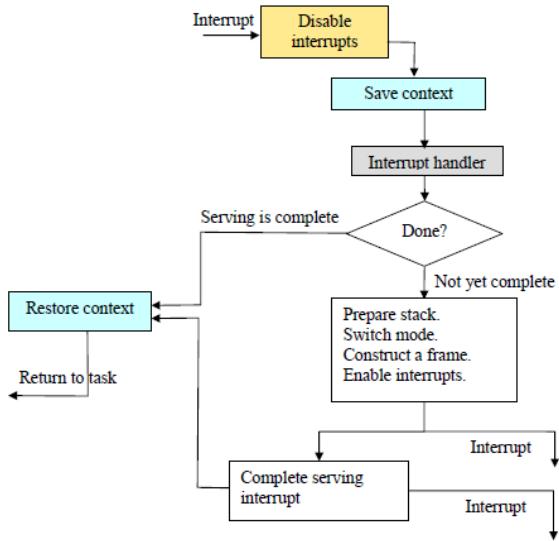


Fig 2 Nested Interrupt

10.0 Exception and Interrupt Handling in Ia-32 Architecture

The interrupt and exception-handling mechanism when operating in protected mode on an Intel 64 or IA-32 processor. Most of the information provided here also applies to interrupt and exception mechanisms used in real-address, virtual-8086 mode, and 64-bit mode.

10.1 INTERRUPT AND EXCEPTION OVERVIEW

Interrupts and exceptions are events that indicate that a condition exists somewhere in the system, the processor, or within the currently executing program or task that requires the attention of a processor. They typically result in a forced transfer of execution from the currently running program or task to a special software routine or task called an interrupt handler or an exception handler. The action taken by a processor in response to an interrupt or exception is referred to as servicing or handling the interrupt or exception.

Interrupts occur at random times during the execution of a program, in response to signals from hardware. System hardware uses interrupts to handle events external to the processor, such as requests to service peripheral devices.

Software can also generate interrupts by executing the INT n instruction.

Exceptions occur when the processor detects an error condition while executing an instruction, such as division by zero. The processor detects a variety of error conditions including protection violations, page faults, and internal machine faults. The machine-check architecture of the Pentium 4, Intel Xeon, P6 family, and Pentium processors also permits a machine-check exception to be generated when internal hardware errors and bus errors are detected.

When an interrupt is received or an exception is detected, the currently running procedure or task is suspended. While the processor executes an interrupt or exception handler. When execution of the handler is complete, the processor resumes execution of the interrupted procedure or task. The resumption of the interrupted procedure or task happens without loss of program continuity, unless recovery from an exception was not possible or an interrupt caused the currently running program to be terminated.

10.2 Sources of Interrupts

The processor receives interrupts from two sources:

- External (hardware generated) interrupts.
- Software-generated interrupts.

10.2.1 External Interrupts

External interrupts are received through pins on the processor or through the local APIC. The primary interrupt pins on Pentium 4, Intel Xeon, P6 family, and Pentium processors are the LINT[1:0] pins, which are connected to the local APIC (see Chapter 10, “Advanced Programmable Interrupt Controller (APIC)”). When the local APIC is enabled, the LINT[1:0] pins can be programmed through the APIC’s local vector table (LVT) to be associated with any of the processor’s exception or interrupt vectors. When the local APIC is global/hardware disabled, these pins are configured as INTR and NMI pins, respectively.

Asserting the INTR pin signals the processor that an external interrupt has occurred. The processor reads from the system bus the interrupt vector number provided by an external interrupt controller, such as an 8259A (see Section 6.2, “Exception and Interrupt Vectors”). Asserting the NMI pin signals a non-maskable interrupt (NMI), which is assigned to interrupt vector 2.

Table 1 Exceptions and Interrupts

Vector	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.

Vector	Mnemonic	Description	Source
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction. ¹
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory. ²
18	#MC	Machine Check	Error codes (if any) and source are model dependent. ³
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction ⁴
20	#VE	Virtualization Exception	EPT violations ⁵
21-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT <i>n</i> instruction.

NOTES:

1. The UD2 instruction was introduced in the Pentium Pro processor.
2. Processors after the Intel386 processor do not generate this exception.
3. This exception was introduced in the Intel486 processor.
4. This exception was introduced in the Pentium processor and enhanced in the P6 family processors.
5. This exception was introduced in the Pentium III processor.
6. This exception can occur only on processors that support the 1-setting of the “EPT-violation #VE” VM-execution control.

The processor's local APIC is normally connected to a system-based I/O APIC. Here, external interrupts received at the I/O APIC's pins can be directed to the local API C through the system

bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel® Atom™, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors). The I/O APIC determines the vector number of the interrupt and sends this number to the local APIC. When a system contains multiple processors, processors can also send interrupts to one another by means of the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors).

The LINT[1:0] pins are not available on the Intel486 processor and earlier Pentium processors that do not contain an on-chip local APIC. These processors have dedicated NMI and INTR pins. With these processors, external interrupts are typically generated by a system-based interrupt controller (8259A), with the interrupts being signaled through the INTR pin. Note that several other pins on the processor can cause a processor interrupt to occur. However, these interrupts are not handled by the interrupt and exception mechanism described in this chapter. These pins include the RESET#, FLUSH#, STPCLK#, SMI#, R/S#, and INIT# pins. Whether they are included on a particular processor is implementation dependent. Pin functions are described in the data books for the individual processors. The SMI# pin is described in Chapter 34, “System Management Mode.”

10.2.1.1 Maskable Hardware Interrupts

Any external interrupt that is delivered to the processor by means of the INTR pin or through the local APIC is called a maskable hardware interrupt. Maskable hardware interrupts that can be delivered through the INTR pin include all IA-32 architecture defined interrupt vectors from 0 through 255; those that can be delivered through the local APIC include interrupt vectors 16 through 255. The IF flag in the EFLAGS register permits all maskable hardware interrupts to be masked as a group (see Section 6.8.1, “Masking Maskable Hardware Interrupts”). Note that when interrupts 0 through 15 are delivered through the local APIC, the APIC indicates the receipt of an illegal vector.

10.2.2 Software-Generated Interrupts

The INT n instruction permits interrupts to be generated from within software by supplying an interrupt vector number as an operand. For example, the INT 35 instruction forces an implicit call to the interrupt handler for interrupt 35.

Any of the interrupt vectors from 0 to 255 can be used as a parameter in this instruction. If the processor’s predefined NMI vector is used, however, the response of the processor will not be the same as it would be from an NMI interrupt generated in the normal manner. If vector number 2 (the NMI vector) is used in this instruction, the NMI interrupt handler is called, but the processor’s NMI-handling hardware is not activated. Interrupts generated in software with the INT n instruction cannot be masked by the IF flag in the EFLAGS register.

10.3 Sources of Exceptions

The processor receives exceptions from three sources:

- Processor-detected program-error exceptions.
- Software-generated exceptions.
- Machine-check exceptions.

10.3.1 Program-Error Exceptions

The processor generates one or more exceptions when it detects program errors during the execution in an application program or the operating system or executive. Intel 64 and IA-32 architectures define a vector number for each processor-detectable exception. Exceptions are classified as **faults**, **traps**, and **aborts** (see Section 6.5, “Exception Classifications”).

10.3.2 Software-Generated Exceptions

The INTO, INT 3, and BOUND instructions permit exceptions to be generated in software. These instructions allow checks for exception conditions to be performed at points in the instruction stream. For example, INT 3 causes a breakpoint exception to be generated. The INT n instruction can be used to emulate exceptions in software; but there is a limitation. If INT n provides a vector for one of the architecturally-defined exceptions, the processor generates an interrupt to the correct vector (to access the exception handler) but does not push an error code on the stack. This is true even if the associated hardware-generated exception normally produces an error code. The exception handler will still attempt to pop an error code from the stack while handling the exception. Because no error code was pushed, the handler will pop off and discard the EIP instead (in place of the missing error code). This sends the return to the wrong location.

10.3.3 Machine-Check Exceptions

The P6 family and Pentium processors provide both internal and external machine-check mechanisms for checking the operation of the internal chip hardware and bus transactions. These mechanisms are implementation dependent. When a machine-check error is detected, the processor signals a machine-check exception (vector 18) and returns an error code. See Chapter 6, “Interrupt 18—Machine-Check Exception (#MC)” and Chapter 15, “Machine-Check Architecture,” for more information about the machine-check mechanism.

10.4 Exception Classifications

Exceptions are classified as **faults**, **traps**, or **aborts** depending on the way they are reported and whether the instruction that caused the exception can be restarted without loss of program or task continuity.

- **Faults** — A fault is an exception that can generally be corrected and that, once corrected, allows the program to be restarted with no loss of continuity. When a fault is reported, the processor restores the machine state to the state prior to the beginning of execution of the faulting instruction. The return address (saved contents of the CS and EI P registers) for the fault handler points to the faulting instruction, rather than to the instruction following the faulting instruction.
- **Traps** — A trap is an exception that is reported immediately following the execution of the trapping instruction. Traps allow execution of a program or task to be continued without loss of program continuity. The return address for the trap handler points to the instruction to be executed after the trapping instruction.
- **Aborts** — An abort is an exception that does not always report the precise location of the instruction causing the exception and does not allow a restart of the program or task that caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

11.0 Nonmaskable Interrupt (NMI)

The nonmaskable interrupt (NMI) can be generated in either of two ways:

- External hardware asserts the NMI pin.
- The processor receives a message on the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors) with a delivery mode NMI.

When the processor receives a NMI from either of these sources, the processor handles it immediately by calling the NMI handler pointed to by interrupt vector number 2. The processor also invokes certain hardware conditions to insure that no other interrupts, including NMI interrupts, are received until the NMI handler has completed executing (see Section 6.7.1, “Handling Multiple NMIs”). Also, when an NMI is received from either of the above sources, it cannot be masked by the IF flag in the EFLAGS register. It is possible to issue a maskable hardware interrupt (through the INTR pin) to vector 2 to invoke the NMI interrupt handler; however, this interrupt will not truly be an NMI interrupt. A true NMI interrupt that activates the processor’s NMI-handling hardware can only be delivered through one of the mechanisms listed above.

11.0.1 Handling Multiple NMIs

While an NMI interrupt handler is executing, the processor blocks delivery of subsequent NMIs until the next execution of the IRET instruction. This blocking of NMIs prevents nested execution of the NMI handler. It is recommended that the NMI interrupt handler be accessed through an interrupt gate to disable maskable hardware interrupts (see Section 6.8.1, “Masking Maskable Hardware Interrupts”). An execution of the IRET instruction unblocks NMIs even if the instruction causes a fault. For example, if the I RET instruction executes with EFLAGS.VM = 1 and IOPL of less than 3, a general-protection exception is generated (see Section 20.2.7, “Sensitive Instructions”). In such a case, NMIs are unmasked before the exception handler is

Invoked.

11.1 SCI

According to the ACPI spec, The **FADT** (Fixed ACPI Description Table) table contains a field that reports the **SCI interrupt** number to OS. The field is defined as below:

SCI_INT	2	46	System vector the SCI interrupt is wired to in 8259 mode. On systems that do not contain the 8259, this field contains the Global System interrupt number of the SCI interrupt. OSPM is required to treat the ACPI SCI interrupt as a sharable, level, active low interrupt.
---------	---	----	--

I dumped the FADT table on an Intel x86 platform and see the SCI interrupt is associated with the number 9:

PM Profile	0x04 (4)
SCI Interrupt	0x0009 (9)
SMI Port	0x000000B2 (178)
ACPI Enable	0xA0 (160)

But according to the Intel Manual, 0-31 are reserved vectors for IA architecturally defined interrupts. Specifically, the 9 is defined as:

9	Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction. ²
---	---	-------	----	--

2. Processors after the Intel386 processor do not generate this exception.

So, according to the note, the 9 is not generated after I386 processor. So I guess that's why 9 can be ***salvaged*** for SCI. This can be seen as a **x86-specific implementation** of the ACPI spec.

11.2 System Management Interrupt (SMI)

The only way to enter SMM is by signaling an SMI through the SMI# pin on the processor or through an SMI message received through the APIC bus. The SMI is a nonmaskable external interrupt that operates independently from the processor's interrupt- and exception-handling mechanism and the local APIC. The SMI takes precedence over an NMI and a maskable interrupt. SMM is non-reentrant; that is, the SMI is disabled while the processor is in SMM.

In the Pentium 4, Intel Xeon, and P6 family processors, when a processor that is designated as an application processor during an MP initialization sequence is waiting for a startup IPI (SIPI), it is in a mode where SMIs are masked. However if a SMI is received while an application processor is in the wait for SIPI mode, the SMI will be pended. The processor then responds on receipt of a SIPI by immediately servicing the pended SMI and going into SMM before handling the SIPI. An SMI may be blocked for one instruction following execution of STI, MOV to SS, or POP into SS.

12.0 APIC

The Advanced Programmable Interrupt Controller (APIC), referred to in the following sections as the local APIC, was introduced into the IA-32 processors with the Pentium processor and is included in the P6 family, Pentium 4, Intel Xeon processors, and other more recent Intel 64 and IA-32 processor families. The local APIC performs two primary functions for the processor:

- It receives interrupts from the processor's interrupt pins, from internal sources and from an external I/O APIC (or other external interrupt controller). It sends these to the processor core for handling.
- In multiple processor (MP) systems, it sends and receives interprocessor interrupt (IPI) messages to and from other logical processors on the system bus. IPI messages can be used to distribute interrupts among the processors in the system or to execute system wide functions (such as, booting up processors or distributing work among a group of processors). The external I/O APIC is part of Intel's system chip set. Its primary function is to receive external interrupt events from the system and its associated I/O devices and relay them to the local APIC as interrupt messages. In MP systems, the I/O APIC also provides a mechanism for distributing external interrupts to the local APICs of selected processors or groups of processors on the system bus.

12.1 Local and I/O APIC Overview

Each local APIC consists of a set of APIC registers (see Table 10-1) and associated hardware that control the delivery of interrupts to the processor core and the generation of IPI messages. The APIC registers are memory mapped and can be read and written to using the MOV instruction. Local APICs can receive interrupts from the following sources:

- **Locally connected I/O devices** — These interrupts originate as an edge or level asserted by an I/O device that is connected directly to the processor's local interrupt pins (LINT0 and LINT1). The I/O devices may also be connected to an 8259-type interrupt controller that is in turn connected to the processor through one of the local interrupt pins.
- **Externally connected I/O devices** — These interrupts originate as an edge or level asserted by an I/O device that is connected to the interrupt input pins of an I/O APIC. Interrupts are sent as I/O interrupt messages from the I/O APIC to one or more of the processors in the system.
- **Inter-processor interrupts (IPIs)** — An Intel 64 or IA-32 processor can use the IPI mechanism to interrupt another processor or group of processors on the system bus. IPIs are used for software self-interrupts, interrupt forwarding, or preemptive scheduling.
- **APIC timer generated interrupts** — The local APIC timer can be programmed to send a local interrupt to its associated processor when a programmed count is reached (see Section 10.5.4, “APIC Timer”).
- **Performance monitoring counter interrupts** — P6 family, Pentium 4, and Intel Xeon processors provide the ability to send an interrupt to its associated processor when a performance-monitoring counter overflows (see Section 18.6.3.5.8, “Generating an Interrupt on Overflow”).
- **Thermal Sensor interrupts** — Pentium 4 and Intel Xeon processors provide the ability to send an interrupt to themselves when the internal thermal sensor has been tripped (see Section 14.7.2, “Thermal Monitor”).

APIC internal error interrupts — When an error condition is recognized within the local APIC (such as an attempt to access an unimplemented register), the APIC can be programmed to send an interrupt to its associated processor (see Section 10.5.3, “Error Handling”). Of these interrupt sources: the processor's LINT0 and LINT1 pins, the APIC timer, the performance-monitoring counters, the thermal sensor, and the internal APIC error detector are referred to as **local interrupt sources**. Upon receiving a signal from a local interrupt source, the local APIC delivers

the interrupt to the processor core using an interrupt delivery protocol that has been set up through a group of APIC registers called the **local vector table** or **LVT** (see Section 10.5.1, “Local Vector Table”).

A separate entry is provided in the local vector table for each local interrupt source, which allows a specific interrupt delivery protocol to be set up for each source. For example, if the LINT1 pin is going to be used as an NMI pin, the LINT1 entry in the local vector table can be set up to deliver an interrupt with vector number 2 (NMI interrupt) to the processor core. The local APIC handles interrupts from the other two interrupt sources (externally connected I/O devices and IPIs) through its IPI message handling facilities.

A processor can generate IPIs by programming the interrupt command register (ICR) in its local APIC (see Section 10.6.1, “Interrupt Command Register (ICR)”). The act of writing to the ICR causes an IPI message to be generated and issued on the system bus (for Pentium 4 and Intel Xeon processors) or on the APIC bus (for Pentium and P6 family processors). See Section 10.2, “System Bus Vs. APIC Bus.” IPIs can be sent to other processors in the system or to the originating processor (self-interrupts). When the target processor receives an IPI message, its local APIC handles the message automatically (using information included in the message such as vector number and trigger mode). See Section 10.6, “Issuing Interprocessor Interrupts,” for a detailed explanation of the local APIC’s IPI message delivery and acceptance mechanism.

The local APIC can also receive interrupts from externally connected devices through the I/O APIC (see Figure 10-1). The I/O APIC is responsible for receiving interrupts generated by system hardware and I/O devices and forwarding them to the local APIC as interrupt messages.

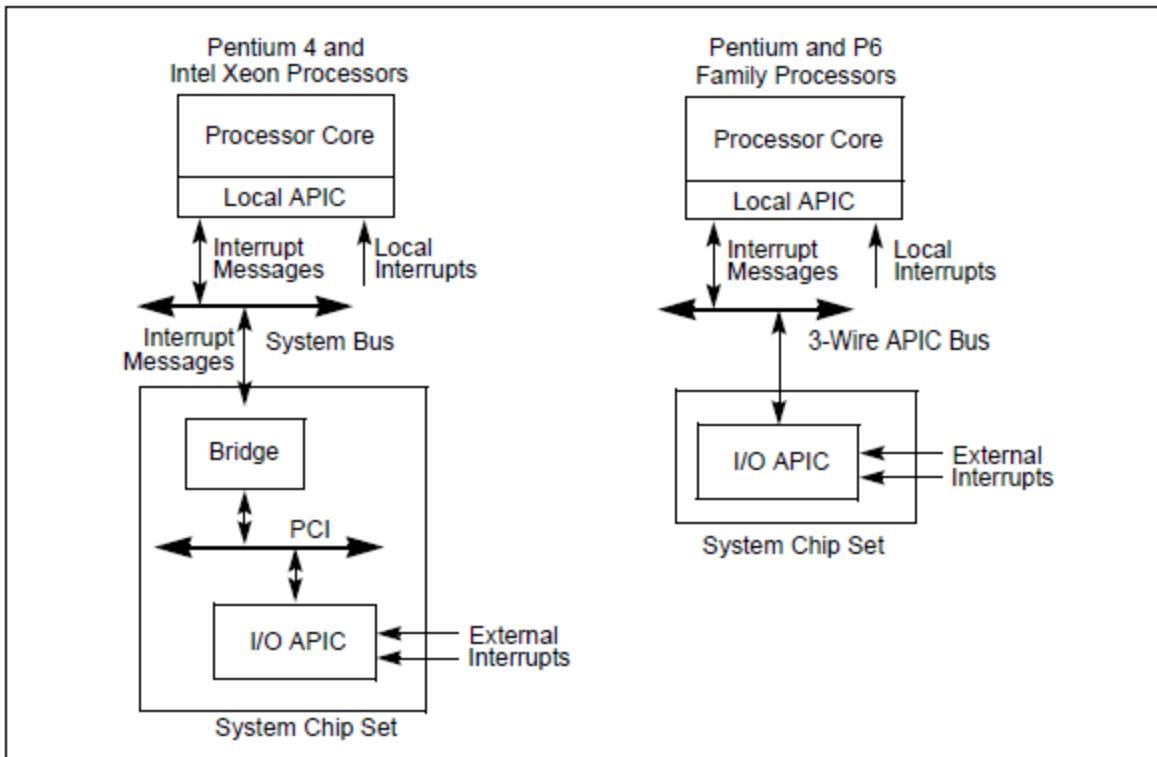


Fig 3 Relationship of Local APIC and I/O APIC in single Processor System

Individual pins on the I/O APIC can be programmed to generate a specific interrupt vector when asserted. The I/O APIC also has a “virtual wire mode” that allows it to communicate with a standard 8259A-style external interrupt controller. Note that the local APIC can be disabled, (“Enabling or Disabling the Local APIC”). This allows an associated processor core to receive interrupts directly from an 8259A interrupt controller. Both the local APIC and the I/O APIC are designed to operate in MP systems (see Figures 3 and 4). Each local APIC handles interrupts from the I/O APIC, IPIs from processors on the system bus, and self-generated interrupts. Interrupts can also be delivered to the individual processors through the local interrupt pins; however, this mechanism is commonly not used in MP systems.

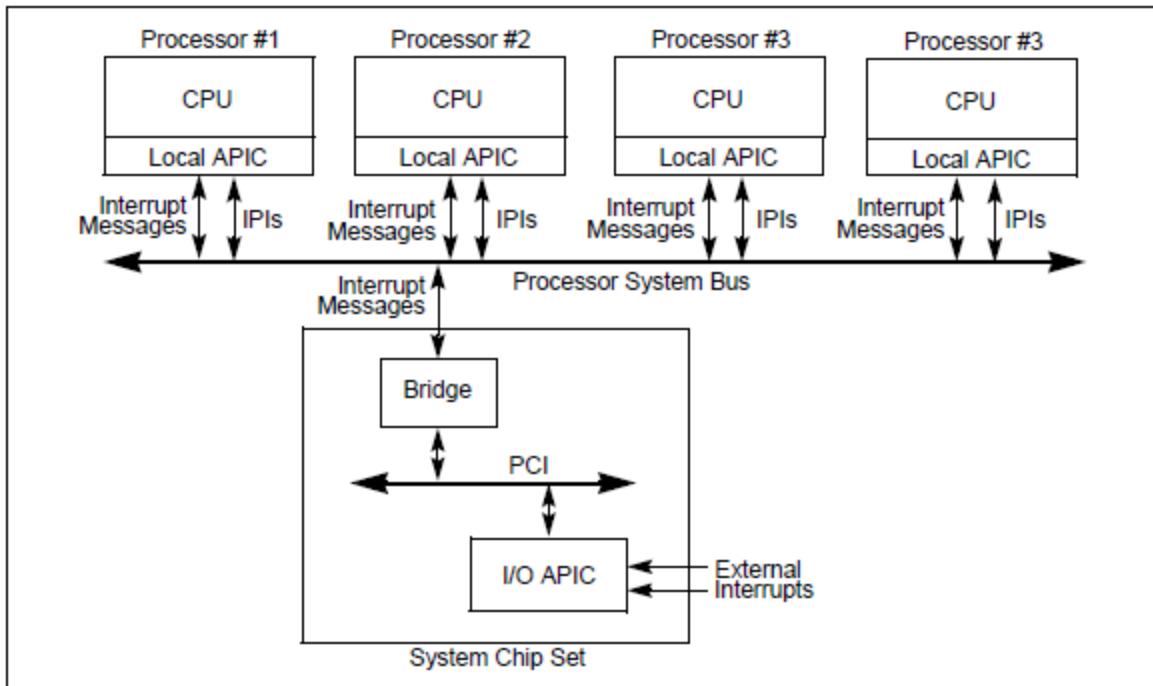


Figure 4. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems

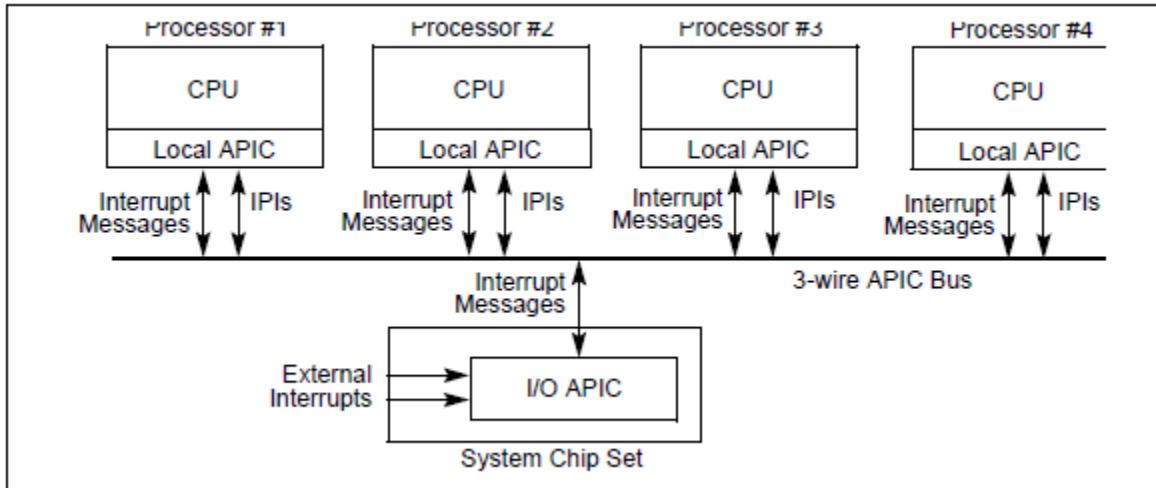


Figure 5. Local APICs and I/O APIC When P6 Family Processors Are Used in Multiple-Processor Systems

The IPI mechanism is typically used in MP systems to send fixed interrupts (interrupts for a specific vector number) and special-purpose interrupts to processors on the system bus. For example, a local APIC can use an IPI to forward a fixed interrupt to another processor for servicing. Special-purpose IPIs (including NMI, INIT, SMI and SIPI IPIs) allow one or more processors on the system bus to perform system-wide boot-up and control functions.

The following sections focus on the local APIC and its implementation in the Pentium 4, Intel Xeon, and P6 family processors. In these sections, the terms “local APIC” and “I/O APIC” refer to local and I/O APICs used with the P6 family processors and to local and I/O xAPICs used with the Pentium 4 and Intel Xeon processors (see Section 10.3, “The Intel® 82489DX External APIC, the APIC, the xAPIC, and the X2APIC”).

12.2 Local APIC

The following sections describe the architecture of the local APIC and how to detect it, identify it, and determine its status.

12.2.1 The Local APIC Block Diagram

Figure 10-4 gives a functional block diagram for the local APIC. Software interacts with the local APIC by reading and writing its registers. APIC registers are memory-mapped to a 4-KByte region of the processor’s physical address space with an initial starting address of FEE00000H. For correct APIC operation, this address space must be mapped to an area of memory that has been designated as strong uncacheable (UC). See Section 11.3, “Methods of Caching Available.”

In MP system configurations, the APIC registers for Intel 64 or IA-32 processors on the system bus are initially mapped to the same 4-KByte region of the physical address space. Software has the option of changing initial mapping to a different 4-KByte region for all the local APICs or of mapping the APIC registers for each local APIC to its own 4-KByte region. Section 10.4.5, “Relocating the Local APIC Registers,” describes how to relocate the base address for APIC registers.

On processors supporting x2APIC architecture (indicated by CPUID.01H:ECX[21] = 1), the local APIC supports operation both in xAPIC mode and (if enabled by software) in x2APIC mode. x2APIC mode provides extended processor addressability (see Section 10.12).

For P6 family, Pentium 4, and Intel Xeon processors, the APIC handles all memory accesses to addresses within the 4-KByte APIC register space internally and no external bus cycles are produced. For the Pentium processors with an on-chip APIC, bus cycles are produced for accesses

to the APIC register space. Thus, for software intended to run on Pentium processors, system software should explicitly not map the APIC register space to regular system memory. Doing so can result in an invalid opcode exception (#UD) being generated or unpredictable execution.

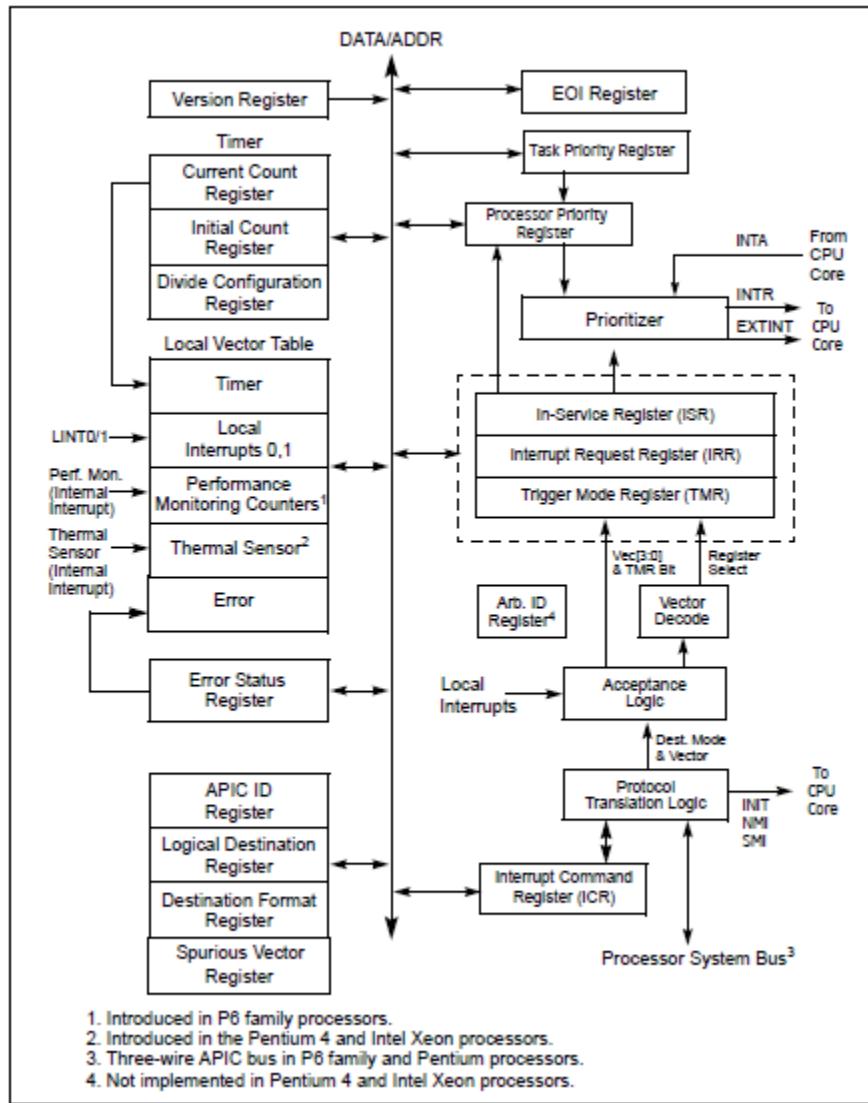


Fig 6 Local APIC Structure

Table 4 shows how the APIC registers are mapped into the 4-KByte APIC register space. Registers are 32 bits, 64 bits, or 256 bits in width; all are aligned on 128-bit boundaries. All 32-bit registers should be accessed using 128-bit aligned 32-bit loads or stores. Some processors may support loads and stores of less than 32 bits to some of the APIC registers. This is model specific behavior and is not guaranteed to work on all processors. Any FP/MMX/SSE access to an APIC register, or

any access that touches bytes 4 through 15 of an APIC register may cause undefined behavior and must not be executed. This undefined behavior could include hangs, incorrect results or unexpected exceptions, including machine checks, and may vary between implementations. Wider registers (64-bit or 256-bit) must be accessed using multiple 32-bit loads or stores, with all accesses being 128-bit aligned.

The local APIC registers listed in Table 10-1 are not MSRs. The only MSR associated with the programming of the local APIC is the IA32_APIC_BASE MSR (see Section 10.4.3, “Enabling or Disabling the Local APIC”). In processors based on Intel microarchitecture code name Nehalem1 the Local APIC ID Register is no longer Read/Write; it is Read Only. See Table 2-1, “CPUID Signature Values of DisplayFamily_DisplayModel,” on page 1, and Section 2.7, “MSRs In the Intel® Microarchitecture Code Name Nehalem” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4* to determine which processors are based on Nehalem microarchitecture.

Table 4 Local APIC Register Address Map

Address	Register Name	Software Read/Write
FEE0 0000H	Reserved	
FEE0 0010H	Reserved	
FEE0 0020H	Local APIC ID Register	Read/Write.
FEE0 0030H	Local APIC Version Register	Read Only.
FEE0 0040H	Reserved	
FEE0 0050H	Reserved	
FEE0 0060H	Reserved	
FEE0 0070H	Reserved	
FEE0 0080H	Task Priority Register (TPR)	Read/Write.
FEE0 0090H	Arbitration Priority Register ¹ (APR)	Read Only.
FEE0 00A0H	Processor Priority Register (PPR)	Read Only.
FEE0 00B0H	EOI Register	Write Only.
FEE0 00C0H	Remote Read Register ¹ (RRD)	Read Only
FEE0 00D0H	Logical Destination Register	Read/Write.
FEE0 00E0H	Destination Format Register	Read/Write (see Section 10.6.2.2).
FEE0 00F0H	Spurious Interrupt Vector Register	Read/Write (see Section 10.9).
FEE0 0100H	In-Service Register (ISR); bits 31:0	Read Only.
FEE0 0110H	In-Service Register (ISR); bits 63:32	Read Only.
FEE0 0120H	In-Service Register (ISR); bits 95:64	Read Only.
FEE0 0130H	In-Service Register (ISR); bits 127:96	Read Only.
FEE0 0140H	In-Service Register (ISR); bits 159:128	Read Only.
FEE0 0150H	In-Service Register (ISR); bits 191:160	Read Only.
FEE0 0160H	In-Service Register (ISR); bits 223:192	Read Only.
FEE0 0170H	In-Service Register (ISR); bits 255:224	Read Only.
FEE0 0180H	Trigger Mode Register (TMR); bits 31:0	Read Only.
FEE0 0190H	Trigger Mode Register (TMR); bits 63:32	Read Only.
FEE0 01A0H	Trigger Mode Register (TMR); bits 95:64	Read Only.

Address	Register Name	Software Read/Write
FE00 01B0H	Trigger Mode Register (TMR); bits 127:96	Read Only.
FE00 01C0H	Trigger Mode Register (TMR); bits 159:128	Read Only.
FE00 01D0H	Trigger Mode Register (TMR); bits 191:160	Read Only.
FE00 01E0H	Trigger Mode Register (TMR); bits 223:192	Read Only.
FE00 01F0H	Trigger Mode Register (TMR); bits 255:224	Read Only.
FE00 0200H	Interrupt Request Register (IRR); bits 31:0	Read Only.
FE00 0210H	Interrupt Request Register (IRR); bits 63:32	Read Only.
FE00 0220H	Interrupt Request Register (IRR); bits 95:64	Read Only.
FE00 0230H	Interrupt Request Register (IRR); bits 127:96	Read Only.
FE00 0240H	Interrupt Request Register (IRR); bits 159:128	Read Only.
FE00 0250H	Interrupt Request Register (IRR); bits 191:160	Read Only.
FE00 0260H	Interrupt Request Register (IRR); bits 223:192	Read Only.
FE00 0270H	Interrupt Request Register (IRR); bits 255:224	Read Only.
FE00 0280H	Error Status Register	Read Only.
FE00 0290H through FE00 02E0H	Reserved	
FE00 02F0H	LVT Corrected Machine Check Interrupt (CMCI) Register	Read/Write.
FE00 0300H	Interrupt Command Register (ICR); bits 0-31	Read/Write.
FE00 0310H	Interrupt Command Register (ICR); bits 32-63	Read/Write.
FE00 0320H	LVT Timer Register	Read/Write.
FE00 0330H	LVT Thermal Sensor Register ²	Read/Write.
FE00 0340H	LVT Performance Monitoring Counters Register ³	Read/Write.
FE00 0350H	LVT LINT0 Register	Read/Write.
FE00 0360H	LVT LINT1 Register	Read/Write.
FE00 0370H	LVT Error Register	Read/Write.
FE00 0380H	Initial Count Register (for Timer)	Read/Write.
FE00 0390H	Current Count Register (for Timer)	Read Only.
FE00 03A0H through FE00 03D0H	Reserved	
FE00 03E0H	Divide Configuration Register (for Timer)	Read/Write.
FE00 03F0H	Reserved	

12.2.2 Presence of the Local APIC

Beginning with the P6 family processors, the presence or absence of an on-chip local APIC can be detected using the CPUID instruction. When the CPUID instruction is executed with a source operand of 1 in the EAX register, bit 9 of the CPUID feature flags returned in the EDX register indicates the presence (set) or absence (clear) of a local APIC.

12.2.3 Enabling or Disabling the Local APIC

The local APIC can be enabled or disabled in either of two ways:

1. Using the APIC global enable/disable flag in the IA32_APIC_BASE MSR
 - When IA32_APIC_BASE[11] is 0, the processor is functionally equivalent to an IA-32 processor without an on-chip APIC. The CPUID feature flag for the APIC (see Section 10.4.2, “Presence of the Local APIC”) is also set to 0.
 - When IA32_APIC_BASE[11] is set to 0, processor APICs based on the 3-wire APIC bus cannot be generally re-enabled until a system hardware reset. The 3-wire bus loses track of arbitration that would be necessary for complete re-enabling. Certain APIC functionality can be enabled (for example: performance and thermal monitoring interrupt generation).

- For processors that use Front Side Bus (FSB) delivery of interrupts, software may disable or enable the APIC by setting and resetting IA32_APIC_BASE[11]. A hardware reset is not required to re-start APIC functionality, if software guarantees no interrupt will be sent to the APIC as IA32_APIC_BASE[11] is cleared.
- When IA32_APIC_BASE[11] is set to 0, prior initialization to the APIC may be lost and the APIC may return to the state described in Section 10.4.7.1, “Local APIC State After Power-Up or Reset.”

2. Using the APIC software enable/disable flag in the spurious-interrupt vector register (see Figure 10-23):

- If IA32_APIC_BASE[11] is 1, software can temporarily disable a local APIC at any time by clearing the APIC software enable/disable flag in the spurious-interrupt vector register (see Figure 10-23). The state of the local APIC when in this software-disabled state is described in Section 10.4.7.2, “Local APIC State After It Has Been Software Disabled.”
- When the local APIC is in the software-disabled state, it can be re-enabled at any time by setting the APIC software enable/disable flag to 1. For the Pentium processor, the APICEN pin (which is shared with the PICD1 pin) is used during power-up or reset to disable the local APIC. Note that each entry in the LVT has a mask bit that can be used to inhibit interrupts from being delivered to the processor from selected local interrupt sources (the LINT0 and LINT1 pins, the APIC timer, the performance-monitoring counters, the thermal sensor, and/or the internal APIC error detector).

12.2.4 Local APIC Status and Location

The status and location of the local APIC are contained in the IA32_APIC_BASE MSR (see Fig 7). MSR bit functions are described below:

- **BSP flag, bit 8** — Indicates if the processor is the bootstrap processor (BSP). See Section 8.4, “Multiple-Processor (MP) Initialization.” Following a power-up or reset, this flag is set to 1 for the processor selected as the BSP and set to 0 for the remaining processors (APs).
- **APIC Global Enable flag, bit 11** — Enables or disables the local APIC (see Section 10.4.3, “Enabling or Disabling the Local APIC”). This flag is available in the Pentium 4, Intel Xeon, and P6 family processors. It is not guaranteed to be available or available at the same location in future Intel 64 or IA-32 processors.

- **APIC Base field, bits 12 through 35** — Specifies the base address of the APIC registers. This 24-bit value is extended by 12 bits at the low end to form the base address. This automatically aligns the address on a 4-Kbyte boundary. Following a power-up or reset, the field is set to FEE0 0000H.
- Bits 0 through 7, bits 9 and 10, and bits MAXPHYADDR2 through 63 in the IA32_APIC_BASE MSR are reserved.

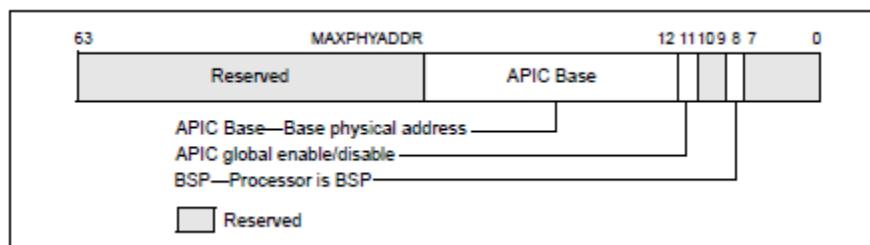


Figure 7. IA32_APIC_BASE MSR (APIC_BASE_MSR in P6 Family)

12.2.5 Relocating the Local APIC Registers

The Pentium 4, Intel Xeon, and P6 family processors permit the starting address of the APIC registers to be relocated from FEE00000H to another physical address by modifying the value in the base address field of the IA32_APIC_BASE MSR. This extension of the APIC architecture is provided to help resolve conflicts with memory maps of existing systems and to allow individual processors in an MP system to map their APIC registers to different locations in physical memory.

12.2.6 Local APIC ID

At power up, system hardware assigns a unique APIC ID to each local APIC on the system bus (for Pentium 4 and Intel Xeon processors) or on the APIC bus (for P6 family and Pentium processors). The hardware assigned APIC ID is based on system topology and includes encoding for socket position and cluster information (see Figure 8-2 and Section 8.9.1, “Hierarchical Mapping of Shared Resources”). In MP systems, the local APIC ID is also used as a processor ID by the BIOS and the operating system. Some processors permit software to modify the APIC ID. However, the ability of software to modify the APIC ID is processor model specific. Because of this, operating system software should avoid writing to the local APIC ID register. The value returned by bits 31-24 of the EBX register (when the CPUID instruction is executed with a source operand value of 1 in the EAX register) is always the Initial APIC ID (determined by the platform initialization).

This is true even if software has changed the value in the Local APIC ID register. The processor receives the hardware assigned APIC ID (or Initial APIC ID) by sampling pins A11# and A12# and pins BR0# through BR3# (for the Pentium 4, Intel Xeon, and P6 family processors) and pins BE0# through BE3# (for the Pentium processor). The APIC ID latched from these pins is stored in the

APIC ID field of the local APIC ID register (see Figure 8), and is used as the Initial APIC ID for the processor.

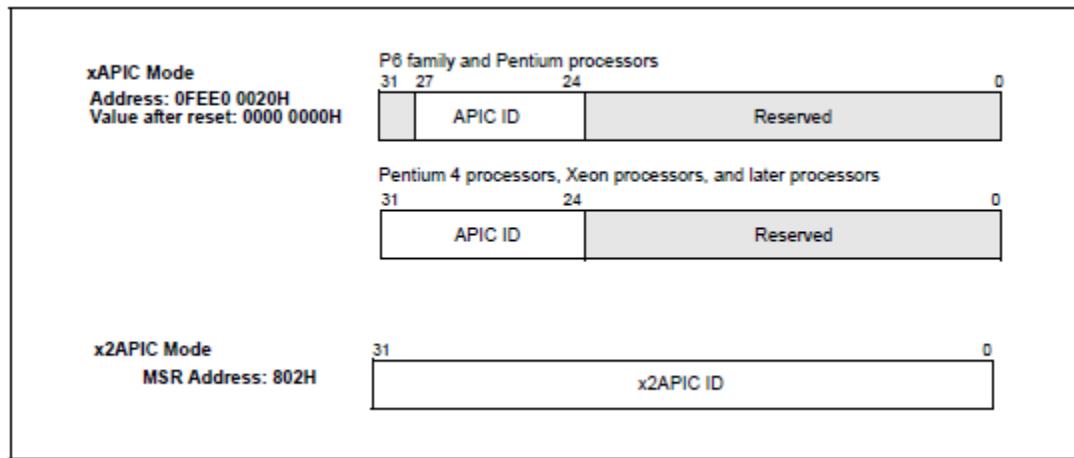


Figure 8. Local APIC ID Register

For the P6 family and Pentium processors, the local APIC ID field in the local APIC ID register is 4 bits. Encodings 0H through EH can be used to uniquely identify 15 different processors connected to the APIC bus. For the Pentium 4 and Intel Xeon processors, the xAPIC specification extends the local APIC ID field to 8 bits. These can be used to identify up to 255 processors in the system.

12.2.7 Local APIC State

The following sections describe the state of the local APIC and its registers following a power-up or reset, after the local APIC has been software disabled, following an INIT reset, and following an INIT-deassert message. x2APIC will introduce 32-bit ID; see Section 10.12.

12.2.7.1 Local APIC State After Power-Up or Reset

Following a power-up or reset of the processor, the state of local APIC and its registers are as follows:

- The following registers are reset to all 0s.
- IRR, ISR, TMR, ICR, LDR, and TPR.
- Timer initial count and timer current count registers.
- Divide configuration register.
- The DFR register is reset to all 1s.
- The LVT register is reset to 0s except for the mask bits; these are set to 1s.
- The local APIC version register is not affected.

- The local APIC ID register is set to a unique APIC ID. (Pentium and P6 family processors only). The Arb ID register is set to the value in the APIC ID register.
- The spurious-interrupt vector register is initialized to 000000FFH. By setting bit 8 to 0, software disables the local APIC.
- If the processor is the only processor in the system or it is the BSP in an MP system (see Section 8.4.1, “BSP and AP Processors”); the local APIC will respond normally to INIT and NMI messages, to INIT# signals and to STPCLK# signals. If the processor is in an MP system and has been designated as an AP; the local APIC will respond the same as for the BSP. In addition, it will respond to SIPI messages. For P6 family processors only, an AP will not respond to a STPCLK# signal.

[12.2.7.2 Local APIC State After It Has Been Software Disabled](#)

When the APIC software enable/disable flag in the spurious interrupt vector register has been explicitly cleared (as opposed to being cleared during a power up or reset), the local APIC is temporarily disabled (see Section 10.4.3, “Enabling or Disabling the Local APIC”). The operation and response of a local APIC while in this software-disabled state is as follows:

- The local APIC will respond normally to INIT, NMI, SMI, and SIPI messages.
- Pending interrupts in the IRR and ISR registers are held and require masking or handling by the CPU.
- The local APIC can still issue IPIs. It is software’s responsibility to avoid issuing IPIs through the IPI mechanism and the ICR register if sending interrupts through this mechanism is not desired.
- The reception of any interrupt or transmission of any IPIs that are in progress when the local APIC is disabled are completed before the local APIC enters the software-disabled state.
- The mask bits for all the LVT entries are set. Attempts to reset these bits will be ignored.
- (For Pentium and P6 family processors) The local APIC continues to listen to all bus messages in order to keep its arbitration ID synchronized with the rest of the system.

[12.2.7.3 Local APIC State After an INIT Reset \(“Wait-for-SIPI” State\)](#)

An INIT reset of the processor can be initiated in either of two ways:

- By asserting the processor’s INIT# pin.
- By sending the processor an INIT IPI (an IPI with the delivery mode set to INIT).

Upon receiving an INIT through either of these mechanisms, the processor responds by beginning the initialization process of the processor core and the local APIC. The state of the local APIC

following an INIT reset is the same as it is after a power-up or hardware reset, except that the APIC ID and arbitration ID registers are not affected. This state is also referred to at the “wait-for-SIPI” state (see also: Section 8.4.2, “MP Initialization Protocol Requirements and Restrictions”).

12.2.7.4 Local APIC State After It Receives an INIT-Deassert IPI

Only the Pentium and P6 family processors support the INIT-deassert IPI. An INIT-deassert IPI has no affect on the state of the APIC, other than to reload the arbitration ID register with the value in the APIC ID register.

12.2.8 82093AA I/O ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (IOAPIC)

1. Provides Multiprocessor Interrupt Management
 - Dynamic Interrupt Distribution Routing Interrupt to the Lowest Priority Processor
 - Software Programmable Control of Interrupt Inputs
 - Off Loads Interrupt Related Traffic From the Memory Bus
2. 24 Programmable Interrupts
 - 13 ISA Interrupts Supported
 - 4 PCI Interrupts
 - 1 Interrupt/SMI# Rerouting
 - 2 Motherboard Interrupts
 - 1 Interrupt Used for INTR Input
 - 3 General Purpose Interrupts
 - Independently Programmable for Edge/Level Sensitivity Interrupts
 - Each Interrupt Can Be Programmed to Respond to Active High or Low Inputs
3. X-Bus Interface
 - CS For Flexible Decode of the IOAPIC Device.
 - Index Register Interface for Optimum Memory Usage
 - Registers are 32-Bit Wide to Match the PCI to Host Bridge Architecture
4. Package 64-Pin PQFP

The 82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC) provides multi-processor interrupt management and incorporates both static and dynamic symmetric interrupt distribution across all processors. In systems with multiple I/O subsystems, each subsystem can have its own set of interrupts. Each interrupt pin is individually programmable as either edge or level triggered. The interrupt vector and interrupt steering information can be specified per interrupt. An indirect register accessing scheme optimizes the memory space needed to access the IOAPIC’s internal registers. To increase system flexibility when assigning memory space usage, the The IOAPIC’s 2-register memory space is re-locatable.

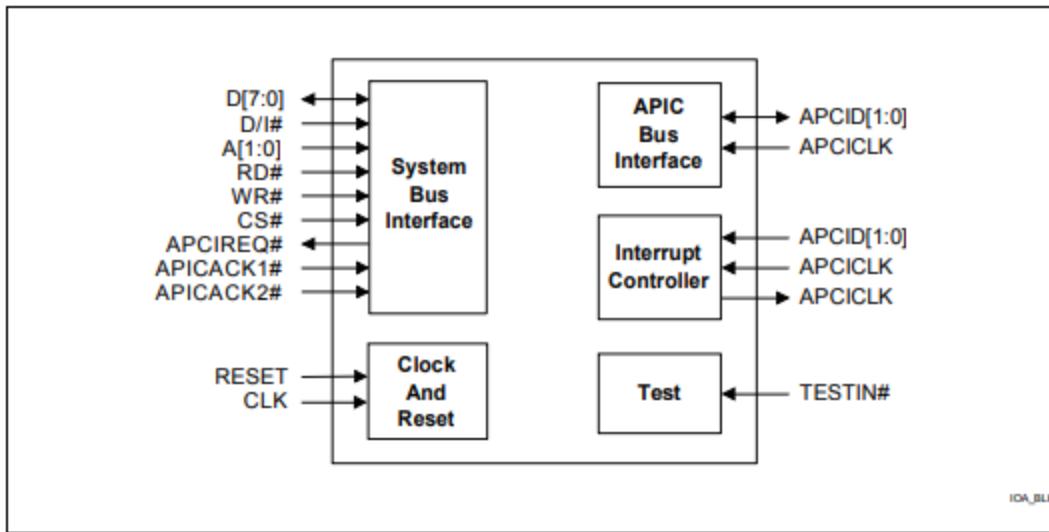


Fig 9 IOAPIC Simplified Block Diagram

13.0 FACILITIES IN THE INTEL CPU FOR EXECUTING CALLS TO PROCEDURES/SUBROUTINES

13.1 PROCEDURE CALL TYPES

The processor supports procedure calls in the following two different ways:

- CALL and RET instructions.
- ENTER and LEAVE instructions, in conjunction with the CALL and RET instructions.

Both of these procedure call mechanisms use the procedure stack, commonly referred to simply as “the stack,” to save the state of the calling procedure, pass parameters to the called procedure, and

store local variables for the currently executing procedure. The processor’s facilities for handling interrupts and exceptions are similar to those used by the CALL and RET instructions.

13.2 STACKS

The stack (see Figure 10) is a contiguous array of memory locations. It is contained in a segment and identified by the segment selector in the SS register. When using the flat memory model, the stack can be located anywhere in the linear address space for the program. A stack can be up to 4 GBytes long, the maximum size of a segment. Items are placed on the stack using the PUSH instruction and removed from the stack using the POP instruction. When an item is pushed onto the stack, the processor decrements the ESP register, then writes the item at the new top of stack.

When an item is popped off the stack, the processor reads the item from the top of stack, then increments the ESP register. In this manner, the stack grows **down** in memory (towards lesser addresses) when items are pushed on the stack and shrinks **up** (towards greater addresses) when the items are popped from the stack. A program or operating system/executive can set up many stacks. For example, in multitasking systems, each task can be given its own stack. The number of stacks in a system is limited by the maximum number of segments and the available physical memory.

When a system sets up many stacks, only one stack—the **current stack**—is available at a time. The current stack is the one contained in the segment referenced by the SS register.

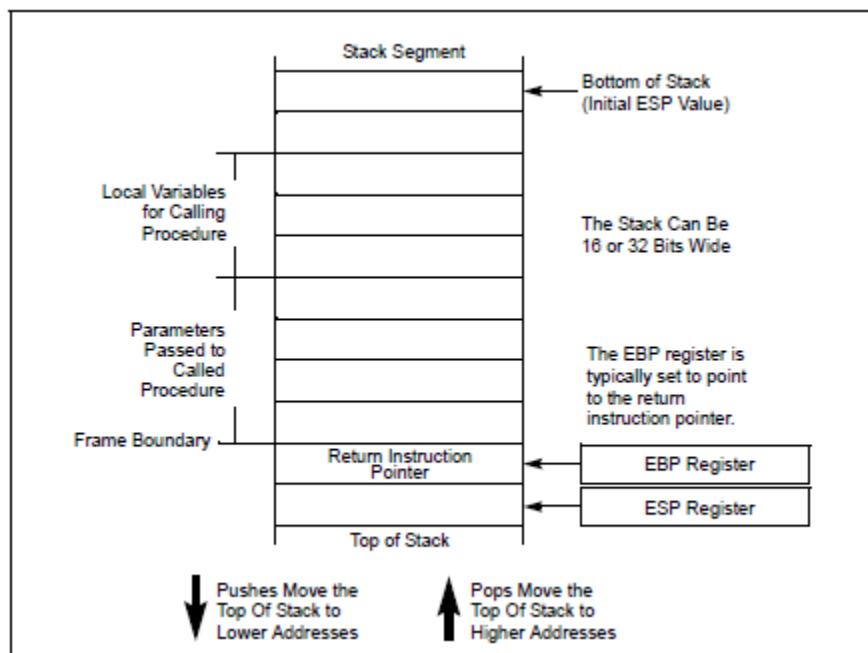


Figure 10. Stack Structure

The processor references the SS register automatically for all stack operations. For example, when the ESP register is used as a memory address, it automatically points to an address in the current stack. Also, the CALL, RET, PUSH, POP, ENTER, and LEAVE instructions all perform operations on the current stack.

13.2.1 Setting Up a Stack

To set a stack and establish it as the current stack, the program or operating system/executive must do the following:

1. Establish a stack segment.
2. Load the segment selector for the stack segment into the SS register using a MOV, POP, or LSS instruction.
3. Load the stack pointer for the stack into the ESP register using a MOV, POP, or LSS instruction.
The LSS instruction can be used to load the SS and ESP registers in one operation.

See “Segment Descriptors” in Chapter 3, “Protected-Mode Memory Management,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for information on how to set up a segment descriptor and segment limits for a stack segment.

13.2.2 Stack Alignment

The stack pointer for a stack segment should be aligned on 16-bit (word) or 32-bit (double-word) boundaries, depending on the width of the stack segment. The D flag in the segment descriptor for the current code segment sets the stack-segment width (see “Segment Descriptors” in Chapter 3, “Protected-Mode Memory Management,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

The PUSH and POP instructions use the D flag to determine how much to decrement or increment the stack pointer on a push or pop operation, respectively. When the stack width is 16 bits, the stack pointer is incremented or decremented in 16-bit increments; when the width is 32 bits, the stack pointer is incremented or decremented in 32-bit increments. Pushing a 16-bit value onto a 32-bit wide stack can result in stack misaligned (that is, the stack pointer is not aligned on a double-word boundary). One exception to this rule is when the contents of a segment register (a 16-bit segment selector) are pushed onto a 32-bit wide stack. Here, the processor automatically aligns the stack pointer to the next 32-bit boundary. The processor does not check stack pointer alignment. It is the responsibility of the programs, tasks, and system procedures running on the processor to maintain proper alignment of stack pointers. Misaligning a stack pointer can cause serious performance degradation and in some instances program failures.

13.2.3 Address-Size Attributes for Stack Accesses

Instructions that use the stack implicitly (such as the PUSH and POP instructions) have two address-size attributes each of either 16 or 32 bits. This is because they always have the implicit address of the top of the stack, and they may also have an explicit memory address (for example, PUSH Array1[EBX]). The attribute of the explicit address is determined by the D flag of the current code segment and the presence or absence of the 67H address-size prefix.

The address-size attribute of the top of the stack determines whether SP or ESP is used for the stack access. Stack operations with an address-size attribute of 16 use the 16-bit SP stack pointer register and can use a maximum stack address of FFFFH; stack operations with an address-size attribute of 32 bits use the 32-bit ESP register and can use a maximum address of FFFFFFFFH. The default address-size attribute for data segments used as stacks is controlled by the B flag of the segment's descriptor. When this flag is clear, the default address-size attribute is 16; when the flag is set, the address-size attribute is 32.

13.2.4 Procedure Linking Information

The processor provides two pointers for linking of procedures: the stack-frame base pointer and the return instruction pointer. When used in conjunction with a standard software procedure-call technique, these pointers permit reliable and coherent linking of procedures.

13.2.4.1 Stack-Frame Base Pointer

The stack is typically divided into frames. Each stack frame can then contain local variables, parameters to be passed to another procedure, and procedure linking information. The stack-frame base pointer (contained in the EBP register) identifies a fixed reference point within the stack frame for the called procedure. To use the stack frame base pointer, the called procedure typically copies the contents of the ESP register into the EBP register prior to pushing any local variables on the stack. The stack-frame base pointer then permits easy access to data structures passed on the stack, to the return instruction pointer, and to local variables added to the stack by the called procedure. Like the ESP register, the EBP register automatically points to an address in the current stack segment (that is, the segment specified by the current contents of the SS register).

13.2.4.2 Return Instruction Pointer

Prior to branching to the first instruction of the called procedure, the CALL instruction pushes the address in the EIP register onto the current stack. This address is then called the return-instruction pointer and it points to the instruction where execution of the calling procedure should resume following a return from the called procedure. Upon returning from a called procedure, the RET instruction pops the return-instruction pointer from the stack back into the EIP register. Execution of the calling procedure then resumes. The processor does not keep track of the location of the return-instruction pointer. It is thus up to the programmer to insure that stack pointer is pointing to the return-instruction pointer on the stack, prior to issuing a RET instruction.

A common way to reset the stack pointer to the point to the return-instruction pointer is to move the contents of the EBP register into the ESP register. If the EBP register is loaded with the stack pointer immediately following a procedure call, it should point to the return instruction pointer on the stack. The processor does not require that the return instruction pointer point back to the calling procedure. Prior to executing the RET instruction, the return instruction pointer can be manipulated in software to point to any address in the current code segment (near return) or another code segment (far return). Performing such an operation, however, should be undertaken very cautiously, using only well-defined code entry points.

13.2.5 Stack Behavior in 64-Bit Mode

In 64-bit mode, address calculations that reference SS segments are treated as if the segment base is zero. Fields (base, limit, and attribute) in segment descriptor registers are ignored. SS DPL is modified such that it is always equal to CPL. This will be true even if it is the only field in the SS descriptor that is modified. Registers E(SP), E(IP) and E(BP) are promoted to 64-bits and are renamed RSP, RIP, and RBP respectively. Some forms of segment load instructions are invalid (for example, LDS, POP ES). PUSH/POP instructions increment/decrement the stack using a 64-bit width. When the contents of a segment register is pushed onto 64-bit stack, the pointer is automatically aligned to 64 bits (as with a stack that has a 32-bit width).

13.3 Calling Procedures Using Call and Ret

The CALL instruction allows control transfers to procedures within the current code segment (**near call**) and in a different code segment (**far call**). Near calls usually provide access to local procedures within the currently running program or task. Far calls are usually used to access operating system procedures or procedures in a different task. See “CALL—Call Procedure” in Chapter 3, “Instruction Set Reference, A-L,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, for a detailed description of the CALL instruction.

The RET instruction also allows near and far returns to match the near and far versions of the CALL instruction. In addition, the RET instruction allows a program to increment the stack pointer on a return to release parameters from the stack. The number of bytes released from the stack is determined by an optional argument (*n*) to the RET instruction.

See “RET—Return from Procedure” in Chapter 4, “Instruction Set Reference, M-U,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, for a detailed description of the RET instruction.

13.3.1 Near CALL and RET Operation

When executing a near call, the processor does the following (see Figure 11):

1. Pushes the current value of the EIP register on the stack.
2. Loads the offset of the called procedure in the EIP register.
3. Begins execution of the called procedure.

When executing a near return, the processor performs these actions:

1. Pops the top-of-stack value (the return instruction pointer) into the EIP register.
2. If the RET instruction has an optional *n* argument, increments the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack.
3. Resumes execution of the calling procedure.

13.3.2 Far CALL and RET Operation

When executing a far call, the processor performs these actions (see Figure 11):

1. Pushes the current value of the CS register on the stack.
2. Pushes the current value of the EIP register on the stack.
3. Loads the segment selector of the segment that contains the called procedure in the CS register.
4. Loads the offset of the called procedure in the EIP register.
5. Begins execution of the called procedure.

When executing a far return, the processor does the following:

1. Pops the top-of-stack value (the return instruction pointer) into the EIP register.
2. Pops the top-of-stack value (the segment selector for the code segment being returned to) into the CS register.
3. If the RET instruction has an optional *n* argument, increments the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack.
4. Resumes execution of the calling procedure.

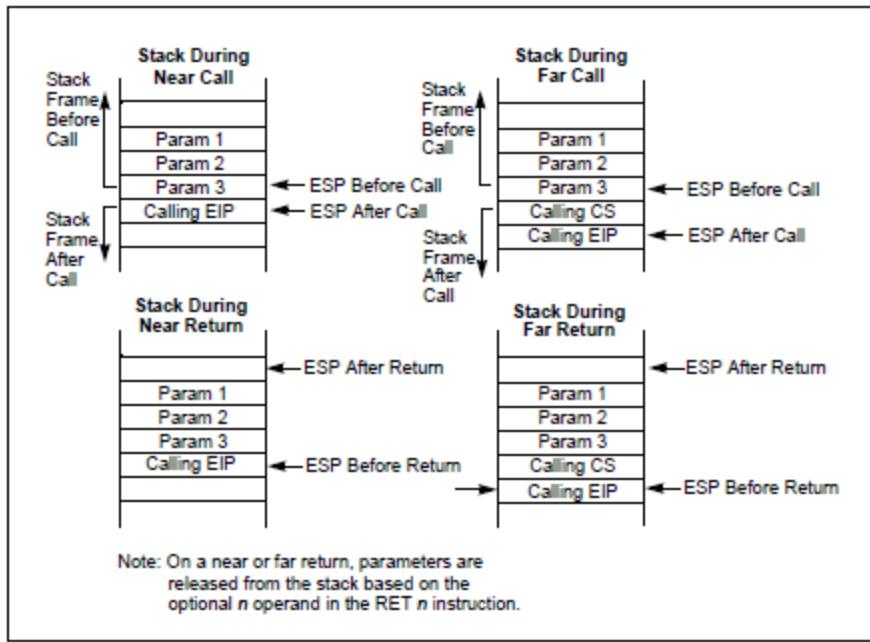


Figure 11. Stack on Near and Far Calls

13.3.3 Parameter Passing

Parameters can be passed between procedures in any of three ways: through general-purpose registers, in an argument list, or on the stack.

13.3.3.1 Passing Parameters Through the General-Purpose Registers

The processor does not save the state of the general-purpose registers on procedure calls. A calling procedure can thus pass up to six parameters to the called procedure by copying the parameters into any of these registers (except the ESP and EBP registers) prior to executing the CALL instruction. The called procedure can likewise pass parameters back to the calling procedure through general-purpose registers.

13.3.3.2 Passing Parameters on the Stack

To pass a large number of parameters to the called procedure, the parameters can be placed on the stack, in the stack frame for the calling procedure. Here, it is useful to use the stack-frame base pointer (in the EBP register) to make a frame boundary for easy access to the parameters. The stack can also be used to pass parameters back from the called procedure to the calling procedure.

13.3.3.3 Passing Parameters in an Argument List

An alternate method of passing a larger number of parameters (or a data structure) to the called procedure is to place the parameters in an argument list in one of the data segments in memory. A pointer to the argument list can then be passed to the called procedure through a general-purpose register or the stack. Parameters can also be passed back to the calling procedure in this same manner.

13.3.4 Saving Procedure State Information

The processor does not save the contents of the general-purpose registers, segment registers, or the EFLAGS register on a procedure call. A calling procedure should explicitly save the values in any of the general-purpose registers that it will need when it resumes execution after a return. These values can be saved on the stack or in memory in one of the data segments. The PUSHA and POPA instructions facilitate saving and restoring the contents of the general-purpose registers.

PUSHA pushes the values in all the general-purpose registers on the stack in the following order: EAX, ECX, EDX, EBX, ESP (the value prior to executing the PUSHA instruction), EBP, ESI, and EDI. The POPA instruction pops all the register values saved with a PUSHA instruction (except the ESP value) from the stack to their respective registers. If a called procedure changes the state of any of the segment registers explicitly, it should restore them to their former values before executing a return to the calling procedure. If a calling procedure needs to maintain the state of the EFLAGS register, it can save and restore all or part of the register using the PUSHF/PUSHFD and POPF/POPFD instructions. The PUSHF instruction pushes the lower word of the EFLAGS register on the stack, while the PUSHFD instruction pushes the entire register. The POPF instruction pops a word from the stack into the lower word of the EFLAGS register, while the POPFD instruction pops a double word from the stack into the register.

13.3.5 Calls to Other Privilege Levels

The IA-32 architecture's protection mechanism recognizes four privilege levels, numbered from 0 to 3, where a greater number mean less privilege. The reason to use privilege levels is to improve the reliability of operating systems. For example, Figure 12 shows how privilege levels can be interpreted as rings of protection.

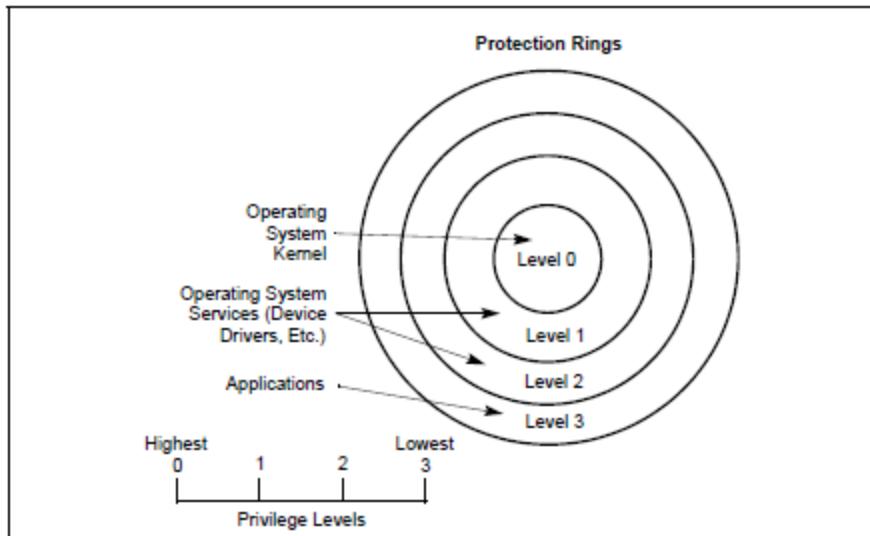


Figure 12. Protection Rings

In this example, the highest privilege level 0 (at the center of the diagram) is used for segments that contain the most critical code modules in the system, usually the kernel of an operating system. The outer rings (with progressively lower privileges) are used for segments that contain code modules for less critical software. Code modules in lower privilege segments can only access modules operating at higher privilege segments by means of a tightly controlled and protected interface called a **gate**. Attempts to access higher privilege segments without going through a protection gate and without having sufficient access rights causes a general-protection exception (#GP) to be generated.

If an operating system or executive uses this multilevel protection mechanism, a call to a procedure that is in a more privileged protection level than the calling procedure is handled in a similar manner as a far call (see Section 6.3.2, “Far CALL and RET Operation”). The differences are as follows:

- The segment selector provided in the CALL instruction references a special data structure called a **call gate descriptor**. Among other things, the call gate descriptor provides the following:
 - access rights information
 - the segment selector for the code segment of the called procedure
 - an offset into the code segment (that is, the instruction pointer for the called procedure)
- The processor switches to a new stack to execute the called procedure. Each privilege level has its own stack. The segment selector and stack pointer for the privilege level 3 stack are stored in the SS and ESP registers, respectively, and are automatically saved when a call to a more privileged level occurs. The segment selectors and stack pointers for the privilege level 2, 1, and

0 stacks are stored in a system segment called the task state segment (TSS). The use of a call gate and the TSS during a stack switch are transparent to the calling procedure, except when a general-protection exception is raised.

13.3.6 CALL and RET Operation Between Privilege Levels

When making a call to a more privileged protection level, the processor does the following (see Figure 13):

1. Performs an access rights check (privilege check).
2. Temporarily saves (internally) the current contents of the SS, ESP, CS, and EIP registers.

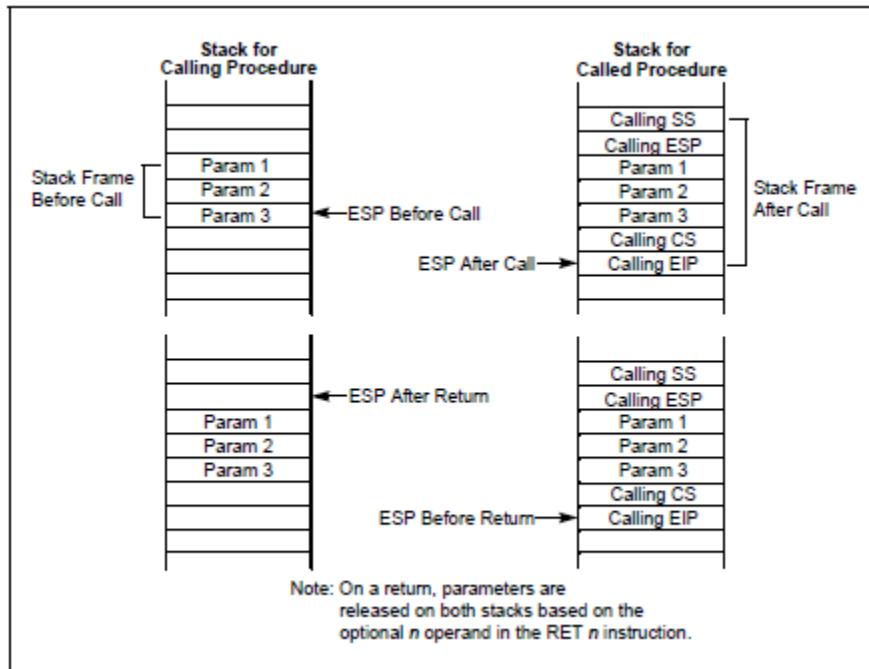


figure 13. Stack Switch on a Call to a Different Privilege Level

3. Loads the segment selector and stack pointer for the new stack (that is, the stack for the privilege level being called) from the TSS into the SS and ESP registers and switches to the new stack.
4. Pushes the temporarily saved SS and ESP values for the calling procedure's stack onto the new stack.
5. Copies the parameters from the calling procedure's stack to the new stack. A value in the call gate descriptor determines how many parameters to copy to the new stack.
6. Pushes the temporarily saved CS and EIP values for the calling procedure to the new stack.

7. Loads the segment selector for the new code segment and the new instruction pointer from the call gate into the CS and EIP registers, respectively.
8. Begins execution of the called procedure at the new privilege level.

When executing a return from the privileged procedure, the processor performs these actions:

1. Performs a privilege check.
2. Restores the CS and EIP registers to their values prior to the call.
3. If the RET instruction has an optional *n* argument, increments the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack. If the call gate descriptor specifies that one or more parameters be copied from one stack to the other, a RET *n* instruction must be used to release the parameters from both stacks. Here, the *n* operand specifies the number of bytes occupied on each stack by the parameters. On a return, the processor increments ESP by *n* for each stack to step over (effectively remove) these parameters from the stacks.
4. Restores the SS and ESP registers to their values prior to the call, which causes a switch back to the stack of the calling procedure.
5. If the RET instruction has an optional *n* argument, increments the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack (see explanation in step 3).
6. Resumes execution of the calling procedure.

[13.3.7 Branch Functions in 64-Bit Mode](#)

The 64-bit extensions expand branching mechanisms to accommodate branches in 64-bit linear-address space.

These are:

- Near-branch semantics are redefined in 64-bit mode
 - In 64-bit mode and compatibility mode, 64-bit call-gate descriptors for far calls are available
- In 64-bit mode, the operand size for all near branches (CALL, RET, JCC, JCXZ, JMP, and LOOP) is forced to 64 bits. These instructions update the 64-bit RIP without the need for a REX operand-size prefix.

The following aspects of near branches are controlled by the effective operand size:

- Truncation of the size of the instruction pointer
- Size of a stack pop or push, due to a CALL or RET
- Size of a stack-pointer increment or decrement, due to a CALL or RET
- Indirect-branch operand size

In 64-bit mode, all of the above actions are forced to 64 bits regardless of operand size prefixes (operand size prefixes are silently ignored). However, the displacement field for relative branches is still limited to 32 bits and the address size for near branches is not forced in 64-bit mode.

Address sizes affect the size of RCX used for JCXZ and LOOP; they also impact the address calculation for memory indirect branches. Such addresses are 64 bits by default; but they can be overridden to 32 bits by an address size prefix. Software typically uses far branches to change privilege levels. The legacy IA-32 architecture provides the call-gate mechanism to allow software to branch from one privilege level to another, although call gates can also be used for branches that do not change privilege levels. When call gates are used, the selector portion of the direct or indirect pointer references a gate descriptor (the offset in the instruction is ignored). The offset to the destination's code segment is taken from the call-gate descriptor.

64-bit mode redefines the type value of a 32-bit call-gate descriptor type to a 64-bit call gate descriptor and expands the size of the 64-bit descriptor to hold a 64-bit offset. The 64-bit mode call-gate descriptor allows far branches that reference any location in the supported linear-address space. These call gates also hold the target code selector (CS), allowing changes to privilege level and default size as a result of the gate transition.

Because immediates are generally specified up to 32 bits, the only way to specify a full 64-bit absolute RIP in 64-bit mode is with an indirect branch. For this reason, direct far branches are eliminated from the instruction set in 64-bit mode. 64-bit mode also expands the semantics of the SYSENTER and SYSEXIT instructions so that the instructions operate within a 64-bit memory space.

The mode also introduces two new instructions: SYSCALL and SYSRET (which are valid only in 64-bit mode). For details, see “SYSENTER—Fast System Call,” “SYSEXIT—Fast Return from Fast System Call,” “SYSCALL—Fast System Call,” and “SYSRET—Return From Fast System Call” in Chapter 4, “Instruction Set Reference, M-U,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.

13.4 Interrupts and Exceptions

The processor provides two mechanisms for interrupting program execution, interrupts and exceptions:

- An **interrupt** is an asynchronous event that is typically triggered by an I/O device.
- An **exception** is a synchronous event that is generated when the processor detects one or more predefined conditions while executing an instruction. The IA-32 architecture specifies three classes of exceptions: faults, traps, and aborts.

The processor responds to interrupts and exceptions in essentially the same way. When an interrupt or exception is signaled, the processor halts execution of the current program or task and switches to a handler procedure that has been written specifically to handle the interrupt or exception condition. The processor accesses the handler procedure through an entry in the interrupt descriptor table (IDT). When the handler has completed handling the interrupt or exception, program control is returned to the interrupted program or task. The operating system, executive, and/or device drivers normally handle interrupts and exceptions independently from application programs or tasks. Application programs can, however, access the interrupt and exception handlers incorporated in an operating system or executive through assembly-language calls. The remainder of this section gives a brief overview of the processor's interrupt and exception handling mechanism. See Chapter 6, "Interrupt and Exception Handling," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for a description of this mechanism.

The IA-32 Architecture defines 18 predefined interrupts and exceptions and 224 user defined interrupts, which are associated with entries in the IDT. Each interrupt and exception in the IDT is identified with a number, called a **vector**. Table 6-1 lists the interrupts and exceptions with entries in the IDT and their respective vectors. Vectors 0 through 8, 10 through 14, and 16 through 19 are the predefined interrupts and exceptions; vectors 32 through 255 are for software-defined interrupts, which are for either **software interrupts** or **maskable hardware interrupts**. Note that the processor defines several additional interrupts that do not point to entries in the IDT; the most notable of these interrupts is the SMI interrupt. See Chapter 6, "Interrupt and Exception Handling," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information about the interrupts and exceptions.

When the processor detects an interrupt or exception, it does one of the following things:

- Executes an implicit call to a handler procedure.
- Executes an implicit call to a handler task.

[13.4.1 Call and Return Operation for Interrupt or Exception Handling Procedures](#)

A call to an interrupt or exception handler procedure is similar to a procedure call to another protection level (see Section 6.3.6, "CALL and RET Operation Between Privilege Levels"). Here, the vector references one of two kinds of gates in the IDT: an **interrupt gate** or a **trap gate**. Interrupt and trap gates are similar to call gates in that they provide the following information:

- Access rights information
- The segment selector for the code segment that contains the handler procedure
- An offset into the code segment to the first instruction of the handler procedure

The difference between an interrupt gate and a trap gate is as follows. If an interrupt or exception handler is called through an interrupt gate, the processor clears the interrupt enable (IF) flag in the EFLAGS register to prevent subsequent interrupts from interfering with the execution of the handler. When a handler is called through a trap gate, the state of the IF flag is not changed.

Table 5. Exceptions and Interrupts

Vector	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV Instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 Instruction.
4	#OF	Overflow	INTO Instruction.
5	#BR	BOUND Range Exceeded	BOUND Instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD Instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT Instruction.

Vector	Mnemonic	Description	Source
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point Instruction. ¹
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT Instruction.
17	#AC	Alignment Check	Any data reference in memory. ²
18	#MC	Machine Check	Error codes (if any) and source are model dependent. ³
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction ⁴
20	#VE	Virtualization Exception	EPT violations ⁵
21-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT n Instruction.

NOTES:

1. IA-32 processors after the Intel386 processor do not generate this exception.

2. This exception was introduced in the Intel486 processor.

3. This exception was introduced in the Pentium processor and enhanced in the P6 family processors.

4. This exception was introduced in the Pentium III processor.

5. This exception can occur only on processors that support the 1-setting of the "EPT-violation #VE" VM-execution control.

If the code segment for the handler procedure has the same privilege level as the currently executing program or task, the handler procedure uses the current stack; if the handler executes at a more privileged level, the processor switches to the stack for the handler's privilege level. If no stack switch occurs, the processor does the following when calling an interrupt or exception handler (see Figure 14):

1. Pushes the current contents of the EFLAGS, CS, and EIP registers (in that order) on the stack.
2. Pushes an error code (if appropriate) on the stack.
3. Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
4. If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
5. Begins execution of the handler procedure.

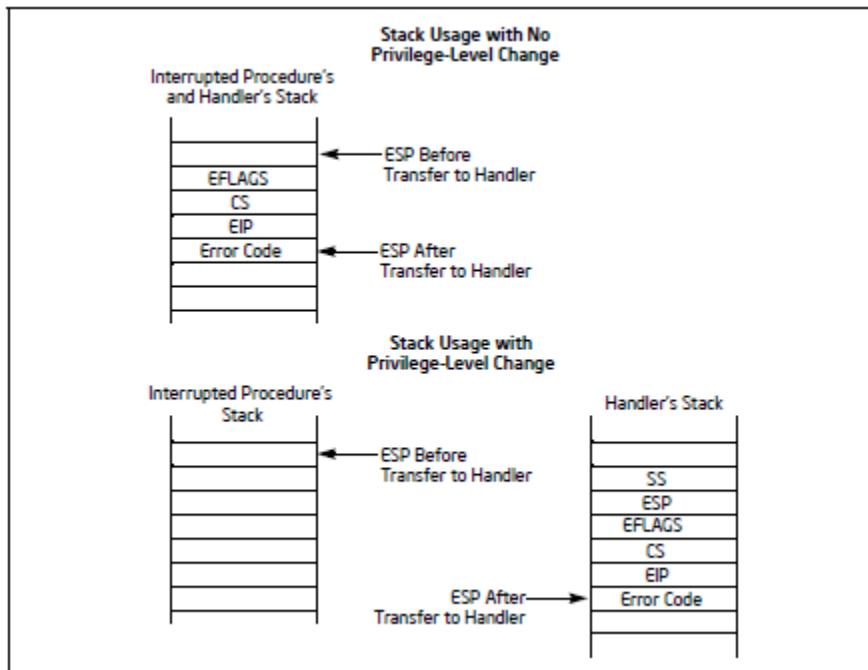


Figure 14. Stack Usage on Transfers to Interrupt and Exception Handling Routines

If a stack switch does occur, the processor does the following:

1. Temporarily saves (internally) the current contents of the SS, ESP, EFLAGS, CS, and EIP registers.
2. Loads the segment selector and stack pointer for the new stack (that is, the stack for the privilege level being called) from the TSS into the SS and ESP registers and switches to the new stack.
3. Pushes the temporarily saved SS, ESP, EFLAGS, CS, and EIP values for the interrupted procedure's stack onto the new stack.

4. Pushes an error code on the new stack (if appropriate).
5. Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
6. If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
7. Begins execution of the handler procedure at the new privilege level.

A return from an interrupt or exception handler is initiated with the IRET instruction. The IRET instruction is similar to the far RET instruction, except that it also restores the contents of the EFLAGS register for the interrupted procedure. When executing a return from an interrupt or exception handler from the same privilege level as the interrupted procedure, the processor performs these actions:

1. Restores the CS and EIP registers to their values prior to the interrupt or exception.
2. Restores the EFLAGS register.
3. Increments the stack pointer appropriately.
4. Resumes execution of the interrupted procedure.

When executing a return from an interrupt or exception handler from a different privilege level than the interrupted procedure, the processor performs these actions:

1. Performs a privilege check.
2. Restores the CS and EIP registers to their values prior to the interrupt or exception.
3. Restores the EFLAGS register.
4. Restores the SS and ESP registers to their values prior to the interrupt or exception, resulting in a stack switch back to the stack of the interrupted procedure.
5. Resumes execution of the interrupted procedure.

[13.4.2 Calls to Interrupt or Exception Handler Tasks](#)

Interrupt and exception handler routines can also be executed in a separate task. Here, an interrupt or exception causes a task switch to a handler task. The handler task is given its own address space and (optionally) can execute at a higher protection level than application programs or tasks. The switch to the handler task is accomplished with an implicit task call that references a **task gate descriptor**. The task gate provides access to the address space for the handler task. As part of the task switch, the processor saves complete state information for the interrupted program or task. Upon returning from the handler task, the state of the interrupted program or task is restored and execution continues. See Chapter 6, “Interrupt and Exception Handling,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for more information on handling interrupts and exceptions through handler tasks.

13.4.3 Interrupt and Exception Handling in Real-Address Mode

When operating in real-address mode, the processor responds to an interrupt or exception with an implicit far call to an interrupt or exception handler. The processor uses the interrupt or exception vector as an index into an interrupt table. The interrupt table contains instruction pointers to the interrupt and exception handler procedures. The processor saves the state of the EFLAGS register, the EIP register, the CS register, and an optional error code on the stack before switching to the handler procedure.

A return from the interrupt or exception handler is carried out with the IRET instruction. See Chapter 20, “8086 Emulation,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*, for more information on handling interrupts and exceptions in real-address mode.

13.4.4 INT *n*, INTO, INT 3, and BOUND Instructions

The INT *n*, INTO, INT 3, and BOUND instructions allow a program or task to explicitly call an interrupt or exception handler. The INT *n* instruction uses a vector as an argument, which allows a program to call any interrupt handler. The INTO instruction explicitly calls the overflow exception (#OF) handler if the overflow flag (OF) in the EFLAGS register is set. The OF flag indicates overflow on arithmetic instructions, but it does not automatically raise an overflow exception.

An overflow exception can only be raised explicitly in either of the following ways:

- Execute the INTO instruction.
- Test the OF flag and execute the INT *n* instruction with an argument of 4 (the vector of the overflow exception) if the flag is set.

Both the methods of dealing with overflow conditions allow a program to test for overflow at specific places in the instruction stream.

The INT 3 instruction explicitly calls the breakpoint exception (#BP) handler.

The BOUND instruction explicitly calls the BOUND-range exceeded exception (#BR) handler if an operand is found to be not within predefined boundaries in memory. This instruction is provided for checking references to arrays and other data structures. Like the overflow exception, the BOUND-range exceeded exception can only be raised explicitly with the BOUND instruction or the INT *n* instruction with an argument of 5 (the vector of the bounds check exception). The processor does not implicitly perform bounds checks and raise the BOUND-range exceeded exception.

13.4.5 Handling Floating-Point Exceptions

When operating on individual or packed floating-point values, the IA-32 architecture supports a set of six floating point exceptions. These exceptions can be generated during operations performed by the x87 FPU instructions or by SSE/SSE2/SSE3 instructions. When an x87 FPU instruction (including the FISTTP instruction in SSE3) generates one or more of these exceptions, it in turn generates floating-point error exception (#MF); when an SSE/SSE2/SSE3 instruction generates a floating-point exception, it in turn generates SIMD floating-point exception (#XM).

See the following sections for further descriptions of the floating-point exceptions, how they are generated, and how they are handled:

- Section 4.9.1, “Floating-Point Exception Conditions,” and Section 4.9.3, “Typical Actions of a Floating-Point Exception Handler”
- Section 8.4, “x87 FPU Floating-Point Exception Handling,” and Section 8.5, “x87 FPU Floating-Point Exception Conditions”
- Section 11.5.1, “SIMD Floating-Point Exceptions”
- Interrupt Behavior

13.4.6 Interrupt and Exception Behavior in 64-Bit Mode

64-bit extensions expand the legacy IA-32 interrupt-processing and exception-processing mechanism to allow support for 64-bit operating systems and applications. Changes include:

- All interrupt handlers pointed to by the IDT are 64-bit code (does not apply to the SMI handler).
- The size of interrupt-stack pushes is fixed at 64 bits. The processor uses 8-byte, zero extended stores.
- The stack pointer (SS:RSP) is pushed unconditionally on interrupts. In legacy environments, this push is conditional and based on a change in current privilege level (CPL).
- The new SS is set to NULL if there is a change in CPL.
- IRET behavior changes.
- There is a new interrupt stack-switch mechanism.
- The alignment of interrupt stack frame is different.

Chapter-4

PC components

1.0 Ports and slots

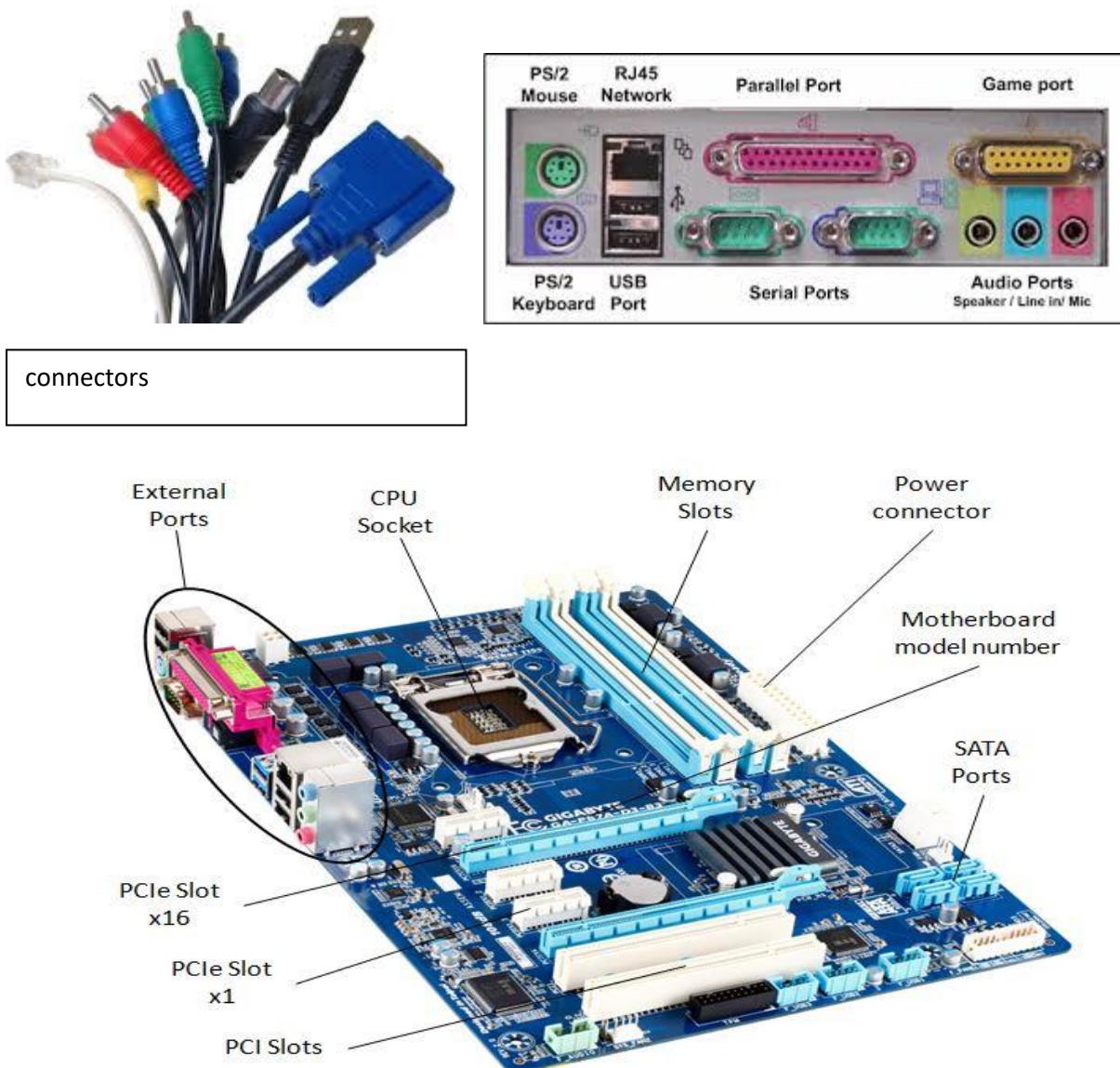


Fig-1:Slots and ports on the motherboard

2.0 LAN card

Some cards, such as a PCMCIA card, can be used in a laptop. There are many other ways of connecting the LAN card to a computer. Some cards are connected via the USB port, some via the PCI port inside of the computer, and some are even embedded inside of the computer.

- A typical Ethernet cable, or network cable, is the plugin that goes into the LAN card, or the Network Interface Controller (NIC).
- Any type of network activity requires a LAN card: the Internet, network printer, connecting computers together, and so on

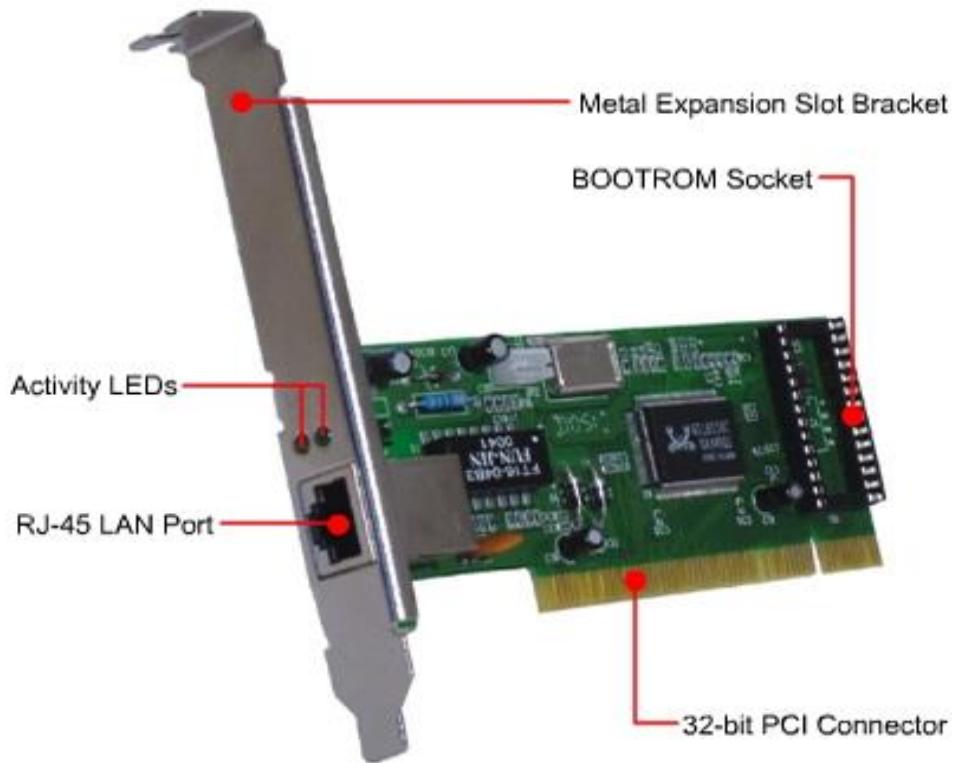


Fig-2:LAN card

3.0 Sound card

- A sound card (also known as an audio card) is an internal expansion card that provides input and output of audio signals to and from a computer under control of computer programs. The term *sound card* is also applied to external audio interfaces used for professional audio applications.
- Sound cards use a digital-to-analog converter (DAC), which converts recorded or generated digital signal data into an analog format.

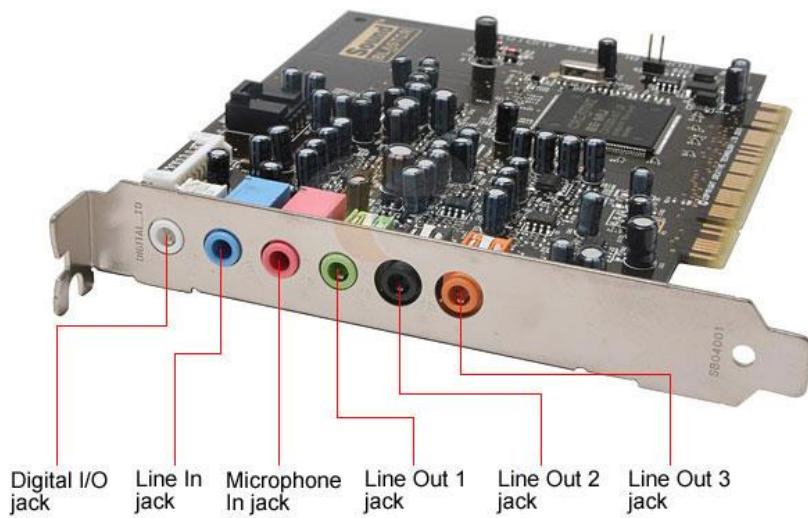


Fig-3:Sound card

4.0 Video card

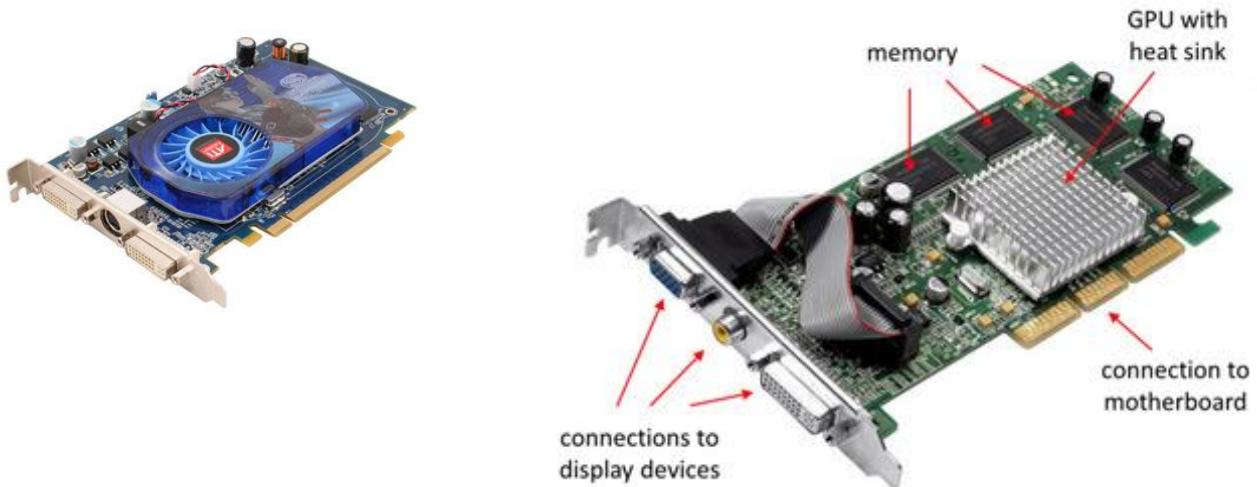


Fig-4:video card

- A **video card** connects to the motherboard of a computer system and generates output images to display. Video cards are also referred to as **graphics cards**. Video cards include a processing unit, memory, a cooling mechanism and connections to a display device.
- The processing unit on a video card is referred to as a **graphics processing unit (GPU)**. This is very similar to a CPU, but its design is optimized to work with images.

- The memory on a video card is very similar to the regular random-access memory (or RAM) on a motherboard.
- A video card connects to the motherboard of a computer system using a slot, typically an Accelerated Graphics Port (AGP) or a Peripheral Component Interconnect Express (PCIe) connection.

5.0 I/O cards



Fig-5:I/o card

I/O is the communication between an information processing system, such as a computer, and the outside world, possibly a human or another information processing system.

6.0 PCI Thunderbolt expansion card and USB card.

Thunderbolt is the brand name of a hardware interface developed by Intel (in collaboration with Apple) that allows the connection of external peripherals to a computer. Thunderbolt 1 and 2 use the same connector as Mini DisplayPort (MDP), whereas Thunderbolt 3 re-uses the USB-C connector from USB. It was initially developed and marketed under the name Light Peak, and first sold as part of a consumer product on 24 February 2011.^[1]

Thunderbolt combines PCI Express (PCIe) and DisplayPort (DP) into two serial signals,^{[5][6]} and additionally provides DC power, all in one cable. Up to six peripherals may be supported by one connector through various topologies



Fig-6: PCI thunderbolt card

7.0 VGA Cable



Fig-7:VGA port and connector

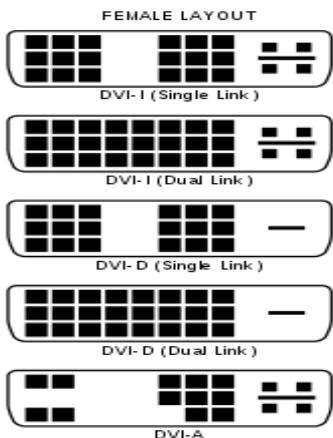
- Also known as D-sub cable, analog video cable
- Connect one end to: computer monitor, television (PC input port)
- VGA connectors and cables carry analog component RGBHV (red, green, blue, horizontal sync, vertical sync) video signals.

8.0 DVI Cable Digital Visual Interface



Fig-8:DVI port and connector

- there are 2 types of DVI, DVI-I and DVI-D
- DVI-D does not have the extra pins around the long pin, this is also a pure digital signal over DVI-I.
- The DVI connector on a device is given one of three names, depending on which signals it implements



DVI-I (integrated, combines digital and analog in the same connector; digital may be single or dual link)

DVI-D (digital only, single link or dual link)

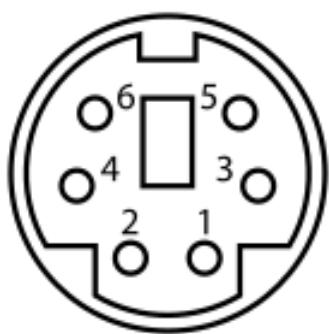
DVI-A (analog only)

9.0 HDMI

- HDMI is a digital replacement for analog video standards.
- It is a proprietary audio/video interface for transmitting uncompressed video data and compressed or uncompressed digital audio data from an HDMI-compliant source device, such as a display controller, to a compatible computer monitor, video projector, digital television, or digital audio device.



10.0 PS/2 Cable



Pin 1	+DATA	Data
Pin 2		Not connected
Pin 3	GND	Ground
Pin 4	<u>Vcc</u>	+5 V DC at 275 mA
Pin 5	+CLK	<u>Clock</u>
Pin 6		Not connected

- **Purple PS/2 port: keyboard**
- **Green PS/2 port: mouse**



11.0 Ethernet Cable

Also known as RJ-45 cable

- An Ethernet cable is one of the most common forms of network cable used on wired networks. Ethernet cables connect devices within a local area network, like PCs, routers, and switches.
- Ethernet cables normally support one or more industry standards including Category 5 and Category 6. Most technicians refer to those standards as CAT5 and CAT6, respectively



12.0 Audio Cable

- A **phone connector**, also known as **phone jack**, **audio jack**, **headphone jack** or **jack plug**, is a family of electrical connectors typically used for analog audio signals.

- Green audio port: computer speakers or headphones
- Pink audio port: microphone
- Blue audio port: MP3 player, CD player, DVD player, turntable, electric guitar etc (line-in port to play and record sounds from the above devices)

13.0 What is multimedia?

Multimedia refers to any application that combine text with other media elements.

- Graphics
- Animation
- Audio
- Video
- Virtual reality

13.1 What is the difference between dedicated graphics and shared graphics?

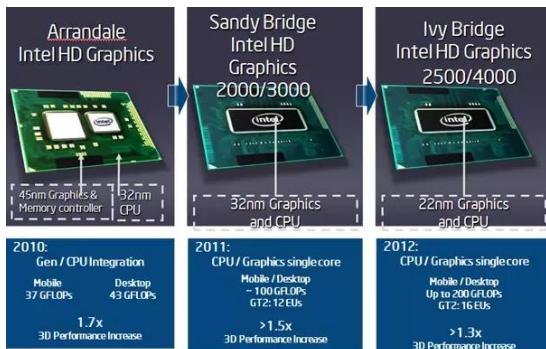


Fig-9:Integrated graphics



fig-10:dedicated graphics

- ❑ A dedicated graphics is more powerful than a shared GPU and it has a dedicated VRAM (in your case 1 GB) to handle all the graphical works. The integrated GPU is powerful, but not powerful enough to render 3D games.
- ❑ The shared graphics or the Integrated graphics comes along with your CPU... The CPUs have integrated graphical processing unit to handle light graphical processing works.
- ❑ The dedicated GPU (may be nvidia or AMD) has lot of processing power and they are fabricated to serve only one purpose i.e. to handle complex calculations which includes graphical processing too



Fig-11:Dedicated graphics

Just as your system has a CPU, some RAM, a CPU fan and a board that connects these components, so does your dedicated graphics card have a board that has a GPU (the equivalent of a CPU), some graphics memory (the equivalent of RAM, known as VRAM), a GPU fan with all of these interconnected on its own board. This board then fits on the larger motherboard through a slot meant for such purposes.

14.0 PCI PCIe

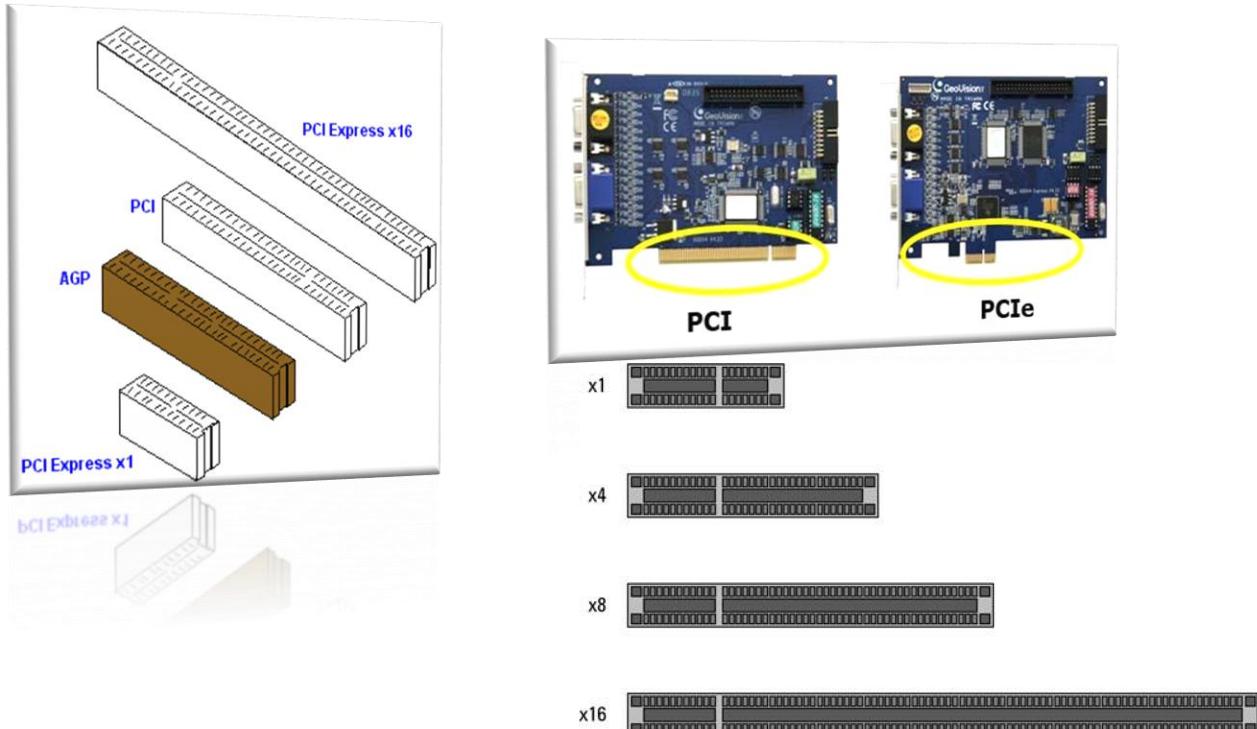


Fig-12:PCI lanes

14.1 PCIe lanes

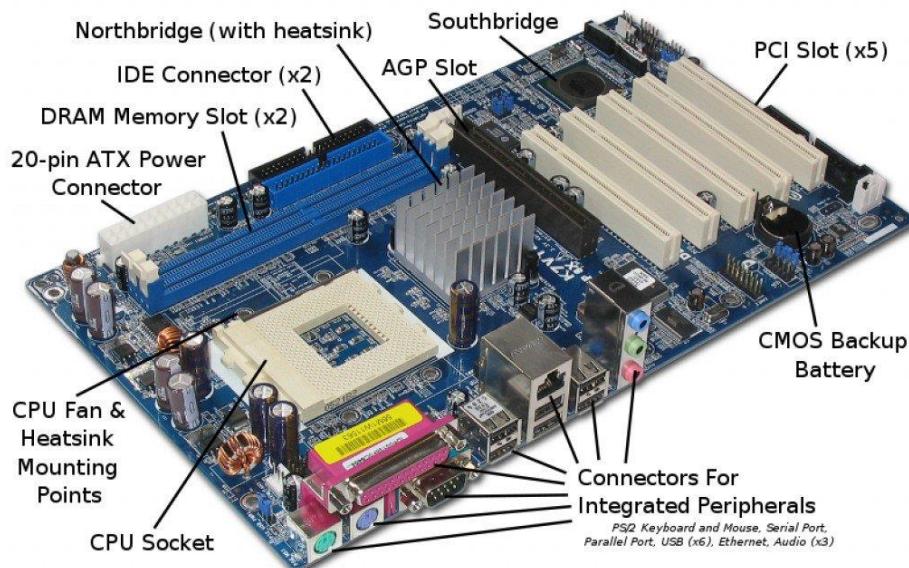
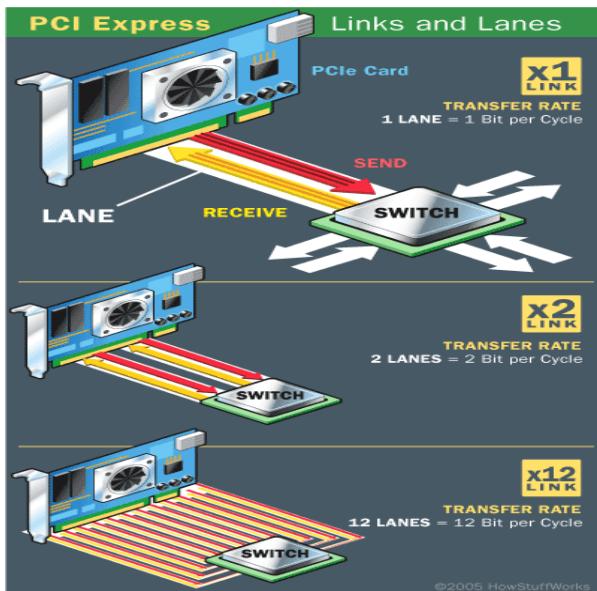
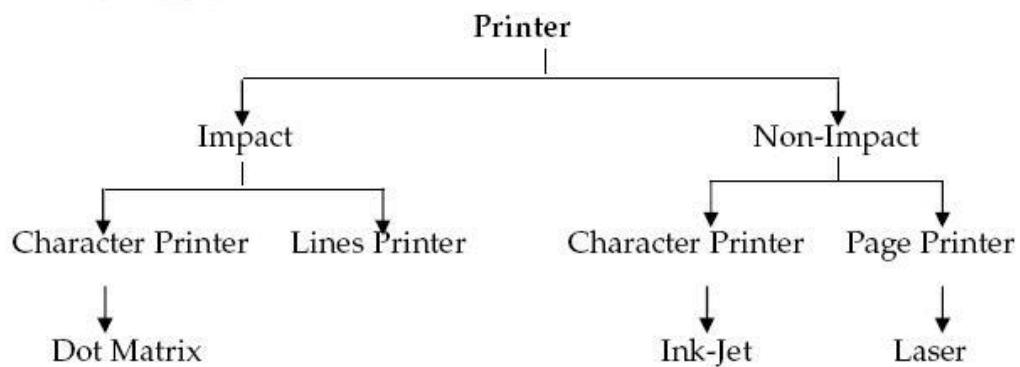


Fig-13:PCI slots on the motherboard

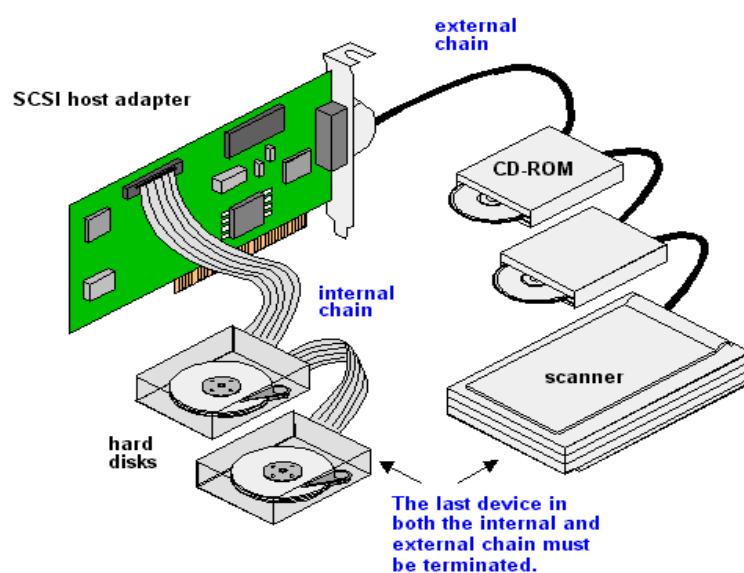
15.0 monitors



16.0 Types of printers

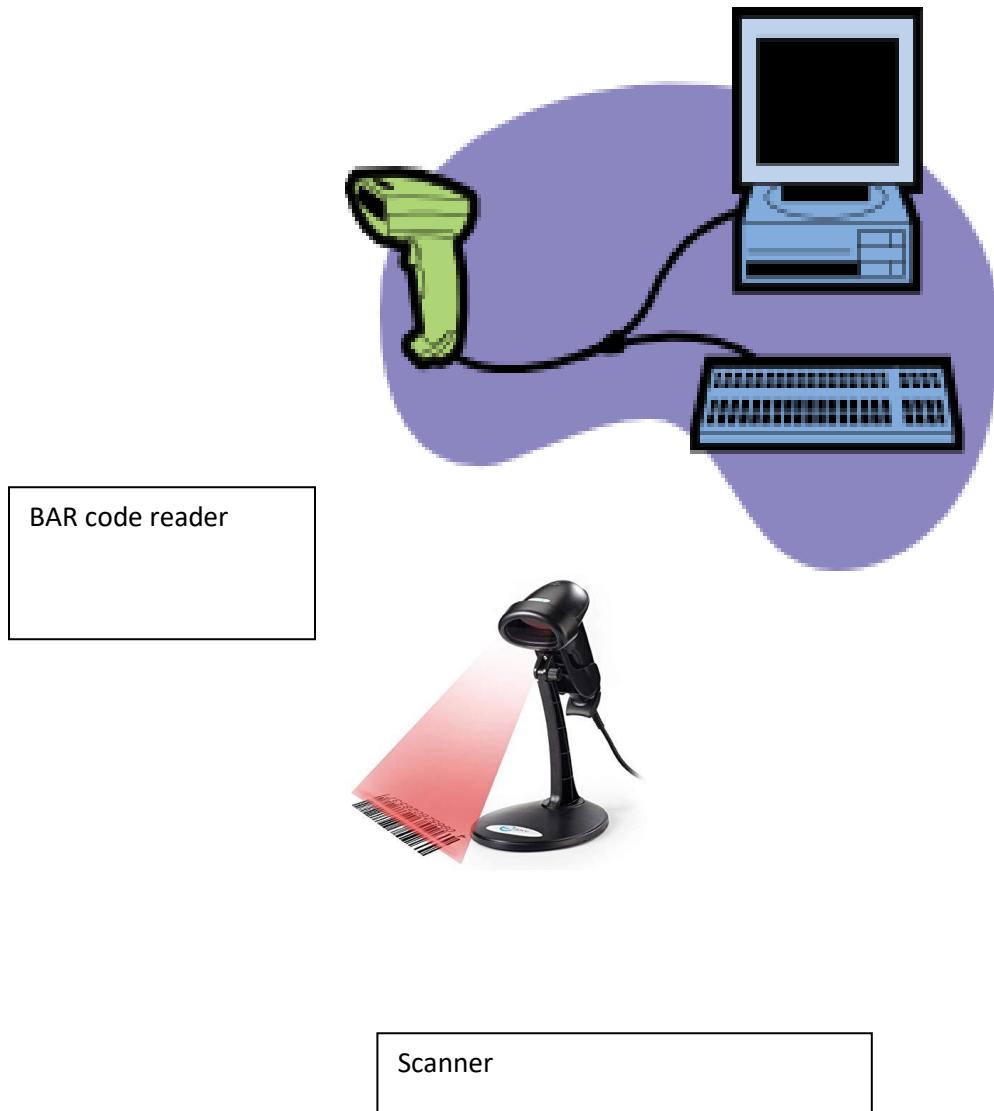


17.0 SCSI interface



Centronics 50-Pin	DB 25-Pin	DB 50-Pin
High Density DB 50-Pin (Clip Type)	High Density DB 50-Pin (Screw Type)	VHDCI 68-Pin
High Density DB 68-Pin (Clip Type)	High Density DB 68-Pin (Screw Type)	High Density Centronics 50-Pin
High Density Centronics 60-Pin	High Density Centronics 68-Pin	HDI30 (Apple/Mac)

18.0 BAR code reader and scanner



19.0 Computer Network

- Computer Network is a system in which multiple computers are connected to each other to share information and resources.



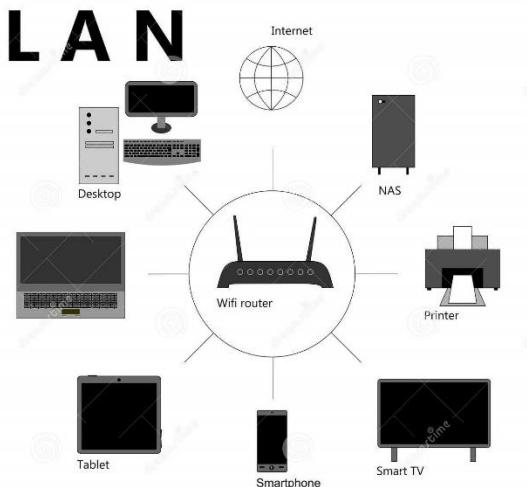
19.1 Advantages of Computer Network

- Better Communication
- File Sharing
- Entertainment
- Flexible Access
- Inexpensive System
- Instant and Multiple Access
- Internet Access
- Resource Sharing

19.2 Types of Computer Network

- Computer networks can be categorized by their size as well as their purpose.
- The size of a network can be expressed by the geographic area.
- Some of the different networks based on size are:
 - LAN – Local Area Network
 - MAN – Metropolitan Area Network
 - WAN – Wide Area Network

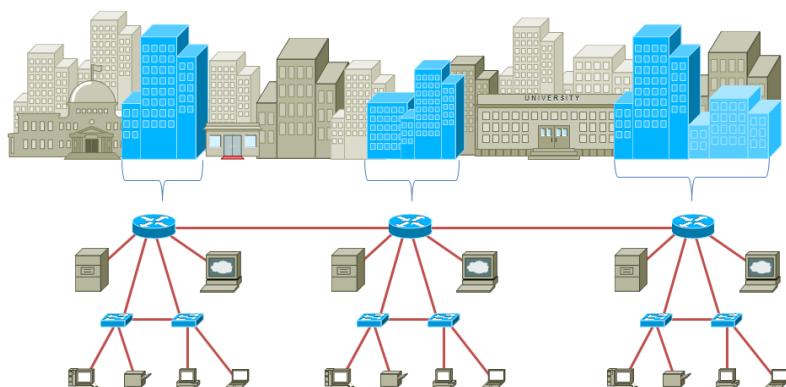
19.3 LAN



- Group of computers and network
- Communication devices interconnected within a geographically limited area, such as a building or a campus.
- transfer data at high speeds
- Key purpose is to serve its users in resource sharing.
- Hardware and software resources are shared.

19.4 MAN

- Spread over city, for example cable TV networks.
- Purpose is sharing hardware and software resource among it's users
- MAN is a larger than LAN but smaller than the area covered by a WAN
- It is also used to interconnection of several local area network.



19.5 WAN

- Spread across countries.
- Group of LANs that are spread across several locations and connected together to look like one big LAN.
- Facilitate fast and efficient exchange of information at lesser cost and higher speed.
- The largest WAN in existence is internet.



Types of Computer Network

Basis Of Comparison	LAN	MAN	WAN
Full Name	Local Area Network	Metropolitan Area Network	Wide Area Network
Meaning	A network that connects a group of computers in a small	It covers relatively large region such as cities, towns	It spans large locality & connects countries together. e.g. Internet

	geographical area		
Ownership of Network	Private	Private or Public	Private or Public (VPN)
Design and Maintenance	Easy	Difficult	Difficult
Propagation Delay	Short	Moderate	Long
Speed	High	Moderate	Low
Components used	NIC, Switch, Hub	Modem, Router	m.wave/ Radio Tx & Rx
Range(Approximately)	1 to 10 km	10 to 100 km	Beyond 100 km
Used for	College, School, Hospital	Small towns, City	State, Country, Continent

20.0 Network Topology

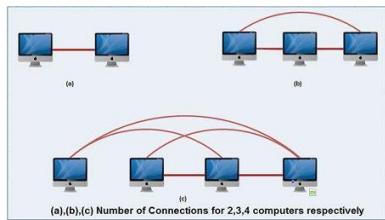
- Network topology is the arrangement of the various elements of a computer or biological network
 1. Point-to-Point Link
 2. Star Topology Communication Link
 3. Bus or Linear Topology
 4. Ring or Circular Topology
 5. Tree Topology Bus or Liner

6. Graph Topology Mesh (Fully)

7. Mesh Topology

20.1 Point- to- Point Link

- Depends on two functions- transmit and receive
- Each station receives exactly from one transmitter
- Each transmitter transmits to exactly one receiver
- Can grow by install a P-P link between each pair of computer in network. This approach is called a Mesh
- Transmit and receive operations can occur either over separate wire or over the same wire



20.2 Star Topology

- Consists of central node to which all other nodes are connected by a single path
- Used in information networks involving data processing or voice communication
- Eg: IBM 370 installations
- Advantages
 - Easy to install and wire
 - No disruptions to the network when connecting or removing devices
 - Easy to detect faults to remove parts
- Disadvantages
 - Require more cable length than a linear topology
 - If the hub, switch, or concentrator fails, nodes attached are disabled
 - More expensive than linear/ bus topologies because of cost of the hubs, etc

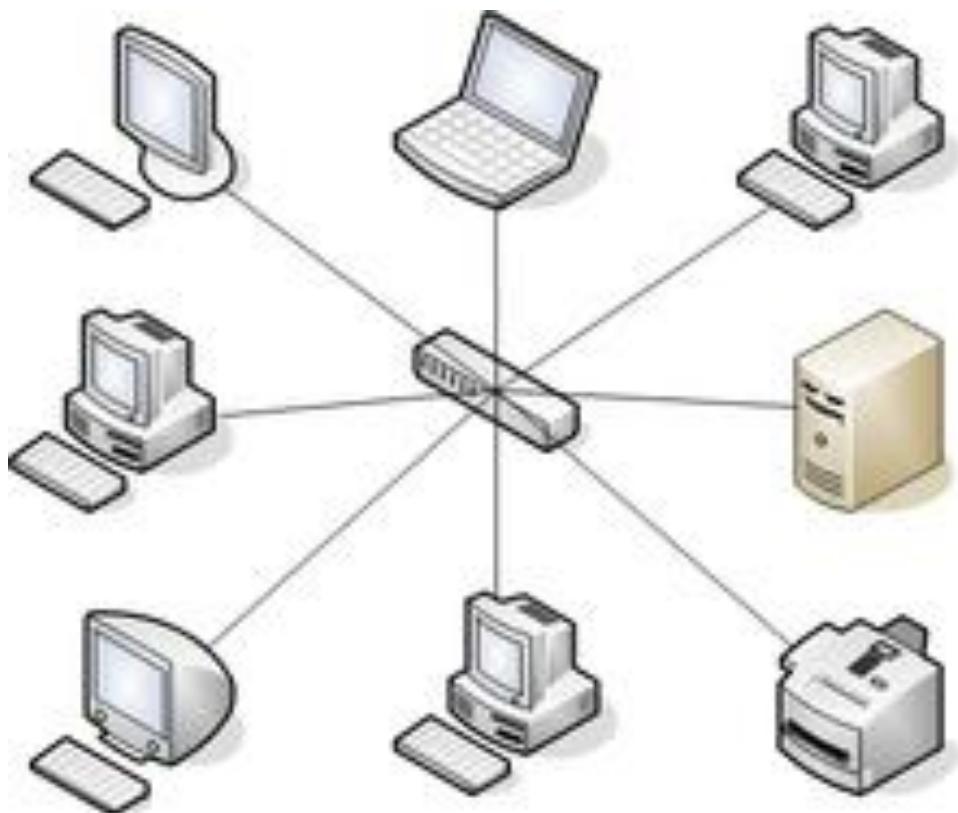
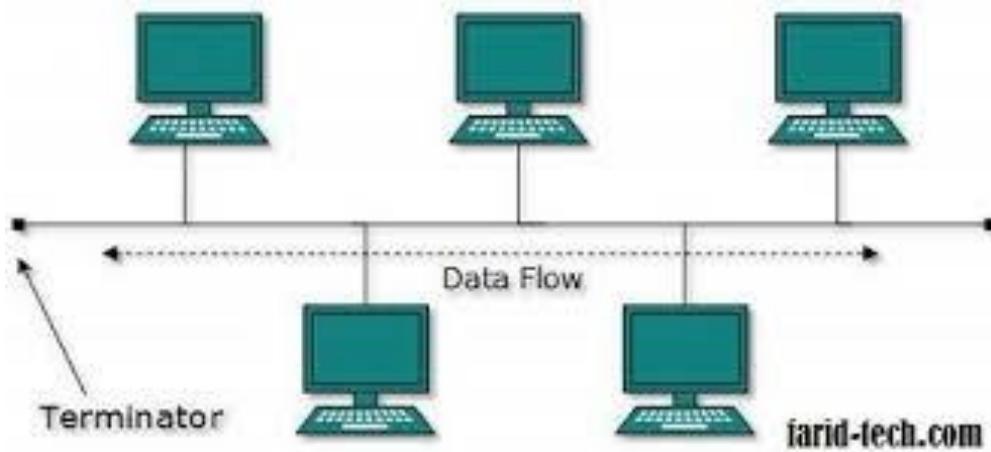


Fig-14:Star topology

20.3 Linear Bus Topology

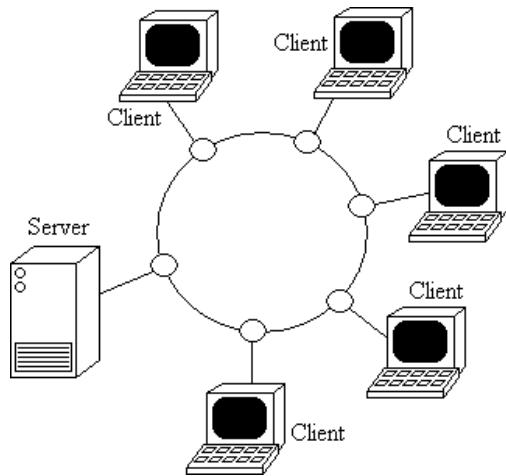
- Consists of single length of the transmission medium on to which the various nodes are attached
- Data is transmitter in small blocks known as packets
- Each packet has some data and destination address
- Transmit some data by send it in packet along with bus
- Destination devices copies the data onto its disk
- Advantages
 - Easy to connect a computer / peripheral to a linear bus
 - Requires less cable length than a star topology
 - Simple architecture
- Disadvantages
 - Entire network shuts down if there is a break in main cable

- Terminators are required at both ends of the backbone Cable
- Difficult to identify the problem if the entire network shuts down
- Not meant to be used as a stand-alone solution in a large building



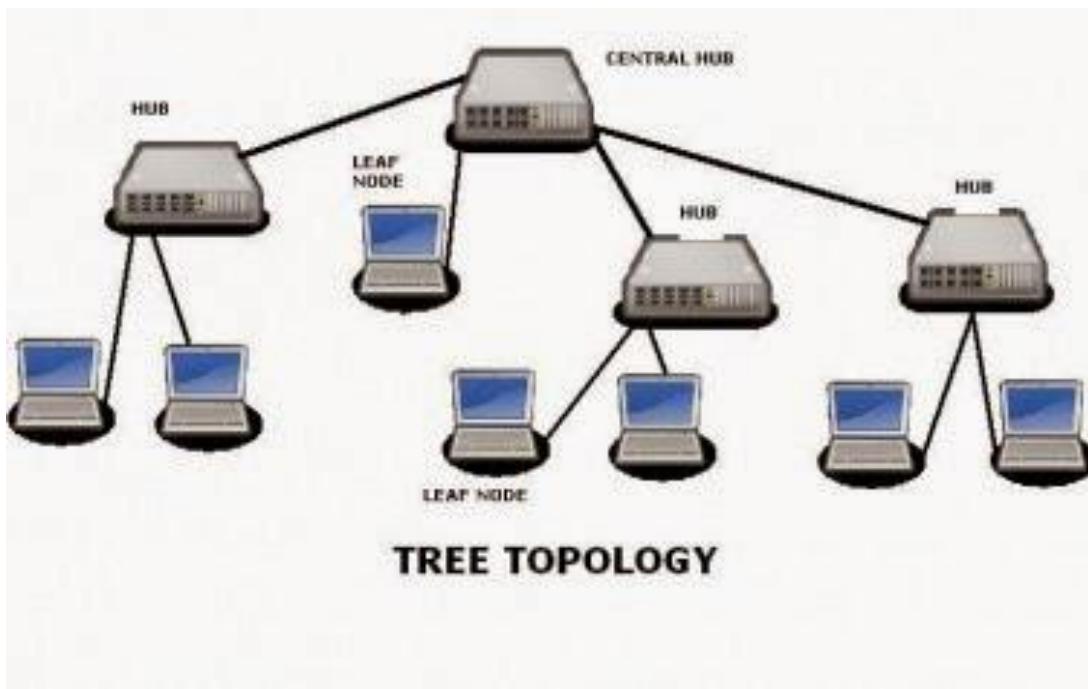
20.4 Ring/ Circular Topology

- Each node is connected to two and only two neighbouring nodes
- Data is accepted from one of the neighbouring nodes and is transmitted onwards to another
- Data travels only in one direction
- Signal may be improved before being used in outward channel
- Advantage
 - Suitable for optical fibre
 - Each computer has equal access to resources
 - Additional components do not affect the performance of network
 - Very organised
 - Requires less cable length than a star topology
- Disadvantages
 - If one node fails to pass data, entire network fails
 - Slower than star topology
 - Difficult to identify the problem in the case of network failure
 - *Reconfiguration of network is difficult*



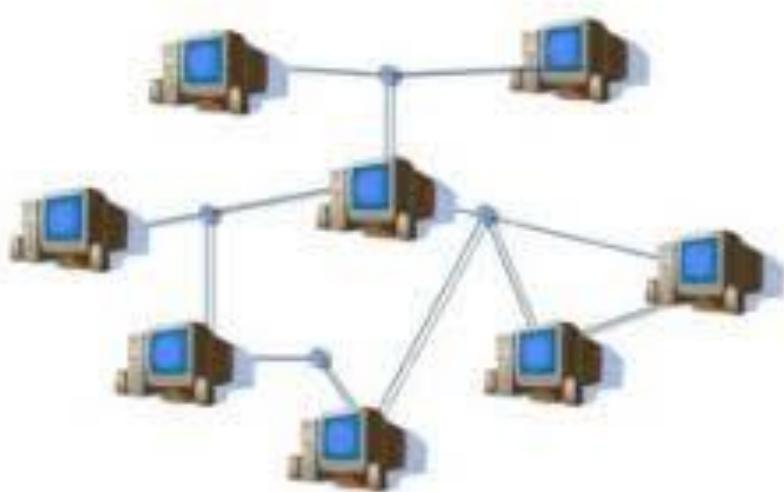
20.5 Tree Topology

- Variation of bus topology
- Shape of the network is like a tree with central branching and sub branching
- Transmission is done in the same way as in bus topology
- Best suited for application which have hierarchical flow of data and control
- Also called hybrid topology
- No need to remove packets from the medium
- Advantages
 - Point- to- Point wiring for individual segments
 - Supported by several hardware and software vendors
- Disadvantages
 - Overall length of each segment is limited by the type of cabling used
 - If the backbone line breaks, the entire segment goes down
 - More difficult to configure and wire than other topologies



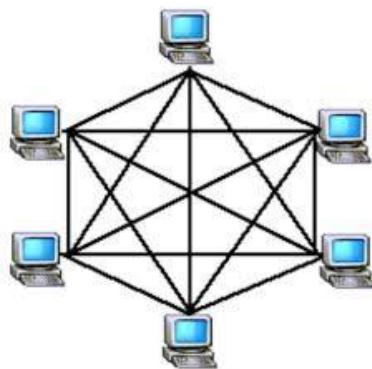
20.6 Graph Topology

- Nodes are connected together randomly
- A link may or may not connect two or more nodes
- There may be multiple nodes
- Not necessary that all the nodes are connected
- If path can be established in two nodes through one or more links, it is known as connected graph



20.7 Mesh Topology

- Each node is connected to more than one node
- Extensions to P-P network
- Outstanding for long distance
- Provide extensive backup, rerouting and pass-through capabilities
- Communication is possible between any two nodes on the network
- Used in large internetworking environments
- Ideal for distributed networks



21.0 Network Devices

21.1 Hubs

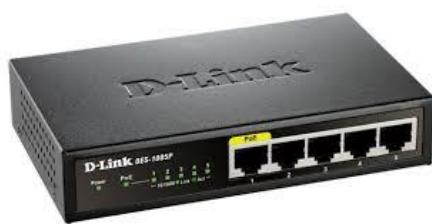
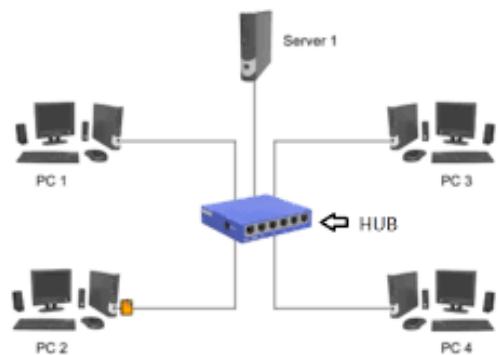


Fig-14:network Hub



- Hubs are devices used to link several computers together. They repeat any signal that comes in on one port and copy it to the other ports (a process that is also called broadcasting)
- There are two types of hubs:..

- **Passive hubs** simply connect all ports together electrically and are usually not powered. (no signal regeneration)
- **Active hubs** use electronics to amplify and clean up the signal before it is broadcast to the other ports.(provide signal regeneration)
- In the category of active hubs, there is also a class called “intelligent” hubs, which are hubs that can be remotely managed on the network.

21.2 Bridges

- They join similar topologies and are used to divide network segments
- For example, with 200 people on one Ethernet segment, the performance will be mediocre, because of the design of Ethernet and the number of workstations that are fighting to transmit. If you divide the segment into two segments of 100 workstations each, the traffic will be much lower on either side and performance will increase.
- If it is aware of the destination address, it is able to forward packets; otherwise a bridge will forward the packets to all segments. They are more intelligent than repeaters but are unable to move data across multiple networks simultaneously.
- Unlike repeaters, bridges can filter out noise.
- The main disadvantage to bridges is that they can't connect dissimilar network types or perform intelligent path selection. For that function, you would need a router
-

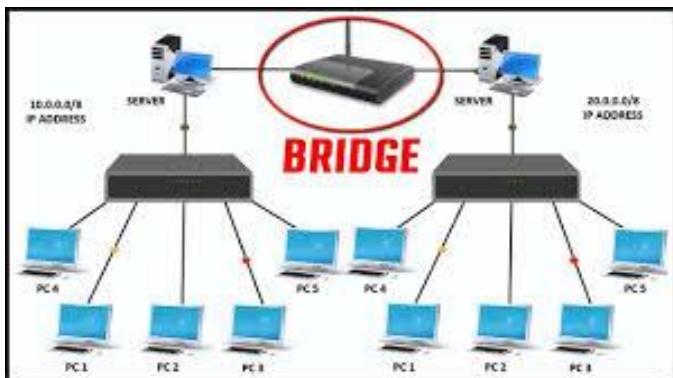


Fig-15: Network bridge

21.3 Routers

- Routers are highly intelligent devices that connect multiple network types and determine the best path for sending data.
- The advantage of using a router over a bridge is that routers can determine the best path that data can take to get to its destination.
- Like bridges, they can segment large networks and can filter out noise.
- However, they are slower than bridges because they are more intelligent devices; as such, they analyze every packet, causing packet-forwarding delays. Because of this intelligence, they are also more expensive.
- Routers are normally used to connect one LAN to another. Typically, when a WAN is set up, there will be at least two routers used.
- Internetwork connectivity device

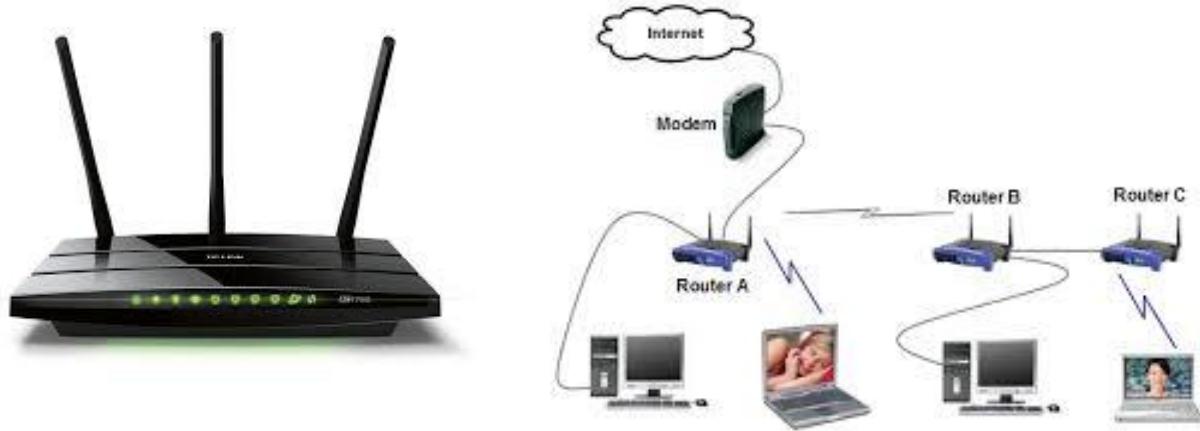


Fig-16: Router

21.4 Switch



Fig-17:network switch

- A network switch is a computer networking device that connects network segments.
- Low-end network switches appear nearly identical to network hubs, but a switch contains more "intelligence" (and a slightly higher price tag) than a network hub.
- Network switches are capable of inspecting data packets as they are received, determining the source and destination device of that packet, and forwarding it appropriately.
- By delivering each message only to the connected device it was intended for, a network switch conserves network bandwidth and offers generally better performance than a hub.
- A vital difference between a hub and a switch is that all the nodes connected to a hub share the bandwidth among themselves, while a device connected to a switch port has the full bandwidth all to itself.
- For example, if 10 nodes are communicating using a hub on a 10-Mbps network, then each node may only get a portion of the 10 Mbps if other nodes on the hub want to communicate as well. But with a switch, each node could possibly communicate at the full 10 Mbps

21.5 Gateways

- A gateway is a network point that acts as an entrance to another network. On the internet, in terms of routing, the network consists of
 - **Host nodes:** computer of network users and the computers that serve contents (such as Web pages).
 - **Gateway nodes:** computers that control traffic within your company's network or at your local internet service provider (ISP)
- A network gateway is a computer which have internetworking capability of joining together two networks that use different base protocols.
- Gateway is node on network that provide entry to another network.
- Operates at highest layer of network abstraction.
- Perform data translation and protocol conversion.
- Associated with both router and switch.
- Can be implemented on software, hardware, or as a combination of both.
- The key point about a gateway is that only the data format is translated, not the data

21.6 Repeater

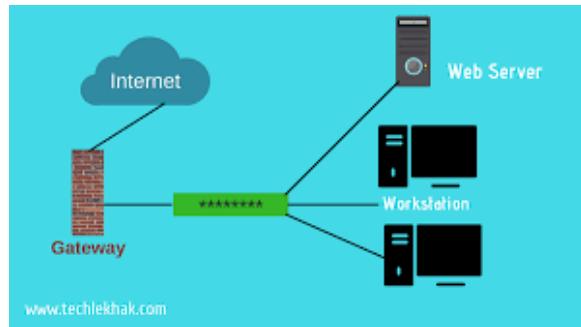


Fig-18:Repeater

- It is a device that amplifies and restores the power of a signal being transmitted on the network.
- Can be installed along the way to ensure that data packets reach their destination.
- Used in long network lines.
- Two types of Repeaters:
 - **Amplifier** : amplifies the incoming signals over the network.
 - **Signal repeater** : collects the inbound packet and retransmits them.

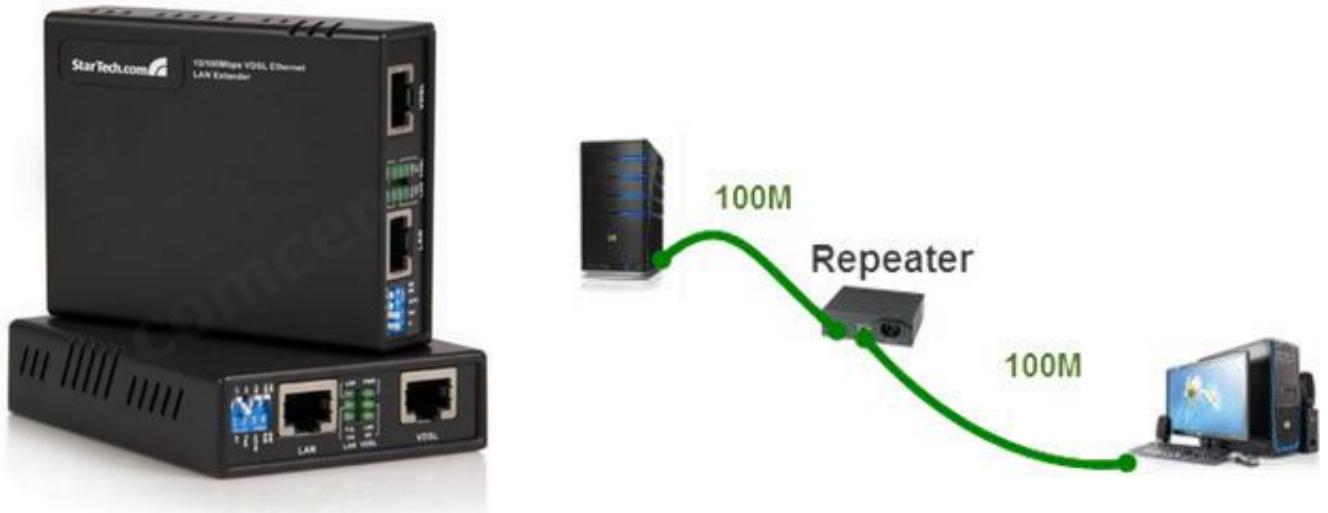


Fig-18:Repeater

22.0 Network Media

- Signals between computers in a network carried through cable based media or wireless media

- Generally 3 types of cables are used
 - Twisted pair cable
 - Unshielded twisted pair (UTP)
 - Shielded twisted pair (STP)
 - Coaxial cable
 - Fibre-optic cable

22.1 Unshielded twisted pair (UTP)

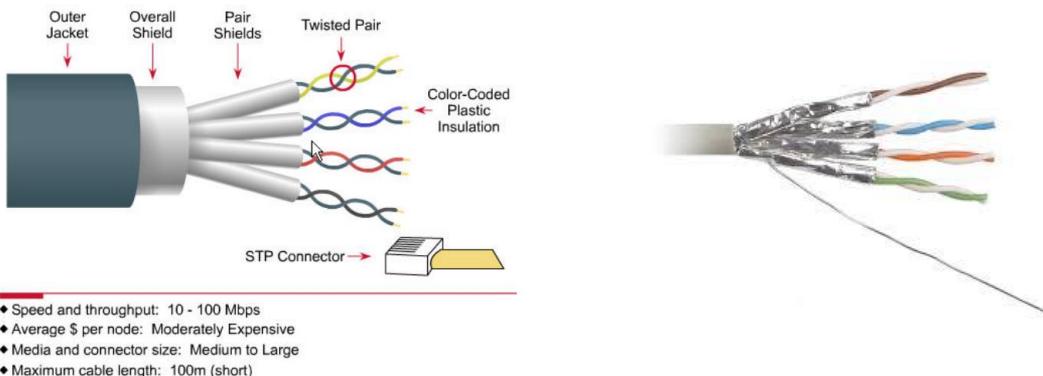
- is a set of three or four pairs of wires with each wire in each pair twisted around the other to prevent electromagnetic interference.
- UTP cabling uses RJ-45, RJ-11, RS232, and RS-449 connectors.
- Because it is less expensive and easier to install, UTP is more popular than Shielded Twisted Pair (STP) or Coaxial Cabling.
- An example of UTP application is telephone networks, which use RJ-11 connectors, and 10BASE-T networks, which use RJ-45 connectors.
- UTP comes in the form of Cat 2, 3, 4, and 5 grades; however, only Cat 5 is now recommended for any data applications.
- The maximum length is 100 meters, without using any kind of signal regeneration device, and a maximum data transfer rate of 1000 Mbps for Gigabit Ethernet.



22.2 Shielded Twisted Pair (STP)

- Shielded Twisted Pair (STP), like UTP, also has four pairs of wires with each wire in each pair twisted together.
- However, STP is surrounded with a foil shield and copper braided around the wires that allows more protection from any external electromagnetic interference.

- Because of the shielding, the cable is physically larger, more difficult to install and terminate, and more expensive than UTP.
- For applications in electrically noisy environments, STP uses RJ-45, RJ-11, RS-232, and RS-449 connectors.
- Like UTP, STP also comes in Cat 2, 3, 4, or 5 grades; however, only Cat5 is recommended for any data applications.
- The maximum cable length with no signal regenerating device is 100 meters, with a maximum data transfer rate is 500 Mbps.



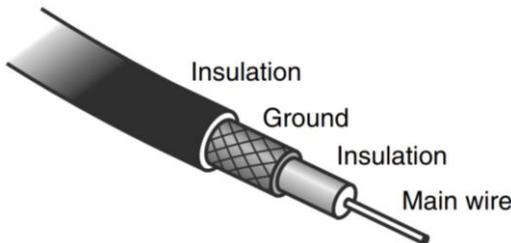
22.3 STP v/s UTP

- STP cables are shielded while UTP cables are unshielded
- STP cables are more immune to interference and noise than UTP cables
- STP cables are better at maximizing bandwidth compared to UTP cables
- STP cables cost more per meter compared to UTP cables
- STP cables are heavier per meter compared to UTP cables
- UTP cables are more prevalent in SOHO networks while STP is used in more high-end applications

22.4 Coaxial cable

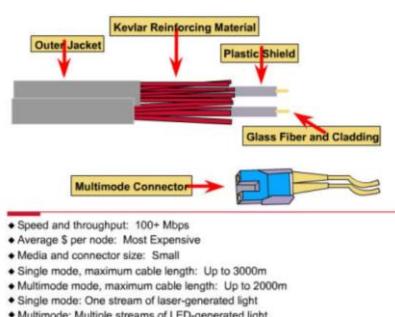
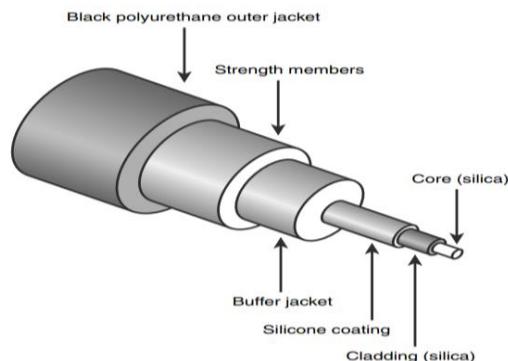
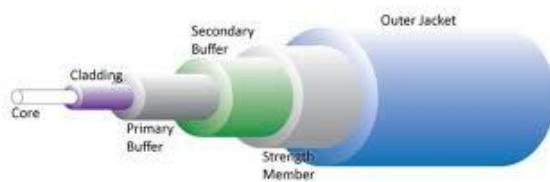
- Coaxial cable is braided-grounded strands of wire that can provide some shielding and noise immunity;
- however, the installation and the termination of the cable itself can be costly.

- Coaxial cabling, which uses connectors called BNC (Bayonet Nut Connector) is known as, in forms of Ethernet, thicknet and thinnet, in the older LAN technology, ARCnet, and cable TV.



22.5 Fibre-optic cable

- Fiber optic cabling carries signals, which have been converted from electrical to optical (pulses of light) form.
- It consists of the core, either an extremely thin cylinder of glass or optical quality plastic, which is surrounded by a second glass or plastic layer called the cladding.
- The interface between the core and cladding can trap light signals by a process called Total Internal Reflection (TIR), resulting in the optical fiber acting as a light pipe.
- Protective buffer and jackets materials are used to cover the cladding layer.
- This type of cabling is less frequently used because it is somewhat more expensive; however, it is rapidly decreasing in both raw cost and installed cost.
- Fiber optic cables are not susceptible to interference, such as radio waves, fluorescent lighting, or any other source of electrical noise.
- It is the common cable used for network backbones and can support up to 1000 stations, carrying signals beyond 25 km.



23.0 Wi-Fi standards

	802.11n	802.11n	802.11ac Wave 1	802.11ac Wave2	802.11ac
	IEEE Specification		Today	WFA Certification Process Continues	IEEE Specification
Band	2.4 GHz & 5 GHz	2.4 GHz & 5 GHz	5 GHz	5 GHz	5 GHz
MIMO	Single User (SU)	Single User (SU)	Single User (SU)	Multi User (MU)	Multi User (MU)
PHY Rate	450 Mbps	600 Mbps	1.3 Gbps	2.34 Gbps - 3.47 Gbps	6.9 Gbps
Channel Width	20 or 40 MHz	20 or 40 MHz	20, 40, 80 MHz	20, 40, 80, 80-80, 160 MHz	20, 40, 80, 80-80, 160 MHz
Modulation	64 QAM	64 QAM	256 QAM	256 QAM	256 QAM
Spatial Streams	3	4	3	3-4	8
MAC Throughput*	293 Mbps	390 Mbps	845 Mbps	1.52 Gbps- 2.26 Gbps	4.49 Gbps

* Assuming a 65% MAC efficiency with highest MCS.

23.1 Access Point (AP)



Fig-19:network access Point

- Access Point is a wireless LAN transceiver or “base station” that can connect one or many wireless devices simultaneously to the Internet.
- An access point is a device that creates a wireless local area network, or WLAN, usually in an office or large building.
- An access point connects to a wired router, switch, or hub via an Ethernet cable, and projects a Wi-Fi signal to a designated area.

- For example, if you want to enable Wi-Fi access in your company's reception area but don't have a router within range, you can install an access point near the front desk and run an Ethernet cable through the ceiling back to the server room.

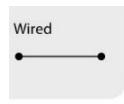
23.2 wireless card



- Wi-Fi cards - They accept the wireless signal and relay information. They can be internal and external.(e.g PCMCIA Card for Laptop and PCI Card for Desktop PC)

23.3 Print server

- A print server is a device that allows you to share a printer with multiple computers.
- It may be a standalone adapter or may be integrated within a printer or a router.
- When activated, the print server allows a printer to connect to a local network rather than a single computer, which then can be accessed by multiple devices
- Standalone print servers come in several varieties. Most have a USB port, which connects directly to the USB port of the printer. However, some print servers can connect to a printer's Ethernet or parallel port as well.
- Wired print servers include an Ethernet port for connecting directly to a router, while wireless versions are able to connect to a Wi-Fi network.
- Printers that include a built-in print server are often called "network printers" or "wireless printers"
- Some routers can also function as print servers. Besides the typical Ethernet ports, they also include a USB port for connecting a printer.



Internet



ADSL Router

TL-PS310U

USB HUB



PC Group



MFP Printer



Scanner



External Hard Drive



Webcam

Fig-20:Print server

Chapter-5

Peripheral Component Interconnect (PCI)

1. Introduction to PCI

A computer bus is used to transfer data from one location or device on the motherboard to the central processing unit where all calculations take place. There are two different parts of a Bus.

- Address bus-transfers information about where the data should go
- Data bus-transfers the actual data

CPU interacts with different peripherals through internal and expansion bus. An Internal Bus is a bus that operates only within the internal circuitry of the CPU, communicating among the internal caches of memory that are part of the CPU chip's design. Microprocessors communicate with its internal and external components through the bus, are mainly divided into two categories:

1. Front-side Bus: Another name for the system bus. The bus that connects the CPU to main memory on the motherboard. The system bus is also called the memory bus, local bus, or host bus.
2. Backside bus: A microprocessor bus that connects the CPU to a Level 2 cache. Typically, a backside bus runs at a faster clock speed than the frontside bus that connects the CPU to main memory.

An expansion bus is a collection of wires and protocols that allows the expansion of a computer by inserting printed circuit boards (expansion boards).

- AT Bus: The AT bus, which runs at 8 megahertz and has a 16-bit data path. The AT bus is sometimes referred to as the ISA bus.
- ISA Bus: Short for Industry Standard Architecture bus. 16 bit bus. ISA began to be replaced by the PCI local bus architecture.
- MCA: IBM introduced the Micro Channel Architecture (MCA) in 1987. 32-bit bus.
- EISA Bus: A more successful alternative to the MCA bus is the Extended Industry Standard Architecture (EISA), a high-speed 32-bit bus architecture developed by a group of IBM's competitors
- VL-bus: Short for VESA Local-Bus, a local bus architecture created by the Video Electronics Standards Association (VESA). Although it was quite popular in PCs made in 1993 and 1994, it has been overshadowed by a competing local bus architecture called PCI.
- PCI Bus: Short for Peripheral Component Interconnect, a local bus standard developed by Intel Corporation. PCI is a 64-bit bus, though it is usually implemented as a 32-bit bus. It can run at clock speeds of 33 or 66 MHz. At 32 bits and 33 MHz, it yields a throughput rate of 133 MBps.
- PCI-X Bus: Short for PCI extended, an enhanced PCI bus. PCI-X is backward-compatible with existing PCI cards. It improves upon the speed of PCI from 133 MBps to as much as 1 GBps.
- NuBus: The expansion bus for versions of the Macintosh computers starting with the Macintosh II and ending with the Performa. Current Macs use the PCI bus.
- SMBus: The System Management Bus (SMBus) is a two-wire interface through which simple power-related chips can communicate with rest of the system. With the SMBus, a device can provide manufacturer information, tell the system what its model or part number is, save its state

for a suspend event, report different types of errors, accept control parameters and return its status.

Peripheral Component Interconnect (PCI) also called Compatible PCI. It's a bus protocol developed by Intel around 1993. The purpose is to attach/interconnect local hardware devices to a computer. PCI is integrated into the chipset, forming a "backbone". Logically speaking, the Chipset is a PCI System

intended for use as an interconnect mechanism between highly integrated peripheral controller components, peripheral add-in cards, and processor/memory systems. It is a parallel bus, synchronous to a single bus clock. PCI component interface is processor independent

2.0 PCI

The first version of conventional PCI found in consumer desktop computers was a 32-bit bus using a 33 MHz bus clock and 5 V signaling, although the PCI 1.0 standard provided for a 64-bit variant as well. These have one locating notch in the card. Version 2.0 of the PCI standard introduced 3.3 V slots, physically distinguished by a flipped physical connector to prevent accidental insertion of 5 V cards. Universal cards, which can operate on either voltage, have two notches. Version 2.1 of the PCI standard introduced optional 66 MHz operation. Conventional PCI supports 4 types of slots and 6 possible PCI card configurations as shown in Figure 1. Because a card can be damaged if installed in the wrong voltage slot, a notch in a PCI slot distinguishes between a 5 V slot and a 3.3 V slot.

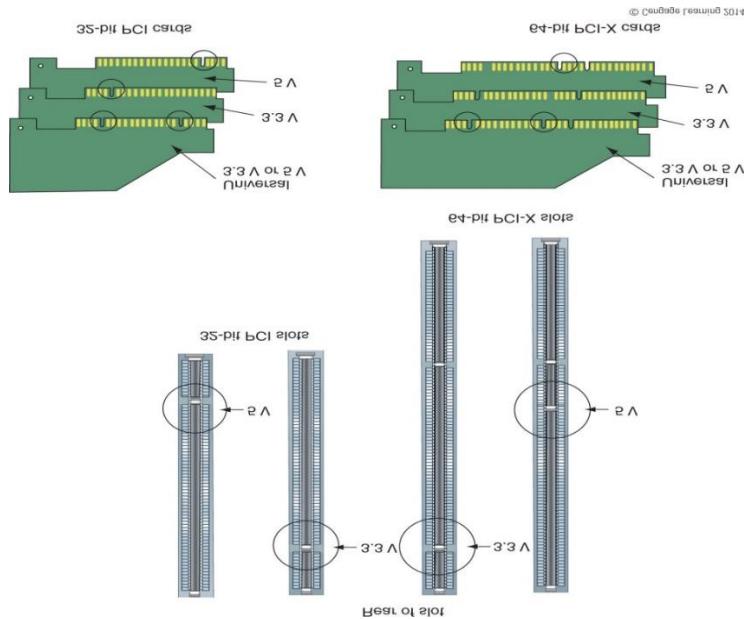


Figure 1. Conventional PCI add in card and slots

A server-oriented variant of conventional PCI, called PCI-X (PCI Extended) operated at frequencies up to 133 MHz for PCI-X 1.0 and up to 533 MHz for PCI-X 2.0. PCI-X uses a 64-bit data path and all revisions are backward compatible. An internal connector for laptop cards, called Mini PCI, was introduced in version 2.2 of the PCI specification. The PCI bus was also

adopted for an external laptop connector standard – the CardBus. The first PCI specification was developed by Intel, but subsequent development of the standard became the responsibility of the PCI Special Interest Group (PCI-SIG).

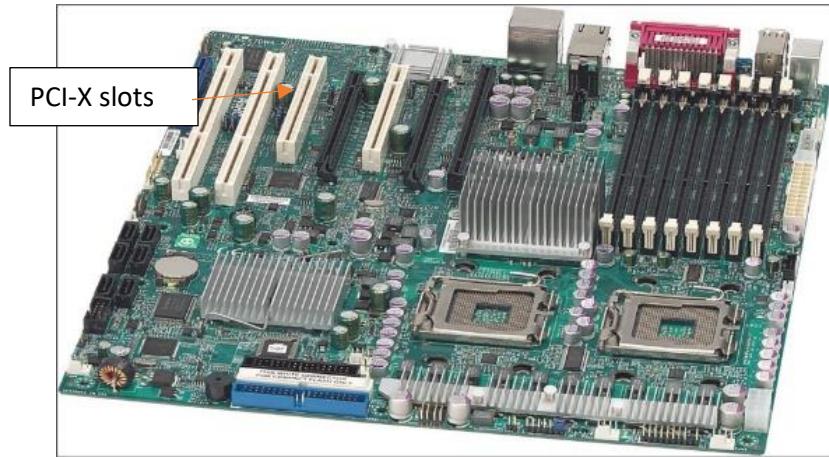


Figure 2. Motherboard with PCI-X slots

Conventional PCI and PCI-X are sometimes called Parallel PCI in order to distinguish them technologically from their more recent successor PCI Express, which adopted a serial, lane-based architecture. PCI and PCI Express are not compatible; the slots have different configurations and do not fit with the cards. However, most motherboards have a combination of PCI and PCI Express cards as shown in figure 3. PCIe is expected to replace both PCI and PCI-X in the future. PCIe comes in four different slot sizes- PCI Express x1, x4, x8, and x16.

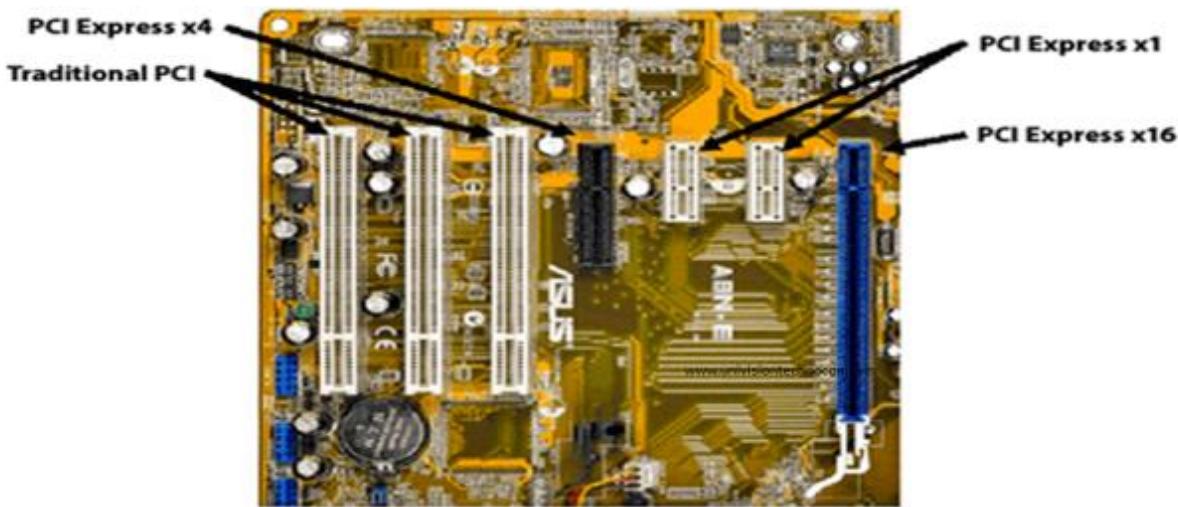


Figure 3. Motherboard with PCIe slots

PCI also supports PCI Riser cards which can be used as further expansion slot. It installs in a PCI slot and provides another slot at a right angle as shown in figure 4. It is used to fit PCI, PCIe, and PCI-X cards into a low-profile or slimline case. PCI riser

card provides a 3.3-V slot or 5-V slot depending on which direction the card is inserted in the PCI slot

Conventional PCI's heyday in the desktop computer market was approximately 1995–2005. PCI and PCI-X have become obsolete for most purposes; however, they are still common on modern desktops for the purposes of backwards compatibility and the low relative cost to produce. Many kinds of devices previously available on PCI expansion cards are now commonly integrated onto motherboards or available in USB and PCI Express versions.

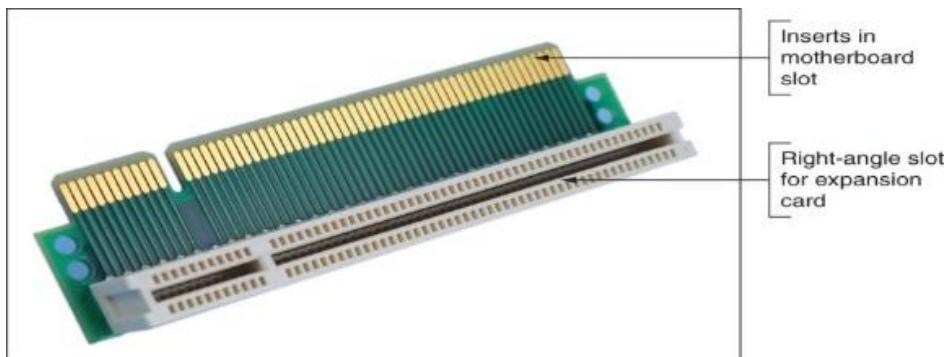


Figure 4. PCI Riser card

2.1 Features of Conventional PCI

1. **Singling Environment:** Support both 3.3 and 5 volt signaling environments.
2. **Reliability:** It offers the ability to replace modules without disturbing a system's operation called as hot plug and hot swap.
3. **Speed:** It can transfer up to 132 MB per second.
4. **Configurability:** The ability to configure a system automatically means automatically identify the interfacing systems and assigns new addresses.
5. **Synchronous bus architecture:** PCI is a synchronous bus where data transfer takes place according to a system clock.
6. **32 and 64 bit addressing:** The PCI bus also supports 64 bit addressing with the same 32 bit connector.
7. **Large bandwidth:** It can handle both 32 bit as well as 64 bit data hence the maximum bandwidth will be 132 MB per second.

2.2 Generic PCI Topology

Figure. 5 shows the basic PCI topology. It supports up to 256 Buses and each Bus can have up to 32 devices attached. Each Device on the Bus can extend up to 8 Functions. Buses are interconnected by PCI-PCI Bridges. Multiple bridges can be connected through a bridge interface and are enumerated

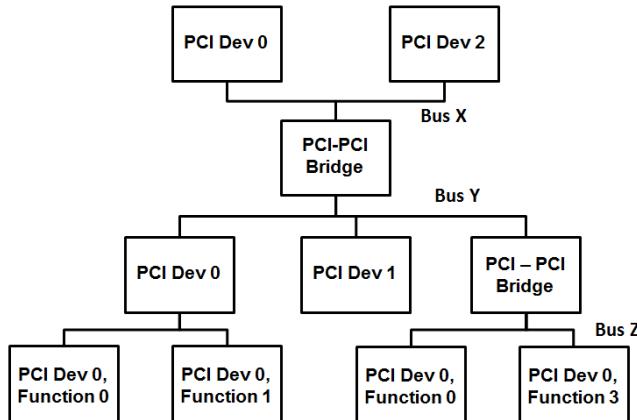


Figure 5. Generic PCI Topology

2.3 PCI Local Bus Overview

The PCI local Bus is a high performance 32-bit or 64-bit bus with multiplexed address and data lines. The bus is intended for use as an interconnect mechanism between highly integrated peripheral controller components, peripheral add-in boards, and processor/memory systems. Figure. 6 shows a typical PCI local Bus system architecture. In this example the processor/cache/memory subsystem is connected to PCI through a PCI bridge. This bridge provides a low latency path through which the processor may directly access PCI devices mapped anywhere in the memory or I/O address spaces. The bridge may optionally include such functions as data buffering/ posting and PCI central functions (e.g. arbitration).

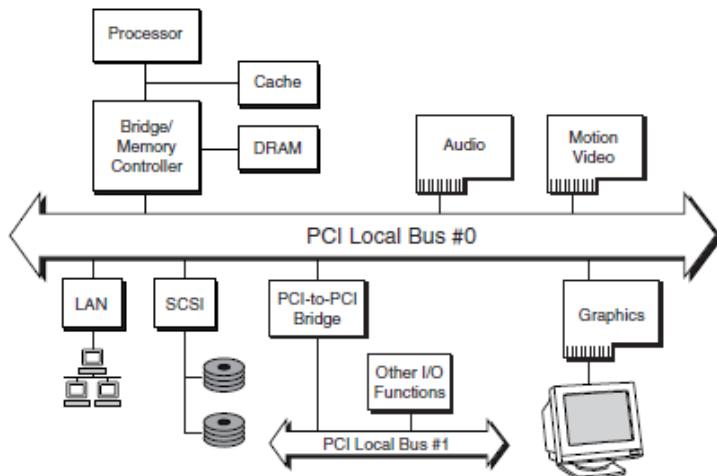


Figure 6. PCI local bus system architecture

3.0 PCI Express

It is a high performance, general purpose I/O interconnect defined for wide range of computing and communication platforms. Along with key features of PCI, usage model, load-store architecture, and software interfaces, its parallel bus implementation is replaced by a highly scalable, fully serial interface.

3.1 Features of PCIe

1. point-to-point interconnects
2. Switch-based technology
3. packetized protocol
4. Power Management
5. Quality Of Service (QoS)
6. Hot-Plug/Hot-Swap support
7. Data Integrity
8. Error Handling

3.2 Generic PCIe Topology

Figure. 7 shows the basic PCIe topology. It supports up to 256 Buses and each Bus can have up to 32 devices attached similar to that of PCI topology. Each Device on the Bus can extend up to 8 Functions. Each Function can implement up to 4 KB of configuration space. The Root Complex connects the processor to the system memory and components

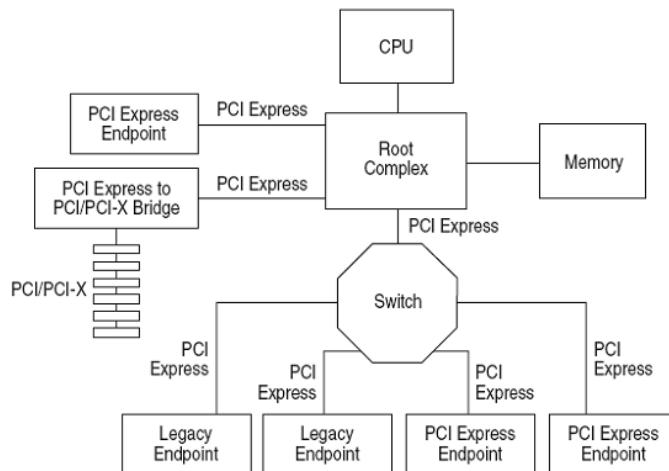


Figure 7. PCIe Topology

Root Complex- denotes the root of an I/O hierarchy that connects the CPU/memory subsystem to the I/O

Endpoints- classified as either legacy, PCI Express, or Root Complex Integrated Endpoints. Must be a Function with Type 00h Configuration Space header

Switch- defined as a logical assembly of multiple virtual PCI-to-PCI Bridge devices

PCI Express to PCI/PCI-X Bridge- must be a Function with Type 01h Configuration Space header

3.3 PCIe Enumeration

When system first boots up and PCIE enumeration is not done yet, our example PCIE system looks like below (Figure 8.1).

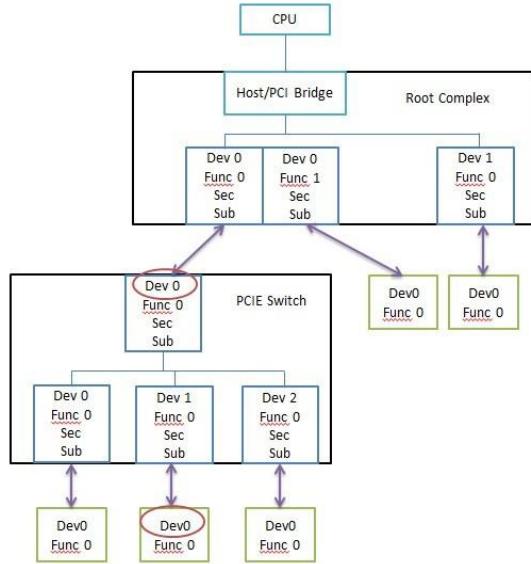


Figure 8.1 PCIe Enumeration

Some observation. We have root complex (RC) which connects to host CPU via host/PCI bridge. Note root complex also resides on host side. Inside root complex, we have two devices, dev 0 and dev 1. Dev 0 is a multi-function device and consists of function 0 and function 1. Dev 1 is single function device. RC dev 0 and dev 1 are not end point devices and they are of bridge type, also called PCIE to PCIE (P2P) virtual bridge.

There is a PCIE switch in the system which is connected to RC dev0 func0 via a PCIE link. This PCIE switch has four P2P bridges and three downstream bridges are connected to a PCIE end point respectively.

Note

1. As highlighted in diagram, all devices attached to downstream side of a PCIE link must be device 0.
2. PCIE link is a point to point connection and P2P bridge, either in RC or in switch, is needed to connect multiple PCIE devices. However, the connection among P2P bridges, either inside RC or inside switch, is multi-drop and it is NOT a PCIE link.

What PCIE enumeration does is to assign PCIE bus number to each PCIE link and P2P bridge connection, and properly fill up secondary bus number and subordinate bus number inside each P2P bridge so that software running in CPU can uniquely identify each PCIE device and P2P bridges combined can properly route the transaction to the correct target PCIE device.

Now let's get started. Software first assigns bus 0 to RC connection. Then it reads bus 0 dev0 configuration space and figures this device is a bridge since configuration header is type 1. Then it assigns bus 1 to this device's downstream PCIE link and updates secondary (sec) bus number to be 1 and subordinate (sub) bus number to be 255. Sec bus number specifies the minimum bus number in the tree under this bridge and sub bus number is the max bus number. Software uses

255 for now since it hasn't gone through the tree yet so it doesn't know how many devices down there.

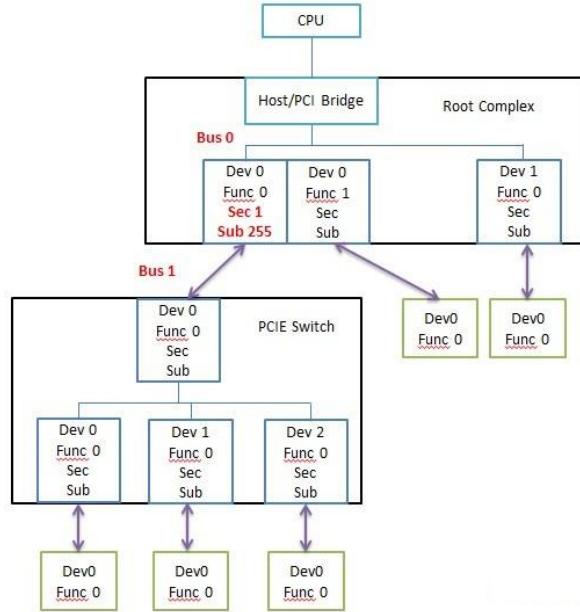


Figure 8.2 PCIe Enumeration

Software reads bus 1 dev0 and figures it is a P2P bridge and then assigns bus 2 to its downstream link (note it is not a PCIE link but a connection among P2P bridges) and updates bus 1 dev0 sec bus number to 2 and sub bus number to 255.

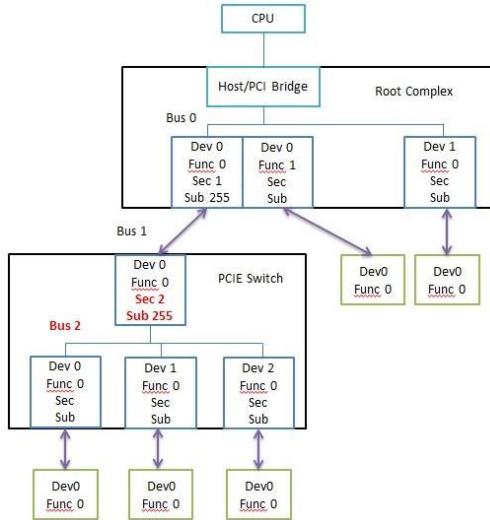


Figure 8.3 PCIe Enumeration

Software reads bus 2 dev 0 and figures it is a P2P bridge. It does the same thing as above, assign bus 3 and updates sec bus num to 3 and sub bus num to 255.

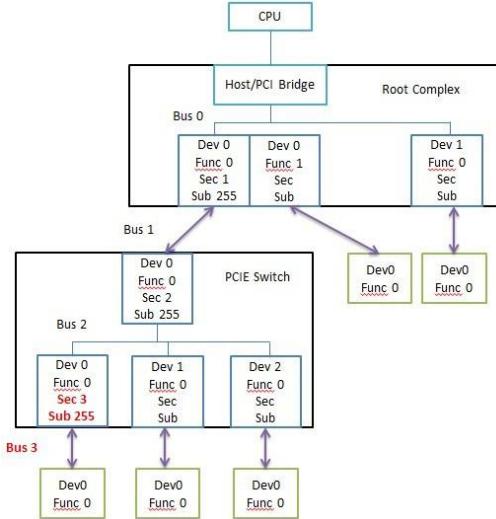


Figure 8.4 PCIe Enumeration

Software reads bus 3 dev 0 and figures it is a PCIE end point. So it goes back to update bus 2 dev0 sub bus number from 255 to 3. Then it checks if bus 2 has dev 1. Find it. Assign bus 4 and update bus 2 dev 1 sec bus num to 4 and sub bus num to 255.

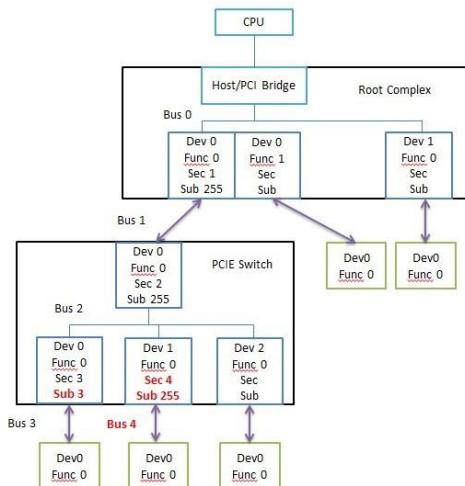


Figure 8.5 PCIe Enumeration

Software reads bus 4 dev 0 and figures it is a PCIE end point. So it goes back to update bus 2 dev1 sub bus number from 255 to 4. Then it checks if bus 2 has dev 2. Find it. Assign bus 5 and update bus 2 dev 2 sec bus num to 5 and sub bus num to 255. It then reads and figures bus 5 dev 0 is end point so it goes back to update bus 2 dev 2 sub bus num to 5.

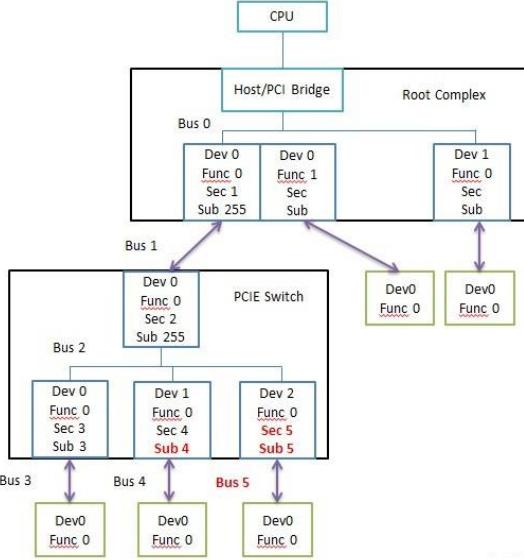


Figure 8.6 PCIe Enumeration

Software further checks if bus 2 has dev 3. It goes ahead and read bus dev 3 configuration space. Since there is no dev 3, this transaction eventually times out causing a master abort inside RC and RC returns data of all one's to software. So software knows no bus 2 dev 3. It then goes back to update bus 1 dev 0 sub bus number from 255 to 5 and further goes back to update bus 0 dev 0 func 0 sub bus number from 255 to 5.

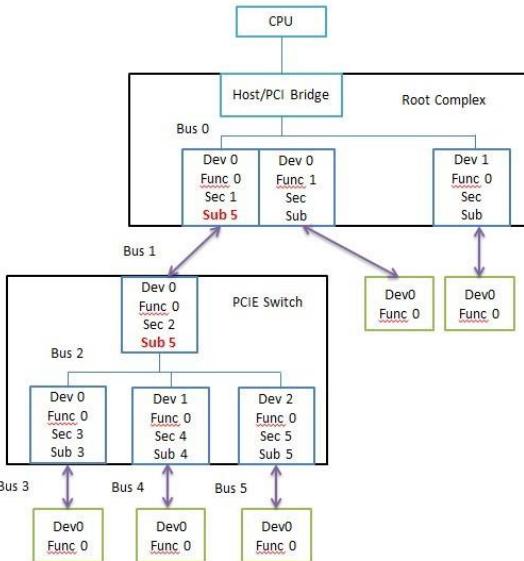


Figure 8.7 PCIe Enumeration

Software already know bus 0 dev 0 is a multi-function device in step 1. So it moves to bus 0 dev0 func 1. Assign bus 6 to its downstream link. Reads bus 6 dev 0 and figures it is a end point. Eventually bus 0 dev0 func1 sec bus num is assigned to 6 and sub bus num also to 6.

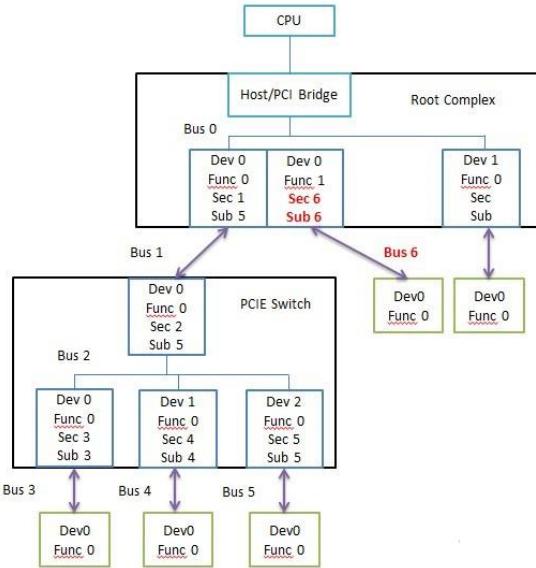


Figure 8.8 PCIe Enumeration

Software checks and sees bus 0 dev 1. Assign bus 7 and update both sec bus number and sub bus number to be 7. Software further checks if there is bus 0 dev 2. No such device. Transaction times out. That's it. Software gets the whole system figured out and properly configured. PCIe enumeration is done.

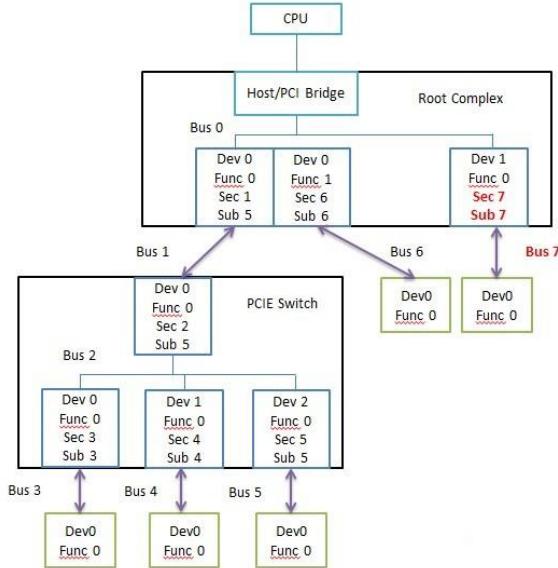


Figure 8.9 PCIe Enumeration

4.0 PCI Programming Model

4.1 PCI Address Spaces

PCI implements three address spaces:

- 1) PCI Configuration Space (up to 256 Bytes)
 - Required/standard. Defined in the specifications. Every PCI device has a configuration space.
- 2) PCI Memory-mapped space
 - Optional. Dependent on whether the device manufacturer needs to map system memory to the PCI device
- 3) PCI I/O-mapped space
 - Optional. Same as PCI Memory Space

4.2 PCI Express (PCIe) Address Spaces

PCIe implements four address spaces:

- 1) PCIe Configuration Space (up to 4KBytes)
 - Required/standard. Defined in the specifications. Every PCIe device has its configuration space mapped to memory.
 - Also provides the first 256 bytes of compatible PCI (memory-mapped and via port IO for backwards compatibility)
- 2) PCIe Memory-mapped space
 - Optional. Dependent on whether the device manufacturer needs to map system memory to the PCI device
- 3) PCIe I/O-mapped space
 - Optional. Same as PCI Memory Space
- 4) PCIe Message Space
 - For low-level protocol messaging/interrupts. We don't get into this in this class

4.3 PCI Configuration Space

Configuration Space is intended for configuration, initialization, and catastrophic error handling functions. Its use should be restricted to initialization software and error handling software. All operational software must continue to use I/O and/or Memory Space accesses to manipulate device registers.

It defines the programming model and usage rules for the configuration register space organization. A device's Configuration Space must be accessible at all times, not just during system boot. It imposes a specific record structure or template on the 256-byte space which is divided into **predefined header region** and **device dependent region**

predefined header region

Consists of fields that uniquely identifies the device and allow the device to be generically controlled. It is divided into 2 parts, first 16 bytes are defined same for all devices and the remaining bytes are dependent on the base function of the device

Device dependent region

Contains device specific information

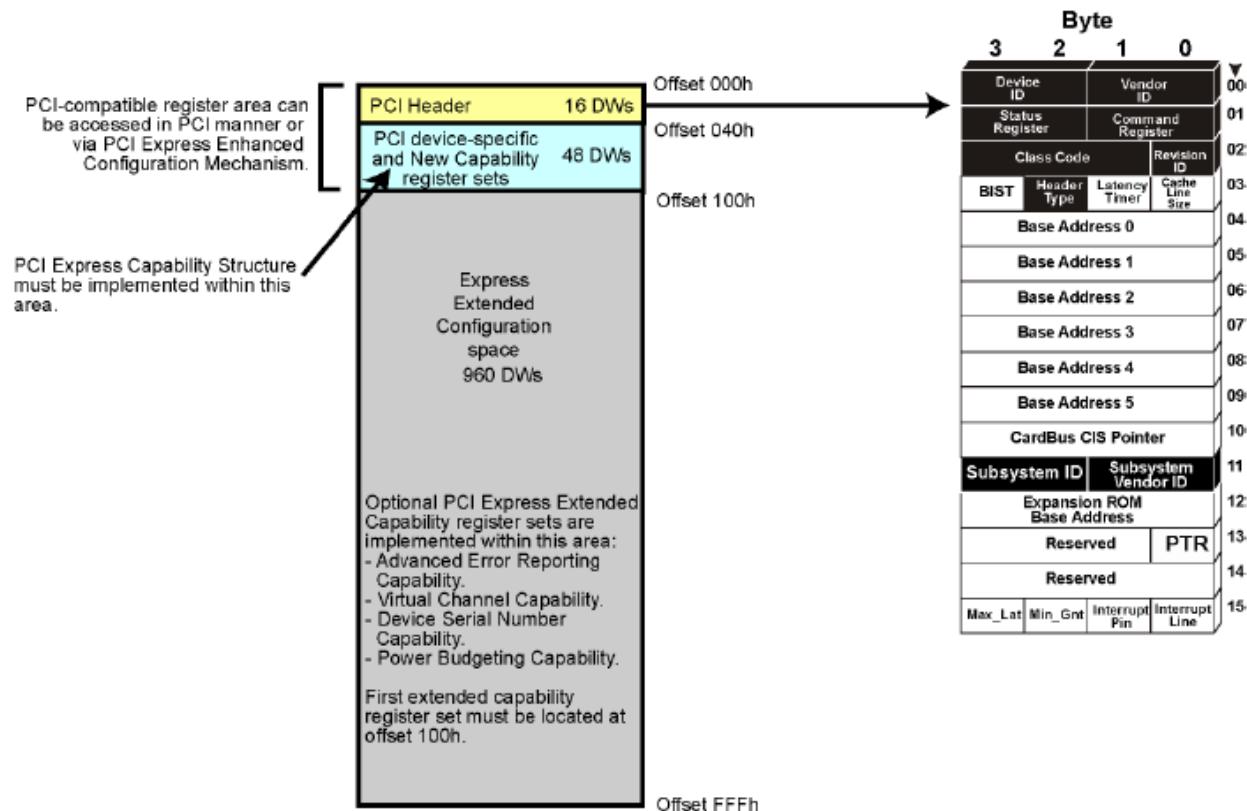


Figure 9. PCIe Address Space

4.4 Accessing Configuration Space

There are 2 ways to access the compatible PCI configuration space registers (0 to 255)

- Port IO or Memory-mapped IO

And 1 way to access the extended configuration space offered by PCI Express (255 to 4KB)

- Memory-mapped IO

Accessing PCI via Port I/O performed in a couple situations:

- When in Real Mode, when accesses to 32-bit memory space is limited. Real Mode may be reentered even after the system has transitioned to Protected mode which was observed in Legacy BIOS and not in UEFI generally
- Before PCIEXBAR has been configured

4.4.1 Compatible PCI Configuration Space

This refers to the software generation of PCI configuration transactions those generated by the CPU/BIOS. Compatible PCI provides 256 bytes of Configuration address space to the CPU/BIOS, who in turn programs the registers contained therein to configure the device and system parameters. Compatible PCI is configured using two 32-bit port I/O address/data pair (**CONFIG_ADDRESS**, **CONFIG_DATA**) to generate configuration transactions

4.4.1.1 I/O Port CONFIG_ADDRESS (CF8h)

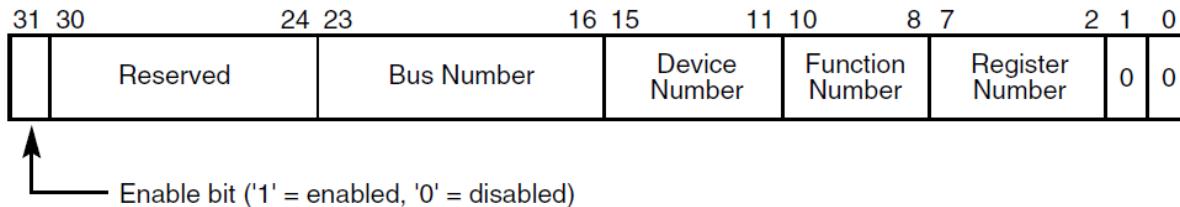


Figure 10. CONFIG_ADDRESS (CF8h)

- Bit 31 when set, all reads and writes to CONFIG_DATA are PCI Configuration transactions
- Bits 30:24 are read-only and must return 0 when read
- Bits 23:16 select a specific Bus in the system (this 8 bit field decides that a PCI system can support up to 256 buses)
- Bits 15:11 specify a Device on the given Bus (this 5 bit field decides that up to 32 devices can be connected to a PCI bus)
- Bits 10:8 Specify the function of a device (this 3 bit field decides that a device can have up to 8 functions)
- Bits 7:0 Select an offset within the Configuration Space (256 bytes max, DWORD-aligned as bits 1:0 are hard-coded 0)
- Addresses are often given in B/D/F, Offset notation

4.4.1.2 I/O Port CONFIG_DATA (CFCh)

CONFIG_DATA can be accessed in DWORD, WORD, or BYTE configurations. Reads and Writes to CONFIG_DATA with Bit 31 in CONFIG_ADDRESS set/enabled results in a PCI Configuration transaction to the device specified in CONFIG_ADDRESS. PCI spec says that if Bit 31 is not enabled, then the transaction is forwarded out as Port I/O

4.4.2 Compatible PCI Configuration Space Header

Implemented in of Configuration Space Header in PCIe is same as that in PCI giving it backward compatibility as well. In general supports three header types (0-2)

- Type 0 represents General Endpoint Device
- Type 1 represent PCI-to-PCI Bridge
- Type 2 represent CardBus Bridge which is not used at present

Following Configuration Space header represent an Endpoint Device (Type 00h). It is divided into 2 parts:

- 1) First 16 bytes (0-F) are standard and defined the same for all devices.
- 2) The remaining header bytes are optional per the vendor, depending on what function the device performs

These registers are used for Device identification, Control, Status and Miscellaneous operations.

31	16	15	0
	Device ID	Vendor ID	00h
	Status	Command	04h
	Class Code	Revision ID	08h
BIST	Header Type	Latency Timer	Cacheline Size
			0Ch
			10h
			14h
	Base Address Registers		18h
			1Ch
			20h
			24h
	Cardbus CIS Pointer		28h
	Subsystem ID	Subsystem Vendor ID	2Ch
	Expansion ROM Base Address		30h
	Reserved	Capabilities Pointer	34h
	Reserved		38h
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line
			3Ch

Figure 11. Type 0 Configuration Space Header

Vendor ID

This field identifies the manufacturer of the device. Valid vendor identifiers are allocated by the PCI SIG to ensure uniqueness. 0 FFFFh is an invalid value for Vendor ID.

Device ID

This field identifies the particular device. This identifier is allocated by the vendor.

Revision ID

This register specifies a device specific revision identifier. The value is chosen by the vendor. Zero is an acceptable value. This field should be viewed as a vendor defined extension to the Device ID.

Header Type

This byte identifies the layout of the second part of the predefined header (beginning at byte 10h in Configuration Space) and also whether or not the device contains multiple functions. Bit 7 in this register is used to identify a multi-function device. If the bit is 0, then the device is single function. If the bit is 1, then the device has multiple functions. Bits 6 through 0 identify the layout of the second part of the predefined header. The encoding 00h specifies the layout shown in Figure 6-1. The encoding 01h is defined for PCI-to-PCI bridges and is defined in the document PCI to PCI Bridge Architecture Specification. The encoding 02h is defined for a Card Bus bridge and is documented in the PC Card Standard. All other encodings are reserved.

Class Code

The Class Code register is read-only and is used to identify the generic function of the device and, in some cases, a specific register level programming interface. The register is broken into three-byte size fields. The upper byte (at offset 0Bh) is a base class code which broadly classifies the type of function the device performs. The middle byte (at offset 0Ah) is a sub-class code which identifies more specifically the function of the device. The lower byte (at offset 09h) identifies a specific register-level programming interface (if any) so that device independent software can interact with the device. Encodings for base class, sub-class, and programming interface are provided in Appendix D. All unspecified encodings are reserved.

Command Register

The Command register provides coarse control over a device's ability to generate and respond to PCI cycles. When a 0 is written to this register, the device is logically disconnected from the PCI bus for all accesses except configuration accesses. All devices are required to support this base level of functionality. Individual bits in the Command register may or may not be implemented depending on a device's functionality. For instance, devices that do not implement an I/O Space will not implement a writable element at bit location 0 of the Command register.

Status Register

The Status register is used to record status information for PCI bus related events. Devices may not need to implement all bits, depending on device functionality. For instance, a device that acts as a target, but will never signal Target-Abort, would not implement bit 11. Reserved bits should be read-only and return zero when read.

CacheLine Size

This read/write register specifies the system cacheline size in units of DWORDs. This register must be implemented by master devices that can generate the Memory Write and Invalidate command.

Latency Timer

This register specifies, in units of PCI bus clocks, the value of the Latency Timer for this PCI bus master

Built-in Self-Test (BIST)

This optional register is used for control and status of BIST. Devices that do not support BIST must always return a value of 0

CardBus CIS Pointer

This optional register is used by those devices that want to share silicon between CardBus and PCI. The field is used to point to the Card Information Structure (CIS) for the CardBus card.

Interrupt Line

The Interrupt Line register is an eight-bit register used to communicate interrupt line routing information. The register is read/write and must be implemented by any device (or device function) that uses an interrupt pin. POST software will write the routing information into this register as it initializes and configures the system

Interrupt Pin

The Interrupt Pin register tells which interrupt pin the device (or device function) uses. A value of 1 corresponds to INTA#. A value of 2 corresponds to INTB#. A value of 3 corresponds to INTC#. A value of 4 corresponds to INTD#.

MIN_GNT and MAX_LAT

These read-only byte registers are used to specify the device's desired settings for Latency Timer values. For both registers, the value specifies a period of time in units of $\frac{1}{4}$ microsecond. Values of 0 indicate that the device has no major requirements for the settings of Latency Timers

Subsystem Vendor ID and Subsystem ID

These registers are used to uniquely identify the add-in card or subsystem where the PCI device resides. They provide a mechanism for add-in card vendors to distinguish their addin cards from one another even though the add-in cards may have the same PCI controller on them (and, therefore, the same Vendor ID and Device ID).

Capabilities Pointer

This optional register is used to point to a linked list of new capabilities implemented by this device. This register is only valid if the "Capabilities List" bit in the Status Register is set. If implemented, the bottom two bits are reserved and should be set to 00b. Software should mask these bits off before using this register as a pointer in Configuration Space to the first entry of a linked list of new capabilities.

Base Address Register

Base Address Registers point to the location in the system address space where the PCI device will be located. The device RAM, etc. (anything really, per the vendor). BARs are R/W and the BIOS programs them to set up the Memory Map PCI Configuration Registers provides space for up to 6 BARs (bytes 10h through 27h) BAR[0-5]. Each BAR is 32-bits wide to support 32-bit address space locations. Concatenating two 32-bit BARs provides 64-bit addressing capability

For a Bridge device (Type 01h), initial 16 bytes are same, apart from early mentioned registers, following few registers are given priority.

31	0
Device ID	Vendor ID
Status	Command
Class Code	Revision ID
BIST	Header Type
Primary Latency Timer	Cache Line Size
Base Address Register 0	
Base Address Register 1	
Secondary Latency Timer	Subordinate Bus Number
Secondary Status	I/O Limit
Memory Limit	Memory Base
Prefetchable Memory Limit	Prefetchable Memory Base
Prefetchable Base Upper 32 Bits	
Prefetchable Limit Upper 32 Bits	
I/O Limit Upper 16 Bits	I/O Base Upper 16 Bits
Reserved	
Expansion ROM Base Address	
Bridge Control	Interrupt Pin
	Interrupt Line

Figure 12. Type 1 Configuration Space Header

Primary Bus Number- denotes the bus in which device sits

Secondary Bus Number- denotes the bus in which is spawned from the device

Subordinate Bus Number - denotes the maximum depth of the bus in the hierarchy

Chapter-6

Universal Serial Bus (USB)

USB 2.0

1.0 Introduction

USB is an interface that connects a device to a computer. With this connection, the computer sends or retrieves data from the device. USB gives developers a standard interface to use in many different types of applications. A USB device is easy to connect and use because of a systematic design process. This application note is intended to help make that process simpler.

The following concepts are covered in this application note:

- USB History
- USB Architecture
- USB Physical Interface
- USB Speeds
- USB Power
- USB Endpoints
- USB Communication Protocol
- USB Descriptors
- USB Class Devices
- USB Enumeration and Configuration Process
- USB Compliance and Windows Logo Testing

Note that USB 3.0 is not discussed in this application note. See the application notes listed in the Related Resources section for USB 3.0 related application notes. This application note also does not provide any code example for USB 2.0 devices. Refer to the code examples listed in the Related Resources section for links to code examples for each of the product families.

2.0 USB History

USB is an industry standard developed for the connection of electronic peripherals such as keyboard, mice, modems, and hard drives to a computer. This standard was developed in order to replace larger and slower connections such as serial and parallel ports. The standard was

developed through a joint effort, starting in 1994, between Compaq, DEC, IBM, Intel, Microsoft, NEC, and Nortel. The goals were to develop a single interface that could be used across multiple devices, eliminate the many different connectors currently available at the time, and increase the data throughput of electronic devices.

Over the years, the USB specification has undergone multiple revisions. It all started with USB 1.0, which was finalized in January of 1996. The original specification only included support for two speeds: Low-Speed (LS), which supported 1.5 Mb/s and Full-Speed (FS), which supported 12 Mb/s. While Low-Speed was slower than Full-Speed, it was less susceptible to electromagnetic interference (EMI), which made it attractive to many USB device developers because lower cost components could be used. In 1998, USB 1.1 was developed and added some clarifications and improvements to the USB 1.0 specification. It was not until the release of USB 2.0 in April 2000 that the next major change occurred. This revision added a new speed, High-Speed (HS), to the specification making it capable of 480 Mb/s. This specification revision is backward-compatible with USB 1.1 and 1.0. That same backward compatibility was maintained when USB 3.0 was announced in November 2008, providing speeds up to 5 Gb/s. With USB 3.0 came a new physical connector as well. More recently, plans for USB 3.1 were announced by USB-IF, which will increase speeds up to 10 Gb/s. USB is currently regulated by the USB Implementers Forum (USB-IF), which is a nonprofit organization that maintains the USB documents and compliance programs.

3.0 USB Overview

USB systems consist of a host, which is typically a personal computer (PC) and multiple peripheral devices connected through a tiered-star topology. This topology may also include hubs that allow additional connection points to the USB system. The host itself contains two components, the host controller and the root hub. The host controller is a hardware chipset with a software driver layer that is responsible for these tasks:

- Detect attachment and removal of USB devices
- Manage data flow between host and devices
- Provide and manage power to attached devices
- Monitor activity on the bus

At least one host controller is present in a host and it is possible to have more than one host controller. Each controller allows connection of up to 127 devices with the use of external USB hubs. The root hub is an internal hub that connects to the host controller(s) and acts as the first interface layer to the USB in a system. Currently on your PC, there are multiple USB ports. These

ports are part of the root hub in your PC. For simplicity, look at the root hub and host controller from the abstract view of a —black box|| that we call the host.

USB devices consist of one or more device functions, such as a mouse, keyboard, or audio device for example. Each device is given an address by the host, which is used in the data communication between that device and the host. USB device communication is done through pipes. These pipes are a connection pathway from the host controller to an addressable buffer called an endpoint. An endpoint stores received data from the host and holds the data that is waiting to transmit to the host. A USB device can have multiple endpoints and each endpoint has a pipe associated with it. This is shown in Figure 1.

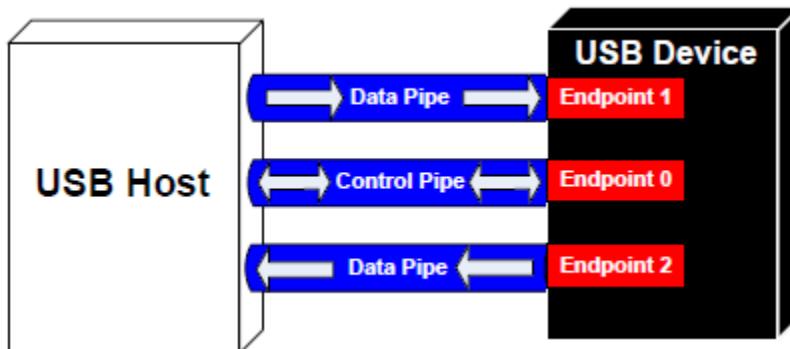


Figure 1. USB Pipe Model

There are two types of pipes in a USB system, control pipes and data pipes. The USB specification defines four different data transfer types. Which pipe is used depends on the data transfer type.

Control Transfers Used for sending commands to the device, make inquiries, and configure the device. This transfer uses the control pipe.

Interrupt Transfers Used for sending small amounts of burst data that requires a guaranteed minimum latency transfer uses a data pipe.

Bulk Transfers Used for large data transfers that use all available USB bandwidth with no guarantee on transfer speed or latency. This transfer uses a data pipe.

Isochronous Transfers Used for data that requires a guaranteed data delivery rate. Isochronous transfers are capable of this guaranteed delivery time due to their guaranteed latency, guaranteed bus bandwidth, and lack of error correction. Without the error correction, there is no halt in transmission while packets containing errors are resent. This transfer uses a data pipe.

Every device has a control pipe and it is through this pipe that control transfers to send and receive messages from the device are performed. Optionally, a device may have data pipes for

transferring data through interrupt, bulk, or isochronous transfers. The control pipe is the only bidirectional pipe in the USB system. All the data pipes are unidirectional.

Each endpoint is accessed with a device address (assigned by the host) and an endpoint number (assigned by the device). When information is sent to the device, the device address and endpoint number are identified with a token packet (discussed later in Communication Protocol section). The host initiates this token packet before a data transaction.

When a USB device is first connected to a host, the USB enumeration process is initiated. Enumeration is the process of exchanging information between the device and the host that includes learning about the device. Additionally, enumeration includes assigning an address to the device, reading descriptors (which are data structures that provide information about the device), and assigning and loading a device driver. This entire process can occur in seconds. For more information, see the USB Enumeration and Configuration section. When this process is complete, the device is ready to transfer data to the host. The flow chart of the general enumeration process is shown in Figure 2. Two files are affiliated with enumeration and the loading of a driver. They exist on the host side.

.INF – A text file that contains all the information necessary to install a device, such as driver names and locations, Windows registry information, and driver version information.

.SYS – The driver needed to communicate effectively with the USB device.

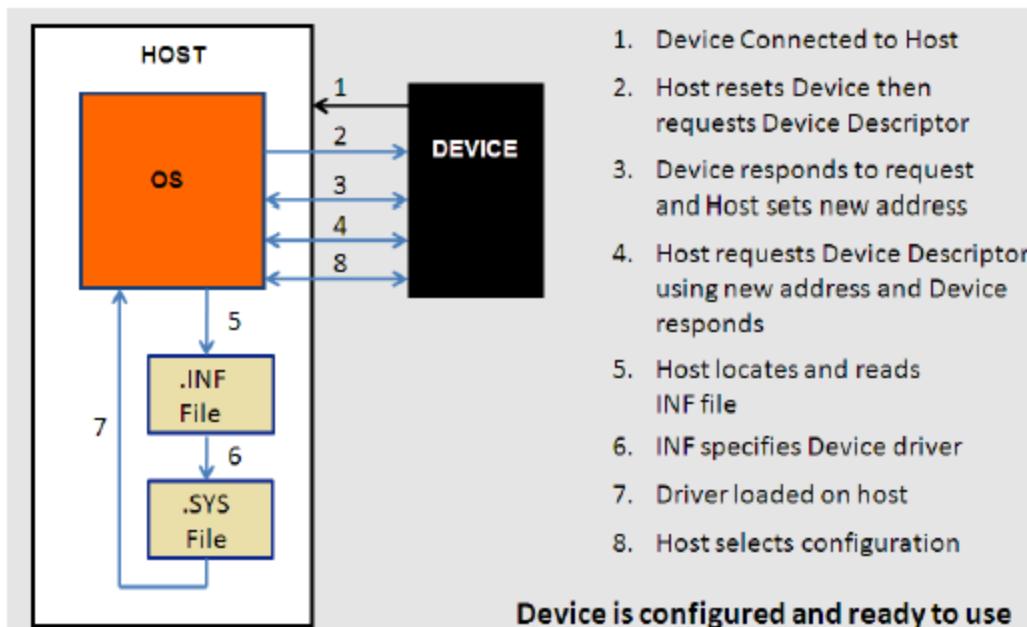


Figure 2. Sequence of Enumeration Events

After a device is enumerated, the host directs all traffic flow to the devices on the bus. Because of this, no device can transfer data without a request from the host controller.

4.0 USB Architecture

Only one host can exist in the system and communication with devices is from the host's perspective. A host is an —upstream|| component, while a device is a —downstream|| component; Figure 3 shows a representation of this. Data moved from the host to the peripheral is an OUT transfer. Data moved to the host from the peripheral is an IN transfer. The host, specifically the host controller, controls all traffic and issues commands to devices. There are three common types of USB host controllers:

Universal Host Controller Interface (UHCI): Produced by Intel for USB 1.0 and USB 1.1. Using UHCI requires a license from Intel. This controller supports both Low-Speed and Full-Speed.

Open Host Controller Interface (OHCI): Produced for USB 1.0 and 1.1 by Compaq, Microsoft, and National Semiconductor. Supports Low-Speed and Full-Speed and tends to be more efficient than UHCI by performing more functionality in hardware.

Extended Host Controller Interface (EHCI): Created for USB 2.0 after USB-IF requested that a single host controller specification be created. EHCI is used for High-Speed transactions and delegates Low-Speed and Full-Speed transactions to an OHCI or UHCI sister controller.



Figure 3. Many Peripherals Can Connect to One Host

One or more devices are attached to a host. Each device has an address and responds to host commands that are addressed to it. Devices are expected to have some form of functionality and

not simply be passive. Devices contain one upstream port. Ports are the physical USB connection point on the device.

A hub is a specialized device that allows the host to communicate with multiple peripheral devices on the bus. Unlike USB peripheral devices, such as a mouse that has actual functionality, a hub device is transparent and is intended to act as a pass-through. A hub also acts as a channel between the host and the device. Hubs have additional attachment points to allow the connection of multiple devices to a single host. A hub repeats traffic to and from downstream devices through one upstream port and up to seven downstream ports. The hub, however, does not have any host capabilities.

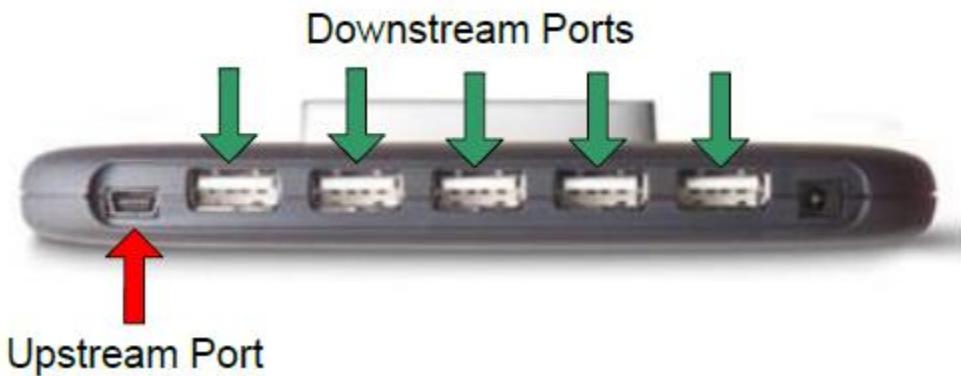


Figure 4. Hub Connections

As mentioned earlier, up to 127 devices can be connected to the host controller with the use of hubs. This limitation is based on the USB protocol, which limits the device address to 7 bits. Additionally, a maximum of five hubs can be chained together, which is limited due to timing constraints of hub and cable propagation delays. Figure 5 shows a diagram of the USB tier system that represents the limitation of chaining hubs and devices together. You can see that with the limitation on chaining hubs together, this produces a seven tiered system.

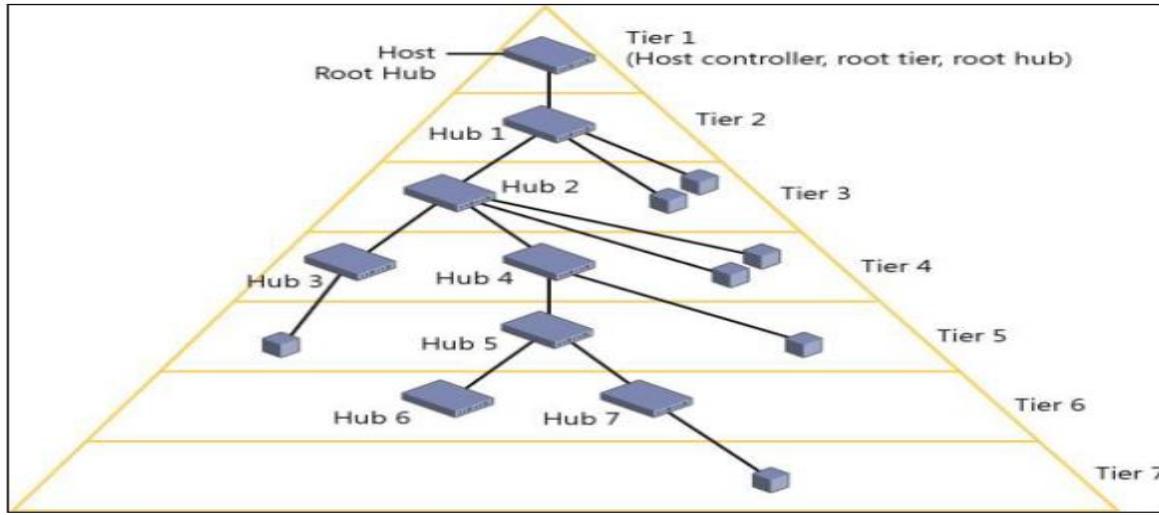


Figure 5. USB Connection Tier

Another way to look at the USB interface is to divide it into different layers, as shown in Figure 6. The Bus Interface Layer provides the physical connection, electrical signaling, and packet connectivity. This is the layer that is handled by the hardware in a device. This is accomplished with the physical interface external to the device. The Device Layer is the view the USB system software has for performing USB operations such as sending and receiving information. This is accomplished with a Serial Interface Engine, which is also internal to the device. Finally, the Function Layer is the software side of things. This is the portion of a USB device that does something with the information it received or does something to gather data to transfer to the host. Figure 6 shows this abstraction.

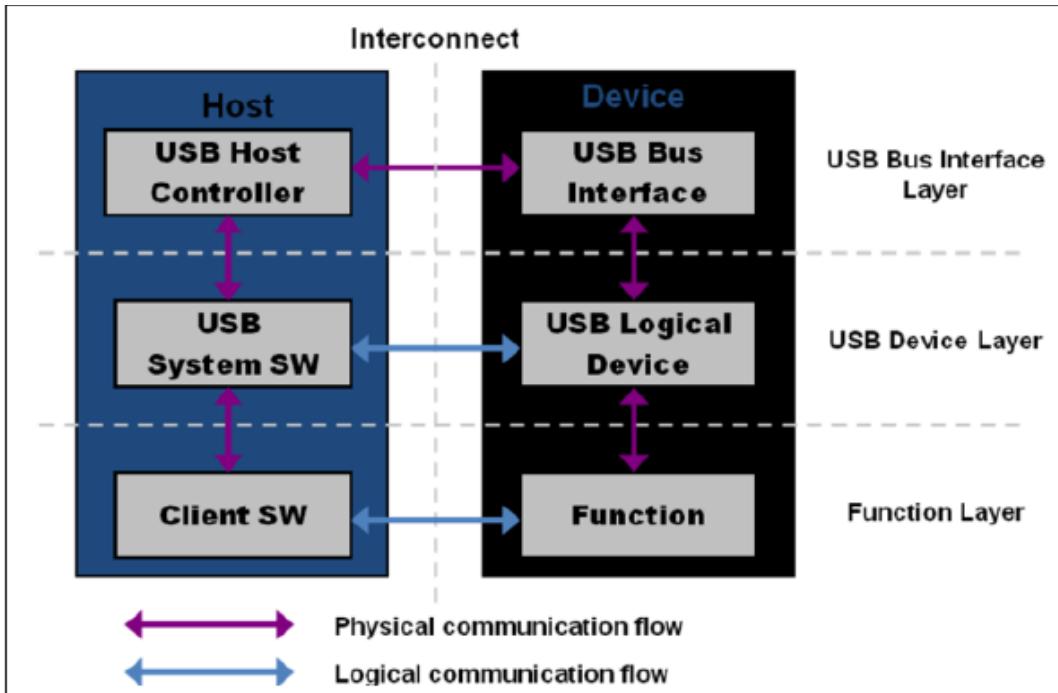


Figure 6. Interface Abstraction

5.0 Physical Interface

From a high-level overview, the physical interface of USB has two components: cables and connectors. These connectors connect devices to a host. A USB cable consists of multiple components that are protected by an insulating jacket. Underneath the jacket is an outer shield that contains a copper braid. Inside the outer shield are multiple wires: a copper drain wire, a VBUS wire (red), and a ground wire (black). An inner shield made of aluminum contains a twisted pair of data wires as seen in Figure 7. There is a D+ wire (green) and a D- wire (white).

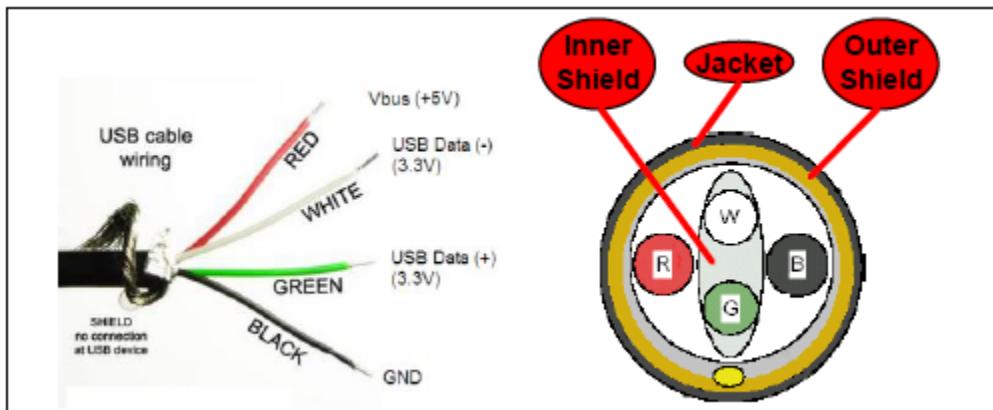


Figure 7. Inside a USB Cable

In Full-Speed and High-Speed devices, the maximum cable length is 5 meters. To increase the distance between the host and a device, you must use a series of hubs and 5-meter cables. While USB extension cables exist in the market, using them to exceed 5 meters is against the USB specification. Low-Speed devices have slightly different specifications. Their cable length is limited to 3 meters and Low-Speed cables are not required to be a twisted pair as Figure 8 shows.



Figure 8. USB Twisted Pair Data Wires

The VBUS wire gives a constant 4.40 - 5.25 V supply to all attached devices. While USB supplies up to 5.25 V to devices, the data lines (D+ and D-) function at 3.3 V. The USB interface uses a differential transmission that is nonreturn- to-zero inverted (NRZI) encoded with bit stuffing across the twisted pair. NRZI encoding is a method for mapping a binary signal for transmission over some medium, in this case, a USB cable. With this encoding scheme, a logic 1 is represented by no change in voltage level and a logic 0 is represented by a change in voltage level as Figure 9 shows. On the top is the data that will be transmitted over USB. On the bottom is the encoded NRZI data.

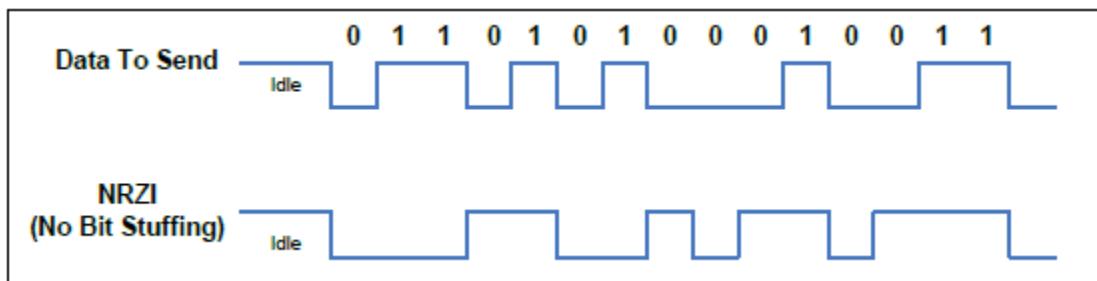


Figure 9. Data to NRZI Encoding

The bit stuffing occurs by inserting a logic 0 following seven consecutive logic 1s. The purpose of the bit stuffing is for synchronization of the USB hardware by maintaining phase-locked loop (PLL). If there are too many logic 1s in the data, then there may not be enough transitions in the NRZI encoded stream to synchronize from. The receiver on the USB hardware automatically detects this extra bit and disregards it. This extra bit stuffing contributes to the extra overhead on the USB. Figure 10 shows an example of NRZI data with bit stuffing. Notice in the —Data to Send|| stream there are eight 1s. In the encoded data, after the sixth logic 1, a logic 0 is inserted. The seventh and eighth logic 1 then follow this logic 0.

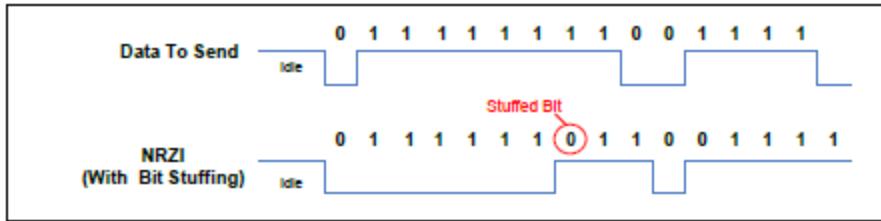


Figure 10. Data to NRZI Encoding with Bit Stuffing

The hardware in USB devices will handle all the encoding and bit stuffing upon receiving any data and before transmitting any data. The reason for using the differential D+ and D- signal is for rejecting common-mode noise. If noise becomes coupled into the cable, it will normally be present on all wires in the cable. With the use of a differential amplifier in the USB hardware internal to the host and device, the common-mode noise can be rejected as shown in Figure 11.

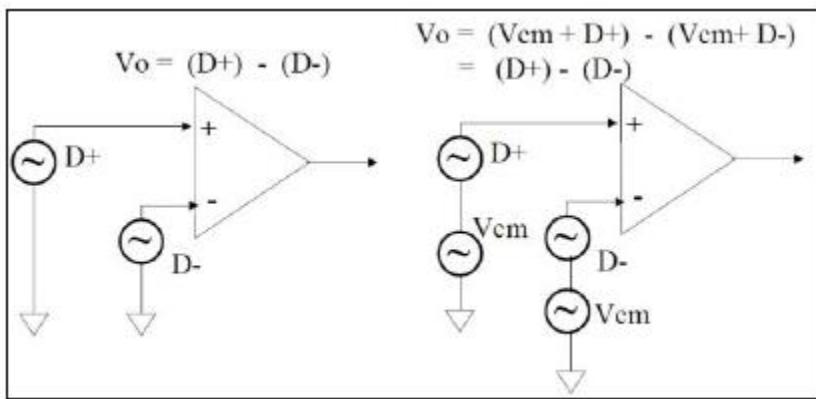


Figure 11. USB Input Differential Amplifier Buffer

USB communication occurs through many different signaling states on the D+ and D- lines. Some of these states transmit the data while others are used as specific signaling conditions. These states are described below with a quick reference list located in Table 1.

Differential 0 and Differential 1: These two states are used for general data communication across USB. Differential 1 is when the D+ line is high and the D- line is low. Differential 0 is when the D+ line is low and the D- line is high. An example of USB data communication is shown in Figure 12.

J-State and K-State: In addition to the differential signals, the USB specification defines two additional differential states: J-State and K-State. Their definitions depend on the device speed. On a Full-Speed and High-Speed device, a J-State is a Differential 1 and a K-State is a Differential 0. The opposite is true for a Low-Speed device.

Single Ended Zero (SE0): Condition that occurs when both D+ and D- are driven low. This condition indicates a reset, disconnect, or End of Packet.

Single Ended One (SE1): Condition that occurs when D+ and D- are both driven high. This condition does not ever occur intentionally and should not be seen occurring in a USB design.

Idle: Condition that occurs before and after a packet is sent. An Idle condition is signified by one of the data lines being low and the other line being high. The definition of high versus low depends on device speed. On a Full-Speed device, an idle condition consists of D+ being high and D- being low. The opposite is true for a Low-Speed device.

Resume: Used to wake a device from a suspend state. This is done by issuing a K-State.

Start of Packet (SOP): Occurs before the start of any Low-Speed or Full-Speed packet when the D+ and D- lines transition from an idle state to a K-State.

End of Packet (EOP): Occurs at the end of any Low-Speed or Full-Speed packet. An EOP occurs when an SEO state occurs for 2 bit times (bit times are discussed later), followed by a J-State for 1 bit time.

Reset: Occurs when an SEO state lasts for 10 ms. After a SEO has occurred for at least 2.5 ms, the device may recognize the reset and begin to enter a reset state.

Keep Alive: Signal used in Low-Speed devices. Low-Speed devices lack a Start-of-Frame packet that is required to prevent suspend. They use an EOP every 1 ms to keep the device from entering suspend.

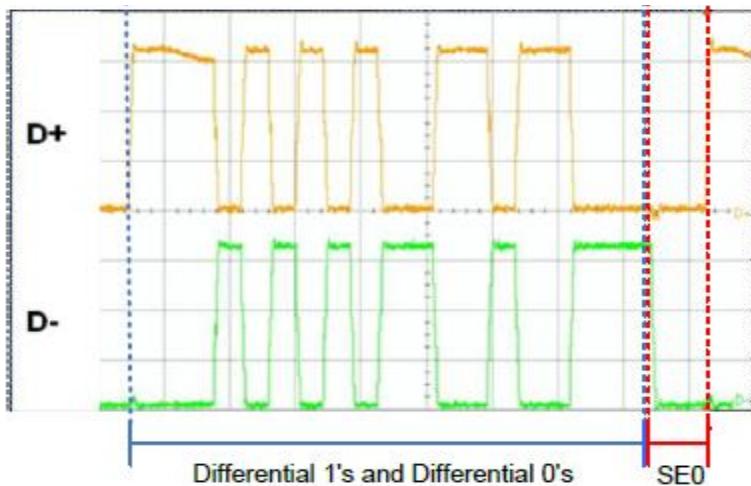


Figure 12. USB D+ and D- Communication

Table 1. USB Communication States

Bus State	Indication
Differential 1	D+ High, D- Low
Differential 0	D+ Low, D- High
Single Ended 0 (SE0)	D+ and D- Low
Single Ended 1 (SE1)	D+ and D- High
J-State: Low-Speed Full-Speed High-Speed	Differential 0 Differential 1 Differential 1
K-State: Low-Speed Full-Speed High-Speed	Differential 1 Differential 0 Differential 0
Resume State:	K-State
Start of Packet (SOP)	Data lines switch from idle to K-State.
End of Packet (EOP)	SE0 for 2 bit time followed by J-State for 1 bit time.

Figure 13 and Figure 14 show the different USB ports and connectors available. The upstream connection always uses a Type A port and connector, while the device uses Type B ports and connectors. Initially, the USB specification included only the larger Type A and Type B connectors for devices but later included the Mini and Micro connections.

These Mini and Micro connectors were initially developed for USB On-the-Go (USB OTG), which is a USB specification that allows devices that would normally act as slaves to become hosts. This is why Figure 14 shows the Mini and Micro ports as Mini-AB and Micro-AB. However, due to the smaller size of the Mini-B and Micro-B connectors compared to Type B, they were adopted in many electronics despite lacking USB OTG capabilities.

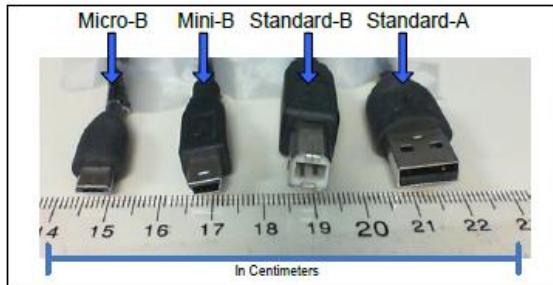


Figure 13. USB Connector Size Comparison

Type	Port Image	Connector Image
Type A		
Type B		
Mini-AB		
Mini-B		
Micro-AB		
Micro-B		

Figure 14. USB Ports and Connectors

Figure 14 shows that the Mini and Micro connectors have five pins, rather than four. The extra pin is the ID pin and identifies the host and the device in OTG applications. Since PSoC does not support USB OTG, this application note does not include details about it. The reason for having different connection types (Type A and Type B) is to prevent loopback connections on hubs. Some USB devices also contain a captive, or attached cable, with the only visible connector being the Type A. Table 2 and Table 3 show the pinout for USB connectors depending on the connector type.

Table 2. USB Standard Connector Pinout

Pin	Name	Color	Function
1	V _{BUS}	Red	+5 V
2	D-	White	Data (-)
3	D+	Green	Data (+)
4	GND	Black	Ground

Table 3. USB Mini/Micro Connector Pinout

Pin	Name	Color	Function
1	V _{BUS}	Red	+5 V
2	D-	White	Data (-)
3	D+	Green	Data (+)
4	ID	NA	Identifies Type A and Type B Plug: A plug: connected to Signal ground B plug: not connected
5	GND	Black	Ground

There are two main hardware blocks required to interface with USB: a transceiver, also known as a PHY (abbreviation for Physical Layer), and a Serial Interface Engine, also known as an SIE. The transceiver provides the hardware interface between the USB connector and the chip circuitry that controls USB communication. The SIE is the core of the USB hardware. It performs many functions such as decoding and encoding the USB data, error correction, bit stuffing, and signaling. SIEs can take many different forms. They are not regulated by the USB specification unlike transceivers. In fact, some devices incorporate SIE that are more software based to reduce cost, while other devices use more of a hardware-centric SIE.

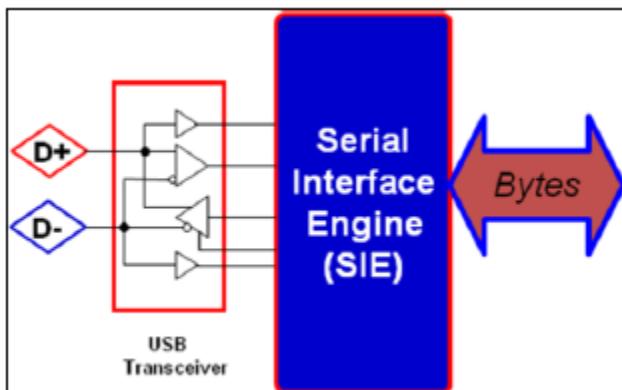


Figure 15. USB Hardware Interface at the Device

6.0 USB Speeds

Currently, the USB specification defines four speeds for a USB system: Low-Speed, Full-Speed, Hi-Speed, and SuperSpeed. Currently, Cypress only supports Full-Speed on the PSoC family of

devices and Low-Speed, Full-Speed, High-Speed, and SuperSpeed on our various dedicated USB devices. As a result, these three speeds will be the focus of this application note.

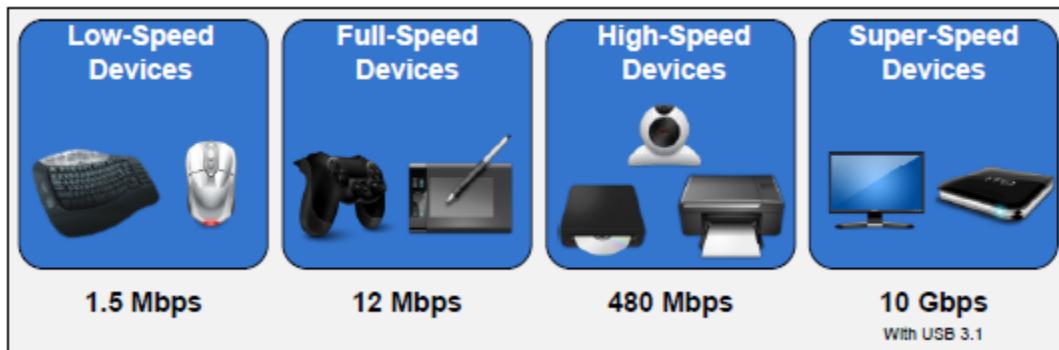


Figure 16. USB Transfer Speeds

Newer hosts can always communicate with devices of lower speed. For example, a High-Speed Host can communicate with a Low-Speed device, but a Full-Speed Host cannot communicate with a High-Speed device. Low-, Full-, and High-Speed devices are often advertised as 1.5 Mb/s, 12 Mb/s, and 480 Mb/s, respectively. However, these are bus rates and not data rates. The actual data rates are affected by bus loading, transfer type, overhead, OS, and so forth.

The actual limits of the data transfer are.

- Low-Speed devices:
- Examples: keyboards, mice, and game peripherals
- Bus Rate: 1.5 Mb/s
- Maximum Effective Data Rate: 800 B/s
- Full-Speed devices
- Examples: phones, audio devices, and compressed video
- Bus Rate: 12 Mb/s
- Maximum Effective Data Rate: 1.2 MB/s
- Hi-speed devices
- Examples: video, imaging, and storage devices
- Bus Rate: 480 Mb/s
- Maximum Effective Data Rate: 53 MB/s

When a USB device is connected to a host, the speed of the device needs to be detected. This is done with pull-up resistors on the D+ or D- line. A 1.5-kΩ pull-up on the D+ line indicates that the attached device is a Full-Speed device. A 1.5-kΩ pull-up resistor on the D- line indicates the attached device is a Low-Speed device. This can be seen in Figure 17. High-Speed devices start as Full-Speed devices, so they have a 1.5-kΩ pull-up on the D+ line. When the device is connected,

it emits a sequence of J-States and K-States during the reset phase of enumeration. If the hub supports High-Speed, then the pull-up resistor is removed.

The pull-up resistor is essential to USB enumeration. Without the pull-up resistor, USB assumes that there is nothing attached to the bus. Some devices require an external pull-up resistor on the D+/D- line. PSoC, however, implements the required pull-up resistor internal to the device, which eliminates the need for this external component.

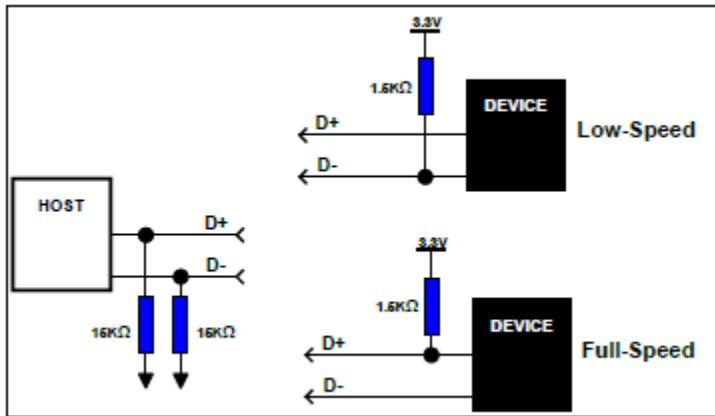


Figure 17. USB Speed Detection

One common misconception about speed on a USB device is that a device listed as USB 2.0 indicates that the device is High-Speed. All Hi-Speed devices are USB 2.0, but this is because Hi-Speed support was added with USB 2.0. The USB 2.0 specification includes Full- and Low-Speed devices as well. These speeds also have an effect on the bit timing for USB signaling, such as the End of Packet (EOP) signal. A Low-Speed and Full-Speed USB device will use a 48-MHz clock for the SIE and the other USB clocking purposes. This 48-MHz clock and the bus speed is what will determine USB bit times:

Full-Speed: $48 \text{ MHz} / 12 \text{ Mb/s} = 4 \text{ clocks per bit time.}$

Low-Speed: $48 \text{ MHz} / 1.5 \text{ Mb/s} = 32 \text{ clocks per bit time.}$

7.0 USB Power

When it comes to USB power, there are two device categories: bus powered and self-powered. Bus power is one of the many benefits of a USB design. It allows the device to sustain itself without the need for a bulky power supply either internally or externally because the device draws its power from the bus. The power available on the bus is either provided by the host or a hub. When dealing with a bus-powered device, users must consider its power consumption before the device is put into a configured state. This is the time from when the device first

connects to the bus to when the host sends the device a SET_CONFIGURATION command after the enumeration steps are complete. Before a device is configured, it must not use more than 100 mA, defined as one

unit load in the USB specification, of power for low, full, or high-speed devices. During the configuration phase, the device requests a power budget. There are two classifications for a bus-powered device: high-powered and low-powered devices. A low-powered device draws, at most, 100 mA and a high-powered device draws, at most, 500 mA. Anything over 500 mA requires the device to be self-powered.

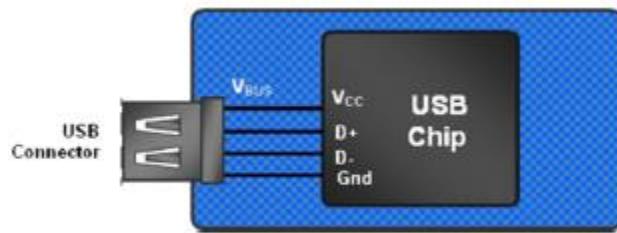


Figure 18. USB Bus-powered Device

Self-powered devices supply their own power with the use of an external power source such as a DC power adapter or a battery. A self-powered device needs a different set of considerations in a design. The USB specification requires that self-powered devices monitor their V_{BUS} line at all times. A device must remove power from its D+/Dpull- up resistor within a certain time frame of V_{BUS} being removed. The reason for this is to prevent back voltage upstream to the host or hub. Failing to meet this requirement can result in USB compliance testing failure. However, a self-powered hub does have the ability to draw up to 100 mA from the bus.

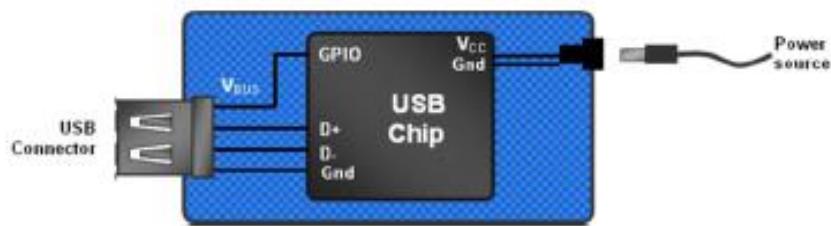


Figure 19. USB Self-Powered Device

Devices can also combine the two power modes and be both a bus-powered and self-powered device. This becomes common when a device runs off a battery. Normally, the device is self-powered, but V_{BUS} is used to charge the battery and provide power to the device while the battery is changing. Technically, this device is a self-powered device and is declared as such in

the USB descriptors, but the device requests a power budget from the host. Similar to a self-powered device, monitoring of VBUS is still required in these hybrid designs and the removal of power to the

D+/D- pull-up resistors must still occur. In this application, some type of a power management system needs to be implemented to monitor the battery voltage, charging status, and control the switching between the battery power and an external source.

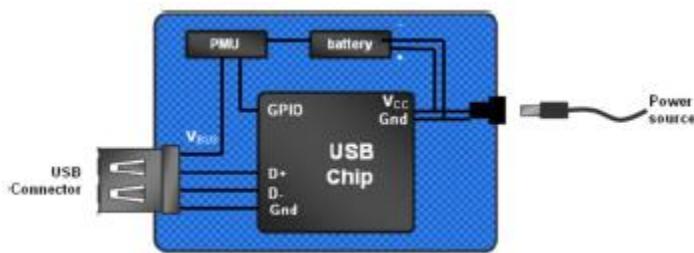


Figure 20. USB Hybrid Powered Device

Additionally, all USB devices, regardless of how the device is powered, must consider them suspend current. The device's suspend current is the current that is taken from VBUS, when the host is put in suspend mode (also called standby mode). Suspend mode occurs when there is no bus activity for 3 ms. Even when there is no active data transmission, the host uses Start of Frame (SOF) tokens to keep devices out of a suspend state. The exception to this is a Low-Speed device that does not have SOF packets. Low-Speed devices use End of Packet (EOP) transitions, as a Keep Alive Signal every 1 ms when there is no low-speed data on the bus. When the bus becomes inactive, a device must enter suspend and pull no more than 2.5 mA of current. To conform to this requirement, designers must make sure they turn off LEDs and other sinks of power before the device goes into the suspend state. A USB device leaves a suspend state once any activity is detected on the bus. If a device has remote wake-up capability, it can signal resume, then wait to see if the host acknowledges the request rather than waiting for the host to resume activity. For more information on suspend current, see Section 11.4.3 of the USB specification. Various USB states relate to USB power that a designer need to know. These states are often seen in USB documentation and apply to the enumeration of a USB device.

Attached State: Occurs when a device is attached to a host/hub, but does not give any power to the VBUS line. This is commonly seen if the hub detects an over current event. The device is still attached, but the hub removes power to it.

Powered: A device is attached to the USB and has been powered, but has not yet received a reset request.

Default: A device is attached to the USB, is powered, and has been reset by the host. At this point, the device does not have a unique device address. The device responds to address 0.

Address: A device is attached to the USB, powered, has been reset, and has had a unique address assigned to it. The device however has not yet been configured.

Configured: The device is attached to the USB, powered, has been reset, assigned a unique address, has been configured, and is not in a suspend state. At this point, bus-powered devices can draw more than 100 mA.

Suspend: As mentioned earlier, occurs when the device is attached and configured, but has not seen activity on the bus for 3 ms. The USB specification (Figure 9-1 in USB specification) has a diagram that illustrates how these power modes are related and transitioned to. See Figure 21.

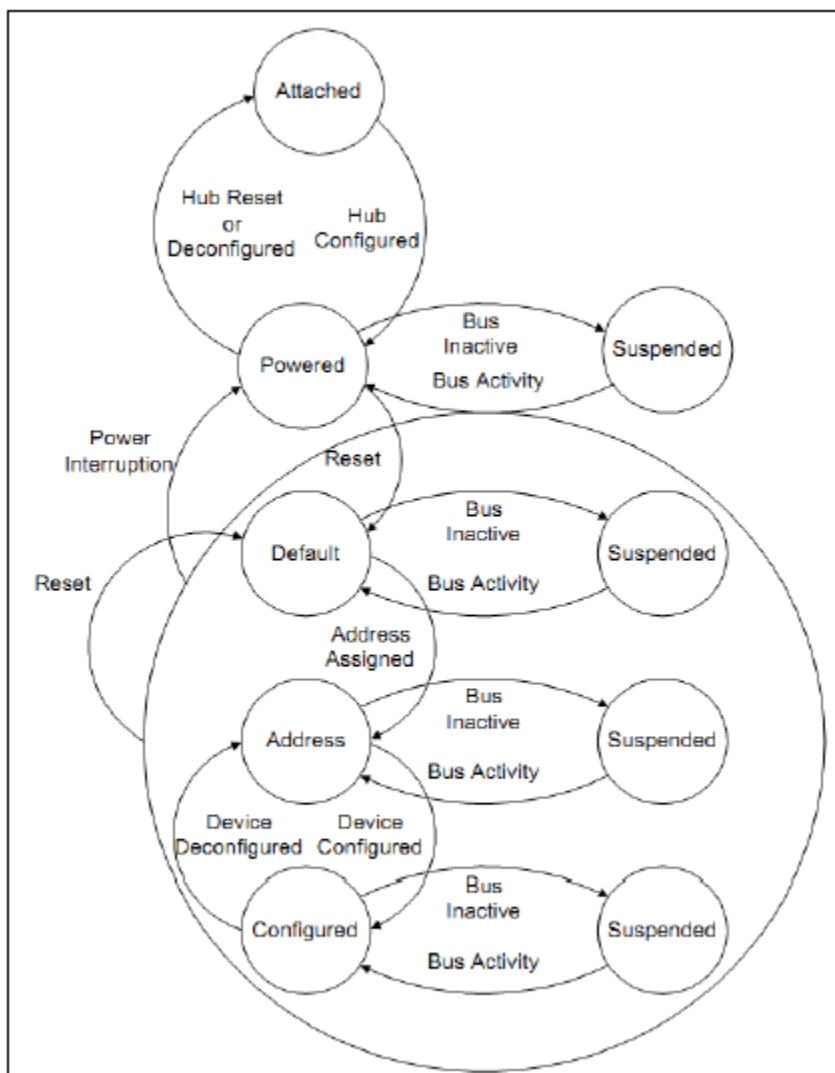


Figure 21. Device State Diagram

USB power is communicated in 2 mA units for Low-Speed, Full-Speed, and High-Speed USB devices. For example, a Full-Speed device that must have 100 mA of operational power

communicates a value of 50 during enumeration. When developing a USB design, consider how much power your device consumes from the bus. The root hub gets its power from the supply of the host PC. If the host is attached to AC power, then the USB specification requires the host provide 500 mA of power to each port on the hub. This is what causes the 500 mA limitation on bus-powered devices. If the host PC is battery powered, then it has the option of supplying either 100 mA or 500 mA to each port on the hub. When attaching a device to a hub that is bus-powered, the device must be low power and not consume more than 100 mA. A bus-powered hub has a total of 500 mA to distribute between all attached devices.

8.0 USB Endpoints

In the USB specification, a device endpoint is a uniquely addressable portion of a USB device that is the source or sink of information in a communication flow between the host and device. The USB Enumeration and Configuration section describes a step in which the device responds to the default address. This occurs before other descriptor information such as the endpoint descriptors are read by the host later in the enumeration process. During this enumeration sequence, a special set of endpoints are used for communication with the device. These special endpoints, collectively known as the Control Endpoint or Endpoint 0, are defined as Endpoint 0 IN and Endpoint 0 OUT. Even though Endpoint 0 IN and Endpoint 0 OUT are two endpoints, they look and act like one endpoint to the developer. Every USB device must support Endpoint 0. For this reason, Endpoint 0 does not require a separate descriptor.

In addition to Endpoint 0, the number of endpoints supported in any particular device is based on its design requirements. A simple design such as a mouse may need only one IN endpoint. More complex designs may need several data endpoints. The USB specification sets a limit on the number of endpoints to 16 for each direction (16 IN/16 OUT – 32 Total) for High-Speed and Full-Speed devices, which does not include the control endpoints 0 IN and 0 OUT. Low-Speed devices are limited to two endpoints. USB Class devices may set a greater limit on the number of endpoints. For example, a Low-Speed human interface device (HID) design may have no more than two data endpoints — typically one IN endpoint and one OUT endpoint. Data endpoints are bidirectional by nature. It is not until they are configured that they take on a single direction (becoming unidirectional). Endpoint 1, for example, can be either an IN or OUT endpoint. It is in the device descriptors that it will officially make Endpoint 1 an IN endpoint.

Endpoints use cyclic redundancy checks (CRCs) to detect errors in transactions. The CRC is a calculated value used for error checking. The actual calculation equation is explained in the USB specification and the handling of these calculations is taken care of by the USB hardware so that the proper response can be issued. The recipient of a transaction checks the CRC against the data.

If the two match, then the receiver issues an ACK. If the data and the CRC do not match, then no handshake is sent. This lack of a handshake tells the sender to try again. The USB specification further defines four types of endpoints and sets the maximum packet size based on both the type and the supported device speed. Developers use the endpoint descriptor to identify the type of endpoint and maximum packet size based on their design requirements. The four types of endpoints and characteristics are:

Control Endpoint – These endpoints support control transfers, which all devices must support. Control transfers send and receive device information across the bus. The advantage of control transfers is guaranteed accuracy. Errors that occur are properly detected and the data is resent. Control transfers have a 10 percent reserved bandwidth on the bus in low and Full-Speed devices (20 percent at High-Speed) and give USB system level control.

Interrupt Endpoints – These endpoints support interrupt transfers. These transfers are used on devices that must use a high reliability method to communicate a small amount of data. This is commonly used in HID designs. The name of this transfer can be misleading. It is not truly an interrupt, but uses a polling rate. However, you get a guarantee that the host checks for data at a predictable interval. Interrupt transfers give guaranteed accuracy as errors are properly detected and transactions are retried at the next transaction. Interrupt transfers have a guaranteed bandwidth of 90 percent on Low- and Full-Speed devices and 80 percent on High-Speed devices. This bandwidth is shared with isochronous endpoints. Interrupt endpoint maximum packet size is a function of device speed. High-Speed capable devices support a maximum packet size of 1024 bytes. Full-Speed capable devices support a maximum packet size of 64 bytes. Low-Speed devices support a maximum packet size of 8 bytes.

Bulk Endpoints – These endpoints support bulk transfers, which are commonly used on devices that move relatively large amounts of data at highly variable times where the transfers can use any available bandwidth space. They are the most common transfer type for USB devices. Delivery time with a bulk transfer is variable because there is no set aside bandwidth for the transfer. The delivery time varies depending on how much bandwidth on the bus is available, which makes the actual delivery time unpredictable. Bulk transfers give guaranteed accuracy because errors are

properly detected and transactions are resent. Bulk transfers are useful in moving large amounts of data that are not time sensitive. A bulk endpoint maximum packet size is a function of device speed. High-Speed capable devices support a maximum BULK packet size of 512 bytes. Full-Speed capable devices support a maximum packet size of 64-bytes. Low-Speed devices do not support bulk transfer types.

Isochronous Endpoints – These endpoints support isochronous transfers, which are continuous, real-time transfers that have a pre-negotiated bandwidth. Isochronous transfers must support streams of error tolerant data because they do not have an error recovery mechanism or

handshaking. Errors are detected through the CRC field, but not corrected. With isochronous, you get the tradeoff of guaranteed delivery versus guaranteed accuracy. Streaming music or video are examples of an application that uses isochronous endpoints because the occasional missed data

is ignored by the human ears and eyes. Isochronous transfers have a guaranteed bandwidth of 90 percent on Low and Full-Speed devices (80 percent on High-Speed devices) that is shared with interrupt endpoints. High-Speed capable devices support a maximum packet size of 1024 bytes. Full-Speed devices support a maximum packet size of 1023 bytes. Low-Speed devices do not support isochronous transfer types. There are special considerations with isochronous transfers. You generally want 3x buffering to ensure data is ready to go by having one actively transmitting buffer, another buffer loaded and ready to transfer, and a third buffer being actively loaded.

Table 4. Endpoint Transfer Type Features

Transfer Type	Control	Interrupt	Bulk	Isochronous
Typical Use	Device Initialization and Management	Mouse and Keyboard	Printer and Mass Storage	Streaming Audio and Video
Low-Speed Support	Yes	Yes	No	No
Error Correction	Yes	Yes	Yes	No
Guaranteed Delivery Rate	No	No	No	Yes
Guaranteed Bandwidth	Yes (10%)	Yes (90%) ^[1]	No	Yes (90%) ^[1]
Guaranteed Latency	No	Yes	No	Yes
Maximum Transfer Size	64 bytes	64 bytes	64 bytes	1023 bytes (FS) 1024 bytes (HS)
Maximum Transfer Speed	832 KB/s	1.216 MB/s	1.216 MB/s	1.023 MB/s

^[1]Shared bandwidth between isochronous and interrupt.

9.0 Communication Protocol

If you look at the USB communication from a time perspective, it contains a series of frames. Each frame consists of a Start of Frame (SOF) followed by one or more transactions. Each transaction is made up of a series of packets. A packet is preceded with a sync pattern and ends with an End of Packet (EOP) pattern. At a minimum, a transaction has a token packet. Depending on the transaction, there may be one or more data packets and some transactions may or may not have a handshake packet.

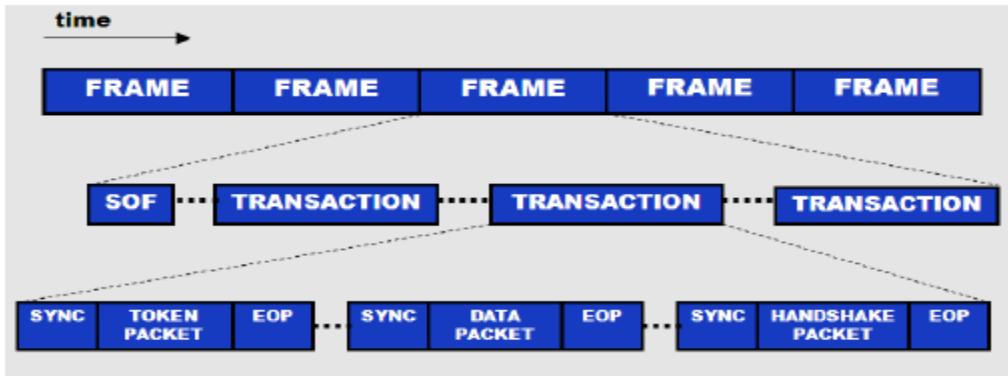


Figure 22. USB Communication from a Time Perspective

Transactions are an exchange of packets and are comprised of three different packets; a token packet, optional data packet, and a handshake packet. Transactions are placed within frames and are never split across frames (with the exception of High-Speed isochronous transfers) or interrupt the middle of another transaction. Figure 23 shows a transaction block diagram.

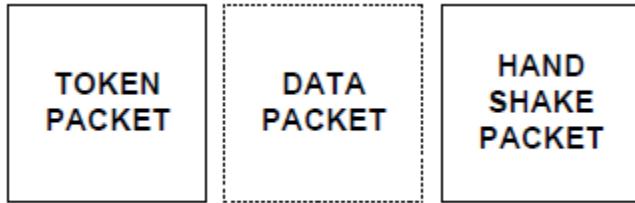


Figure 23. Transaction Block Diagram

Each packet can contain different pieces of information. What information is included depends on the packet type. The following is a list of the potential information that can be included with a packet and Figure 24 shows the potential composition of a packet. Think of Figure 24 as a packet template; information can be added and taken out as needed.

Packet ID (PID) – (8 bits: 4 type bits and 4 error check bits). These bits declare a transaction as an IN/OUT/SETUP/SOF.

Optional Device Address – (7 bits: Max of 127 devices)

Optional Endpoint Address – (4 bits: Max of 16 endpoints). The USB specification supports up to 32 endpoints. While 4 bits gives a maximum value of 16, we achieve 32 endpoints with an IN PID and an endpoint address between 1 and 16 and an OUT PID with an endpoint address between 1 and 16, giving a total of 32. Remember that this is the endpoint address, not the endpoint number.

Optional Payload Data – (0 to 1023 bytes)

Optional CRC (5 or 16 bits)

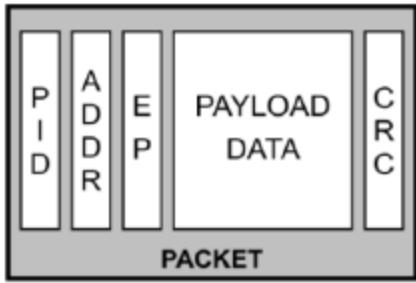


Figure 24. USB Packet Contents

9.1 Packet Types

Figure 24 can potentially represent four packet types.

- Token packets
- Initiate transaction
- Identify device involved in transaction
- Always sourced by the host
- Data packets
- Delivers payload data
- Sourced by host or device
- Handshake packets
- Acknowledge error-free data receipt
- Sourced by receiver of data
- Special packets
- Facilitates speed differentials
- Sourced by host-to-hub devices

As mentioned earlier, everything in the packet with the exception of the PID is optional. Token, data, and handshake packets have different combinations of the packet information. Which information is included is identified in the token packet, data packet, and handshake packet sections.

Token Packets: Token packets always come from the host and are used to direct traffic on the bus. The function of the token packet depends on the activity performed. IN tokens are used to request that devices send data to the host. OUT tokens are used to precede data from the host. SETUP tokens are used to precede commands from the host. SOF tokens are used to mark time frames. With an IN, OUT, and SETUP token packet, there is a 7-bit device address, 4-bit endpoint ID, and 5-bit CRC. Figure 25 shows a diagram of the various token packets.

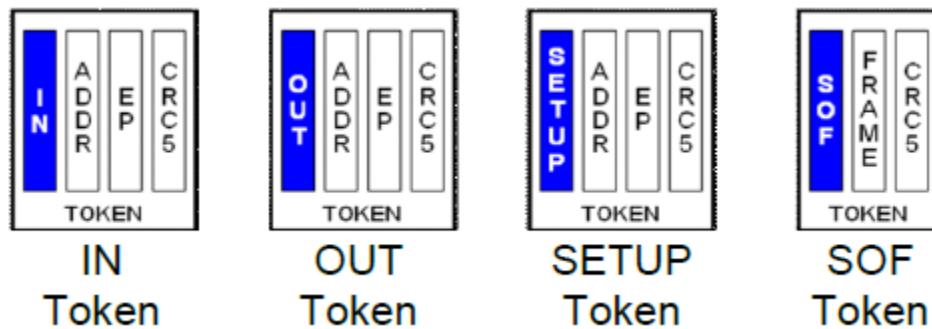


Figure 25. USB Token Packet Types

The SOF gives a way for devices to identify the beginning of a frame and synchronize with the host. They are also used to prevent a device from entering suspend mode (which it must do if 3 ms pass without an SOF). SOF packets are only seen on full- and high-speed devices and are sent every millisecond as seen in Figure 26. The SOF packet contains an 8-bit SOF PID, 11-bit frame count value (which rolls over when it reaches maximum value), and a 5-bit CRC. The CRC is the only error check used. A handshake packet does not occur for an SOF packet. High-speed communication goes a step further with micro frames. With a High-Speed device, an SOF is sent out every 125 us and frame count is only incremented every 1 ms.

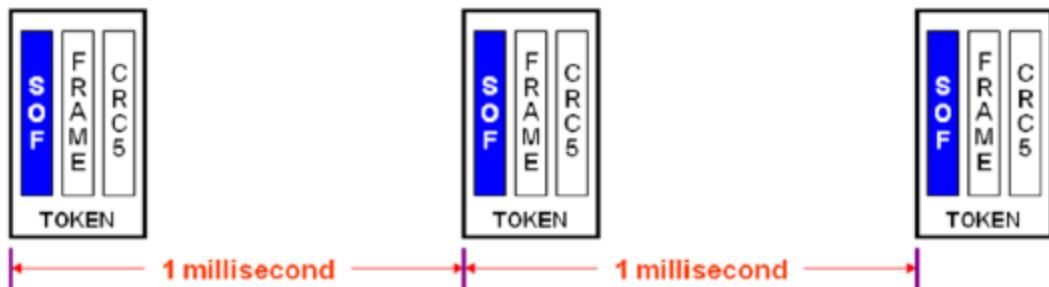


Figure 26. USB SOF in Full-Speed Device

Data Packets: Data packets follow IN, OUT, and SETUP token packets. The size of the payload data ranges from 0 to 1024 bytes depending on the transfer type. The packet ID toggles between DATA0 and DATA1 for each successful data packet transfer, and the packet closes with a 16-bit CRC. The composition of a data packet can be seen in Figure 27.

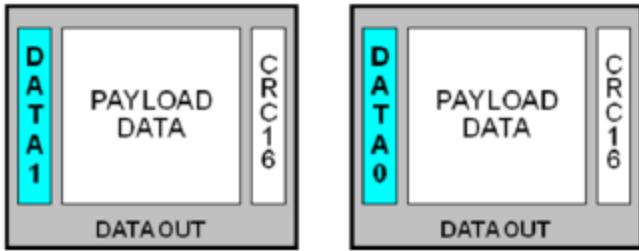


Figure 27. USB Data Packets

The data toggle is updated at the host and the device for each successful data packet transfer. One advantage to the data toggle is that it acts as an additional error detection method. If a different packet ID is received than what is expected, the device will be able to know there was an error in the transfer and it can be handled appropriately. An example where the data toggle is used is if an ACK is sent but not received. In this instance, the sender updates the data toggle from `_1'` to `_0'` but the receiver does not. The receiver remains at `_1'`. This causes the host and device to be out of sync on the next data stage, which indicates an error. An example of the data toggle in a USB transfer can be seen in Figure 28. In this figure, and all other figures in this application note, white boxes represent the transaction is coming from the host and black boxes represent the transaction is coming from the device.

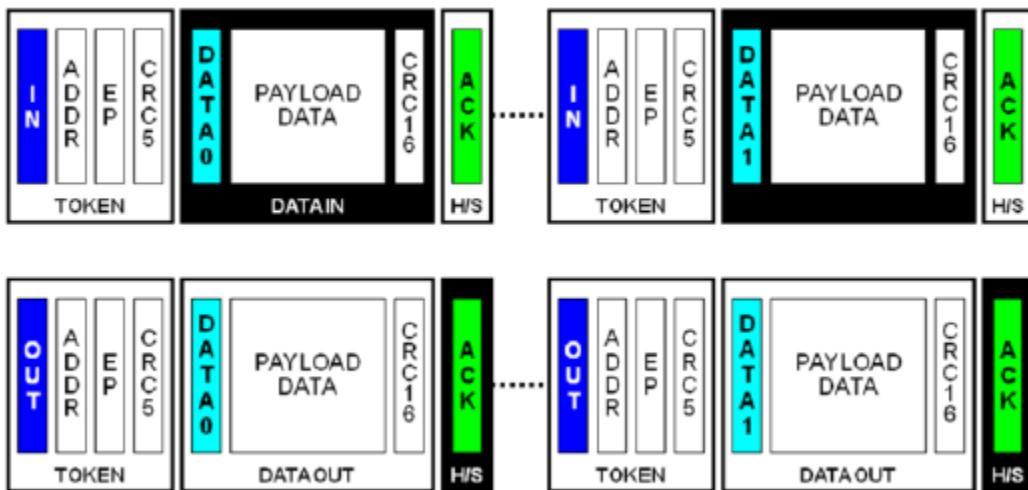


Figure 28. Data Toggle Example

Handshake Packets: Handshake packets conclude each transaction. Each handshake includes an 8-bit packet ID and is sent by the receiver of the transaction. Each USB speed has several options for a handshake response. Which ones are supported depend on the USB speed:

ACK: Acknowledge successful completion. (LS/FS/HS)

NAK: Negative acknowledgement. (LS/FS/HS)

STALL: Error indication sent by a device. (LS/FS/HS)

NYET: indicates the device is not ready to receive another data packet. (HS Only)



Figure 29. Handshake Packet Indicators

Special Packets: The USB specification defines four special packets.

PRE: Is issued to hubs by the host to indicate that the next packet is low speed.

SPLIT: Precedes a token packet to indicate a split transaction. (HS Only)

ERR: Returned by a hub to report an error in a split transaction. (HS Only)

PING: Checks the status for a Bulk OUT or Control Write after receiving a NYET handshake. (HS Only)

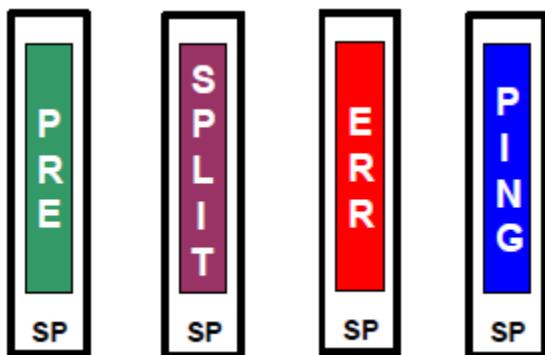


Figure 30. Special Packet Indicator

9.2 Transaction Types

USB transitions are how data from the host and the device get from point A to point B. There are a couple of different transaction types and they often use different names to represent the same concept. The three different transaction types are as follows.

9.2.1 IN/Read/Upstream Transactions

IN, Read, and Upstream are terms that refer to a transaction that is sent from the device to the host. These transactions are initiated by the host by sending an IN token packet. The targeted device responds by sending one or more data packets, and the host responds with a handshake

packet. Figure 31 shows white boxes for transactions from the host and the black box for the transaction from the device.



Figure 31. IN/Read/Upstream Block Diagram

In Figure 32, the device responds with NAKs to show that it is not ready to send data when the host makes the request. The host continues to retry and the device responds with a data packet when it is ready. The host then acknowledges the receipt of the data with an ACK handshake.

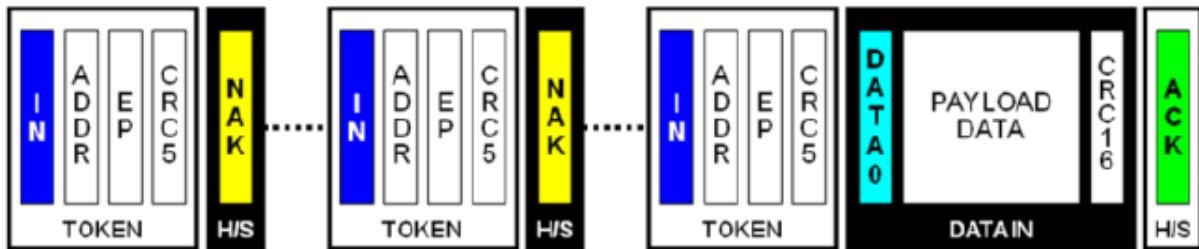


Figure 32. IN Transaction Example

9.2.2 OUT/Write/Downstream Transactions

OUT, Write, and Downstream are terms that refer to a transaction that occurs from the host to the device. In this type of transaction, the host sends the appropriate token packet (either an OUT or SETUP), and follows with one or more data packets. The receiving device ends the transaction by sending the appropriate handshake packet. Figure 33 shows white boxes for transactions from the host and the black box for the transaction from the device.



Figure 33. OUT/Write/Downstream Block Diagram

In Figure 34, the host sends the OUT token packet and a DATA0 packet but receives a NAK from the device. The host then retries to send the data. Notice that the data toggle bit has not changed

since the handshake was NAKed. With the next attempt to send data, the device responds with an ACK to indicate that the OUT transaction was successful.

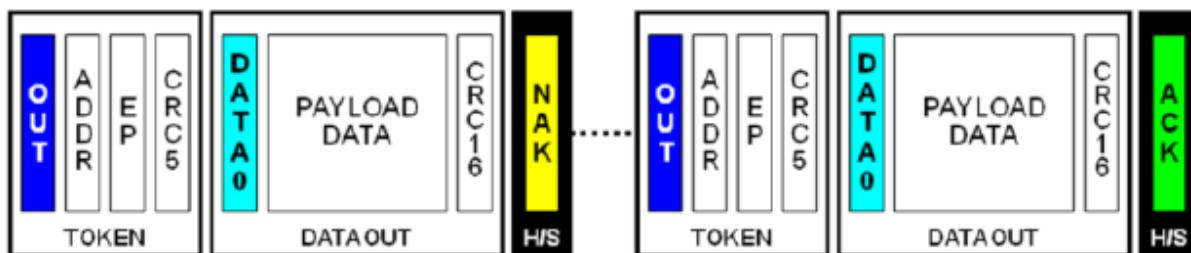


Figure 34. OUT Transaction Example

9.2.3 Control Transactions

Control transfers identify, configure, and control devices. They enable the host to read information about a device, set the device address, establish configuration, and issue certain commands. A control transfer is always directed to the control endpoint of a device. Control transfers have three stages: the setup stage, (optional) data stage, and status stage. Figure 35 shows three stages are transferred by the host. The dotted line around the data stage shows that it is an optional transaction.

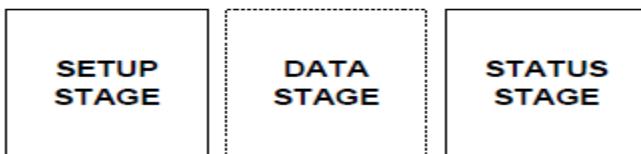


Figure 35. Control Transfer Block Diagram

The setup stage (or setup packet) is only used in a control transaction. This packet sends USB requests from the host to the device and requires the data packet to contain an 8-byte USB request. The device must always acknowledge the setup stage, you cannot NAK a setup stage.

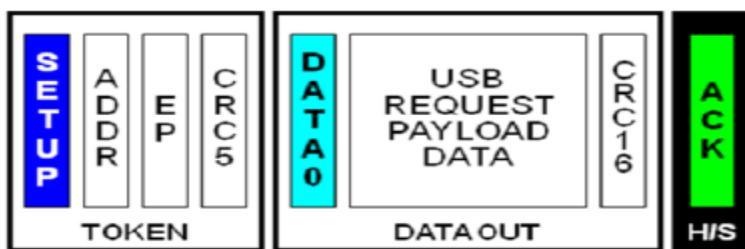


Figure 36. Setup Stage Transaction

The data stage is optional in a control transaction. This stage can consist of multiple data transactions and is only required when a data payload is to be transferred between the host and device. Frequently, relevant data for the control transfer can be transferred in the setup stage.

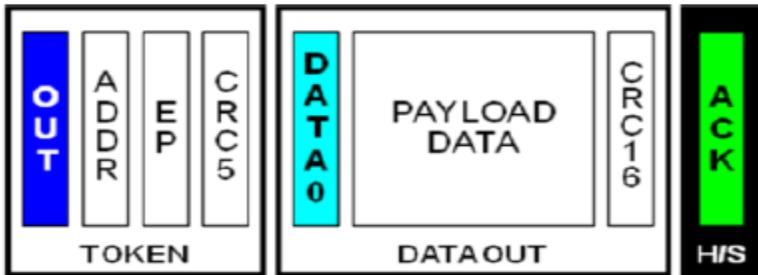


Figure 37. Setup Stage Transaction

The final stage, the status stage, includes a single IN or OUT transaction that reports on the success or failure of the previous stages. The data packet is always DATA1 (unlike normal IN and OUT transactions that toggle between DATA0 and DATA1) and contains a zero length data packet. The status stage ends with a handshake transaction that is sent by the receiver of the preceding packet.

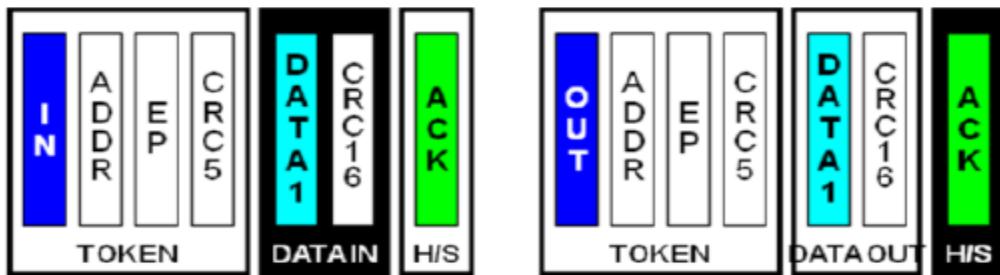


Figure 38. Status Stage Transaction

USB communications have three types of control transfers: control write, control read, and control no data. Figure 39, Figure 40, and Figure 41 show examples of these transactions.

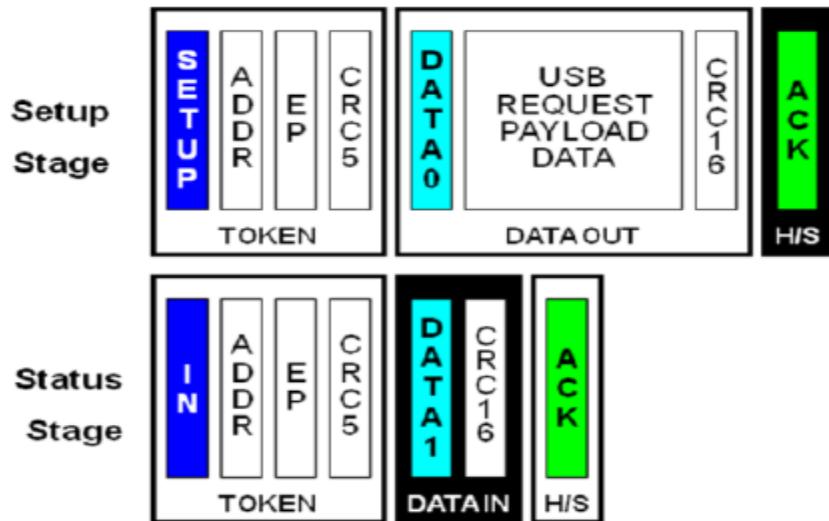


Figure 39. Control No Data Transaction

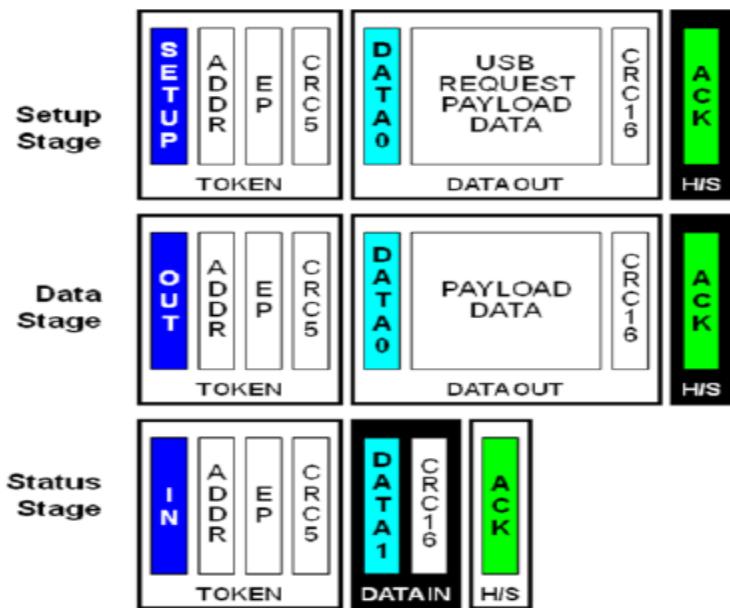
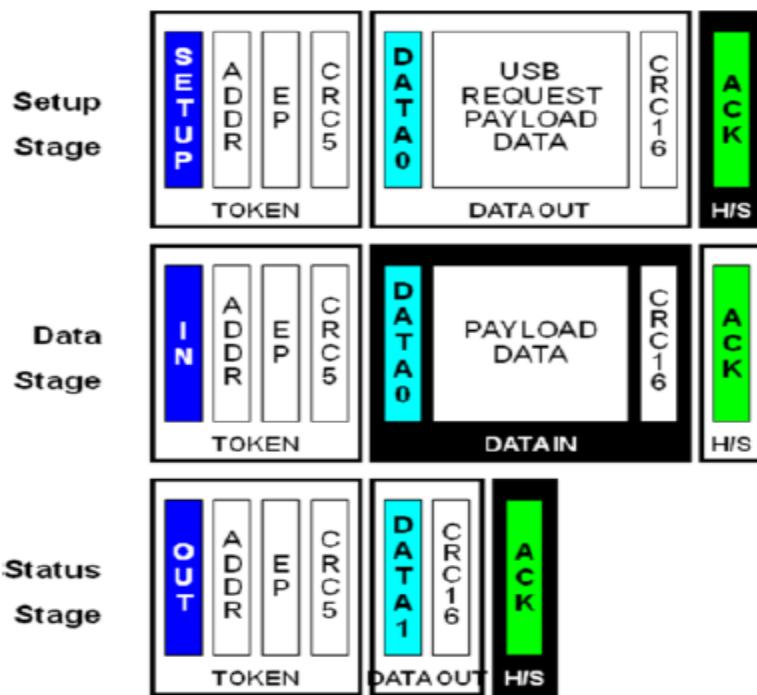


Figure 40. Example of Control Write Transaction

Figure 41. Example of Control Read Transaction



For more information on USB transfer types and their implementations in PSoC 3 and PSoC 5 devices, see the AN56377—PSoC® 3 / PSoC 5LP—Introduction to Implementing USB Data Transfer.

10.0 USB Descriptors

As described earlier, when a device is connected to a USB host, the device gives information to the host about its capabilities and power requirements. The device typically gives this information through a descriptor table that is part of its firmware. A descriptor table is a structured sequence of values that describe the device; these values are defined by the developer. All descriptor tables have a standard set of information that describes the device attributes and power requirements. If a design conforms to the requirement of a particular USB device class, additional descriptor information that the class must have is included in the device descriptor structure. Appendix A includes an example of a fully functional device descriptor for a PSoC USB device.

It is important to know that data fields are transmitted with the least significant bit first. Remember this when reading or creating your own descriptors. Many parameters are 2 bytes long. Make sure that the low byte is first followed by the high byte.

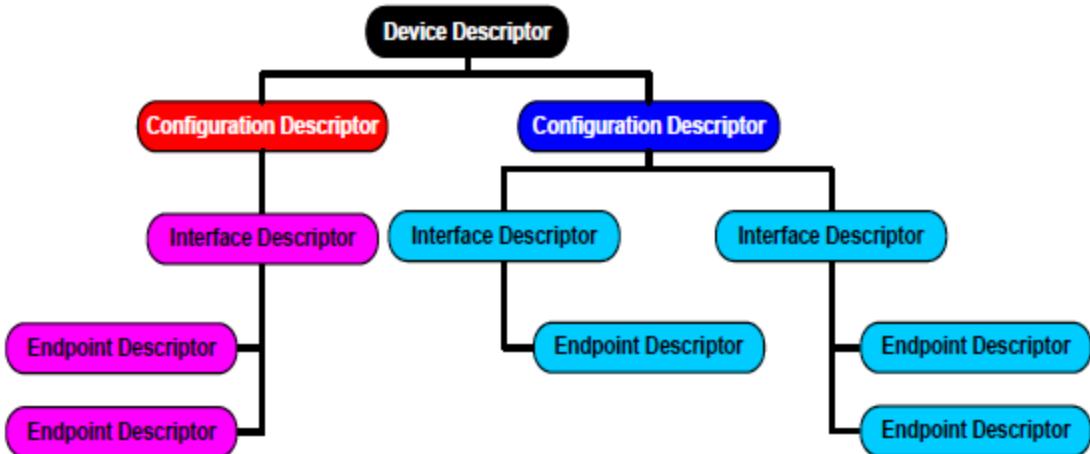


Figure 42. USB Descriptor Tree with Two Configurations

10.1 Device Descriptor

Device descriptors give the host information such as the USB specification to which the device conforms, the number of device configurations, and protocols supported by the device, Vendor Identification (also known as VID, which is something that each company gets uniquely from the USB Implementers Forum), Product Identification (also known as PID, different from a packet ID), and a serial number if the device has one. The device descriptor is where some of the most crucial information about the USB device is contained. Table 5 shows the structure for a device descriptor.

Table 5. Device Descriptor Table

Offset	Field	Size (Bytes)	Description
0	bLength	1	Length of this descriptor = 18 bytes
1	bDescriptorType	1	Descriptor type = DEVICE (01h)
2	bcdUSB	2	USB specification version (BCD)
4	bDeviceClass	1	Device class
5	bDeviceSubClass	1	Device subclass
6	bDeviceProtocol	1	Device Protocol
7	bMaxPacketSize0	1	Max Packet size for endpoint 0
8	idVendor	2	Vendor ID (or VID, assigned by USB-IF)
10	idProduct	2	Product ID (or PID, assigned by the manufacturer)
12	bcdDevice	2	Device release number (BCD)
14	iManufacturer	1	Index of manufacturer string
15	iProduct	1	Index of product string
16	iSerialNumber	1	Index of serial number string
17	bNumConfigurations	1	Number of configurations supported

bLength is the total length in bytes of the device descriptor.

bcdUSB reports the USB revision that the device supports, which should be latest supported revision. This is a binary-coded decimal value that uses a format of 0xAABC, where A is the major version number, B is the minor version number, and C is the sub-minor version number. For example, a USB 2.0 device would have a value of 0x0200 and USB 1.1 would have a value of 0x0110. This is normally used by the host in determining which driver to load.

bDeviceClass, **bDeviceSubClass**, and **bDeviceProtocol** are used by the operating system to identify a driver for a USB device during the enumeration process. Filling in this field in the device descriptor prevents different interfaces from functioning independently, such as a composite device. Most USB devices define their classes in the interface descriptor, and leave these fields as 00h.

bMaxPacketSize reports the maximum number of packets supported by Endpoint 0. Depending on the device, the possible sizes are 8 bytes, 16 bytes, 32 bytes, and 64 bytes.

iManufacturer, **iProduct**, and **iSerialNumber** are indexes to string descriptors. String descriptors give details about the manufacturer, product, and serial number. If string descriptors exist, these variables should point to their index location. If no string exists, then the respective field should be assigned a value of zero.

bNumConfigurations defines the total number of configurations the device can support. Multiple configurations allow the device to be configured differently depending on certain conditions such as being bus powered or self-powered. More details regarding this are discussed later.

10.2 Configuration Descriptor

This descriptor gives information about a specific device configuration such as the number of interfaces, if the device is bus-powered or self-powered, if the device can start a remote wake-up, and how much power the device needs. Table 6 shows the structure for a configuration descriptor.

Table 6.Configuration Descriptor Table

Offset	Field	Size (Bytes)	Description
0	bLength	1	Length of this descriptor = 9 bytes
1	bDescriptorType	1	Descriptor type = CONFIGURATION (02h)
2	wTotalLength	2	Total length including interface and endpoint descriptors
4	bNumInterfaces	1	Number of interfaces in this configuration
5	bConfigurationValue	1	Configuration value used by SET_CONFIGURATION to select this configuration
6	iConfiguration	1	Index of string that describes this configuration
7	bmAttributes	1	Bit 7: Reserved (set to 1) Bit 6: Self-powered Bit 5: Remote wakeup
8	bMaxPower	1	Maximum power required for this configuration (in 2 mA units)

wTotalLength is the length of the entire hierarchy of this configuration. This value reports the total number of bytes of the configuration, interface, and endpoint descriptors for one configuration.

bNumInterfaces defines the total number of possible interfaces in this particular configuration. This field has a minimum value of 1.

bConfigurationValue defines a value to use as an argument to the SET_CONFIGURATION request to select this configuration.

bmAttributes defines parameters for the USB device. If the device is bus powered, bit 6 is set to 0, if the device is self powered, than bit 6 is set to 1. If the USB device supports remote wakeup, bit 5 is set to 1. If remote wakeup is not supported, bit 5 is set to 0.

bMaxPower defines the maximum power consumption drawn from the bus when the device is fully operational, expressed in 2 mA units. If a self-powered device becomes detached from its external power source, it may not draw more than the value indicated in this field.

10.3 Interface Association Descriptor (IAD)

This descriptor describes two or more interfaces that are associated with a single device function. The interface association descriptor (IAD) informs the host that the interfaces are linked together. For example, a USB UART has two interfaces associated with it: a control interface and a data interface. The IAD tells the host that these two interfaces are part of the same function, which is a USBUART, and falls under the communication device class (CDC). This descriptor is not required in all cases. Figure 43 shows how a single interface relates to a single device function. The interface descriptor defines the characteristics for that function. Figure 43 shows how two separate interfaces are linked to a particular device function. This is where the IAD is required. Table 7 shows the structure for an interface association descriptor.

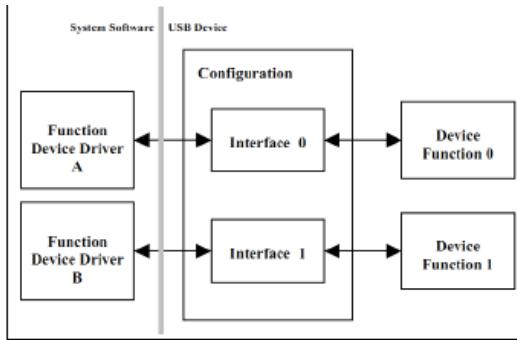


Figure 43. Multiple Interfaces with Multiple Function

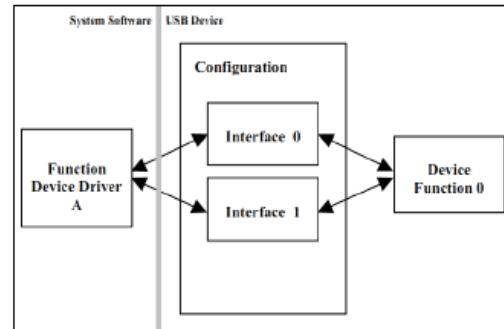


Figure 44. Multiple Interfaces with Single Function

Table 7. Interface Association Descriptor Table

Offset	Field	Size (Bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	Descriptor type = INTERFACE ASSOCIATION (0Bh)
2	bFirstInterface	1	Number identifying the first interface associated with the function
3	bInterfaceCount	1	The number of contiguous interfaces associated with the function
4	bFunctionClass	1	Class code
5	bFunctionSubClass	1	Subclass code
6	bFunctionProtocol	1	Protocol code
7	iFunction	1	Index of string descriptor for the function

10.4 Interface Descriptor

An interface descriptor describes a specific interface within a configuration. The number of endpoints for an interface is identified in this descriptor. The interface descriptor is also where the USB Class of the device is declared. There are many predefined classes that a USB device can be, many of which are listed in Table 12. A USB device class identifies the device functionality and aids in the loading of a proper driver for that specific functionality. Table 8 shows the structure for an interface descriptor.

Table 8. Interface Descriptor Table

Offset	Field	Size (Bytes)	Description
0	bLength	1	Length of this descriptor = 9 bytes
1	bDescriptorType	1	Descriptor type = INTERFACE (04h)
2	bInterfaceNumber	1	Zero based index of this interface
3	bAlternateSetting	1	Alternate setting value
4	bNumEndpoints	1	Number of endpoints used by this interface (not including EP0)
5	bInterfaceClass	1	Interface class
6	bInterfaceSubclass	1	Interface subclass
7	bInterfaceProtocol	1	Interface protocol
8	iInterface	1	Index to string describing this interface

10.5 Endpoint Descriptor

Each endpoint used in a device has its own descriptor. This descriptor gives the endpoint information that the host must have. This information includes direction of the endpoint, transfer type, and maximum packet size. Table 9 shows the structure for an endpoint descriptor.

Table 9. Endpoint Descriptor

Offset	Field	Size (Bytes)	Description
0	bLength	1	Length of this descriptor = 7 bytes
1	bDescriptorType	1	Descriptor type = ENDPOINT (05h)
2	bEndpointAddress	1	Bit 3...0: The endpoint number Bit 6...4: Reserved, reset to zero Bit 7: Direction. Ignored for Control 0 = OUT endpoint 1 = IN endpoint
3	bmAttributes	1	Bits 1..0: Transfer Type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt If not an isochronous endpoint, bits 5...2 are reserved and must be set to zero. If isochronous, they are defined as follows: Bits 3..2: Synchronization Type 00 = No Synchronization 01 = Asynchronous 10 = Adaptive 11 = Synchronous Bits 5..4: Usage Type 00 = Data endpoint 01 = Feedback endpoint 10 = Implicit feedback Data endpoint 11 = Reserved
4	wMaxPacketSize	2	Maximum packet size for this endpoint
6	blInterval	1	Polling interval in milliseconds for interrupt endpoints (1 for isochronous endpoints, ignored for control or bulk)

10.6 String Descriptor

The string descriptor is another optional descriptor and gives user readable information about the device. Possible information contained in the descriptor is the name of the device, the manufacturer, the serial number, or names for the various interfaces or configurations. If strings are not used in a device, any string index field of the descriptors mentioned earlier must be set to 00h. Strings are defined using UNICODE UTF16LE encodings and can support multiple languages with a language ID code. In a Windows system, these strings can be seen in the device manager. Table 10 shows the structure for a string descriptor.

Table 10. String Descriptor Table

Offset	Field	Size (Bytes)	Description
0	bLength	1	Length of this descriptor = 7 bytes
1	bDescriptorType	1	Descriptor type = STRING (03h)
2..n	bString -or- wLangID	Varies	Unicode encoded text string -or- LANGID code

The string descriptor has two possible forms. The first string descriptor contains a value for the language ID, **wLangID**, which contains one or more two byte ID codes that indicate the languages in which the strings are. USB-IF gives a document that defines many different ID codes. U.S. English for example is 0409h. For more ID codes, see the USB-IF LANGID page. All string descriptors that occur after wLangID use **bString**, which is a string field that contains a Unicode string (UTF16LE) and uses 2 bytes to represent each character.

10.7 Other Miscellaneous Descriptor Types

Report Descriptors: A USB device class may require an extended set of descriptor information. Developers must make certain that any additional descriptor information required by a USB device class is included in the descriptor file. For example, with the HID class the developer must include report descriptors that further define the device attributes. If additional descriptors are required, the descriptor format is present in the class definition specification or other class supporting documentation. For additional information on Report Descriptors, see AN57473 - *USB HID*

Basics with PSoC® 3 and PSoC 5LP and AN58726 - USB HID Intermediate with PSoC® 3 and PSoC 5LP.

MS OS Descriptor: Microsoft has a descriptor called the Microsoft OS Feature Descriptor (also called the MS OS descriptor) that is used in vendor-specific devices. This descriptor gives Microsoft Windows specific information such as special icons, registry settings, help files, and

URLs. The MS OS descriptor contains a string and one or more feature descriptors. During enumeration, Windows requests the string descriptor, which contains an index of EEh and an embedded signature. If the device supports the MS OS descriptor, Windows requests additional information after receiving the string descriptor. If the device does not support the MSOS descriptor, the device returns a STALL as the handshake. More information on MS OS descriptors is at the MSDN Microsoft OS descriptors landing page.

Device_Qualifier Descriptor: Another descriptor seen in USB is a Device_Qualifier Descriptor. This describes information about a High-Speed capable device that changes if the device operates at another speed and is required by devices that support both speed configurations. If a device operates at Full-Speed when this descriptor is requested, it tells the host how the device would operate differently if it were operating at High-Speed. The same is true if the descriptor is requested while the device is operating at High-Speed. The descriptor read tells the host about a Full-Speed configuration. If this descriptor is requested and the device supports only Full-Speed, the proper action is to respond with a STALL. Otherwise, the descriptor information will be provided to the host upon request. For more information on this descriptor, see Section 9.6.2 of the USB specification.

BOS Descriptor: Another descriptor seen in USB 2.0 devices that support Link Power Management (LPM) is the Binary device Object Store (BOS) descriptor. In the PSoC family of devices that have USB 2.0 device support, only the PSoC 4200L family of devices support the LPM feature, and consequently the BOS descriptor. LPM is an improvement over the USB suspend mode that enables devices to enter and exit low-power modes with transition latencies of the order of tens of microseconds compared to the 3-20 ms latency associated with suspend mode entry/exit.

Any USB 2.0 device that supports the LPM feature has to report its LPM capabilities through the USB 2.0 extension descriptor. The USB 2.0 extension descriptor is part of the BOS descriptor. The value of the bcdUSB field in the standard USB 2.0 device descriptor is used to indicate that the device supports the request to read the BOS descriptor through the GetDescriptor(BOS) request. Devices that support the BOS descriptor must have bcdUSB value of 0201H or larger. See Section 9.6.2 of the USB 3.1 specification for details on the BOS descriptor and USB 2.0 extension descriptor.

10.8 Using Multiple USB Descriptors

USB devices have only one device descriptor. However, a device can have multiple configurations, interfaces, endpoints, and string descriptors. When a device is enumerated, one of the final

stages is to read the device descriptors and make a decision on which device configuration to enable. Only one configuration can be enabled at a time. An example of this is a design that has one configuration that is used when the device is self-powered and another configuration when the device is bus-powered. The overall USB functionality may be different for the self-powered device than for the bus-powered device. Having multiple configurations and multiple configuration descriptors allows the option to implement this ability.

At the same time, a device can have multiple interfaces, thus multiple interface descriptors. A USB device with multiple interfaces that perform different functions is called a composite device. An example is a USB audio head set. In the head set, you have a single USB device with two interfaces. One interface is for the audio side of the headset and another interface may be the controls to adjust the volume. Multiple interfaces can be active at the same time. Figure 45 shows a diagram of the ability to split two interfaces under a single USB device.

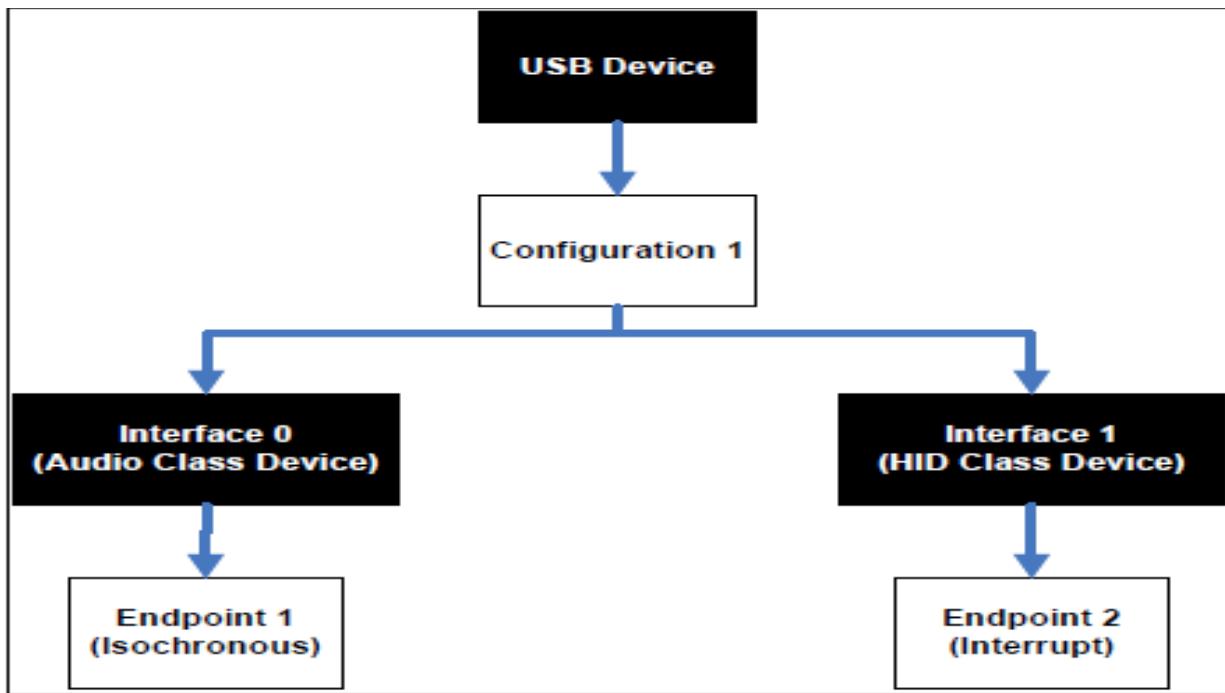


Figure 45. Multiple Interface Settings Diagram

Finally, each interface can have multiple configurations. These multiple configurations are called alternate settings. One possible application is to allow the ability to alter endpoint configuration on a device to reserve different amounts of bandwidth. For example, in one alternate setting, a device can have its endpoints configured to bulk, which has no guaranteed bus bandwidth; in another alternate setting, it can have the endpoints configured for Isochronous, which has guaranteed bus bandwidth. This concept is shown in Figure 46.

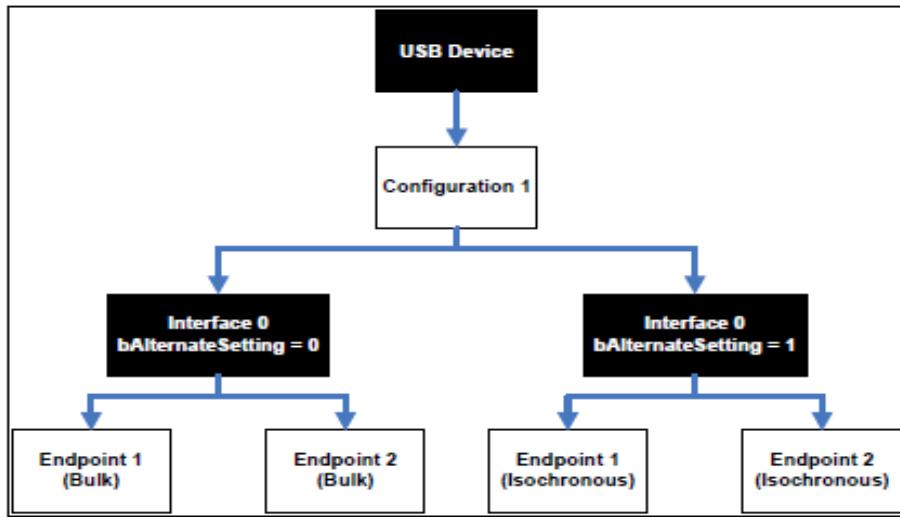


Figure 46. Multiple Interface Settings Diagram

Figure 47 shows an overall diagram view of the customization options that can be used to create a highly customizable USB device to suit a variety of configurations options.

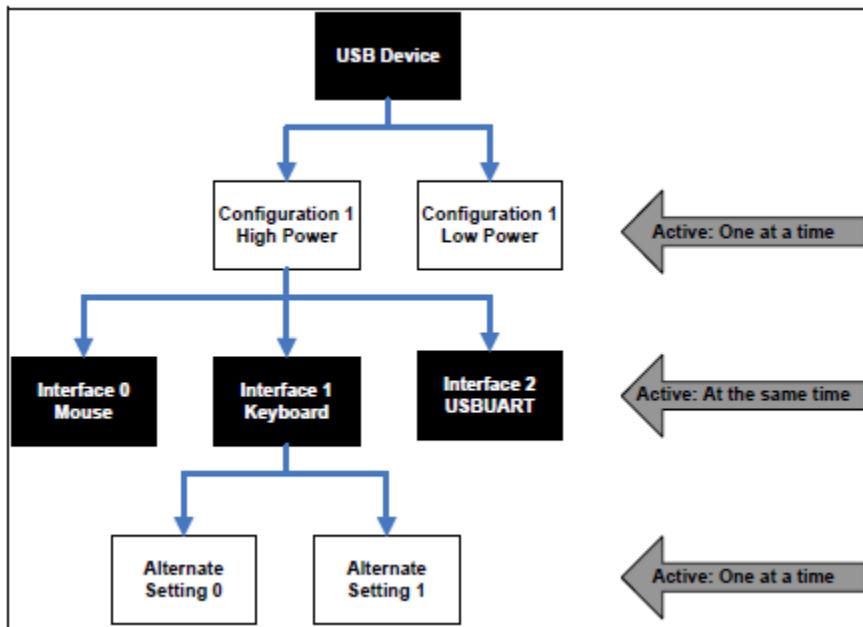


Figure 47. Configuration Diagram

11.0 USB Class Devices

The USB Implementers Forum has a list of recognized and approved USB device classes. The most common device classes are

Human Interface Device (HID)

Mass Storage Device (MSD)

Communication Device Class (CDC)

Vendor (Vendor Specific)

There are several considerations to think about when developing an application for a certain class. First, each class has a fixed maximum bandwidth. Second, each class has limitations on the supported transfer types and certain commands that must be supported. However, the biggest advantage to using a predefined USB device class is the cross platform support across various operating systems. All major operating systems include a driver in the OS for most of the predefined USB classes that eliminates the need to create a custom driver. Table 11 shows some of the more common drivers that are used with Cypress products and some of the capabilities of those drivers.

Table 11. USB Device Class Driver Features

Feature	HID	CDC	WinUSB	LibUSB	CYUSB
Driver Support in Windows	Yes	Need .inf	Need .inf ^[1]	No	No
Support for 64-Bit	Yes	Yes	Yes	Yes	Yes
Support for Control Transfers	Yes	No	Yes	Yes	Yes
Support for Interrupt Transfer	Yes	No	Yes	Yes	Yes
Support for Bulk Transfers	No	Yes	Yes	Yes	Yes
Support for Isochronous Transfers	No	Yes	No	Yes	Yes
Maximum Speed (Full-Speed)	~64 KB/s	~80 KB/s	~1 MB/s	~1 MB/s	~1 MB/s

Devices that do not meet the definition of a specific USB device class are called vendor-specific devices. These devices allow developers to create applications with their own creativity and customization options, which are not bound by a specific USB class, but still conform to the USB specification. Devices that fall under a vendor-specific device use WinUSB, CYUSB, LibUSB, or another type of vendor-specific driver. The advantage to using WinUSB is that it is Windows own vendor-specific driver and does not need to undergo Windows Hardware Quality Labs (WHQL) testing for driver signing. WHQL testing is discussed later in this application note. LibUSB is an open source driver project with support for Windows, Mac, and Linux operating systems. CyUSB is Cypress own vendor specific driver. The advantage to using this driver in an application is the broad range of example applications, supporting documentation, and direct support from Cypress.

In the USB Descriptors section, notice that the fourth byte in the device descriptor and the sixth byte in the interface descriptor are where the class of the USB device is defined. The USB specification defines many different USB classes and the device class codes that go along with

them. Table 12 shows some USB class codes that can be used in these bytes to give an idea of the various USB classes that are available.

Table 12. USB Class Codes

Class	Usage	Description	Examples
00h	Device	Unspecified	Device class is unspecified, interface descriptors are used to determine needed drivers
01h	Interface	Audio	Speaker, microphone, sound card, MIDI
02h	Both	Communications and CDC Control	Modem, ethernet adapter, Wi-Fi adapter
03h	Interface	Human Interface Device (HID)	Keyboard, mouse, joystick
05h	Interface	Physical Interface Device (PID)	Force feedback joystick
06h	Interface	Image	Camera, scanner
07h	Interface	Printer	Printers, CNC machine
08h	Interface	Mass Storage	External hard drives, flash drives, memory cards
09h	Device	USB Hub	USB hubs
0Ah	Interface	CDC-Data	Used in conjunction with class 02h.
0Bh	Interface	Smart Card	USB smart card reader
0Dh	Interface	Content Security	Fingerprint reader

Class	Usage	Description	Examples
0Eh	Interface	Video	Webcam
0Fh	Interface	Personal Healthcare	Heart rate monitor, glucose meter
DCh	Both	Diagnostic Device	USB compliance testing device
E0h	Interface	Wireless Controller	Bluetooth adapter
EFh	Both	Miscellaneous	ActiveSync device
FEh	Interface	Application Specific	IrDA Bridge, Test & Measurement Class (USBTMC), USB DFU (direct firmware update)
FFh	Both	Vendor Specific	Indicates a device needs vendor specific drivers

12 USB Enumeration and Configuration

Normally, developers look at enumeration as a single process in a USB device. Enumeration is actually one part in a three-stage process: dynamic detection, enumeration, and configuration. Dynamic detection is the recognition of a change in the state of a USB port. In Figure 17 you see that there are pull-down resistors on the host/hub side. When a device is attached, one of these lines is pulled high depending on device speed. It is with this voltage transition that the host/hub detects the change of the bus port. Enumeration directly follows the device detection, and is the process of assigning a unique address to a newly attached device. Configuration is the process of determining a device's capabilities by an exchange of device requests. The requests that the host uses to learn about a device are called standard requests and must support these requests on all USB devices. The entire enumeration process is described in the following sections.

12.1 Dynamic Detection

Step 1: The device is connected to a USB port and detected. At this point, the device can draw up to 100 mA from the bus. The device is currently in the powered state.

Step 2: The hub detects the device by monitoring voltages on the ports. A hub has pull-down resistors on the D+ and D- lines as seen in Figure 17. As mentioned earlier, there is a pull-up resistor on either the D+ or D- line depending on device speed. By monitoring the voltage transition on these lines, the hub detects if a device is attached.

12.2 Enumeration

Step 3: The host learns of the newly attached device by using an interrupt endpoint to get a report about the hub's status. This includes changes in port status. After the hub tells the host about the device detection, the host issues a request to the hub to learn more details about the status change that occurred using the GET_PORT_STATUS request.

Step 4: After the host gathers this information, it detects the speed of the device using the method mentioned in the USB Speeds. Initially, only Full-Speed or Low-Speed is detected by the hub by detecting if the pull-up resistor is on the D+ or D- line. This information is then reported to the host by another GET_PORT_STATUS request.

Step 5: The host issues a SET_PORT_FEATURE request to the hub asking it to reset the newly attached device. The device is put into a reset state by pulling both the D+ and D- lines down to GND (0 V). Holding these lines low for greater than 2.5 us issues the reset condition. This reset state is held for 10 ms by the hub.

Step 6: During this reset, a series of J-State and K-State occurs to determine if the device supports High-Speed. During this reset state, if the device supports High-Speed, it issues a single K-State. A High-Speed hub detects this K-State and responds with a sequence of J and K states to form a —KJKJKJ|| pattern. The device detects this pattern and removes its pull-up resistor from its D+ line. This step is skipped on Low-Speed and Full-Speed devices.

Step 7: The host then checks to see if the device is still in a reset state by issuing a GET_PORT_STATUS request. If the request reports that the device is still in reset, then the host continues to issue the request until it receives word that the device is out of reset. After the device leaves reset, it is in the default state as mentioned in the USB Power section. The device can now respond to requests from the host in the form of control transfers to its default address of 00h. All USB devices start with this default address. Only one USB device can have this address.

at a time; this is why when you connect multiple USB devices to a port at the same time, they enumerate sequentially and not simultaneously.

Step 8: The host begins the process of learning more information about the device. It starts by learning the maximum packet size of the default pipe (Endpoint 0). The host starts by issuing a GET_DESCRIPTOR request to the device. The device begins to send the descriptors discussed in the USB Descriptors section of the application note. In the device descriptor, the eighth byte (**bMaxPacketSize0**) contains information about the maximum packet size for EP0. A Windows host requests 64-bytes, but after only receiving 8 bytes of the device descriptor, it moves onto the status stage of the control transfer and requests that the hub reset the device. The USB specification requires that a device return at least 8 bytes of the device descriptor, when requested, if the device has the default address of 00h. The reason for requesting the 64-bytes is to avoid unpredictable behavior from the device. Additionally, the reason for performing a reset after only receiving 8 bytes is an artifact of early USB devices. In the early life of USB, some devices did not respond properly when a second request for the device descriptor occurred. To solve this problem, a reset after the first device descriptor request was required. Regardless, the 8 bytes that were transferred was enough to get the require information about the **bMaxPacketSize0**.

Step 9: The host applies an address to the device with the SET_ADDRESS request. The device completes the status stage of this request using the default 00h address before using the newly assigned address. All communication beyond this point will use the new address. The address may change if the device is detached from a port, the port is reset, or the PC reboots. The device is now in the address state.

12.3 Configuration

Step 10: After the device returns from its reset, the host issues a command, GET_DESCRIPTOR, using the newly assigned address, to read the descriptors from the device. However, this time all the descriptors are read. The host uses this information to learn about the device and its abilities. This information includes the number of peripheral interfaces, power connection method, and the required maximum power. The host starts by requesting the device descriptor, but this time it receives the entire descriptor and not just a partial version. Next the host issues another GET_DESCRIPTOR command asking for the configuration descriptor. This request not only returns the configuration descriptor, but all other descriptors associated with it such as the interface descriptor and the endpoint descriptor. A Windows PC first asks for just the configuration descriptor (9 bytes), than it issued a second GET_DESCRIPTOR request for the configuration descriptor and all other associated descriptors with that configuration (interface and endpoint descriptors).

Step 11: For the host PC to successfully use the device, a Windows PC in this case, the host must load a device driver. The host searches for a driver to manage communication between itself and the device. Windows uses its *.inf* files to locate a match for the devices Product ID and Vendor ID. Device release version numbers can optionally be used. If Windows cannot find a match, then it looks at the driver from a different perspective by looking for a match with any class, subclass, and protocol retrieved from the device. If a device was previously enumerated, Windows uses its registry to search for the proper driver. When a driver is identified, the host may request additional descriptors that are specific to the device class or request that descriptors are resent.

Step 12: After all descriptors are received, the host sets a specific device configuration using the SET_CONFIGURATION request. Most devices have only one configuration. Devices that support multiple configurations can allow the user or the driver to select the proper configuration.

Step 13: The device is now in the configured state. It took on its properties that were defined with the descriptors. The defined maximum power can be drawn from VBUS and the device is now ready for use in an application.

13 Debugging USB Designs

Picture this situation: you have spent hours, days, or weeks developing a design that uses USB and now the time has come to test the design. The firmware is loaded and you go to plug the device into the host. As you stare at the prototype, you realize that nothing is happening; something went wrong. You now need to do some debug to check what happened. The question racing through your mind is where to start. There are several tips and tools to use when attempting to debug a design and this section is intended to help you through the debug process.

13.1 Debugging on the Host Side

When a USB device does not seem to be working properly, the first step is to start the debug process from the host side, because the host is responsible for initiating the communication with the device. Based on what is observed, a progression to debugging the device may be required. If the host being used is running Windows, a list of attached devices and their status can be found in the Device Manager (shown in Figure 48) by following these steps: If the host being used is running **Windows XP/Vista**, a list of attached devices can be found by following these steps:

Step 1: Click **Start > System > Hardware > Device Manager**. If the host being used is running **Windows 7**, a list of attached devices can be found by following these steps:

Step 1: Click **Start > Control Panel**.

Step 2: Select **System > Device Manager**. Note that you may need to change the —View by:|| option to —Large Icons|| or —Small Icons||. The options provided will be different for —Category||.

If the host being used is running **Windows 8/8.1**, a list of attached devices can be found by following these steps:

Step 1: Click the Desktop Tile to open Desktop.

Step 2: Click —>>|| at the bottom right Taskbar.

Step 3: Select **Control Panel** from the menu that appears.

Step 4: Select **Device Manager** from the list of control panel items.

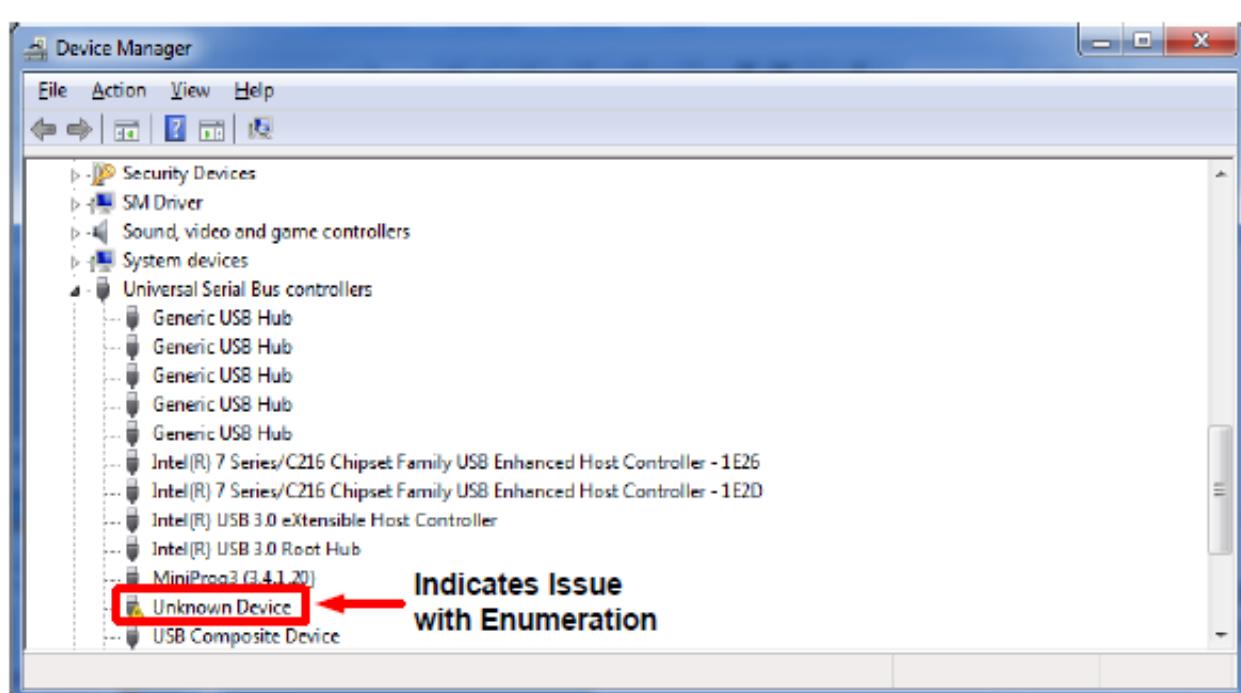


Figure 48. Windows Device Manager

When there is an issue on the host side with a USB device, it is most commonly related to one of the two things: malfunctioning or improperly configured hardware or malfunctioning or missing device drivers. Using Device Manager, you can tell several things about the device connected.

Unknown Device: Similar to what is seen in Figure 48, this issue can signify several different things. First, it can indicate that the device does not have a properly installed driver. Often, it will not list the —Unknown Device|| under the —Universal Serial Bus controllers|| branch, but rather under a —Other devices|| branch. When this occurs, double clicking on the device, which opens the properties of the device, will show an Error Code 1 or Error Code 10. Many error codes can be reported by the device manager. Microsoft provides a list, description, and possible solution

for the error codes to help when needed. The link can be found here. Additionally, a driver that was not installed or configured correctly can cause issues. This sometimes happens when a USB device is removed while Windows is trying to install and configure the hardware. Right-click on the device, select —“Uninstall”, and unplug the device once complete. Plug the device back in. You may even need to force the driver update by right-clicking and selecting —Update Driver Software”.

Device is not listed: In the instance that you cannot find the device at all in Device Manager, the first thing to make sure is that the device is not exceeding the hub power requirements. Under the —Universal Serial Bus controllers|| tree, you will see one or more —USB Root Hub|| or —Generic USB Hub||. Double-click on them and select the —Power|| tab. In there, you will see the attached device and the amount of power they each require. An example can be seen in Figure 49. Additionally, the total power available will be listed. Ensure that the required power does not exceed the available power.

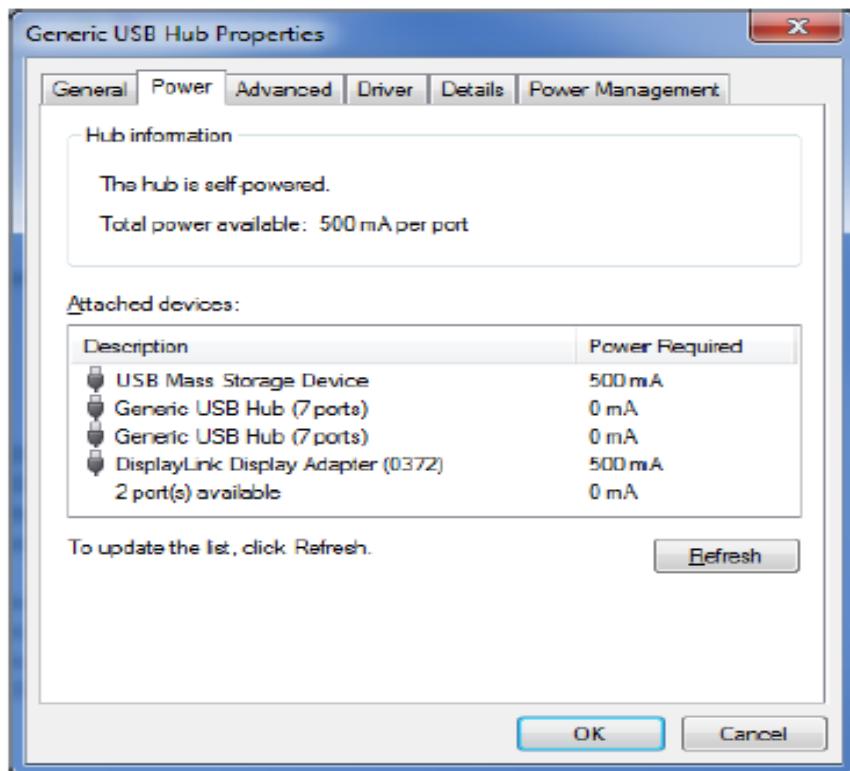


Figure 49. Device Manager - USB Power Properties

If that does not solve the problem, try a different USB port and/or cable. There is always the possibility that the contacts or some cabling has gone bad. Lastly, there is software provided by Cypress, such as the CY SuiteUSB/EZUSB, that can be used to provide useful tools for debug without needing to develop test applications on the host side. Bundled with the software are the following tools:

Bulkloop: Simple tool that provides a loopback of data through a bulk transfer. Users can define a data pattern to send and the tool will inform the user how many bytes were transferred OUT and how many were transferred IN.

USB Data Streamer: Intended for isochronous data transfers. This tool is used to test the bandwidth and speed capabilities of a USB device by sending a constant stream of data. The user can configure the number of packets per transfer and the tool will keep track of the number of packets dropped, the number of packets received successfully, and throughput in KB/s.

USB Console: Intended to communicate with USB devices that are using the CyUSB.sys driver. Provides detailed information about the device using the driver and provides advanced control options to interface with the device.

USB Control Center: Intended to communicate with USB devices that are served by the Cypress CyUSB.sys driver, Windows Mass Storage driver (usbstor.sys), or Windows HID driver (hidusb.sys). This tool provides a capability to view attached USB devices, similar to Device Manager, while also providing a mechanism to communicate with various device endpoints by sending or requesting data.

13.2 Debugging the Communication

When attempting to debug a USB design, there will come a time where one may need to view the USB traffic. To accomplish this, specialized tools are available. These tools come in two forms: hardware-based, where dedicated devices are attached to the system under test, and software-based, where the software resides on the host hardware. A USB analyzer allows you to record the bus traffic that is sent between the device and the host. They decode the USB traffic information such as enumeration, signaling errors, and general data transfer into an easy to read display.

This provides an easy way to debug USB design issues. Software analyzers can be used to capture device data transfers and protocols. They can also be used to send commands to the device. Software analyzers work by replacing the USB software stack on the host machine. This is a required step in order to monitor the USB traffic. As a result, the software analyzer will be dependent on the host hardware with regards to what information is available to be presented. Software analyzers have the advantage of being lower-priced than their hardware counter parts.

However, at a lower cost come some disadvantages. First, a host cannot monitor the USB traffic without being part of the bus itself. This means observing conditions such as suspend or reset is not possible. Additionally, depending on the host hardware and software stack, there are other bus conditions that the host-controller handles without software involvement. One final

disadvantage is related to timing. The accuracy of timing with a software analyzer is dependent on the host operating system, which could induce varying levels of inaccuracies.

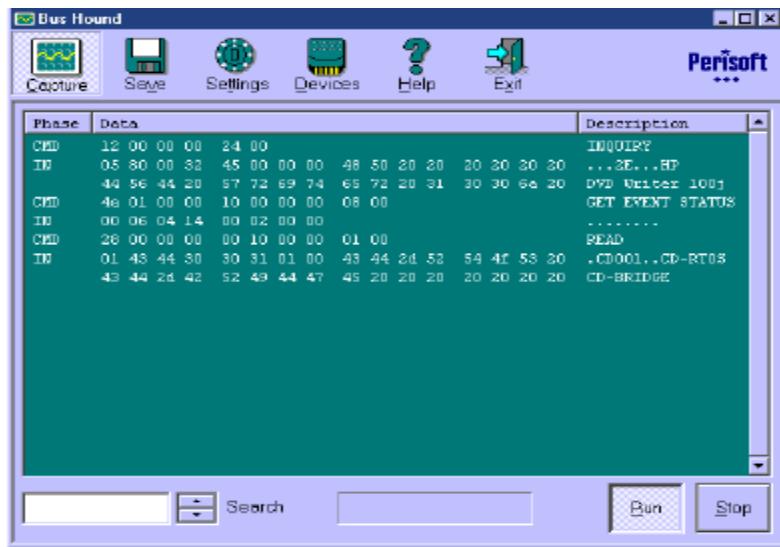


Figure 50. Bus Hound Software Analyzer by Perisoft

Hardware analyzers are more commonly used and are manufactured by a wide variety of companies. These devices are custom-manufactured pieces of hardware that are inserted between the host and the device to monitor the bus traffic. Some of these devices even have the capability to generate bus traffic if there is a specific condition that is trying to be produced. Prices for these types of devices can vary greatly depending on manufacturer, desired USB specification (USB 2.0 versus USB 3.0), and traffic generation capabilities. In general, these analyzers cost more than their software counterparts, but address the disadvantages that software analyzers face. While there are several companies that produce hardware analyzers, two more common companies are TotalPhase and LeCroy.



Figure 51. Examples of USB Analyzers

How these devices are used is quite simple and can be used in one of two ways. In one application, the USB analyzer is inserted between the host and the device being tested, as shown in Figure 52. The host will be responsible for running the application that is associated with the device under test while also running the analyzer software that will be used to view the traffic.



Figure 52. Using Analyzer with Single Host

In a second setup, two PCs can be used where one PC acts as the host for the device under test while the analyzer is connected to a second host responsible for monitoring the bus activity, as shown in Figure 53. You may wonder why one analyzer configuration would be used over another. The answer is that there are situations where there is a need to monitor suspend and resume activities for the device being tested. Because the host has to be put into a suspend state, you will need a separate host to continue monitoring the traffic.

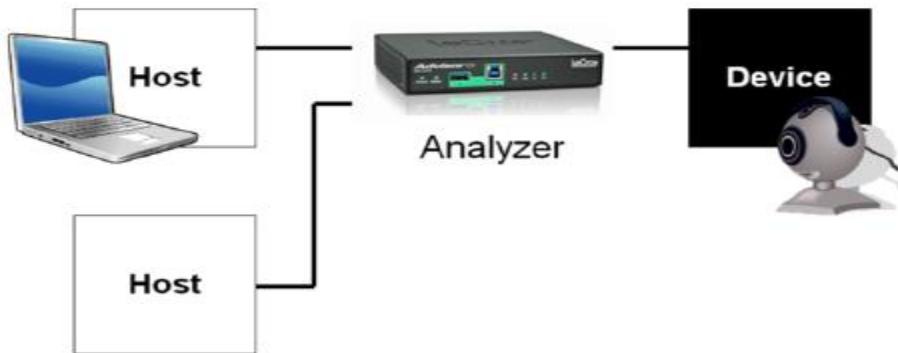


Figure 53. Using Analyzer with Dual Host

The prices of these analyzers can range from several hundred dollars to several thousand dollars. Note that if the application being designed does not require USB 3.0, money can be saved by purchasing a USB analyzer that only supports USB 2.0. Additionally, even more money can be saved with only purchasing a Low Speed-only or Full-Speed-only analyzer. Hardware-based USB analyzers are not solely limited to a dedicated hardware, but can also be found on some higher-end oscilloscopes. Oscilloscopes such as these even have the capability to run the USB electrical tests performed during USB compliance testing. Now that the various types of USB analyzers are

understood, the next step is to look at what kind of information can be expected to be seen with an analyzer. Earlier in the USB Enumeration and Configuration section, the various steps of enumeration were discussed. Using a bus analyzer, the enumeration process can be observed. An example of a USB bus analyzer trace is shown in Figure 54. Appendix B shows the enumeration of the USB device on a USB bus analyzer, whose descriptors are in Appendix A. Using what you have just read in the USB Enumeration and Configuration section, and the information in Appendix A, you have enough instructions to follow and understand the bus analyzer capture. Be aware that the enumeration sequence shown in Appendix B is one of many possible enumeration event examples. This particular enumeration sequence uses the descriptors in Appendix A and enumeration was done on a Windows XP computer with the WinUSB driver. What you see varies depending on the particular device's descriptors, the operating system being enumerated on, and the drivers used.

Reset.		25.644 ms		Idle				
Packet #	L/S	Sync	SETUP	ADDR	ENDP	CRC5	EOP	Idle
190	S	00000001		0xB4	0	0	0x08	3.00
191	S	00000001		0xC3	00 05 02 00 00 00 00 00		0xD768	3.00
192	S	00000001	ACK	EOP	Idle			
194	S	00000001		0x96	0	0	0x08	3.00
195	S	00000001		0xD2		0x0000	3.00	6
196	S	00000001	ACK	EOP	Idle			

Figure 54. USB Analyzer Output Example

To help understand what Figure 54 is showing, let us focus specifically on Packet #190, Packet #191, and Packet #192.

Packet #190

L/S: Implies Low-Speed

Sync: Synchronizes the sending of packets between the host and the device.

SETUP: Implies it is a SETUP token.

ADDR: The current address is zero because it is the first transfer to the device. All control transfers are initiated from endpoint 0, address 0.

ENDP: End point- gives the endpoint location. ENDP 0 implies it is a control transfer.

CRC5: CRC 5 bits - detects errors in tokens.

EOP: End of Packet - Signifies that the packet has ended.

Packet #191

DATA0: Implies it is a data token

CRC 16: 16 bit CRC check for data

Idle: The time between the current packet and the previous packet.

Packet #192

ACK: Indicates the successful completion of the data packet.

Bus analyzer traces are often used in other Cypress PSoC USB application notes. You must understand the information that is displayed on the bus analyzer trace to fully understand the information that is being conveyed with their help. These analyzers have the ability to show different levels of abstraction from a high level transaction view to a low level D+ and D-waveform view. It is up to you to determine how deeply you want to analyze data when using a bus analyzer. That being said, a bus analyzer is an extremely valuable tool when debugging a USB once you know what to look for, since the analyzer can show information related to the packets being sent, the handshakes, and the timing. When USB basics are understood, the user will be able to look at what the analyzer captures, understand what is missing, and hone in on how to correct the issue.

13.3 Debugging on the Device Side

Capabilities of debugging on the device side will depend on the capabilities of the device and the development tools. For PSoC, PSoC Designer™ and PSoC Creator™ can be used to place breakpoints while inspecting memory locations and register configurations. The same stand true for the FX-series of USB parts along with Keil uVision and Eclipse. This provides a powerful method to make sure that ISRs are triggered and that commands are properly sent to the device. Cypress provides Technical Reference Manuals for various products that will outline the various registers and the effect they have on the system. Those documents in conjunction with the debuggers in the IDEs will enable you to make sure that the device is working as expected.

14.0 Acquiring a VID and PID

To sell or market a USB product you must have a Vendor ID. This means that you must acquire one from USBIF. It is important to make sure that you get a Vendor ID for your company to avoid legal consequences. There are two options on how to acquire a VID:

Become a member of USB-IF and pay the annual membership dues. At the time of this writing, the dues were \$4000/yr. Membership has several added benefits. Some of the most important include the eligibility to participate in free USB-IF sponsored quarterly Compliance Workshops, a Free Vendor ID (if one has not been previously assigned), a waived logo-administration fee when joining the USB-IF logo program, and the eligibility to participate in Device Working Groups.

¶ Purchase a Vendor ID from USB-IF for a one-time fee of \$5000. Note however that simply buying a VID does not authorize you to use the USB logo with a product. In order to do so, you need to become a USB-IF non-member logo licensee. This licensing fee comes at a cost of \$3,500 for a two year term. Also, be aware that this licensing fee does not entitle you to other USB-IF membership benefits, such as compliance workshops. After you purchase a VID, you can use it across all USB devices that you produce. The Product ID is not purchased but chosen by the developer or company. Each product needs its own VID/PID combination. The USB-IF recommends each vendor set up a coordinated allocation scheme for PIDs so that different teams do not inadvertently choose the same PID for different products. Duplicate numbers will cause compliance testing to fail. Commonly, Cypress is asked if their VID can be used on a customer's end product. The answer to this question is —No|. Cypress VIDs can be used in a development environment, but cannot be released in a production design. The main reason behind this is due to operating systems, such as Windows, remember the system files (inf file, driver, and so on) used by the device and loads them every time the device is attached. If our VID is used by our customers,

then the possibility that another customer may use the same VID/PID for their product exists. In this case, as you can see, the end product may bind to a different driver and malfunction. Ultimately this causes trouble for the end customer.

15.0 Compliance Testing

When it comes to official compliance testing for USB devices and host software, there are two options available from the USB Implementers Forum and Microsoft. This section will look at options, what the test process is from a high level, and the benefits achieved by going through the test process.

15.1 USB-IF Compliance Testing

The intention of USB is to provide a universal plug-and-play connection so that even the most non-tech savvy person can simply plug a USB cable into a host and have it working. To help ensure this, the USB IF performs compliance testing to insure that all consumers have a good overall experience with USB. As a way to provide reassurance to customers, a USB product can bear a —Certified USB| logo, similar to those in Figure 55.



Figure 55. USB Compliance Logos

These logos cannot be arbitrarily put onto a product. Instead, they must undergo a series of compliance tests that are administered by USB-IF. This compliance testing is performed on products ranging from devices, hubs, host systems, silicon building blocks, and mechanical products (such as cables and connectors). These compliance tests will ensure that the device under test confirms to any relevant sections of the USB specification. Upon passing, the product will receive the ability to use the relevant USB —Certified|| logo, and in addition, have the product put on the USB integrators' list, which is simply a list maintained by USB-IF that contains the list of USB products that have met the mandatory compliance criteria. There are two components that must be completed on the path to being certified as USB compliant: checklists and compliance testing.

15.1.1 USB- IF Compliance Checklists

There are several checklists depending on what is being certified. In essence, the checklists are a questionnaire about the device's specifications, functionality, and behavior. Many Cypress devices will be used in a peripheral application, which means the checklists will ask varying questions regarding the mechanical design and layout, device states and signals, operating voltage, and power consumption. Each question in a checklist is accompanied with a section number that can be referenced to the USB specification itself for a more detailed explanation. These check lists act as the first step towards compliance certification.

15.1.2 USB- IF Compliance Testing

After all the checklist requirements are met, the next step is to go through the second certification criteria: compliance testing. There are two ways to perform compliance testing on a device. Companies that are a member of USB-IF may attend a compliance testing workshop (also known as a Plugfest). The testing can also be contracted out to an independent certified lab. The following link is a list of Independent Test Labs provided by USB-IF. As mentioned earlier, you must be a member of USB-IF to attend the plugfest workshops. There are advantages and disadvantages to each method.

Table 13. Comparing Test Labs Versus Plugfests

Independent Test Lab	USB-IF Plugfest
No personal required to be present during testing	One to four days of an engineer's or technician's time is required to attend the event
~\$2K-6K cost per item	Free with USB-IF membership (excluding time, travel, and so on)
Consulting available for a fee	USB experts available to help with debug, but resources are shared among other attendees
Testing can occur at any time	Testing events only held four or five times a year
Exposed to less prototype hardware.	Exposed to prototype hosts, hubs, and drivers
More secrecy	Less secrecy

Regardless of which testing method is chosen, the checklists completed earlier will be submitted officially at this point in time. The compliance testing process will differ based on the product that is being certified. These requirements can be found on the USB-IF website. In the instance of a Full-Speed device, such as PSoC, the following must be completed to earn certification: Electrical Tests, Interoperability Tests, and Functional Tests, all of which are described in the following sections.

Functional Tests: Also referred to as —Device Framework Tests||, these tests evaluate a series of items such as stress testing the device, Chapter 9 compliance tests, specific USB class tests, and general ability to function after a suspend condition. The primary tool used in this testing is called USBCV, which is discussed further in the Preparing for Certification section. Additionally, in the instance of the device being a HID, another set of tests are performed to ensure compliance to the USB-IF HID specification.

Electrical Tests: These tests check the device's compliance to the USB electrical specification by evaluating the signal quality, ensuring the device does not cause back voltage on the bus, and that suspend/resume/reset commands are responded to within the time limits documented in the USB specification. The more commonly recognized result of the electrical testing is the USB eye diagram as shown in Figure 56. This diagram is created on certain high-end oscilloscopes and show the signal quality of a device by testing the rise time, fall time, undershoot, overshoot, and D+ and D- line jitter.

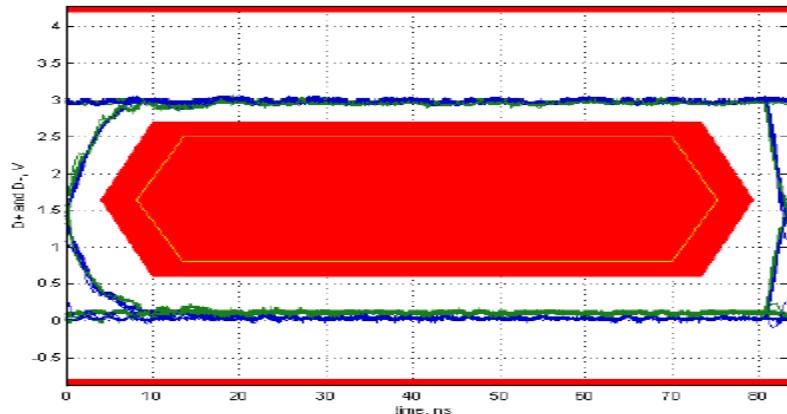


Figure 56. USB Eye Test Diagram

During the electrical tests, an oscilloscope captures the inrush current into the device, as shown in Figure 57. The purpose of the test is to ensure that an excessive amount of current is not drawn from the hub, and if so, to ensure that the proper current surge limiting circuitry exists.

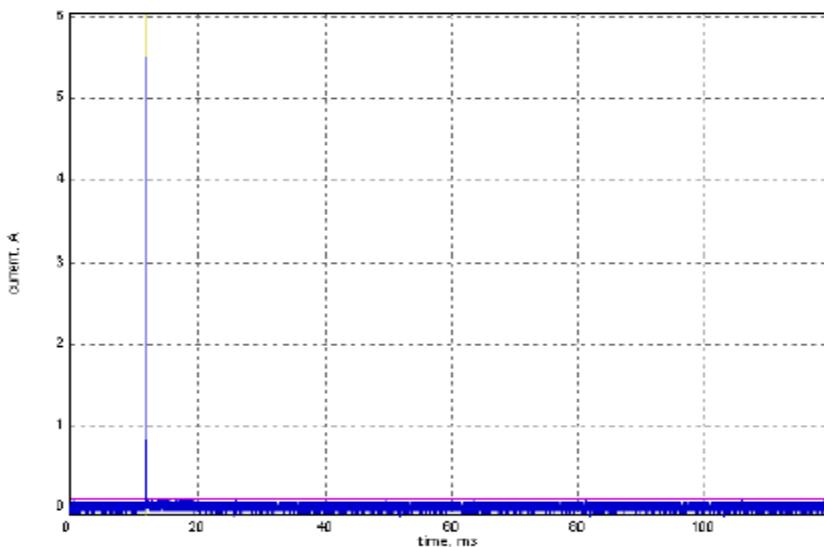


Figure 57. USB In-Rush Current Measurement

Performing the electrical testing uses a series of different adapter boards and test hardware. Particularly, testing is performed using high-end oscilloscopes. USB-IF provides a list of compliance test procedures, depending on the oscilloscope used, on their website.

Interoperability Tests: These tests evaluate the device's ability to interact with a host and coexist with other USB devices. A device can play with many combinations of devices, hosts, and hubs.

Various procedures are available for interoperability depending on whether the device is a Low-Speed, Full-Speed, High-Speed, or SuperSpeed device.

There is a term that is often heard in the USB community that is associated with interoperability. That term is —Gold Tree|| and is an arrangement of USB peripherals, for use in testing that consists of the following characteristics:

The bus contains isochronous, bulk, interrupt, and control traffic.

The bus contains both Full-Speed and High-Speed traffic.

The device is nested behind five levels of USB hubs.

The device is up to 30 meters from the host.

The host contains EHCI, UHCI, and OHCI controllers for testing.

To create the High-Speed and Full-Speed bus traffic, discussed in bullet point 2, the interoperability test also defines what test devices are to be included in the gold tree. The interoperability document even gives specifics on the manufacturer and model of the hardware to be used in testing. From a high level, the attached hardware includes a video camera (such as a web camera), a mass storage device (such as a USB hard drive), a flash media drive (Flash Drive or Jump Drive), a keyboard, and a mouse. All of this hardware is pieced together to create the web shown in the following figure.

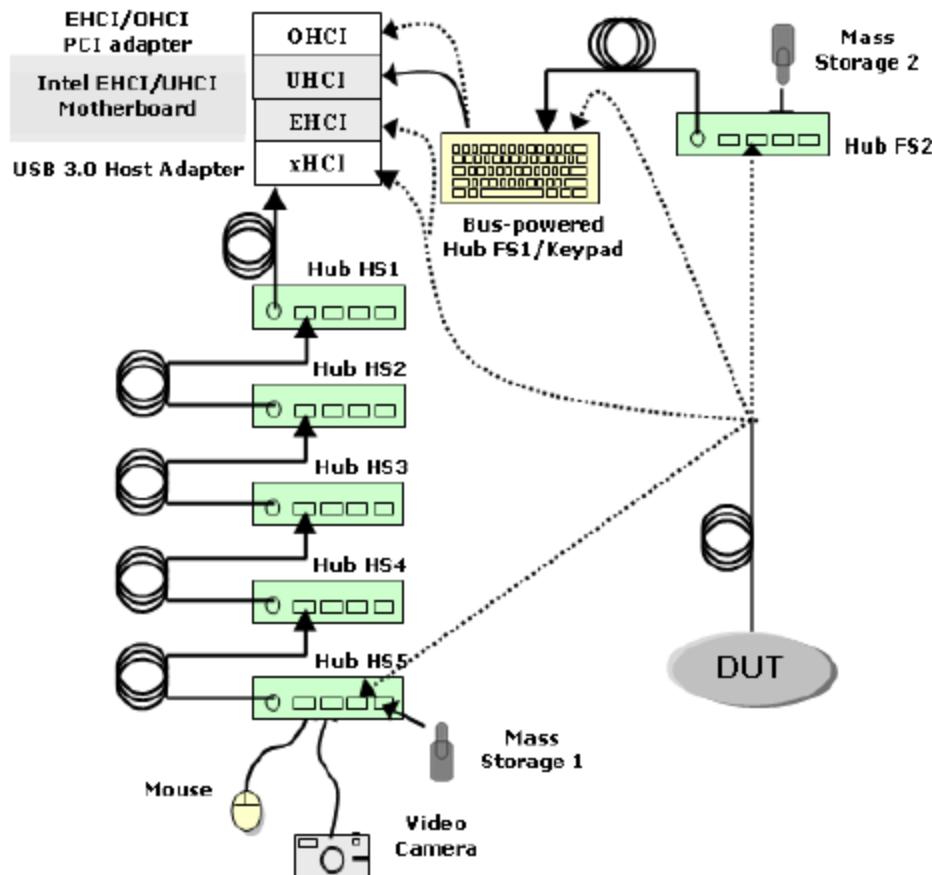


Figure 58. USB Gold Tree Attachment Points

The process of interoperability testing consists of multiple steps.

Step 1: Hot Detach & Reattach: Detach the device under test and reattach it to the same hub port.

Step 2: Topology Change - Detach the device under test and attach it to a different port location.

Step 3: Warm Boot - Restart the host/system through the Windows Start Menu (Start > Shutdown > Restart)

Step 4: Cold Boot – Restart the host/system through the Windows Start Menu (Start > Shutdown > Shutdown).

Restart the PC by using the physical power button on the PC.

Note: The reason to perform a cold boot and a warm boot is that a warm boot typically does not go through the complete boot process. Often, the Power-On Self-Test (POST) will be skipped. In addition, a warm boot may not reset all onboard devices. The remaining tests of interoperability occur in various system power states, solely with regards to sleeping states. To be more specific, interoperability tests S1 and S3 states.

- S0: Working State
- S1/S2: CPU Stopped
- S3: Suspend to RAM
- S4: Suspend to Disk (known as Hibernate)

Microsoft provides a more detailed description of what exactly the PC is doing in these various states in their System Sleeping States page. These states are entered when the PC is put into Sleep Mode. However, deciding whether the PC goes into S1 or S3 is determined by the BIOS for the host PC. The ACPI sleep type is typically found in the Power Management Setup of the BIOS itself, as shown in Figure 59. Thus, between the S1 and S3 tests, which will be described shortly, the test PC needs to be booted, entered into the BIOS configuration menu, and have the ACPI sleep mode configured properly as shown in Figure 59.

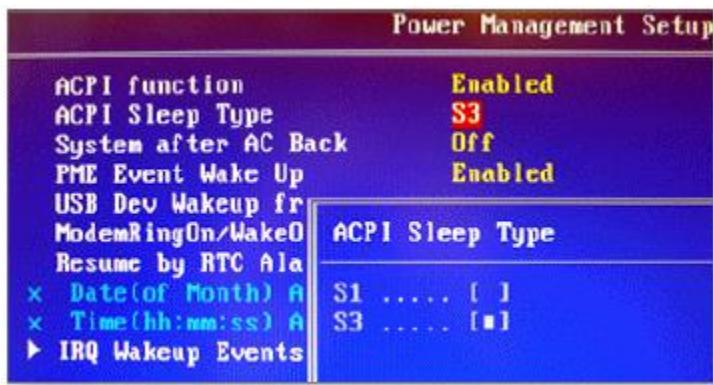


Figure 59. BIOS Selection of ACPI Sleep Mode

Step 5: Active S1 Suspend - With the BIOS configured for S1 mode, place the system into a sleep state. Note that the device should be in an active state prior to initiating the suspend state. Wake the system and confirm that the operation initiated prior to suspend continues without error. If the device being tested supports Remote Wakeup, then that approach should be used to resume the system.

Step 6: Inactive S1 Suspend - With the BIOS still configured for S1 mode, place the system into a sleep state while the device is in an idle state. Wake the system and confirm that the device under test still functions as intended when commanded. If the device being tested supports Remote Wakeup, then that approach should be used to resume the system.

Step 7: Active S3 Suspend - Configure the BIOS for S3 mode and place the system into a sleep state, similar to Step 5. Note that the device should be in an active state prior to initiating the suspend state. Wake the system and confirm that the device under test still functions as

intended. Remote wakeup should be disabled during this test. When testing is complete and checklists submitted, in a matter of a couple weeks, you can find a device listed on the USB-IF Integrators List. The following image provides an example of a listed device. In this case, the tested PSoC 3 can be seen.

Name	Company	TID	Categories	Added
PSoC 3 CY8C3866AXI-040	Cypress Semiconductor	40770053	Development > Peripheral Silicon > Low/Full Speed > Other	29 Apr 2011 00:31:00

Figure 60. PSoC 3 Device Shown on Integrators List

In the event that a device exceeded a specification parameter, there is still a way to pass compliance testing overall. That way is through the use of waivers. Waivers allow devices to qualify for the Integrator's List when they are slightly out of spec. For example, the current High-Speed test procedure allows devices to pass the High-Speed electrical test if they fail in the fifth hub tier. Waivers are temporary agreements between the device manufacturer and the USB I/F. As the current practices of USB design improve, waivers are removed. Waiver decisions are made by the Compliance Review Board.

15.1.3 Preparing for Certification

Before you have a device officially certified, there are multiple applications provided by USB-IF that can be used to test the device compliance. These applications are located in the Tools section of their website. Two of the more commonly used devices with USB peripherals are USBCV and USBET. Both these tools are available as free downloads from the USB-IF website. USBCV is a command-verifier application that is used to test the compliance with Chapter 9 of the USB specification (Device Framework) and tests the compliance with class specifications (such as HID). Figure 61 is a screen capture of USBCV.

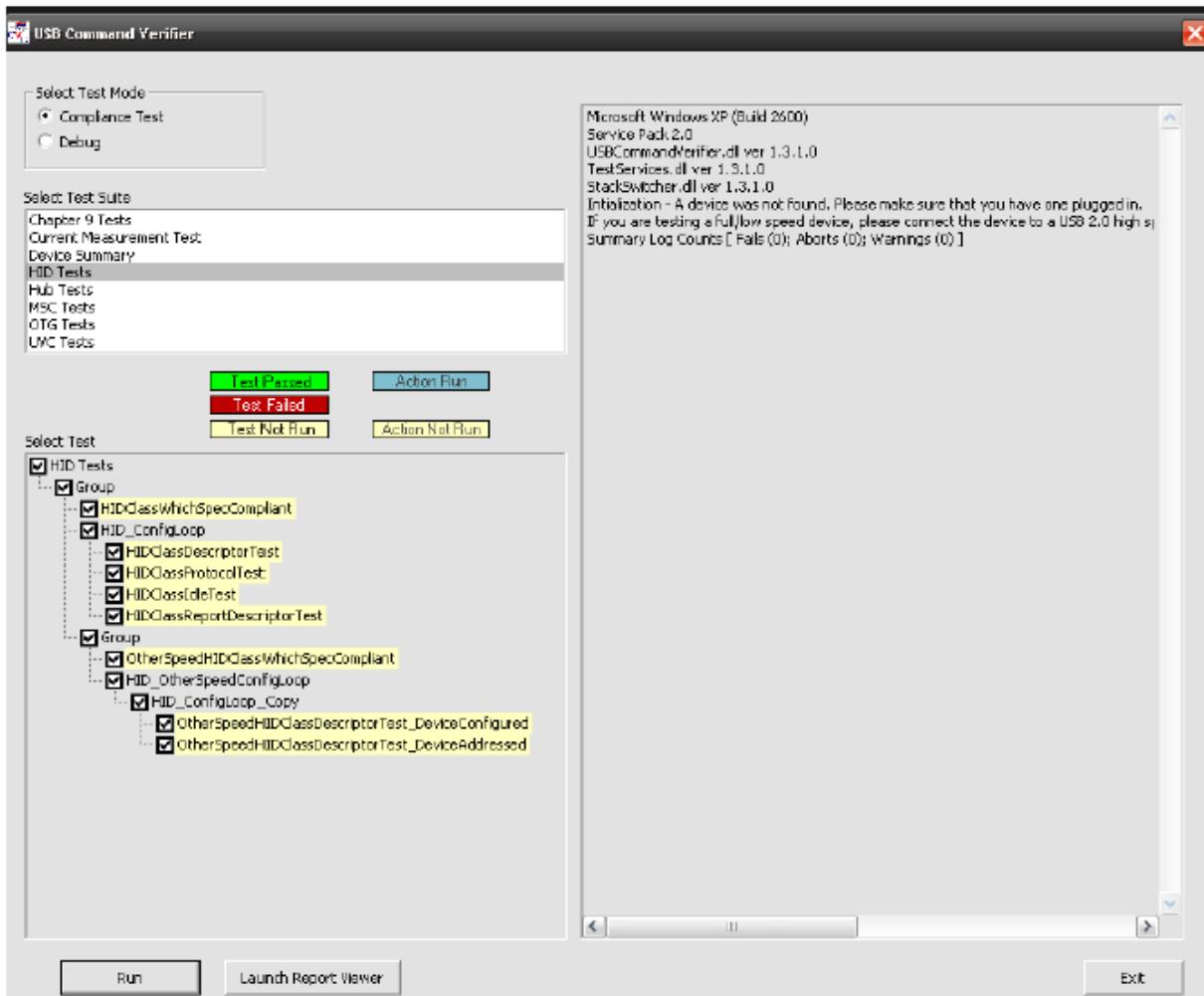


Figure 61. USBCV Application

USBET and **USBHSET** are tools that test electrical properties of a device. This includes the ability to test inrush and suspend mode current, and D+/D- signal quality. **USBHSET** tests the electrical characteristics of a high-speed device, and is an excellent tool for sending certain device commands. You do not need a high-speed device when using **USBHSET**. Not having that dependency makes this application extremely useful for testing full-speed devices. Figure 62 and Figure 63 show a screen capture of these applications. There are also additional tools called **USBHTT**, **OPT**, and **OET**, which are intended for hubs and USB On-The-Go devices.

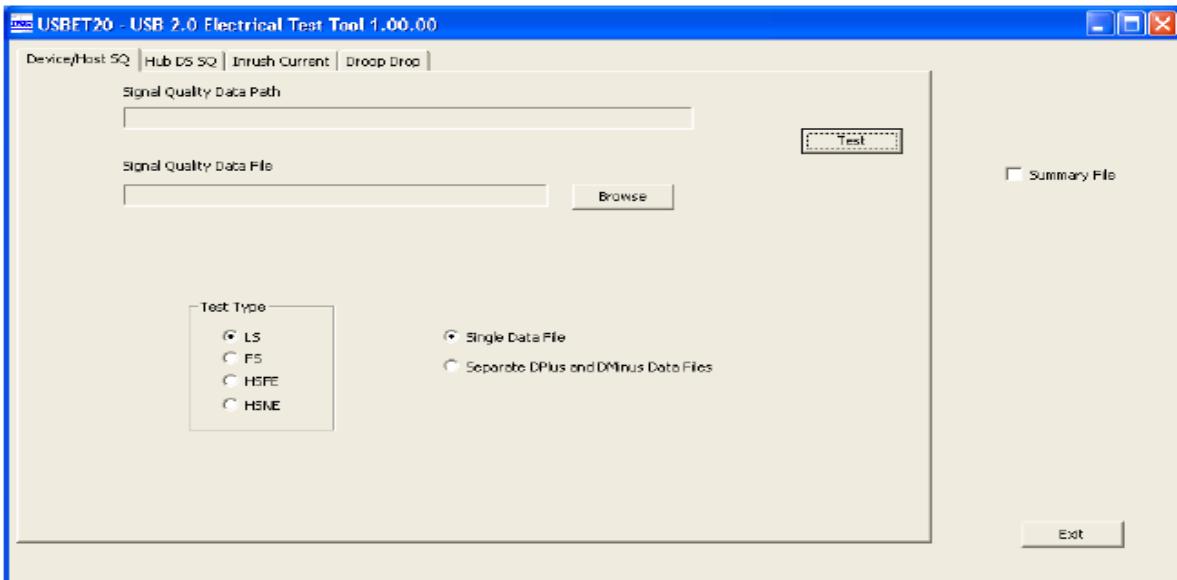


Figure 62. USBET Application

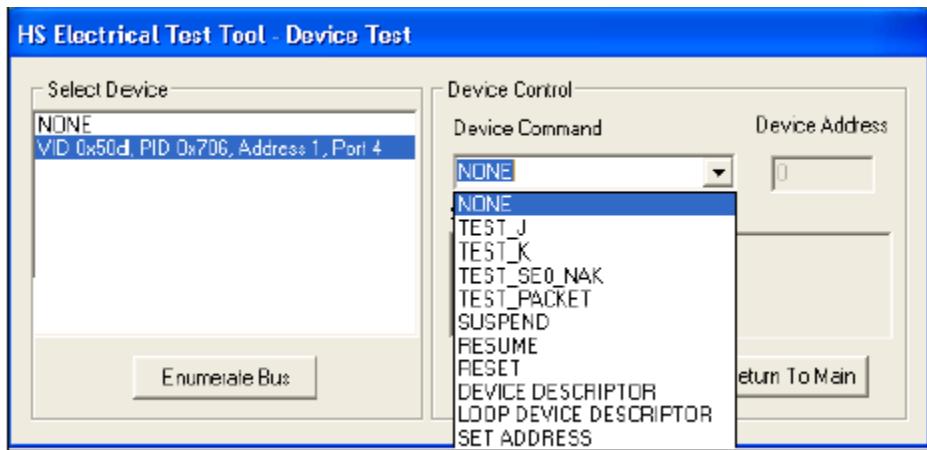


Figure 63. USBHSET Application

15.1.4 Qualification by Similarity

One question that might be on your mind is with regards to having different products that are extremely similar. What happens to them? USB-IF has a special circumstance for that called Qualification by Similarity, of which the details from the USB-IF website is as follows: When products are very similar, testing of one product may also allow other products to be added to the Integrators List. Many OEMs buy USB interface boards that are already on the Integrators List and qualify by similarity. However, if significant differences exist between products, testing of each is required. The definition of significant differences is debatable and the final judgment is

the responsibility of the compliance review board, which reports to the USB-IF board of directors. As decisions are made on what are significant differences, rules of thumb are listed at <http://www.usb.org/developers/compliance/>. The ultimate responsibility for making sure that various production product models do not have significant differences from the product samples tested lies with each vendor. Audits by USB-IF that reveal discrepancies between shipping product and samples tested are cause for retest. The effect on rights to use the USB-I/F logo is covered in the standard logo license agreement.

Retest required:

Microcontroller design change (new architecture or new product family).
Connector footprint on PCB.

Retest not required:

Product packaging changes (color, shape, and so on)
Microcontroller vendor change (no board layout change, no firmware change). Retest not required only if new microcontroller is on the Integrators List.
Microcontroller firmware change (changes in fully modular code not associated with USB functions).
Connector color and aesthetics.

15.2 Microsoft Hardware Certification Testing

When you connect a USB device into Microsoft Windows computer, you often receive a warning message that the device does not have a digitally signed driver. These warnings are similar to those seen in Figure 64.

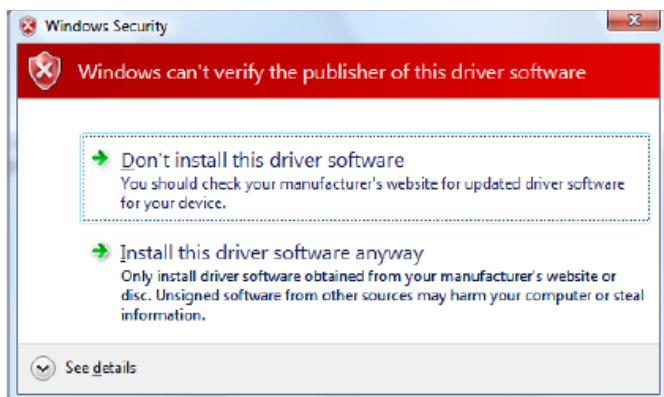


Figure 64. Windows 7 Driver Warning

These warnings are a result of device developers not submitting their device for Windows Hardware Quality Labs testing, also known as WHQL or Windows Logo testing. WHQL testing is Microsoft's test plan for USB hardware or software to verify that the device does not cause compatibility issues with Windows that may cause Windows to crash or fail to function properly. The program covers the certification for the following Windows platforms: Windows 8/8.1, Windows 7, Windows Vista, Windows Server 2012/2012 R2, and Windows Server 2008/2008R2. When a logo testing is performed on a device or driver, it goes through a specific series of tests that are tailored to that specific device class, such as a printer or a touch screen. Devices that do not fit into Microsoft's defined set of devices can still submit a driver as —Unclassified|| to obtain a digital signal. This will remove the error in Figure 64, but the product cannot display the Windows logo. Testing can be performed in one of two ways: either in-house by the company developing the product or by a third party. Microsoft does not perform the testing for you. If testing in-house, the process begins by downloading the Hardware Certification Kit (HCK) from Microsoft's website. This kit provides all the needed software, driver, and tools to tests and bundle the results to be sent to Microsoft. Note that this is a free download from Microsoft. After the toolkit is downloaded, use the installed software to perform the needed tests and put together a submission bundle that can be given to Microsoft. Note that the test environment needed to perform the testing will require a dedicated PC with a copy of Windows Server that is responsible for administrating and controlling the tests. Another PC with the Windows operating system you intend to test against is also required. You will need to test against both 32-bit and 64-bit versions of the Windows software. Additionally, at the time of this writing, Microsoft charges \$250 per Windows family. The HCK will provide a guide that walks you through the test configuration and required tests.

The result of WHQL testing, if all tests are passed, is to create a submission bundle using the HCK that will be given to Microsoft. After Microsoft receives and accepts the submission, you will receive a signed certification file from Microsoft that is included with the diver and does not cause these warning messages. Passing WHQL testing allows devices to bear some of the following logos in Figure 65 depending on the device and the OS that was tested.

Figure 65. Microsoft Windows Certified Logos



The need for WHQL testing has increased in the last few years with the release of 64-bit Windows Vista and 64-bit Windows 7, which require all drivers installed in the 64-bit system to be signed unless you enter a special startup mode that disables the enforcement. More information on Windows Hardware Certification and the process is available on Microsoft's website here.

16.0 Summary

At this point, you should have a foundation to understand other USB material with more confidence. While this application note contains some theory, several other Cypress USB application notes show you how to use the information taught in this application note. See the Related Resources section for some of those resources. Additional information regarding anything that was mentioned in this application note is also contained in the USB specification. I strongly recommend that for anything you wish to explore and learn about, you begin there.

Appendix A

A.1 Example PSoC 3 Full-Speed USB Device Descriptors

```
*****
Device Descriptors
*****
uint8 CYCODE USBFS_1_DEVICE0_DESCR[] = {
/* Descriptor Length */ 0x12u,
/* DescriptorType: DEVICE */ 0x01u,
/* bcdUSB (ver 2.0) */ 0x00u, 0x02u,
/* bDeviceClass */ 0x00u,
/* bDeviceSubClass */ 0x00u,
/* bDeviceProtocol */ 0x00u,
/* bMaxPacketSize0 */ 0x08u,
/* idVendor */ 0xB4u, 0x04u,
/* idProduct */ 0x34u, 0x12u,
/* bcdDevice */ 0x00u, 0x00u,
/* iManufacturer */ 0x01u,
/* iProduct */ 0x02u,
/* iSerialNumber */ 0x00u,
/* bNumConfigurations */ 0x01u
};
```

```

*****
Config Descriptor
*****
uint8 CYCODE USBFS_1_DEVICE0_CONFIGURATION0_DESCR[] = {
/* Config Descriptor Length */ 0x09u,
/* DescriptorType: CONFIG */ 0x02u,
/* wTotalLength */ 0x19u, 0x00u,
/* bNumInterfaces */ 0x01u,
/* bConfigurationValue */ 0x01u,
/* iConfiguration */ 0x00u,
/* bmAttributes */ 0x80u,
/* bMaxPower */ 0x32u,
*****
Interface Descriptor
*****
/* Interface Descriptor Length */ 0x09u,
/* DescriptorType: INTERFACE */ 0x04u,
/* bInterfaceNumber */ 0x00u,
/* bAlternateSetting */ 0x00u,
/* bNumEndpoints */ 0x01u,
/* bInterfaceClass */ 0xFFu,
/* bInterfaceSubClass */ 0x00u,
/* bInterfaceProtocol */ 0x00u,
/* iInterface */ 0x00u,
*****
Endpoint Descriptor
*****
/* Endpoint Descriptor Length */ 0x07u,
/* DescriptorType: ENDPOINT */ 0x05u,
/* bEndpointAddress */ 0x81u,
/* bmAttributes */ 0x02u,
/* wMaxPacketSize */ 0x40u, 0x00u,
/* bInterval */ 0x00u
};
*****
String Descriptor Table
*****

```

uint8 CYCODE USBFS_1_STRING_DESCRIPTORs[] = {

```

/***********************
Language ID Descriptor
***********************/

/* Descriptor Length */ 0x04u,
/* DescriptorType: STRING */ 0x03u,
/* Language Id */ 0x09u, 0x04u,
/***********************/

String Descriptor: "Cypress Semiconductor"
/***********************/

/* Descriptor Length */ 0x2Cu,
/* DescriptorType: STRING */ 0x03u,
'C', 0,'y', 0,'p', 0,'r', 0,'e', 0,'s', 0,'s', 0,' ', 0,'S', 0,'e', 0
,'m', 0,'i', 0,'c', 0,'o', 0,'n', 0,'d', 0,'u', 0,'c', 0,'t', 0,'o', 0
,'r', 0,
/***********************/

String Descriptor: "USB Example"
/***********************/

/* Descriptor Length */ 0x18u,
/* DescriptorType: STRING */ 0x03u,
'U', 0,'S', 0,'B', 0,' ', 0,'E', 0,'x', 0,'a', 0,'m', 0,'p', 0,'l', 0
,'e', 0,
/***********************/

/* Marks the end of the list. */ 0x00u};

/******************/0,

```

Appendix B

B.1 Bus Analyzer Capture of USB Enumeration (Example)

USB Bus Analyzer Capture of USB Enumeration (Example)											
All Commands Initially Sent to Address 0 Enumeration Occurs Through Endpoint 0											
Reset: Transaction F 0 S 0x84 0 0 D>H S D GET_DESCRIPTOR 26.308 ms Idle 1083 /* Reset 1st Descriptor Request to by Hub learn bMaxPacketSize0											
Setup Transaction: Transaction F 1 S 0x96 0 0 T DATA ACK 1.999 ms /* Descriptor Length */ /* DescriptorType: DEVICE */ /* bcdUSB (ver 2.0) */ /* bDeviceClass */ /* bDeviceSubClass */ /* bDeviceProtocol */ /* bMaxPacketSize0 */ 84 Bytes Requested											
Data Transaction: Transaction F 2 S 0x87 0 0 T DATA ACK 0x4B 25.416 ms Idle 5844 /* 2nd Reset After Receiving Descriptor */											
Handshake Transaction: Transaction F 3 S 0x84 0 0 H>D S D SET_ADDRESS 999.750 µs New Address 1 Is Assigned											
Transaction F 4 S 0x96 0 0 T DATA ACK 45.989 ms Re-Request of Device Descriptor											
Transaction F 5 S 0x84 1 0 D>H S D GET_DESCRIPTOR 999.750 µs /* Descriptor Length */ /* DescriptorType: DEVICE */ /* bcdUSB (ver 1.0) */ /* bDeviceClass */ /* bDeviceSubClass */ /* bDeviceProtocol */ /* bMaxPacketSize0 */ /* idVendor */ /* idProduct */ /* bcdDevice */ /* iManufacturer string number */ /* iProduct string number */ /* iSerialNumber string number */ /* bNumConfigurations */ 18 Bytes Requested											
New Address 1 Is Used from Now On											
Transaction F 6 S 0x96 1 0 T DATA ACK 999.750 µs											
Transaction F 7 S 0x96 1 0 T DATA ACK 999.750 µs											
Transaction F 8 S 0x96 1 0 T DATA ACK 999.750 µs											
Transaction F 9 S 0x87 1 0 T DATA ACK 1.999 ms											
Setup Transaction: Transaction F 10 S 0x84 1 0 H>D S D GET_DESCRIPTOR 999.750 µs Configuration type 0x0000 9 0x4B Request for Configuration Descriptor - 1st Request (For more details on Setup/Control transfers see AN68377)											
Packet # F Sync SETUP ADDR ENDP CRC5 EOP Idle 165 S 00000001 0xB4 1 0 0x17 2.75 2											
Packet # F Sync DATA0 DATA CRC16 EOP Idle 166 S 00000001 0xC3 80 06 00 02 00 00 09 00 0x7520 3.00 5											
Packet # F Sync ACK EOP Idle 167 S 00000001 0x4B 3.00 11798											
Setup Transaction: Transaction F 11 S 0x96 1 0 T DATA ACK 999.750 µs /* Config Descriptor Length */ /* DescriptorType: CONFIG */ /* wTotalLength */ /* bNumInterfaces */ /* bConfigurationValue */ /* iConfiguration */ /* bmAttributes, bus power */ /* bMaxPower */											
Transaction F 12 S 0x96 1 0 T DATA ACK 999.750 µs											
Transaction F 13 S 0x87 1 0 T DATA ACK 1.999 ms											

Request for Configuration (2nd Request), Interface, and Endpoint Descriptors

Transaction	F	SETUP	ADDR	ENDP	D	T	R	bRequest	wValue	wIndex	wLength	ACK	Time
14	S	0xB4	1	0	D->H	S	D	GET_DESCRIPTOR	CONFIGURATION type	0x0000	255	0x4B	999.750 µs
15	S	0x96	1	0	T	DATA			0x4B				
16	S	0x96	1	0	T	DATA			0x4B				
17	S	0x96	1	0	T	DATA			0x4B				
18	S	0x96	1	0	T	DATA			0x4B				
19	S	0x87	1	0	T	DATA			0x4B				
20	S	0xB4	1	0	D->H	S	D	GET_DESCRIPTOR	DEVICE_QUALIFIER type	0x0000	10	0x4B	999.750 µs
21	S	0x96	1	0	STALL				0x78				
													This is a Full Speed only device. As a result, a STALL (request error) is issued.
22	S	0xB4	1	0	D->H	S	D	GET_DESCRIPTOR	STRING type, LANGID codes requested	Language ID 0x0000	255	0x4B	999.833 µs
23	S	0x96	1	0	T	DATA			0x4B				
24	S	0x87	1	0	T	DATA			0x4B				
25	S	0xB4	1	0	D->H	S	D	GET_DESCRIPTOR	STRING type, Index 2	Language ID 0x0409	255	0x4B	999.750 µs
26	S	0x96	1	0	T	DATA			0x4B				
27	S	0x96	1	0	T	DATA			0x4B				
28	S	0x96	1	0	T	DATA			0x4B				
29	S	0x96	1	0	T	DATA			0x4B				

```
/* Config Descriptor Length */ 0x09u,
/* DescriptorType: CONFIG */ /* 0x02u,
/* wTotalLength */ /* 0x19u, 0x00u,
/* bNumInterfaces */ /* 0x01u,
/* bConfigurationValue */ /* 0x01u,
/* iConfiguration */ /* 0x00u,
/* bmAttributes, bus power */ /* 0x00u,
/* bMaxPower */ /* 0x32u,
/* Interface Descriptor Length */ /* 0x09u,
/* DescriptorType: INTERFACE */ /* 0x04u,
/* bInterfaceNumber */ /* 0x00u,
/* bAlternateSetting */ /* 0x00u,
/* bNumEndpoints */ /* 0x01u,
/* bInterfaceClass */ /* 0x07u,
/* bInterfaceSubClass */ /* 0x00u,
/* bInterfaceProtocol */ /* 0x00u,
/* iInterface */ /* 0x00u,
/* Endpoint Descriptor length */ /* 0x07u,
/* DescriptorType: ENDPOINT */ /* 0x05u,
/* bEndpointAddress, BPL IN */ /* 0x81u,
/* bEndpointAddress BULK transfer type */ /* 0x02u,
/* wMaxPacketSize 64 bytes */ /* 0x40u, 0x00u,
/* bInterval */ /* 0x00u,
```

Request for Device_Qualifier Descriptor.

This is a Full Speed only device. As a result, a STALL (request error) is issued.

1st String Descriptor Request (Language ID)

```
/* Descriptor Length */ 0x04u,
/* DescriptorType: STRING */ 0x03u,
/* Language Id */ 0x09u, 0x04u,
```

```
/* Descriptor Length */ 0x10u,
/* DescriptorType: STRING */ 0x03u,
'U', 0, 'S', 0, 'R', 0, ' ', 0, 'E', 0, 'x', 0, 'a', 0, 'n', 0, 'p', 0, 'l', 0, ' ', 0,
```

```
UTF16 Conversion:
U = 55h a = 61h
S = 53h m = 6dh
R = 42h p = 70h
E = 45h l = 6ch
x = 78h e = 65h
Space = 20h
```

Transaction F	OUT	ADDR	ENDP	T	DATA	ACK	Time
30 S	0x87	1	0	1	0x4B	1.999 ms	
Transaction F	SETUP	ADDR	ENDP	D	T R	bRequest	wValue
31 S	0xB4	1	0	D->H	S D	GET_DESCRIPTOR	STRING type, LANGID codes requested
Transaction F	IN	ADDR	ENDP	T	DATA	ACK	Time
32 S	0x96	1	0	1	04 03 09 04	0x4B	1.999 ms
Transaction F	OUT	ADDR	ENDP	T	DATA	ACK	Time
33 S	0x87	1	0	1	0x4B	1.999 ms	
Transaction F	SETUP	ADDR	ENDP	D	T R	bRequest	wValue
34 S	0xB4	1	0	D->H	S D	GET_DESCRIPTOR	STRING type, Index 2
Transaction F	IN	ADDR	ENDP	T	DATA	ACK	Time
35 S	0x96	1	0	1	18 03 55 00 53 00 42 00	0x4B	999.833 µs
Transaction F	IN	ADDR	ENDP	T	DATA	ACK	Time
36 S	0x96	1	0	0	20 00 45 00 18 00 61 00	0x4B	999.750 µs
Transaction F	IN	ADDR	ENDP	T	DATA	ACK	Time
37 S	0x96	1	0	1	6D 00 70 00 6C 00 65 00	0x4B	999.750 µs
Transaction F	IN	ADDR	ENDP	T	DATA	ACK	Time
38 S	0x96	1	0	0	0x4B	1.999 ms	
Transaction F	OUT	ADDR	ENDP	T	DATA	ACK	Time
39 S	0x87	1	0	1	0x4B	29.993 ms	
Transaction F	SETUP	ADDR	ENDP	D	T R	bRequest	wValue
40 S	0xB4	1	0	D->H	S D	GET_DESCRIPTOR	DEVICE type
Transaction F	IN	ADDR	ENDP	T	DATA	ACK	Time
41 S	0x96	1	0	1	12 01 00 02 00 00 00 C6	0x4B	999.750 µs
Transaction F	IN	ADDR	ENDP	T	DATA	ACK	Time
42 S	0x96	1	0	0	54 04 34 12 00 00 01 C2	0x4B	999.750 µs
Transaction F	IN	ADDR	ENDP	T	DATA	ACK	Time
43 S	0x96	1	0	1	00 01	0x4B	999.750 µs
Transaction F	OUT	ADDR	ENDP	T	DATA	ACK	Time
44 S	0x87	1	0	1	0x4B	1.999 ms	
Transaction F	SETUP	ADDR	ENDP	D	T R	bRequest	wValue
45 S	0xB4	1	0	D->H	S D	GET_DESCRIPTOR	CONFIGURATION type
Transaction F	IN	ADDR	ENDP	T	DATA	ACK	Time
46 S	0x96	1	0	1	09 02 19 00 01 01 00 80	0x4B	999.750 µs
Transaction F	IN	ADDR	ENDP	T	DATA	ACK	Time
47 S	0x96	1	0	0	32	0x4B	999.833 µs
Transaction F	OUT	ADDR	ENDP	T	DATA	ACK	Time
48 S	0x87	1	0	1	0x4B	1.999 ms	

Re-Request of Descriptors by Driver

Transaction F	SETUP	ADDR	ENDP	D	T R	bRequest	wValue	wIndex	wLength	ACK	Time
49 S	0xB4	1	0	D->H	S D	GET_DESCRIPTOR	CONFIGURATION type	0x0000	25	0x4B	999.833 µs
Transaction F	IN	ADDR	ENDP	T	DATA	ACK	Time				
50 S	0x96	1	0	1	09 02 19 00 01 01 00 80	0x4B	999.833 µs				
Transaction F	IN	ADDR	ENDP	T	DATA	ACK	Time				
51 S	0x96	1	0	0	32 09 04 00 00 01 FF 00	0x4B	999.750 µs				
Transaction F	IN	ADDR	ENDP	T	DATA	ACK	Time				
52 S	0x96	1	0	1	00 00 07 05 81 02 40 00	0x4B	999.750 µs				
Transaction F	IN	ADDR	ENDP	T	DATA	ACK	Time				
53 S	0x96	1	0	0	00	0x4B	999.750 µs				
Transaction F	OUT	ADDR	ENDP	T	DATA	ACK	Time				
54 S	0x87	1	0	1	0x4B	1.999 ms					
Transaction F	SETUP	ADDR	ENDP	D	T R	bRequest	wValue	wIndex	wLength	ACK	Time
55 S	0xB4	1	0	D->H	S D	GET_STATUS	0x0000	Device Status requested	2	0x4B	999.750 µs
Transaction F	IN	ADDR	ENDP	T	DATA	ACK	Time				
56 S	0x96	1	0	1	00 00	0x4B	999.750 µs				
Transaction F	OUT	ADDR	ENDP	T	DATA	ACK	Time				
57 S	0x87	1	0	1	0x4B	1.999 ms					
Transaction F	SETUP	ADDR	ENDP	D	T R	bRequest	wValue	wIndex	wLength	ACK	Time
58 S	0xB4	1	0	H->D	S D	SET_CONFIGURATION	New configuration 1	0x0000	0	0x4B	999.750 µs
Transaction F	IN	ADDR	ENDP	T	DATA	ACK	Time				
59 S	0x96	1	0	1	0x4B	1.999 ms					

Re-Request of Descriptors by Driver (Continued)

Request for Device Status

Configuration is Selected.

Configuration 1

Device is Ready for Use!

USB 3.0

1.0 USB 3.0 OVERVIEW

USB 3.0 enables an increased data rate of 5 Gbps, reduces power consumption, and is backward-compatible with USB 2.0. The USB Specification, published by USB-IF, can be found here. Figure 1 shows the USB 3.0 architecture.

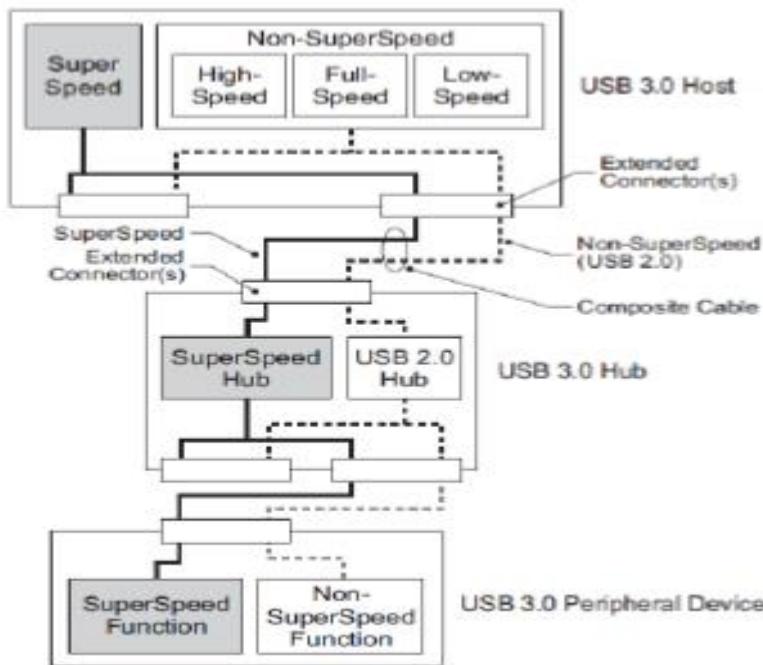


Fig 1 USB 3.0 Dual Bus Architecture

The physical interface of USB 3.0 consists of two differential pairs and a ground for SuperSpeed transfers in addition to the USB 2.0 connections. This allows USB 3.0 to ensure backward compatibility with USB 2.0.

1.1 Electrical Interface

The USB 3.0 pinout is different from that of USB 2.0. In addition to the VBUS, D-, D+, and GND pins required for USB 2.0, USB 3.0 has five additional pins – two differential pairs and one ground (GND_DRAIN). The two differential pairs are for SuperSpeed data transfer, supporting dual simplex SuperSpeed signaling. The GND_DRAIN pin is for drain wire termination, management of signal integrity, and EMI performance. Table 1 shows a description of the nine pins.

Pins Name	Description
VBUS	Power
D-	USB 2.0 differential pair
D+	
GND	Ground for power return
SSRX-	SuperSpeed receiver differential pair
SSRX+	
SSTX-	SuperSpeed transmit differential pair
SSTX+	
GND_DRAIN	Ground for signal return

Table 1. USB 3.0 Pin Description

1.2 Cables and Connectors

USB 3.0 has four additional data lines (SSRX+, SSRX-, SSTX+, SSTX-) for data transfer and one additional ground line for drain wire termination, signal integrity management, and EMI performance. Figure 2 shows the architecture of a USB 3.0 cable. Table 4 shows the description of these lines.

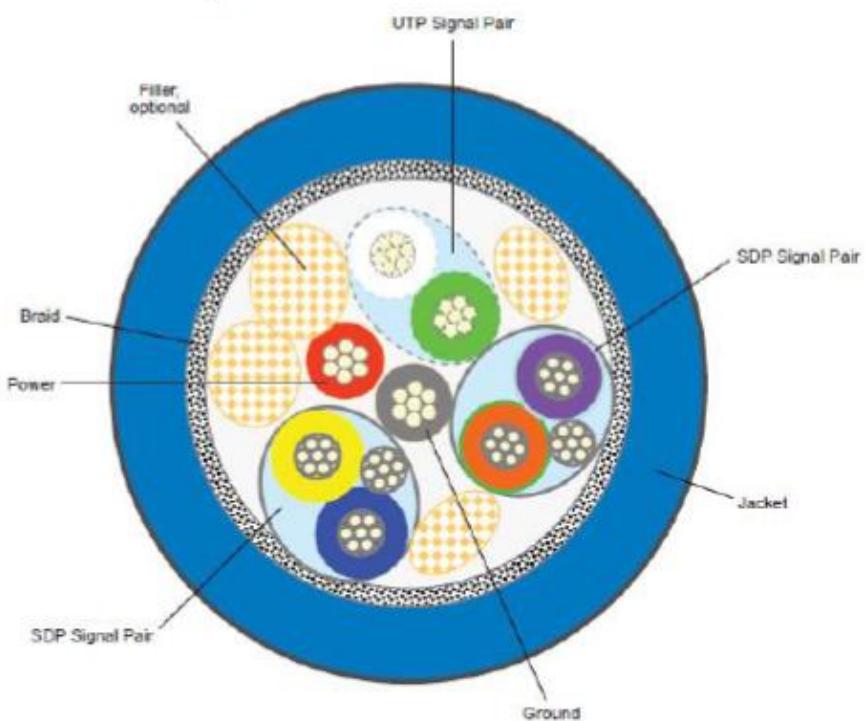


Fig 2 USB 3.0 Cable Architecture

Name	Description	Color
PWR	VBUS	Red
UTP_D-	USB 2.0 D-	White
UTP_D+	USB 2.0 D+	Green
Ground	Ground for power drain	Black
SDP1-	Shielded differential pair 1	Blue
SDP1+		Yellow
SDP1_Drain	Drain line for SDP1	
SDP2-	Shielded differential pair 2	Purple
SDP2+		Orange
SDP2_Drain	Drain line for SDP2	

Table 2 USB 3.0 Cable Description

The USB 3.0 Specification also defines the following connectors:

- **USB 3.0 Standard-A Plug and Receptacle**

The USB 3.0 Standard-A connector is defined in the SuperSpeed standard as the host connector. It is based on the design of the USB 2.0 Standard-A connector but has the additional SuperSpeed signals. A USB 3.0 Standard-A receptacle accepts either a USB 3.0 Standard-A plug or a USB 2.0 Standard-A plug. USB 3.0 capable Standard-A connectors use a unique color for easy identification. Figure 2 shows the color coding recommendation.

- **USB 3.0 Standard-B Plug and Receptacle**

The USB 3.0 Standard-B connector is defined for large, stationary peripherals, such as external hard drives and printers. The USB 3.0 Standard-B receptacle accepts either a USB 3.0 Standard-B plug or a USB 2.0 Standard-B plug. You cannot insert a USB 3.0 Standard-B plug into a USB 2.0 Standard-B receptacle.

- **USB 3.0 Powered-B Plug and Receptacle**

The USB 3.0 Powered-B connector enables a USB 3.0 device to provide power to a USB adapter without an external power supply. It is identical to the USB 3.0 Standard-B connector in form factor but has two more pins: one for power (DPWR) and one for ground (DGND). USB 3.0 Cable Description.

USB 3.0 Micro-B Plug and Receptacle

The USB 3.0 Micro-B connector is defined for small handheld devices.

- **USB 3.0 Micro-AB and USB 3.0 Micro-A Connectors**

The USB 3.0 Micro-AB receptacle is similar to the USB 3.0 Micro-B receptacle, except keying is different. It accepts a USB 3.0 Micro-A plug, a USB 3.0 Micro-B plug, a USB 2.0 Micro-A plug, or a USB 2.0 Micro-B plug. The USB 3.0 Micro-AB receptacle is allowed only on OTG products, which may function as either a host or a device. All other uses of the USB 3.0 Micro-AB receptacle are prohibited. The USB 3.0 Micro-A plug is similar to the USB 3.0 Micro-B plug, except for different keying and ID pin connections. The USB 3.0 Micro-A plug, the USB 3.0 Micro-AB receptacle, and the USB 3.0 Micro-B receptacle and plug, belong to the USB 3.0 Micro connector family. Their interfaces differ only in keying. Similar to the USB 2.0 Micro-A plug, the USB 3.0 Micro-A plug is defined only for OTG applications.

1.3 USB 3.0 Versus 2.0

USB 3.0 has a dual-bus architecture that supports USB 2.0 and 3.0. The following table shows the main differences between USB 3.0 and USB 2.0.

Feature	USB 2.0	USB 3.0
Data rate	480 Mbits/s (High Speed) 12 Mbits/s (Full Speed) 1.5 Mbits/s (Low Speed)	5.0 Gbits/s (SuperSpeed) 480 Mbits/s (High Speed) 12 Mbits/s (Full Speed) 1.5 Mbits/s (Low Speed)
Data interface	Half-duplex Two-wire differential signaling	Dual-simplex Four-wire differential signaling
Cable signal count	Four signals: - Two for USB 2.0 data (D, D–) - Two for VBUS and GND	Nine signals: - Four for SuperSpeed data - Two for USB 2.0 data (D, D–) - Three for VBUS and GND
Bus transaction protocol	Host directed Polled traffic flow Packets broadcast to all downstream devices No multiplexing of data streams	Host directed Asynchronous notifications Packets routed only to target device Multiple data streams possible for BULK transfers
Power management	Two modes - Active - Suspend	Four modes - Active (U0) - Idle, Fast (U1) - Idle, Slow (U2) - Suspend, Slow (U3)
Bus power	Low-power device : 100 mA High-power device : 500 mA	Low-power device : 150 mA High-power device : 900 mA
Port state	Port hardware detects connect events. System software uses port commands to transition the port into an enabled state.	Port hardware detects connect events and brings the port into operational state ready for SuperSpeed data communication
Maximum cable length	5 meters	Based on electrical specification. In practice 3 meters for 26 AWG copper
Data transfer types	Four data transfer types: CONTROL, BULK, INTERRUPT, and ISOCHRONOUS.	USB 2.0 types with SuperSpeed capabilities. BULK has streams capability.

Table 3 Differences Between USB 3.0 and USB 2.0

2.0 FX3 DVK Driver Installation on Windows

If you have not already installed the FX3 DVK kit on a Windows computer, the first time you connect the DVK to the computer you will see the following message.

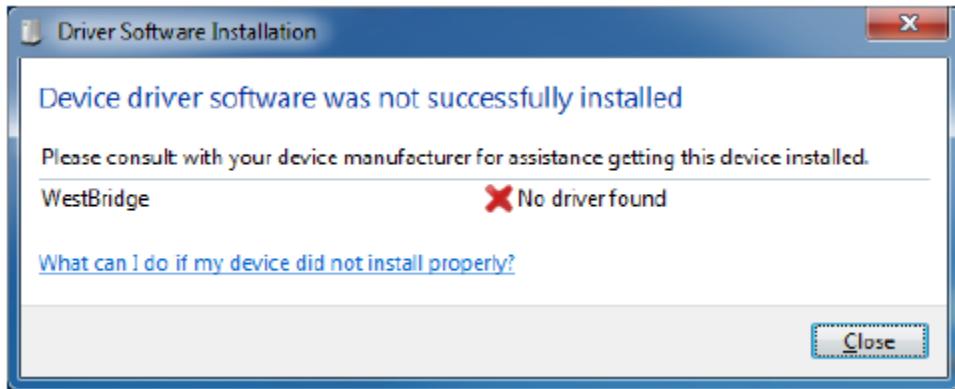


figure 3: Windows Message if Driver is Not Installed

Close the message box and navigate to Windows Device Manager. To do this, click the Windows **Start** button, rightclick on **Computer** in the right-hand column, and select **Properties** to bring up System Information. Then, click **Device Manager** at the top of the left column.

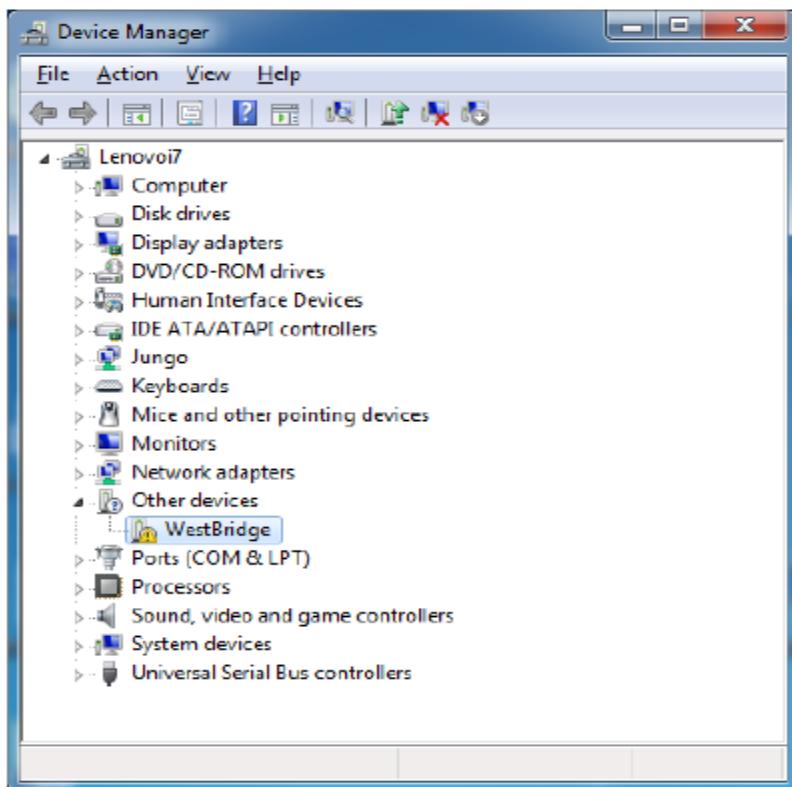


Figure 4: Windows Device Manager

Right-click on **WestBridge** and select **Browse my computer for driver software**. On a 64-bit Windows 7 machine, the driver is located at C:\Program Files (x86)\Cypress\EZ-USB FX3 SDK\1.3\driver\bin\win7\x64. Your SDK version number may be higher than 1.2. The Device Manager window should remove the WestBridge entry and identify the SDK board as the Cypress USB Bootloader (top entry).

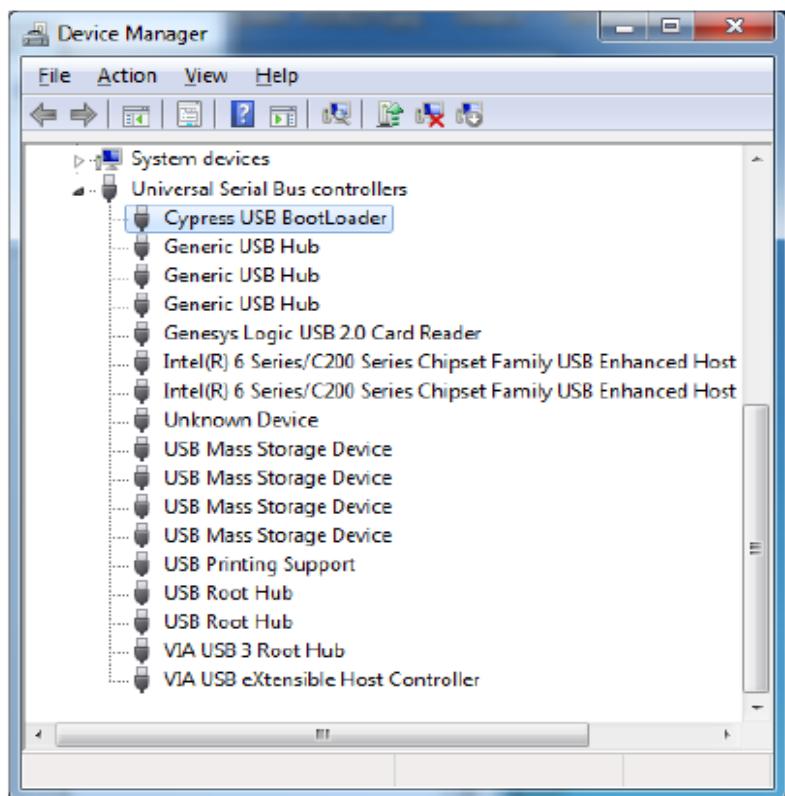


Figure 5: Cypress USB BootLoader in Device Manager After Driver Installation.

3.0 Introduction to FX3 DVK

Cypress' FX3 DVK provides the hardware that you need to get started. The PCB provides the necessary clocks and voltages for the FX3 as well as configurable I/O voltages. The DVK has high-speed connectors for interfacing with external devices. The DVK also implements various boot modes. An I2C EEPROM socket can be used to program and test booting from an I2C EEPROM. An on-board SPI flash chip allows programming and booting over the SPI bus. Cypress example projects supplied with the SDK contain firmware to program these devices using the DVK board. If you do not already have a DVK, you can find one here. Two important pieces of hardware, external to the FX3 DVK board, are the USB 3.0 host (typically the PC) and the external device connected to the GPIF II interface (such as an image sensor or FPGA). For details on using the FX3

DVK, refer to the DVK User Guide. Fig shows a picture of the FX3 DVK board with the key areas called out.

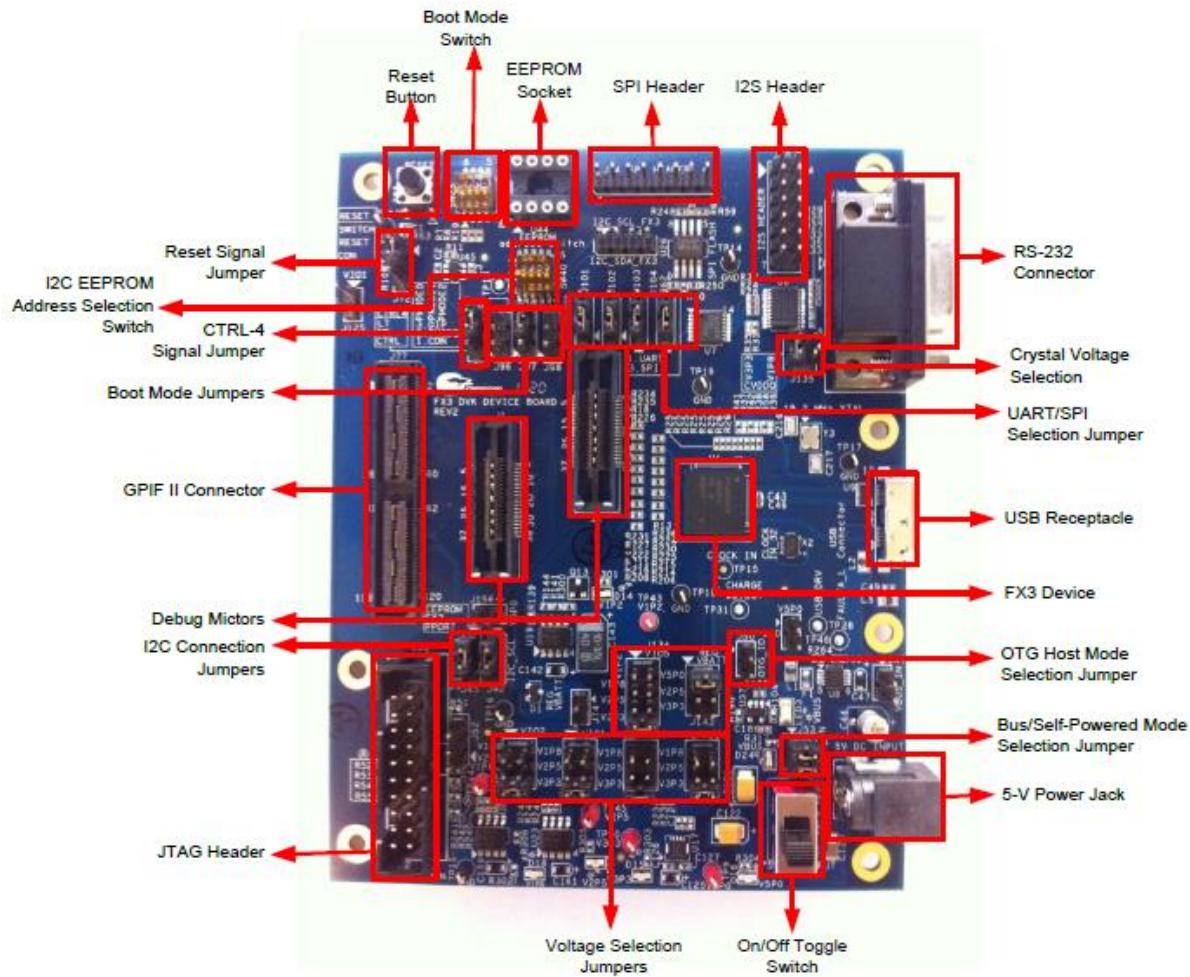


Figure 6: FX3 DVK Board

Table 6 shows the key jumpers and their default settings on the DVK board.

Jumper/Switch	Pins Shorted using Jumpers (Default Setting)	Function (Default Setting)
J101	1 and 2	GPIO_46=UART_RTS
J102	1 and 2	GPIO_47=UART_CTS
J103	1 and 2	GPIO_48=UART_TX
J104	1 and 2	GPIO_49=UART_RX
J136	3 and 4	VIO1(3.3 V)
J144	3 and 4	VIO2(3.3 V)
J145	3 and 4	VIO3(3.3 V)
J146	3 and 4	VIO4(3.3 V)
J134	4 and 5	VIO5(3.3 V)
J135	2 and 3	CVDDQ(3.3 V)
J143	1 and 6	VBATT(2.5 V)
J96 & SW25	2 and 3	<ul style="list-style-type: none"> • PMODE0 Pin state (ON/OFF) selection using SW25. • SW25.1=OFF
J97 & SW25	2 and 3	<ul style="list-style-type: none"> • PMODE0 Pin state (ON/OFF) selection using SW25. • SW25.2=OFF
J98	1 and 2	PMODE2 Pin Floating
J72	1 and 2	RESET
J42	Not Installed	GPIO_58=I2C_SCL
J45	Not Installed	GPIO_59=I2C_SDA
J100	1 and 2	GPIO_21=CTL4

Table 4 FX3 DVK Board Jumpers

3.1 JTAG Debuggers for FX3 DVK

The Segger J-Link probe is the preferred JTAG probe for the FX3 SDK. This probe, along with the Segger J-Link Arm GDB Server, is used for debug. The Eclipse IDE connects to the J-link GDB server to debug your firmware. To get Eclipse working with the GDB server, you need to create a debug configuration for the J-link. Details are in Chapter 12 of the FX3 Programmer’s Manual.

To debug, you may also use other JTAG probes such as the Olimex Arm-USB-OCD probe. In this case, the OpenOCD tools can be used for the GDB connection. Details on how to do this are in Chapter 12 of the FX3 Programmer’s Manual

Reference Links:

<https://usb.org/>

<https://usb.org/documents>

<https://usb.org/usbc> (USB Type-C)

<https://thunderbolttechnology.net>

Chapter-7

Memory devices

1.0 RAM (Random-access memory)

- A random-access memory device allows data items to be read or written in almost the same amount of time irrespective of the physical location of data inside the memory. RAM contains multiplexing and demultiplexing circuitry, to connect the data lines to the addressed storage for reading or writing the entry.
- RAM is normally associated with volatile types of memory (such as DRAM modules), where stored information is lost if power is removed, although non-volatile RAM has also been developed.

1.1 Static RAM and Dynamic RAM

Static RAM	Dynamic RAM
➤ SRAM uses transistor to store a single bit of data	➤ DRAM uses a separate capacitor to store each bit of data
➤ SRAM does not need periodic refreshment to maintain data	➤ DRAM needs periodic refreshment to maintain the charge in the capacitors for data
➤ SRAM's structure is complex than DRAM	➤ DRAM's structure is simplex than SRAM
➤ SRAM are expensive as compared to DRAM	➤ DRAM's are less expensive as compared to SRAM
➤ SRAM are faster than DRAM	➤ DRAM's are slower than SRAM
➤ SRAM are used in Cache memory	➤ DRAM are used in Main memory

1.2 Applications of SRAM & DRAM

SRAM

- SRAM is also used in personal computers, workstations, routers and peripheral equipment: CPU register files, internal CPU caches and external burst mode SRAM caches, hard disk buffers, router buffers, etc.
- LCD screens and printers also normally employ static RAM to hold the image displayed (or to be printed). Static RAM was used for the main memory of some early personal computers such as the ZX80, TRS-80 Model 100 and Commodore VIC-20.

DRAM

- Cell Phones
- Desktop Computers
- Digital Signal Controller (DSC)
- Global Positioning System (GPS)
- Personal Data Assistant (PDA)
- Smartphones
- Tablets and Pads

1.3 DIMM, DOWN memory (SDRAM)

DIMM(dual in-line memory module)

- In general small outline dual in-line memory module (SO-DIMM).
- A **DIMM or dual in-line memory module** comprises a series of dynamic random-access memory integrated circuits.



fig-1: SO-DIMM RAM

These modules are mounted on a printed circuit board and designed for use in personal computers, workstations and servers. DIMMs began to replace SIMMs (single in-line memory modules) as the predominant type of memory module as Intel P5-based Pentium processors began to gain market share.

DDR DIMMs

DDR, DDR2, DDR3 and DDR4 all have different pin counts, and different notch positions. As of August, 2014, DDR4 SDRAM is a modern emerging type of dynamic random access memory (DRAM) with a high-bandwidth ("double data rate") interface, and has been in use since 2013. It is the higher-speed successor to DDR, DDR2 and DDR3. DDR4 SDRAM is neither forward nor backward compatible with any earlier type of random access memory (RAM) because of different signalling voltages, timings, as well as other differing factors between the technologies and their implementation.



fig-2 DDR DIMM

1.4 Form factors

Several form factors are commonly used in DIMMs. Single Data Rate Synchronous DRAM (SDR SDRAM) DIMMs were primarily manufactured in 1.5 inches (38 mm) and 1.7 inches (43 mm) heights. When 1U rackmount servers started becoming popular, these form factor registered DIMMs had to plug into angled DIMM sockets to fit in the 1.75 inches (44 mm) high box. To alleviate this issue, the next standards of DDR DIMMs were created with a "low profile" (LP) height of around 1.2 inches (30 mm). These fit into vertical DIMM sockets for a 1U platform.

1.5 SDRAM

- SD-RAM (Synchronous dynamic random-access memory)
 - Also call it as down memory
 - where the operation of its external pin interface is coordinated by an externally supplied clock signal.



1.6 DDR, DDR2, DDR3, DDR4

DDR SDRAM Standard	Bus clock (MHz)	Internal rate (MHz)	Prefetch (min burst)	Transfer Rate (MT/s)	Density	Voltage	DIMM pins	SO-DIMM pins
DDR	100–200	100–200	2n	200–400	1GB, 512MB, 256MB, 128MB	2.5/2.6	184	200
DDR2	200–533½	100–266½	4n	400–1066½	4GB, 2GB, 1GB, 512MB, 256MB	1.8	240	200
DDR3	400–1066½	100–266½	8n	800–2133½	16GB, 8GB, 4GB, 2GB, 1GB	1.5	240	204
DDR4	1066½–2133½	100–266½	8n	2133½–4266½	64GB, 32GB, 16GB, 8GB, 4GB	1.05/1.2	288	256

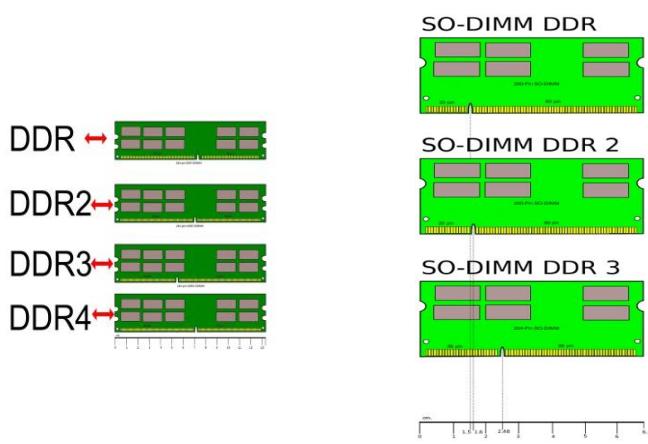


Fig-3: DDR form factors

1.7 Dual channel and single channel

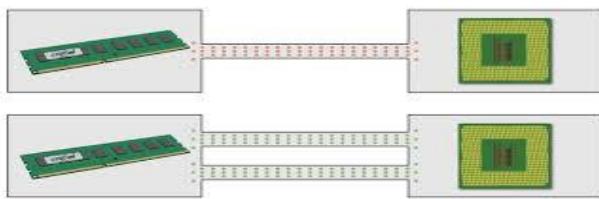
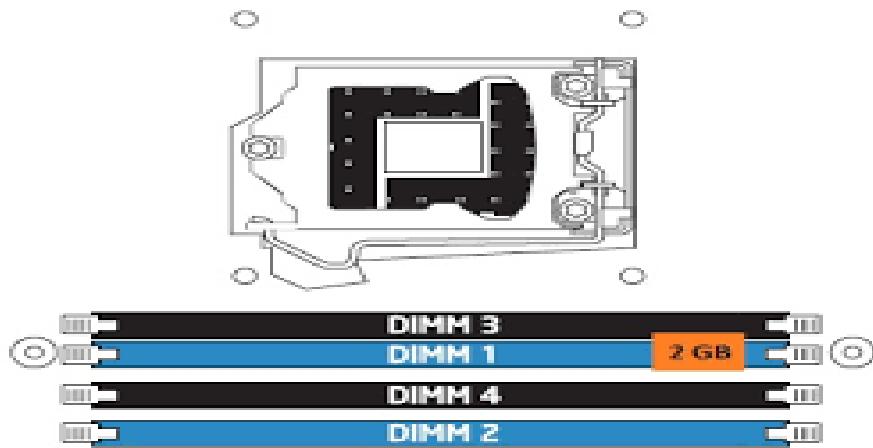
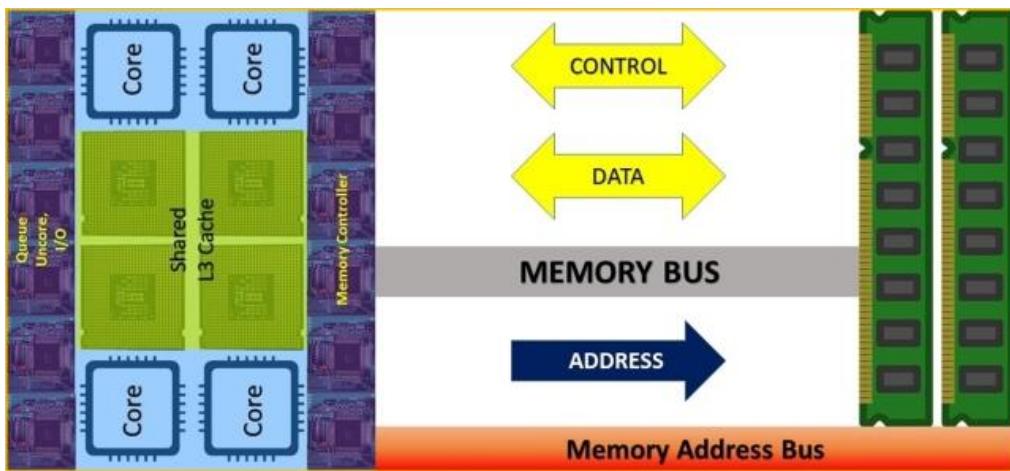


fig-4:Dual Channel between RAM and controller



The **RAM** in the system is **controlled by a circuit that is referred to as Memory Controller**. The RAM and the Memory Controller are connected through a series of wires, collectively known as a Memory Bus. Now, these wires are further divided into three groups – **Control, Data, and Address**. The Control wires are responsible for sending the commands to the memory modules, which contain information about what kind of operation is being performed by the system. The Data wires will carry the data that is either being read from the memory to the memory controller or being written from the memory controller to the memory

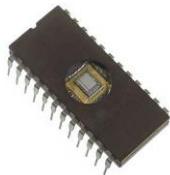


1.8 How to use 2 RAM with different frequency?

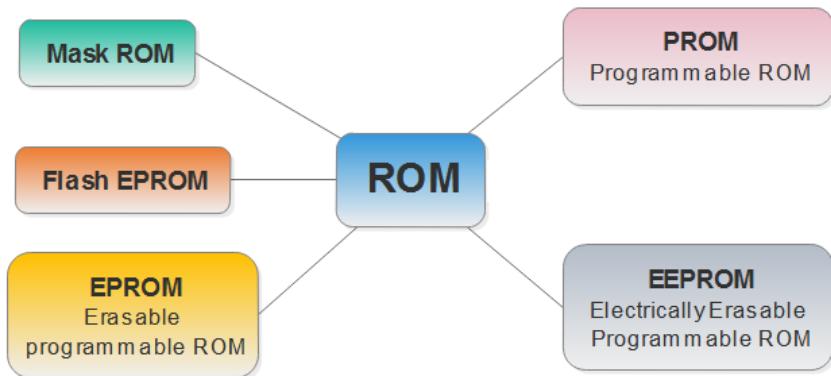


- Both RAM will choose the lowest speed among themselves.
- And mainly the higher frequency RAM should be in the 1st slot else they won't configure.

2.0 ROM stands for Read Only Memory



- The memory from which we can only read but cannot write on it.
- This type of memory is non-volatile.
- A ROM stores such instructions that are required to start a computer. This operation is referred to as **bootstrap**.



2.1 MROM (Masked ROM)

- The very first ROMs were hard-wired devices that contained a pre-programmed set of data or instructions.
- These kind of ROMs are known as masked ROMs, which are inexpensive.

2.2 PROM (Programmable Read Only Memory)

- PROM is read-only memory that can be modified only once by a user.
- The user buys a blank PROM and enters the desired contents using a PROM program.
- Inside the PROM chip, there are small fuses which are burnt open during programming. It can be programmed only once and is not erasable.

2.3 EPROM (Erasable and Programmable Read Only Memory)

- EPROM can be erased by exposing it to ultra-violet light for a duration of up to 40 minutes.
- The charge is retained for more than 10 years because the charge has no leakage path.
- For erasing this charge, ultra-violet light is passed through a quartz crystal window (lid).

2.4 EEPROM (Electrically Erasable and Programmable Read Only Memory)

- EEPROM is programmed and erased electrically. It can be erased and reprogrammed about ten thousand times.
- Both erasing and programming take about 4 to 10 ms (millisecond).

- In EEPROM, any location can be selectively erased and programmed. EEPROMs can be erased one byte at a time, rather than erasing the entire chip. Hence, the process of reprogramming is flexible but slow.

2.5 FLASH ROM

- **Flashrom** is a universal flash programming utility used to detect, read, verify, erase, or write BIOS chips in dual in-line package.
- It supports parallel, Low Pin Count (LPC), FWH, and Serial Peripheral Interface Bus (SPI) flash interfaces. It can be used to flash firmware images such as BIOS or coreboot, or to backup an existing firmware.

2.6 Advantages of ROM

The advantages of ROM are as follows –

- Non-volatile in nature
- Cannot be accidentally changed
- Cheaper than RAMs
- Easy to test
- More reliable than RAMs
- Static and do not require refreshing
- Contents are always known and can be verified

STORAGE DEVICES

Categorized as

- Floppy Disk Drive
- Hard Disk Drive
- Optical Drives
- Removable Storage

3.0 HARD DISK DRIVES

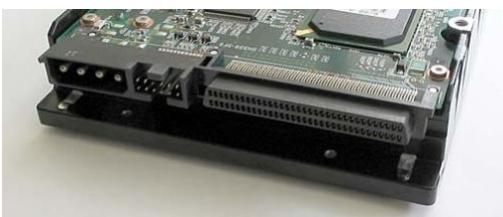


Fig-5:Harddisk

- A hard disk is a sealed unit containing a number of platters in a stack
- In a hard disk, data is stored in thin, concentric bands. A drive head, while in one position can read or write a circular ring, or band called a track.
- Sections within each track are called sectors. A sector is the smallest physical storage unit on a disk.
- A cluster can consist of one or more consecutive sectors. The number of sectors is always an exponent of 2



Fig-6:Internal of Harddisk

1.Primary Partition:

- Windows operating systems must be located in a primary partition.
- Only primary partitions can be used to boot the Operating system.

2.Extended Partition:

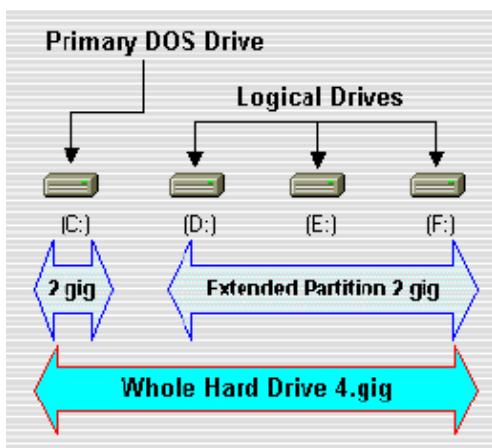
- A hard disk may contain only one extended partition.
- the extended partition can be subdivided into multiple logical partitions (Other than OS is a Extended Partition).

3.Logical Partition:

- Linux operating systems can be installed into (and run from) logical partitions.

4.Active Partition:

- Only one partition on a computer can be set as an **active partition** or **bootable partition**.
- For example, if you are using Microsoft Windows the partition that contains Windows is the active partition.



Cluster

Grouping sectors into larger blocks are called **clusters** .

Master Boot Record (MBR)

The location of the master boot record is cylinder 0, head 0, and sector 1.

DOS Boot Record (DBR)

In the beginning of the boot drive the DOS boot record will load .

Boot Sector

The two tasks of boot sector is to load in the operating system and to inform the user

3.1 Hard Disk Interfaces

- ✓ ST-506/412
- ✓ ESDI

- ✓ IDE
- ✓ EIDE
- ✓ SCSI

Integrated Drive Electronics (IDE)/EIDE

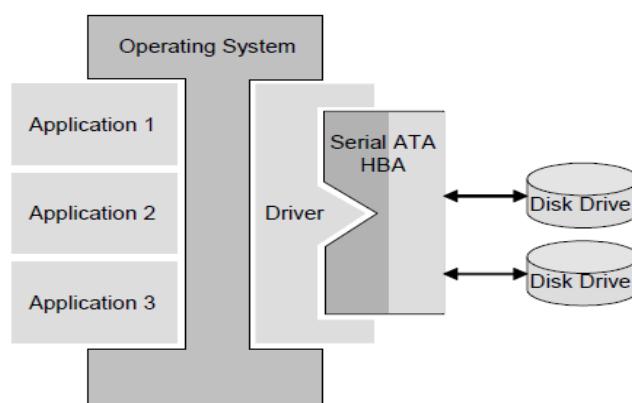
It is used to attach Mass Storage media devices to the computer.

The next generations of IDE interface is Enhanced IDE (EIDE)

- **Small Computer System Interface (SCSI)**
- It is capable of supporting up to 7 to 15 devices.
- **Serial ATA (SATA)**
- It is an internal or inside-the-box interface technology.
- External SATA is commonly known as ESATA.
- **USB**
- It is an External , pocket Hard-disk

Serial Advanced Technology Attachment

- a standard for connecting and transferring data from hard disk drives (HDDs) to computer systems
- SATA is based on serial signaling technology, unlike Integrated Drive Electronics (IDE) hard drives that use parallel signaling.
- succeeds the older Parallel ATA (PATA) offering several advantages
 - reduced cable size and cost
(7 conductors instead of 40 or 80),
 - native hot swapping
- faster data transfer through higher signaling rates



3.2 SATA v/s PATA

Parameters	SATA	PATA
Data transfer rate	150 – 600 Mbps	Up to 133Mbps
Sustained Throughput	55 Mbps	52 Mbps
Cable pins	7	40
Cable length	40 inches(1 meter)	18n inches
Hot pluggable	Yes	No
Power consumption	1.6	2.2
Read/Write	0.65	0.95
Idle	0.20	0.25
Standby/ Sleep		
Power connect pins	15 pins	4 pins
Jumper settings	No	Yes



eSATA- for external connectivity

is an extension to the Serial ATA standard that enables SATA drives to be attached externally.

Prior to eSATA, external hard drives were connected via USB 2.0 or FireWire. While eSATA can provide faster transfer rates than USB or FireWire, it requires its own power connector.

mSATA- mini SATA

An mSATA SSD has a smaller form factor than a standard SSD and is designed for use with portable, power-constrained devices such as laptops, tablets and netbooks.

An mSATA SSD is roughly the size of a business card. Advantages of mSATA SSDs include the small form factor, low power consumption, shock/vibration resistance and fast boot/shutdown capabilities



**Fig-7:An mSATA SSD
on top of a 2.5-inch**

4.0 Optical Disc Drives

- is a flat, circular disc which encodes binary data in the form of pits (0) and lands (1) on special material on one of its flat surfaces
- optical discs are usually between 7.6 and 30 cm (3 to 12 in) in diameter and about 1.2 mm (0.05 in) thick
- designed to support one of three types:
 - read-only (e.g.: CD and CD-ROM)
 - recordable (write-once, e.g. CD-R)
 - re-recordable (rewritable, e.g. CD-RW)
- different types
 - ***Compact Disk (CD)***
 - ***Digital Versatile Disk (DVD)***
 - ***Blu-ray Disk (BD)***

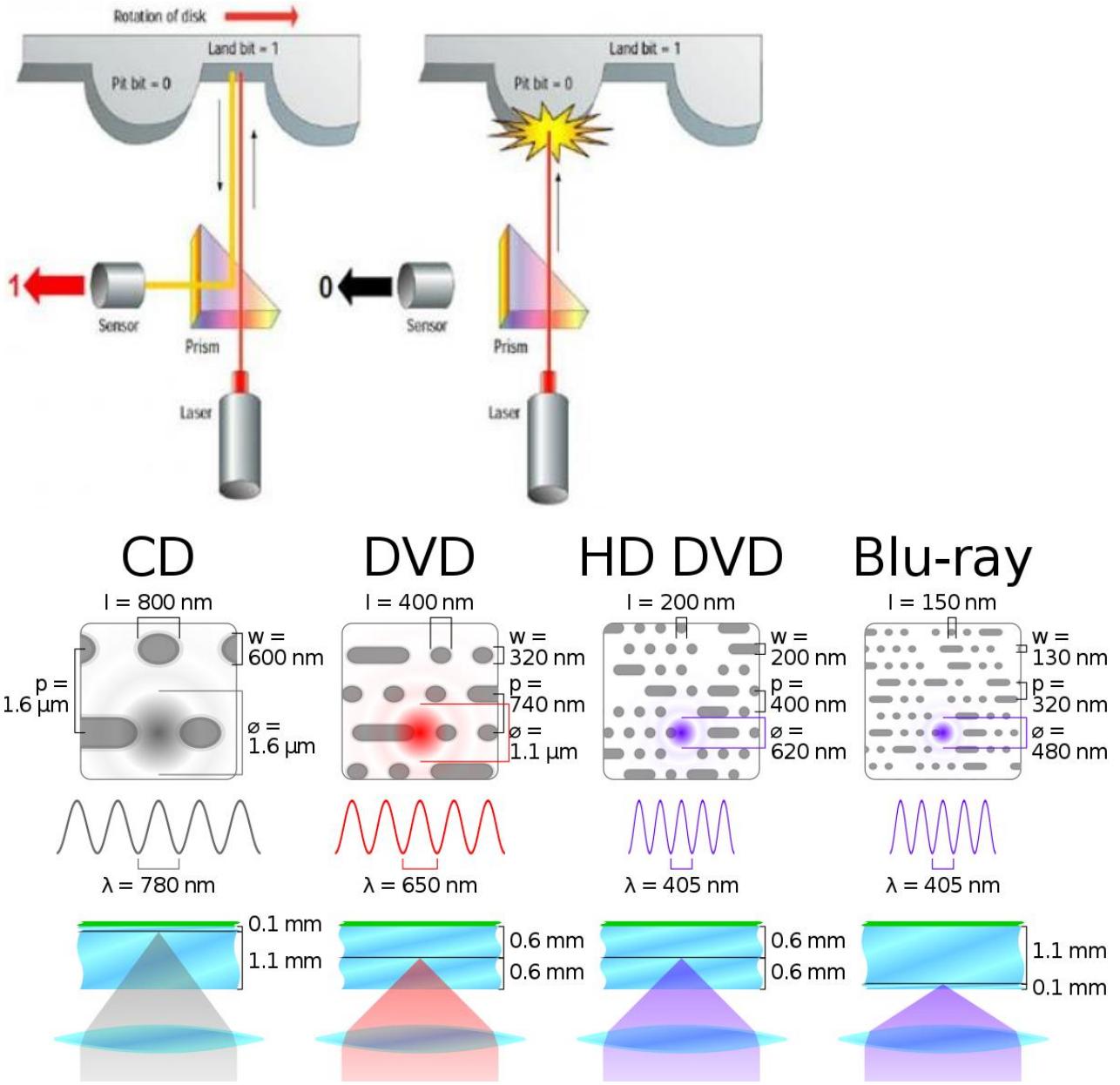


Fig-8:Optical Disk Drive wavelength

4.1 Flash memory

- **Flash memory** is an electronic non-volatile computer storage device that can be electrically erased and reprogrammed.
- Flash memory developed from EEPROM (electrically erasable programmable read-only memory).
- There are two main types of flash memory, which are named after the NAND and NOR logic gates.

- The internal characteristics of the individual flash memory cells exhibit characteristics similar to those of the corresponding gates.

5.0 Removable Storage

- Compact Flash Card*
- Memory stick*
- USB Storage Device*
- MMC devices*
- SD Card*
- Universal Flash Storage*



5.1 Multi Media Card

- is a memory card standard used for solid-state storage
- about the size of a postage stamp: 24 mm × 32 mm × 1.4 mm.
- an MMC is used as a storage medium for a portable device, in a form that can easily be removed for access by a PC.
- MMCs are available in sizes up to and including 512 GB

5.2 (eMMC) embedded Multi Media Card

- widely used in the industry as a primary means of integrated storage in portable devices

- low-cost flash memory system
- Contains 3 parts
 - The MMC interface
 - Flash memory
 - Flash memory controller (embedded)
- appear to its host as a bootable device

5.3 Secure Digital

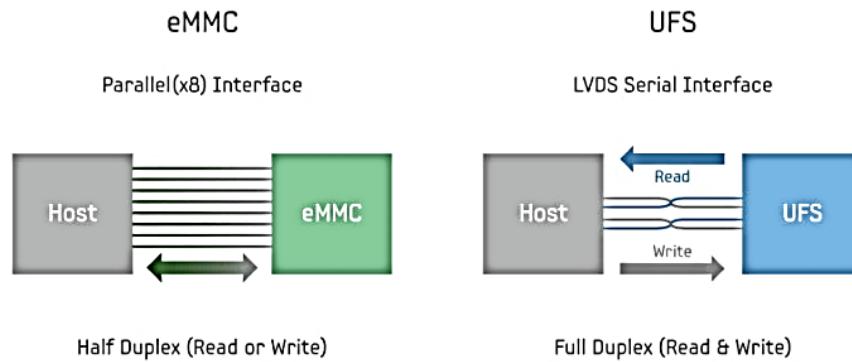
- non-volatile memory card format developed by the SD Card Association (SDA) for use in portable devices.
- was developed to improve on the Multi Media Card (MMC) standard
- Has 4 families
 - *Standard-Capacity (SDSC)*
 - *High-Capacity (SDHC)*
 - *extended-Capacity (SDXC)*
 - *Input- Output(SDIO)*

5.4 Secure Digital Input Output

- Extension of SD specification to cover I/O functions; gradually utilized to connect I/O devices as embedded by using multiple devices connection feature on an SD bus
- Only functional in host devices designed to support I/O functions
- These devices (built for SDIO cards) can use the SD slot to support GPS, Wi-Fi etc. but reverse is not possible

5.5 Universal Flash Storage

- positioned as a replacement for eMMCs and SD cards
- Next generation flash storage that provides the low power of eMMC with the high performance of SCSI SSD
- UFS implements a full-duplex serial LVDS interface that scales better to higher bandwidths than the 8-lane parallel interface of eMMCs.



6.0 Solid State Drive(SSD)

- SSD is an PC storage device that uses Solid State memory to store information.
- SSD uses non volatile NAND Flash Memory , which enables it to retain data when the power is removed.
- SSDs do not have any moving mechanical components, which distinguishes them from traditional magnetic disks such as hard disk drives (HDDs).
- **SSDs use NAND-based flash memory or DRAM to store data.**

6.1 Difference

• Flash memory-based SSDs:	• DRAM-based SSDs:
<ul style="list-style-type: none"> use non-volatile NAND flash memory Ability to retain the data without a constant power supply lower cost compared to DRAM Flash memory SSDs are slower than DRAM solutions. 	<ul style="list-style-type: none"> Based on volatile memory such as DRAM internal battery or an external AC/DC adapter is needed to hold the data ultrafast data access primarily to accelerate applications

Assignment:

- **Difference between SSD and Harddisk**
- **Advantage and disadvantages of SSD and harddisk**

6.2 RAID (Redundant Array of Independents Disks)

RAID (**Redundant Array of Independents Disks**) is a simple technology that improves the performance of external storage solutions. RAID allows you to choose the best way to use your device to fit your needs. Simply put, RAID technology either divides or duplicates the task of one hard disk between many (or as few as two) disks to either improve performance or create data redundancy in case of a drive failure. You can decide how the device handles data by setting its RAID mode.

To better understand how RAID works, first familiarize yourself with the terms below:

Striping◆◆

is the splitting of data between multiple drives. Striped RAID arrays generally aim to merge maximum capacity into one single volume.

Mirroring◆◆

is the copying of data to more than one disk. Mirrored RAID arrays typically allow the failure of at least one disk in the array without any data loss, depending on the RAID level of the array.

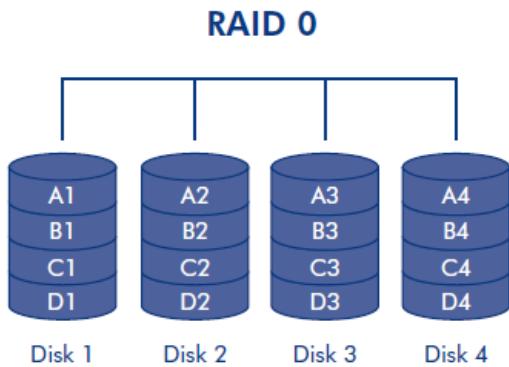
Fault Tolerance◆◆

allows a RAID array to continue working (i.e. the data stored in the array is still available to the user) in the event of a disk failure. Not all mirrored RAID arrays are user friendly. For instance, some RAID devices must be shut down before a failed disk is replaced.

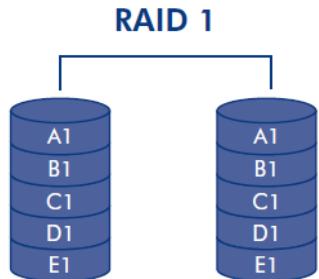
RAID LEVELS

- **RAID 0**
- **RAID 1**
- **RAID 2**
- **RAID 3**
- **RAID 4**
- **RAID 5**
- **RAID 6**
- **RAID 10(1+0)**

6.3 RAID 0



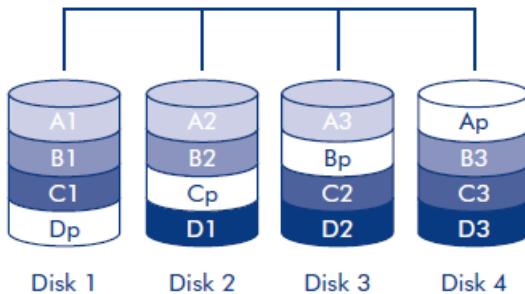
- It is the fastest RAID mode.
- Requiring at least 2 drives, RAID 0 stripes data onto each disk. The available capacities of each disk are added together so that one logical volume mounts on the computer.
- If one physical disk in the array fails, the data of all disks becomes inaccessible because parts of the data have been written to all disks. **RAID 1**.



- It is a secure RAID mode that requires at least 2 drives and works with pairs of drives.
- One logical volume mounts on the computer and the combined available capacity of both drives is limited to the capacity of the lowest-capacity disk.
- If one physical disk fails, the data is available immediately on the second disk. No data is lost if one disk fails.

6.4 RAID 5

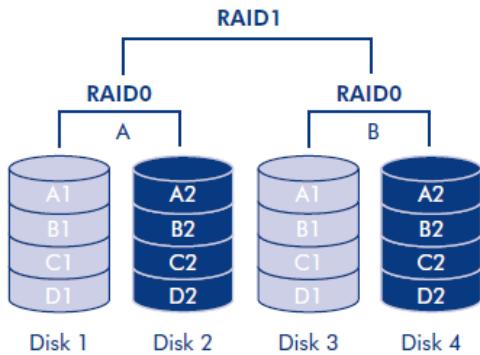
RAID 5



- ❑ RAID 5 combines the striping of RAID 0 with data redundancy in an array with a minimum of three disks.
- ❑ Data is striped across all disks and a parity block (P) for each data block is written on the same stripe. If one physical disk fails, the data from the failed disk can be rebuilt onto a replacement disk.
- ❑ No data is lost in the case of a single disk failure, but if a second disk fails before data can be rebuilt to a replacement drive, all data in the array will be lost.

6.5 Nested RAID Levels

RAID 0+1



RAID 0+1

- RAID 0+1 is a secure RAID mode that is composed of a mirror of striped sets.
- Up to two disks can fail in a RAID 0+1 array without loss of data, as long as the failed disks are not part of different RAID 0 pairs.

6.6 RAID config in BIOS

Press **ctrl+l** at the time of booting to get RAID utility in order to configure RAID

Intel(R) Rapid Storage Technology enterprise - SCU Option ROM - 3.5.0.1005
Copyright(C) 2003-12 Intel Corporation. All Rights Reserved.

RAID Volumes:
None defined.

Physical Devices:

ID	Device Model	Serial #	Size	Type/Status(Vol ID)
0	ST3146855SS	RCXG000097389NWW	136.7GB	Non-RAID Disk
1	ST3146855SS	ZJ3H00009737L0EF	136.7GB	Non-RAID Disk
2	ST3146855SS	JETS00009803UA3L	136.7GB	Non-RAID Disk

Press <CTRL-I> to enter Configuration Utility...

Intel(R) Rapid Storage Technology enterprise - SATA Option ROM - 3.5.0.1005
Copyright(C) 2003-12 Intel Corporation. All Rights Reserved.

RAID Volumes:
None defined.

Physical Devices:

ID	Device Model	Serial #	Size	Type/Status(Vol ID)
0	ST91000640NS	9XG035JC	931.5GB	Non-RAID Disk
1	ST91000640NS	9XG04JB1	931.5GB	Non-RAID Disk
3	ST91000640NS	9XG04WW8	931.5GB	Non-RAID Disk

Press <CTRL-I> to enter Configuration Utility...

Intel(R) Rapid Storage Technology enterprise - SATA Option ROM - 3.5.0.1005
Copyright(C) 2003-12 Intel Corporation. All Rights Reserved.

[MAIN MENU]=

- | | |
|-----------------------|----------------------------|
| 1. Create RAID Volume | 3. Reset Disks to Non-RAID |
| 2. Delete RAID Volume | 4. Exit |

[DISK/VOLUME INFORMATION]=

RAID Volumes:
None defined.

Physical Devices:

ID	Device Model	Serial #	Size	Type/Status(Vol ID)
0	ST91000640NS	9XG035JC	931.5GB	Non-RAID Disk
1	ST91000640NS	9XG04JB1	931.5GB	Non-RAID Disk
3	ST91000640NS	9XG04WW8	931.5GB	Non-RAID Disk

[↑↓]-Select

[ESC]-Exit

[ENTER]-Select Menu

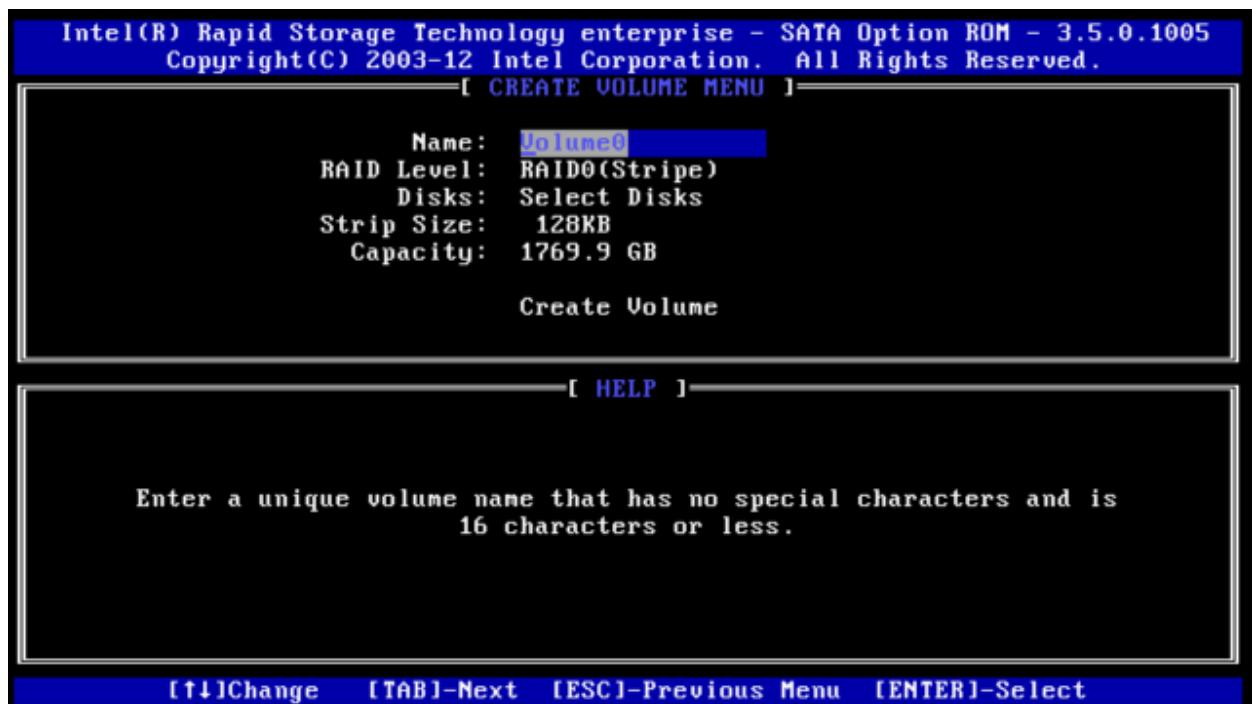


Fig-9:Bios utility for RAID configuration

Chapter-8

On board Communication Protocols

1.0 Inter - Integrated Circuit (I2C)

1.1 Introduction

When multiple devices connecting to a microcontroller, the address and data lines of each device were conventionally connected individually. This would take up precious pins on the microcontroller, result in lot of traces on the PCB, and require more components to connect everything together. This made these systems expensive to produce and susceptible to interference and noise. To solve this problem Philips developed Inter-IC bus, or I2C, in the 1980s.

I2C is a low-bandwidth, short distance protocol for on board communications. All devices are connected through two wires: serial data (SDA) and serial clock (SCL).

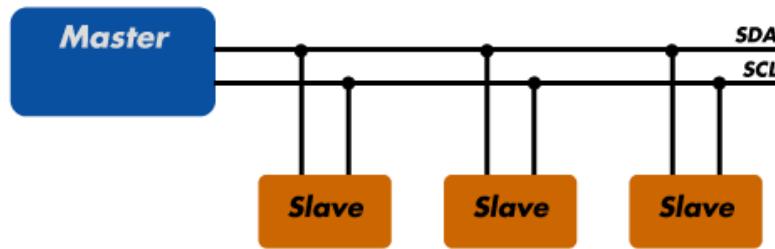


Figure 1. SDA and SCL lines in I2C Communication

1.2 What is I²C

The name stands for “Inter - Integrated Circuit Bus”. It provides half-Duplex Operation. Originally intended for operation on one single board / PCB. Two wires carry information between a number of devices

- One wire used for the data (SDA)
- One wire used for the clock (SCL)

Today, a variety of devices are available with I2C Interfaces Microcontroller, EEPROM, Real-Timer, interface chips, LCD driver, A/D converter

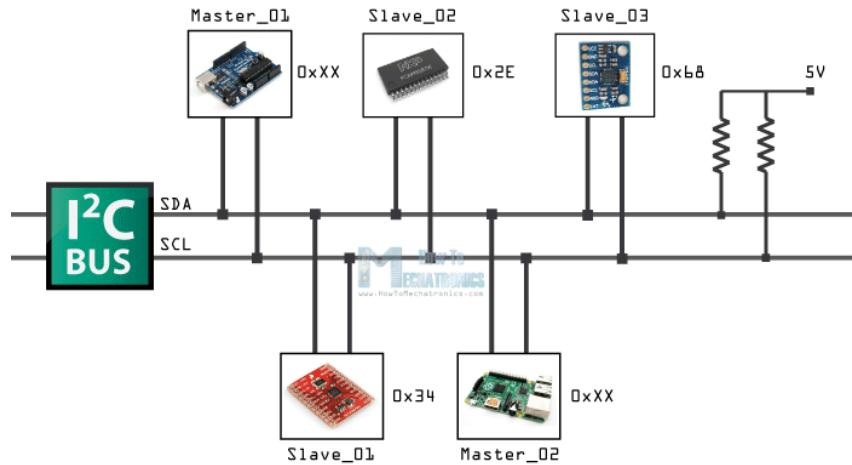


Figure 2. Example for I2C Communication

1.3 I²C Features

- Only two bus lines are required: a serial data line (SDA) and a serial clock line (SCL).
- Each device connected to the bus is software addressable by a unique address and simple master/slave relationships exist at all times; masters can operate as master-transmitters or as master-receivers.
- It's a true multi-master bus including collision detection and arbitration to prevent data corruption if two or more masters simultaneously initiate data transfer.
- Serial, 8-bit oriented, bi-directional data transfers can be made at up to 100 kbit/s in the Standard-mode, up to 400 kbit/s in the Fast-mode, or up to 3.4 Mbit/s in the High-speed mode.
- On-chip filtering (50 ns) rejects spikes on the bus data line to preserve data integrity.
- The number of ICs that can be connected to the same bus segment is limited only by the maximum bus capacitive loading of 400 pF.

1.4 Responsibility

When two devices communicate with each other than one device is called master and other one is slave. In communication protocol one device or process (known as the *master*) which controls one or more other devices or processes (known as *slaves*). Once the master/slave relationship is established, the direction of control is always from the master to the slave

Master:

- Initiates a transfer by generating start and stop conditions
- Generates the clock
- Transmits the slave address
- Determines data transfer direction

Slave:

- Responds only when addressed
- Timing is controlled by the clock line

1.5 I²C Addressing

The first byte of an I²C transfer contains the slave address and the data direction. The address is 7 bits long, followed by the direction bit. Like all data bytes, the address is transferred with the most significant bit first. A seven-bit wide address space theoretically allows 128 I²C addresses – however, some addresses are reserved for special purposes. Thus, only 112 addresses are available with the 7-bit address scheme. To get rid of this a special method for using 10 bit addresses is defined.

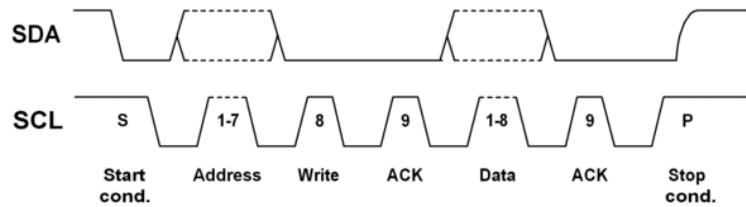


Figure 3. I²C Data transfer

1.6 I²C Operation

I²C has a master/slave protocol. The master initiates the communication. The sequence of events are:

- The Master device issues a start condition. This condition informs all the slave devices to listen on the serial data line for instructions.
- The Master device sends the address of the target slave device and a read/write flag.
- The Slave device with the matching address responds with an acknowledgement signal.
- Communication proceeds between the Master and the Slave on the data bus. Both the master and slave can receive or transmit data depending on whether the communication is a read or write. The transmitter sends 8-bits of data to the receiver which replies with a 1-bit acknowledgement.
- When the communication is complete, the master issues a stop condition indicating that everything is done.

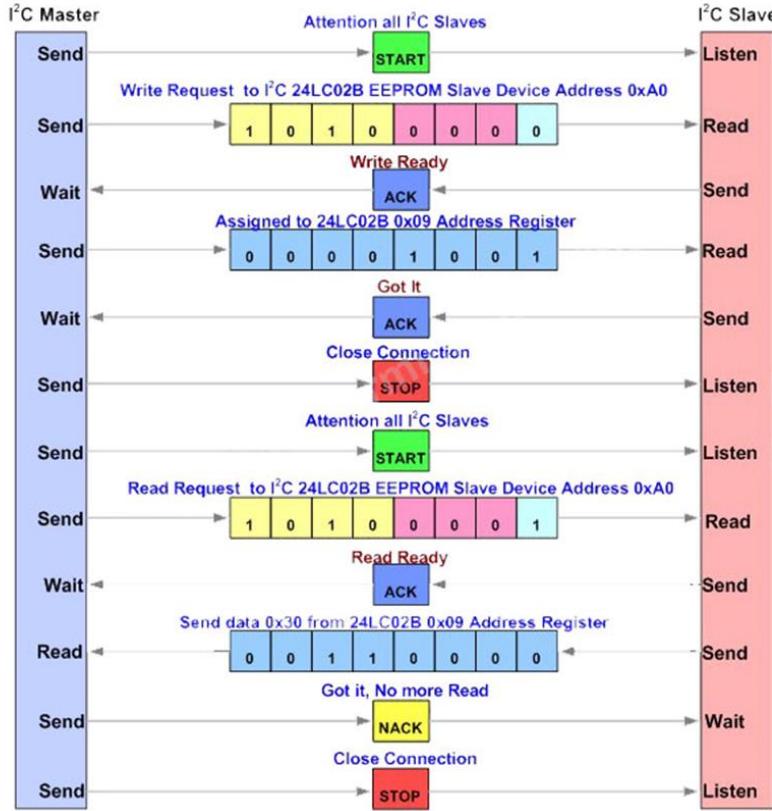


Figure 4. I2C Master and Slave Device Read Operation Handshake Diagram

1.7 START and STOP Conditions

A transition of the data line while the clock line is high is defined as either a start or a stop condition. Both start and stop conditions are generated by the bus master. The bus is considered busy after a start condition, until a stop condition occurs.

START: A HIGH to LOW transition on the SDA line while SCL is HIGH

STOP: A LOW to HIGH transition on the SDA line while SCL is HIGH

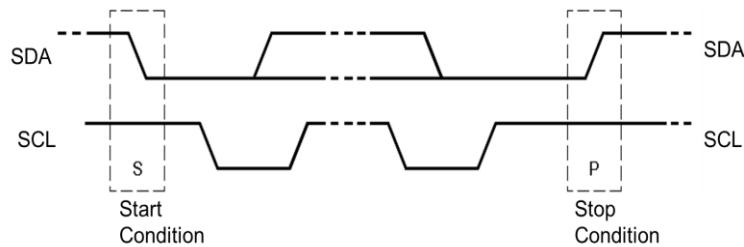


Figure 5. I2C START and STOP Conditions

1.8 I2C Full Data Transfer

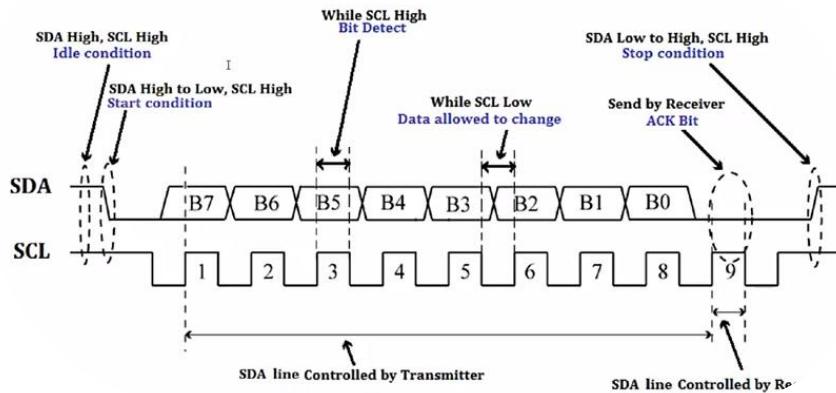


Figure 6. I2C Full Data Transfer

1.9 I2C Read and Write Operations

Writing to a Slave on the I2C Bus

One important point is master reading and also master reading Talk with respect to one only master or slave else it will create lots of confusion To write on the I2C bus, the master will send a start condition on the bus with the slave's address, as well as the last bit (the R/W bit) set to 0, which signifies a write. After the slave sends the acknowledge bit, the master will then send the register address of the register it wishes to write to. The slave will acknowledge again, letting the master know it is ready. After this, the master will start sending the register data to the slave, until the master has sent all the data it needs to (sometimes this is only a single byte), and the master will terminate the transmission with a STOP condition.

- Master Controls SDA Line
- Slave Controls SDA Line

Write to One Register in a Device

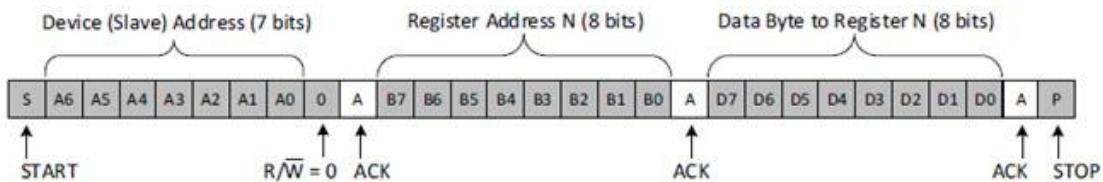


Figure 7. I2C Write to Slave Device Register

Reading from a Slave on The I2C Bus

Reading from a slave is very similar to writing, but with some extra steps. In order to read from a slave, the master must first instruct the slave which register it wishes to read from. This is done by the master starting off the transmission in a similar fashion as the write, by sending the

address with the R/W bit equal to 0 (signifying a write), followed by the register address it wishes to read from. Once the slave acknowledges this register address, the master will send a START condition again, followed by the slave address with the R/W bit set to 1 (signifying a read). This time, the slave will acknowledge the read request, and the master releases the SDA bus, but will continue supplying the clock to the slave. During this part of the transaction, the master will become the master-receiver, and the slave will become the slave-transmitter. The master will continue sending out the clock pulses, but will release the SDA line, so that the slave can transmit data. At the end of every byte of data, the master will send an ACK to the slave, letting the slave know that it is ready for more data. Once the master has received the number of bytes it is expecting, it will send a NACK, signalling to the slave to halt communications and release the bus. The master will follow this up with a STOP condition.

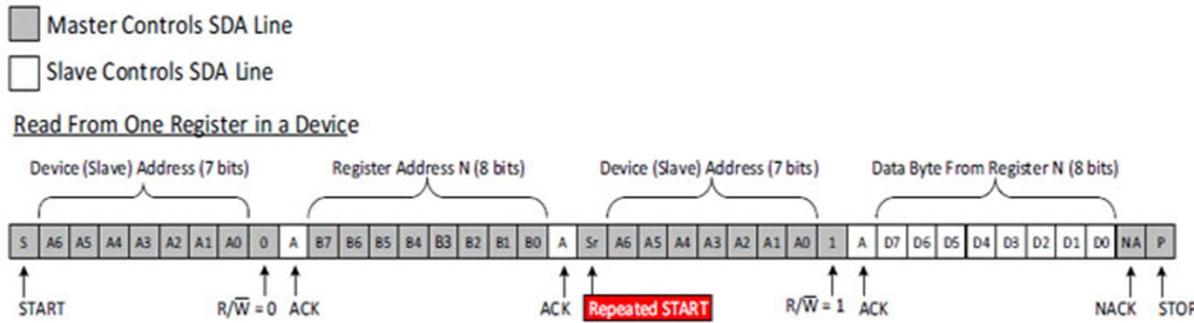


Figure 8. I2C Read from Slave Device's Register

1.10 Multi-master I²C Systems

There are I2C environments where multiple masters are driving the bus. In such case each device needs to be able to cooperate with the fact that another device is currently talking and the bus is therefore busy.

This translates into:

- Being able to follow arbitration logic. If two devices start to communicate at the same time the one writing more zeros to the bus (or the slower device) wins the arbitration and the other device immediately discontinues any operation on the bus.
- Bus busy detection. Each device must detect an ongoing bus communication and must not interrupt it. This is achieved by recognizing traffic and waiting for a stop condition to appear before starting to talk on the bus.

If you plan to use a multimaster device on a bus it is essential that all masters are multimasters. A single-master is simply a device, which does not understand the above mechanisms. If a singlemaster and a multimaster are connected, the single master may well interrupt the multimaster causing unpredictable results.

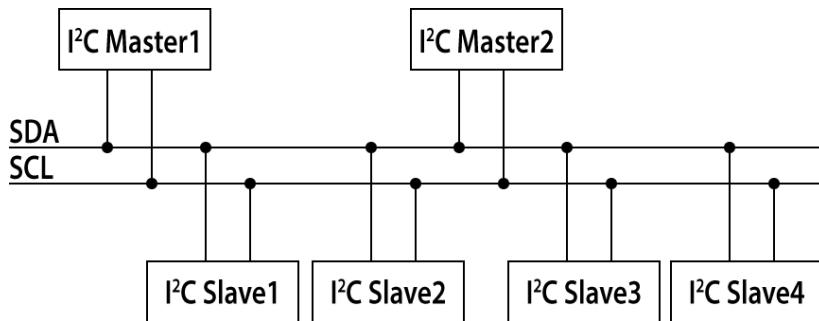


Figure 9. I²C Multi Master Configuration

Arbitration: (Data should not corrupt)

- Arbitration is the procedure by which competing masters decide final control of the bus
- I²C arbitration does not corrupt the data transmitted by the prevailing master
- Arbitration is performed bit by bit until it is uniquely resolved
- Arbitration is lost by a master when it attempts to assert a high on the data line and fails

Error Checking

- I²C defines the basic protocol and timing
 - Protocol errors are typically flagged by the interface
 - Timing errors may be flagged, or in some cases could be interpreted as a different bus event
- Glitches (if not filtered out) could potentially cause:
 - Apparent extra clocks
 - Incorrect data
 - “Locked” bus
- Microprocessors communicating with each other can add a checksum or equivalent

1.11 Bus Recovery

An I²C bus can be “locked” when:

- A Master and a Slave get out of synchronization.
- A Stop is omitted or missed (possibly due to noise)
- Any device on the bus holds one of the lines low improperly, for any reason
- A shorted bus line

If SCL can be driven, the Master may send extra clocks until SDA goes high, then send a Stop. If SCL is stuck low, only the device driving it can correct the problem.

1.12 I²C Devices & Applications

Analog to Digital Converters (A/D, D/A): MMI functions, battery & converters, temperature monitoring, control systems

Bus Controller: Telecom, consumer electronics, automotive, Hi-Fi systems, PCs, servers

Bus Repeater, Hub & Expander: Telecom, consumer electronics, automotive, Hi-Fi systems, PCs, servers

Real Time Clock (RTC)/Calendar: Telecom, EDP, consumer electronics, clocks, automotive, Hi-Fi systems, FAX, PCs, terminals

DIP Switch: Telecom, automotive, servers, battery & converters, control systems

LCD/LED Display Drivers: Telecom, automotive instrument driver clusters, metering systems, POS terminals, portable items, consumer electronics

There are some specific applications for certain types of I2C devices such as TV or radio tuners, but in most cases a general purpose I2C device can be used in many different applications because of its simple construction.

1.13 Advantages and Disadvantages

Advantages

- Only uses two wires
- Supports multiple masters and multiple slaves
- ACK/NACK bit gives confirmation that each frame is transferred successfully
- Hardware is less complicated than with UARTs
- Well known and widely used protocol

Disadvantages

- Slower data transfer rate than SPI
- The size of the data frame is limited to 8 bits
- More complicated hardware needed to implement than SPI

2.0 Serial Peripheral Interface (SPI)

2.1 Introduction

SPI stands for Serial Peripheral Interface, developed by Motorola. SPI is a synchronous protocol that allows a master device to initiate communication with a slave device. It provides Full-Duplex Operation mode using a master-slave architecture with a single master. It allows serial communication between two or more devices at a high speed and is reasonably easy to implement. The clock signal controls when data is changed and when it should be read. Since SPI is synchronous, the clock rate can vary, unlike RS-232 style communications

2.2 SPI Topology

The SPI bus specifies four logic signals:

SCLK: Serial Clock (output from master).

MOSI: Master Output Slave Input, or Master Out Slave In (data output from master).

MISO: Master Input Slave Output, or Master In Slave Out (data output from slave).

SS: Slave Select

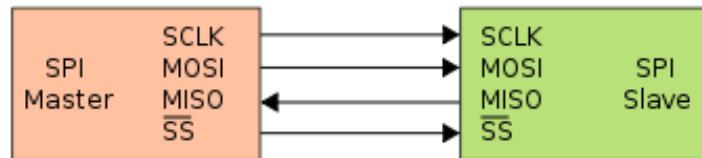


Figure 10. Single master, single slave SPI implementation

2.3 SPI Data Transmission

The steps of SPI Data Transmission are as per the following.

1. The master outputs the clock signal:

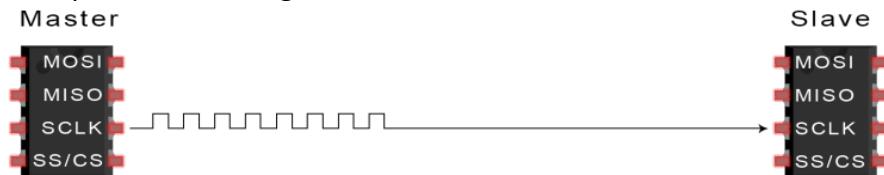


Figure 11.1 Master outputs clock signal

2. The master switches the SS/CS pin to a low voltage state, which activates the slave:

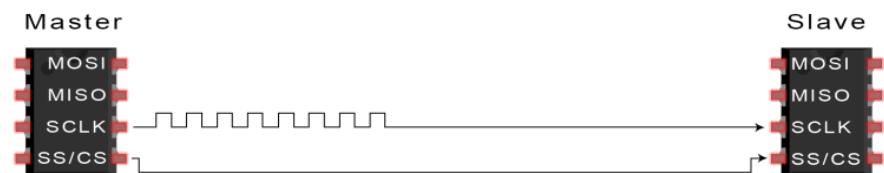


Figure 11.2 Master activating slave

3. The master sends the data one bit at a time to the slave along the MOSI line. The slave reads the bits as they are received:

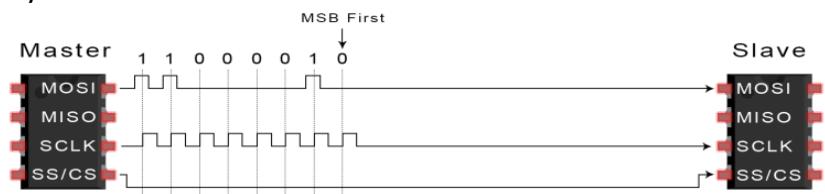


Figure 11.3 Master sends the data along MOSI

4. If a response is needed, the slave returns data one bit at a time to the master along the MISO line. The master reads the bits as they are received:

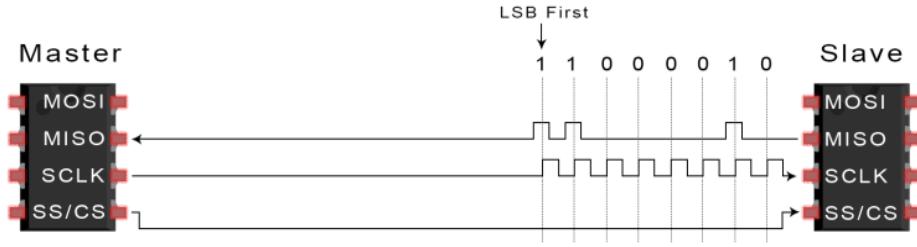


Figure 11.4 Slave sends the data along MISO

2.4 Clock polarity and phase

In addition to setting the clock frequency, the master must also configure the clock polarity and phase with respect to the data.

CPOL determines the polarity of the clock. The polarities can be converted with a simple inverter.

CPOL=0 is a clock which idles at 0, and each cycle consists of a pulse of 1. That is, the leading edge is a rising edge, and the trailing edge is a falling edge.

CPOL=1 is a clock which idles at 1, and each cycle consists of a pulse of 0. That is, the leading edge is a falling edge, and the trailing edge is a rising edge.

CPHA determines the timing of the data bits relative to the clock pulses. Conversion between these two forms is non-trivial.

For CPHA=0, the "out" side changes the data on the trailing edge of the preceding clock cycle, while the "in" side captures the data on (or shortly after) the leading edge of the clock cycle. The outside holds the data valid until the trailing edge of the current clock cycle. For the first cycle, the first bit must be on the MOSI line before the leading clock edge.

An alternative way of considering it is to say that a CPHA=0 cycle consists of a half cycle with the clock idle, followed by a half cycle with the clock asserted.

For CPHA=1, the "out" side changes the data on the leading edge of the current clock cycle, while the "in" side captures the data on (or shortly after) the trailing edge of the clock cycle. The outside holds the data valid until the leading edge of the following clock cycle. For the last cycle, the slave holds the MISO line valid until slave select is deasserted.

An alternative way of considering it is to say that a CPHA=1 cycle consists of a half cycle with the clock asserted, followed by a half cycle with the clock idle.

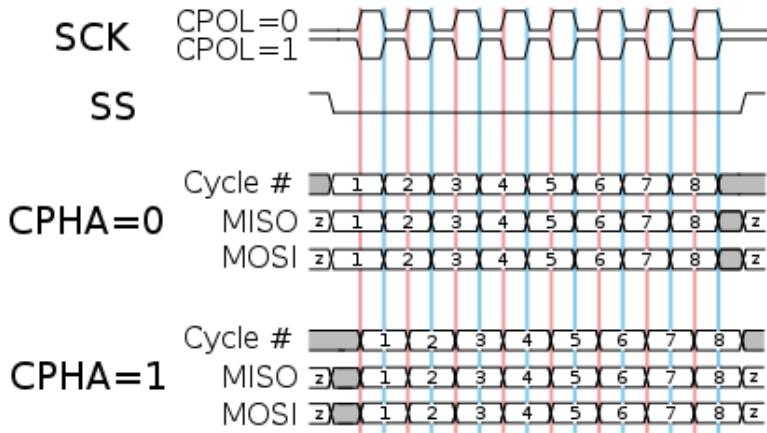


Figure 12. Clock Polarity and Phase

2.5 SPI Configuration

Independent slave configuration:

In the independent slave configuration, there is an independent chip select line for each slave. A pull-up resistor between power source and chip select line is highly recommended for each independent device to reduce cross-talk between devices. This is the way SPI is normally used. Since the MISO pins of the slaves are connected together, they are required to be tristate pins (high, low or high-impedance).

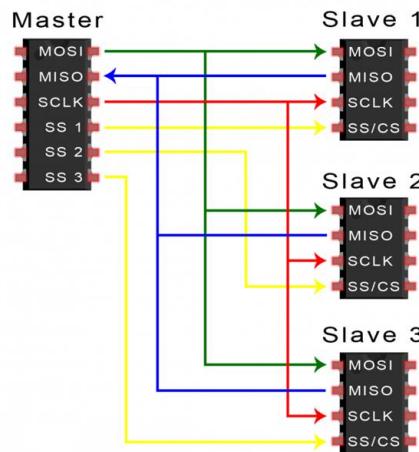


Figure 13. Independent slave configuration

Daisy chain configuration:

Some products that implement SPI may be connected in a daisy chain configuration, the first slave output being connected to the second slave input, etc. The SPI port of each slave is designed to send out during the second group of clock pulses an exact copy of the data it received during the first group of clock pulses. The whole chain acts as a communication shift register; daisy

chaining is often done with shift registers to provide a bank of inputs or outputs through SPI. Such a feature only requires a single SS line from the master, rather than a separate SS line for each slave.

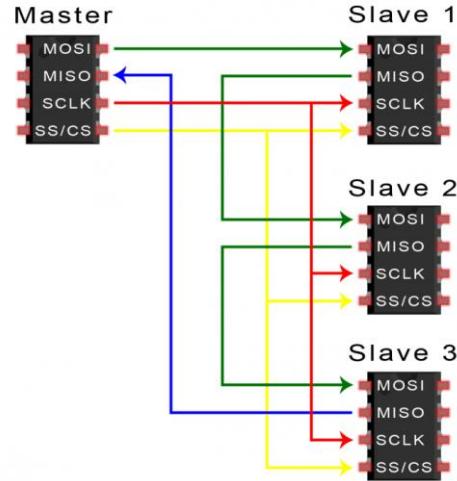


Figure 14. Daisy chain configuration

2.6 Advantages and drawbacks

- SPI is a very simple communication protocol.
- It does not have a specific high-level protocol which means that there is almost no overhead.
- Data can be shifted at very high rates in full duplex mode. This makes it very simple and efficient in a single master and single slave scenario.
- The exchange itself has no pre-defined protocol. This makes it ideal for data-streaming applications.
- Data can be transferred at high speed, often into the range of the tens of megaHertz.

The flipside is that there is no acknowledgment, no flow control, and the master may not even be aware of the slave's presence / or absence.

2.7 SPI Devices & Applications

- Converters (ADC and DAC)
- Memories (EEPROM and FLASH)
- Real Time Clocks (RTC)
- Sensors (temperature, pressure)
- Others (signal mixer, potentiometer, LCD controller, UART, CAN controller, USB controller, amplifier)

3.0 Low Pin Count (LPC)

3.1 Introduction

The Low Pin Count (LPC) bus, is used to connect peripherals around the CPU and to replace the Industry Standard Architecture (ISA) bus. It can connect the boot ROM, "legacy" I/O devices, and Trusted Platform Module (TPM).

"Legacy" I/O devices usually include serial and parallel ports, PS/2 keyboard, PS/2 mouse, and floppy disk controller. Most PC motherboards with an LPC bus have either a Platform Controller Hub (PCH) or a Southbridge chip, which acts as the host and controls the LPC bus.

3.2 Goals of the LPC Interface

- Enable a system without an ISA or X-bus.
- Reduce the cost of traditional X-bus devices.
- Intended for use by devices down on a motherboard only (i.e. no connector).
- Meet the data transfer rate of X-bus, and exceed those data rates where appropriate.
- Perform the same cycle types as the X-bus: Memory, I/O, DMA, and Bus Master
- Support new Firmware Memory cycle type allowing separate boot BIOS firmware memory cycles and application memory cycles.
- Ability to support a variable number of wait-states.
- Ability to support wake-up and other power state transitions.

3.3 Applications

- Intel designed the LPC bus so that the system BIOS image could be stored in a single flash memory chip directly connected to the LPC bus.
- Intel also made it possible to put operating system images and software applications on a single flash memory chip directly connected to the LPC bus, as an alternative to a Parallel ATA port.
- A CPLD or FPGA can implement an LPC host or peripheral.
- The original Xbox game console has an LPC debug port that can be used to force the Xbox to boot new code

3.4 LPC bus limitations

- LPC consists of 7 required pins and 6 optional pins that makes up to a total of 13 pins to implement.
- Present implementations of the LPC include a fabrication process cost burden as it is based on 3.3V I/O signalling technology.
- The frequency of the bus clock is fixed at 33 MHz. The fix LPC bandwidth of 133 Mbps is deemed insufficient to cater for the demands of new devices. Connecting these devices to high speed interfaces such as PCI Express and USB3 is prohibitive from cost perspective.

- There exist a significant number of sideband signals used for communication between chipset and EC, BMC and SIO that amounts to significant pin cost.

4.0 Enhanced Serial Peripheral Interface (eSPI)

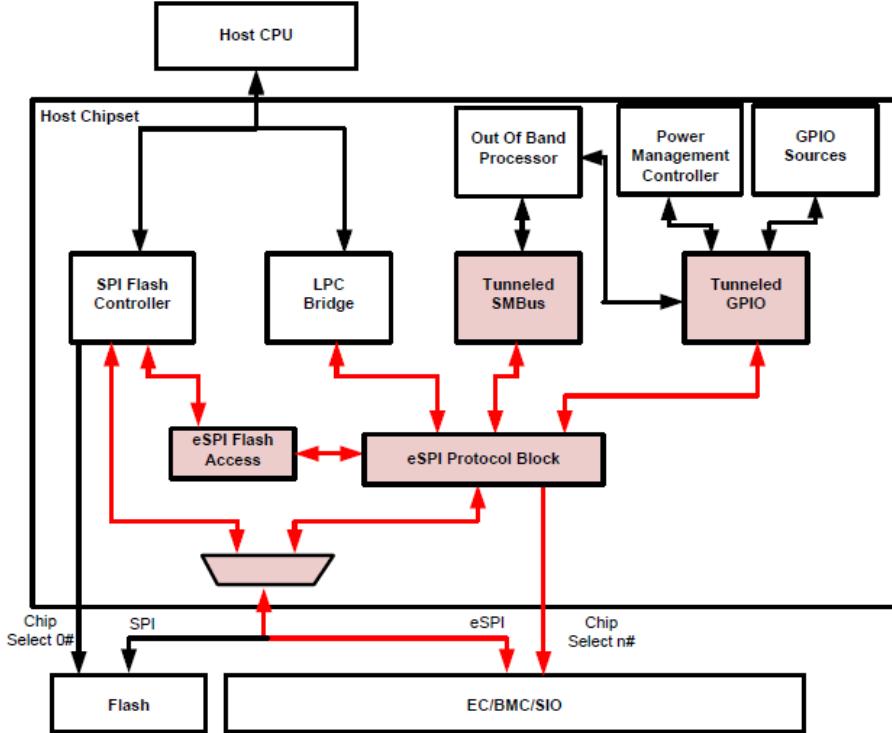


Figure 15. EC/BMC/SIO Communication over eSPI

LPC bus limitations led to the development of eSPI. The eSPI specification provides a path for migrating LPC devices over to the new eSPI interface. eSPI reuses the timing and electrical specification of Serial Peripheral Interface (SPI) but with different protocol to meet a set of different requirements.

4.1 Requirements

eSPI is defined to meet the following requirements:

- Low Power
- Pin Count Reduction
- Medium Bandwidth
- LPC Replacement
- Sideband Pins as In-Band Messaging
- Real Time Flash Sharing
- Chipset and Slave Devices SMBus Replacement
- Scalable bandwidth
- Low Voltage I/O Buffer

4.2 eSPI Topology

- eSPI bus, there is only one eSPI master and one or more eSPI slaves.
- eSPI master dictates the flow of command and data between itself and the eSPI slaves by controlling the Chip Select# pins for each of the eSPI slaves.
- At any one time, the eSPI master must ensure that only one of the Chip Select# pins is asserted based on source decode.
- eSPI master is the only component that is allowed to drive Chip Select# when eSPI Reset# is de-asserted.

4.3 eSPI Configuration

For an eSPI bus, there is only one eSPI master and one or more eSPI slaves. In Single Master-Single Slave configuration, a single eSPI master will be connected to a single eSPI slave. In one configuration, the eSPI slave could be the device that generates the eSPI Reset#. In this case, the eSPI Reset# is driven from eSPI slave to eSPI master. In other configuration, the eSPI Reset# could be generated by the eSPI master and thus, it is driven from eSPI master to eSPI slave.



Figure 16. Single Master-Single Slave with eSPI Reset# from Slave to Master

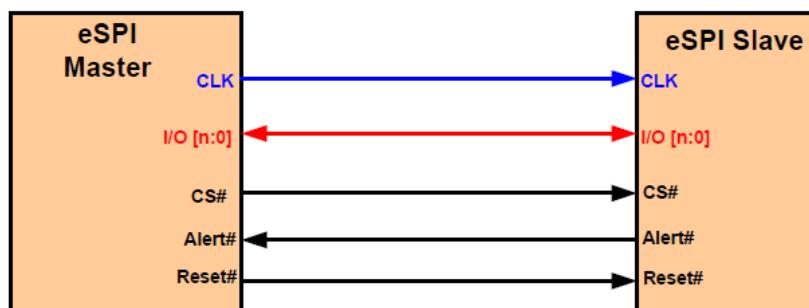


Figure 17. Single Master-Single Slave with eSPI Reset# from Master to Slave

Multiple SPI and eSPI slaves could be connected to the same eSPI bus interface in a multi-drop Single Master-Multiple Slaves configuration. The number of devices that can be supported over a single eSPI bus interface is limited by bus loading and signals trace length.

In this configuration, the clock and data pins are shared by multiple SPI and eSPI slaves. Each of the slaves has its dedicated Chip Select# and Alert# pins.

In an eSPI bus configuration with multiple slaves present, the eSPI master may support 2 eSPI Reset# pins, one from eSPI slave to eSPI master and another one from eSPI master to eSPI slaves. In this case, the master's eSPI interface will only be reset if all the slaves' eSPI interfaces are reset.

SPI slaves such as Flash and TPM are allowed to share the same set of clock and data pins with eSPI slaves. These non-eSPI slaves are selected using the dedicated Chip Select# pins and they communicate with the eSPI master through SPI specific protocols ran over the eSPI bus.

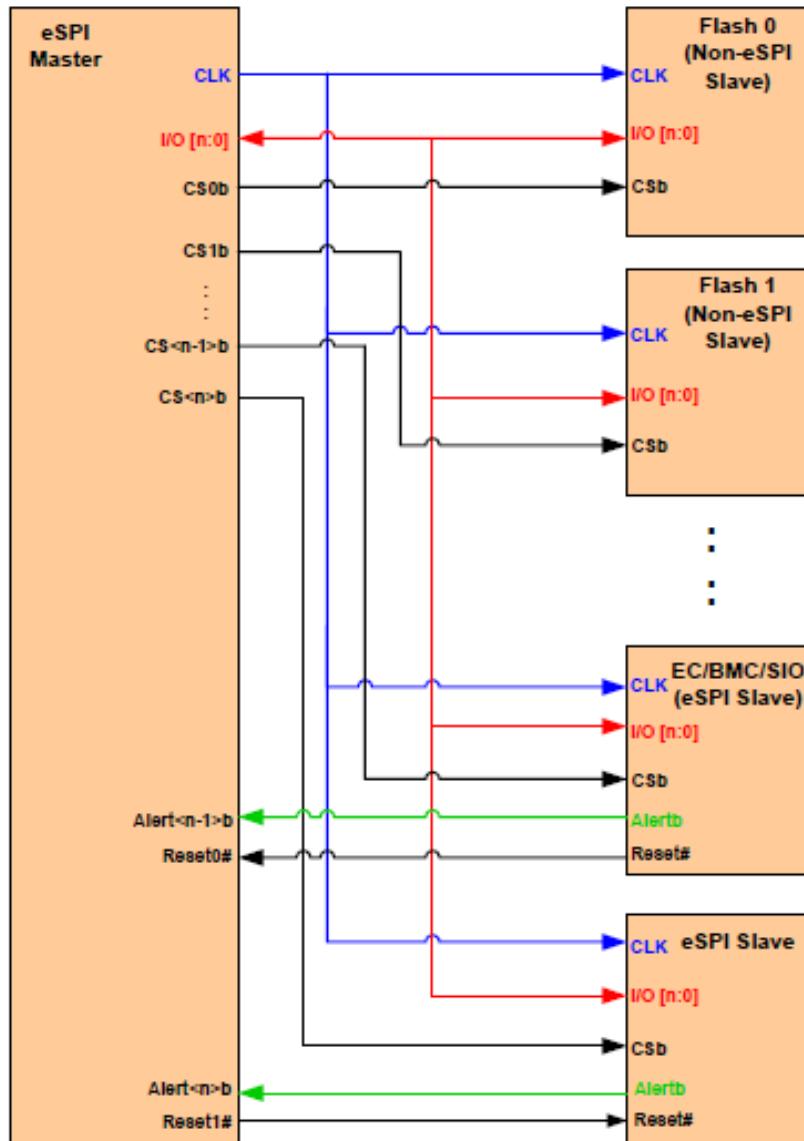


Figure 18. Single Master-Multiple Slaves with Two eSPI Reset#

Chapter-9

Introduction to BIOS

1.0 Introduction

The BIOS tells the operating system in your computer how to boot up, where to load everything, what to load, what's memory and CPU are present and much more. The term BIOS (Basic Input/output System) was created by Gary Kildall and first appeared in the CP/M operating system in 1975, describing the machine-specific part of CP/M loaded during boot time that interfaces directly with the hardware. (A CP/M machine usually has only a simple boot loader in its ROM.)

Versions of MS-DOS, PC DOS or DR-DOS contain a file called variously "IO.SYS", "IBMBIO.COM", "IBMBIO.SYS", or "DRBIOS.SYS"; this file is known as the "DOS BIOS" (also known as the "DOS I/O System") and contains the lower-level hardware-specific part of the operating system. Together with the underlying hardware-specific but operating system-independent "System BIOS", which resides in ROM, it represents the analogue to the "CP/M BIOS".

With the introduction of PS/2 machines, IBM divided the System BIOS into real- and protected-mode portions. The real-mode portion was meant to provide backward compatibility with existing operating systems such as DOS, and therefore was named "CBIOS" (for "Compatibility BIOS"), whereas the "ABIOS" (for "Advanced BIOS") provided new interfaces specifically suited for multitasking operating systems such as OS/2.



Figure 1. BIOS setup screen

The BIOS of the original IBM PC XT had no interactive user interface. Error codes or messages were displayed on the screen, or coded series of sounds were generated to signal errors when the power-on self-test (POST) had not proceeded to the point of successfully initializing a video display adapter. Options on the IBM PC and XT were set by switches and jumpers on the main

board and on peripheral cards. Starting around the mid-1990s, it became typical for the BIOS ROM to include a "BIOS configuration utility" (BCU) or "BIOS setup utility", accessed at system power-up by a particular key sequence. This program allowed the user to set system configuration options, of the type formerly set using DIP switches, through an interactive menu system controlled through the keyboard.

In the interim period, IBM-compatible PCs—including the IBM AT—held configuration settings in battery-backed RAM and used a bootable configuration program on disk, not in the ROM, to set the configuration options contained in this memory. The disk was supplied with the computer, and if it was lost the system settings could not be changed. The same applied in general to computers with an EISA bus, for which the configuration program was called an EISA Configuration Utility (ECU).

A modern Wintel-compatible computer provides a setup routine essentially unchanged in nature from the ROM-resident BIOS setup utilities of the late 1990s; the user can configure hardware options using the keyboard and video display. Also, when errors occur at boot time, a modern BIOS usually displays user-friendly error messages, often presented as pop-up boxes in a TUI style, and offers to enter the BIOS setup utility or to ignore the error and proceed if possible. Instead of battery-backed RAM, the modern Wintel machine may store the BIOS configuration settings in flash ROM, perhaps the same flash ROM that holds the BIOS itself.

2.0 BIOS ROM

BIOS software was stored on a non-volatile ROM chip on the motherboard usually in a socket which was removable. In computer systems, the BIOS contents were stored on a flash memory chip mostly an *Electrically Erasable and Programmable ROM (EEPROM)*, but now in *flash memory*, which enables firmware updates to be performed electronically. This allows BIOS software to be easily upgraded to add new features or fix bugs.

3.0 BIOS and CMOS RAM

BIOS and CMOS are often confused to each other as the setup program in the BIOS(stored in ROM) is to set and store the configuration settings in the CMOS RAM (Stored in RTC/NVRAM).

Although it is called nonvolatile, it is actually volatile, meaning that without power, the time/date settings and the data in the RAM portion will, in fact, be erased. So a battery in the system provides that power.

When you enter your BIOS Setup, configure your hard disk parameters or other BIOS Setup settings and save them, these settings are written to the storage area in the RTC/NVRAM (otherwise called CMOS RAM) chip. Every time your system boots up, it reads the parameters stored in the CMOS RAM chip to determine how the system should be configured.

One problem with ROMs such as those used for the system BIOS and video BIOS, is that it is relatively slow. The access time of ROMs is usually between 120 and 200 ns, compared to system RAM which is typically 50 to 70 ns. Also, system RAM is accessed 32 bits at a time, while ROMs

are usually 16 bits wide. The result of this is that accesses to the BIOS code are very slow relative to accesses to code in the system memory. Since there is RAM hiding underneath the ROMs anyway, most systems have the ability to "mirror" the ROM code into this RAM to improve performance. This is called ROM Shadowing, and is controlled using a set of BIOS parameters. There is normally a separate parameter to control the shadowing of the system BIOS, the video BIOS and adapter ROM areas.

4.0 POST(Power On Self Test)

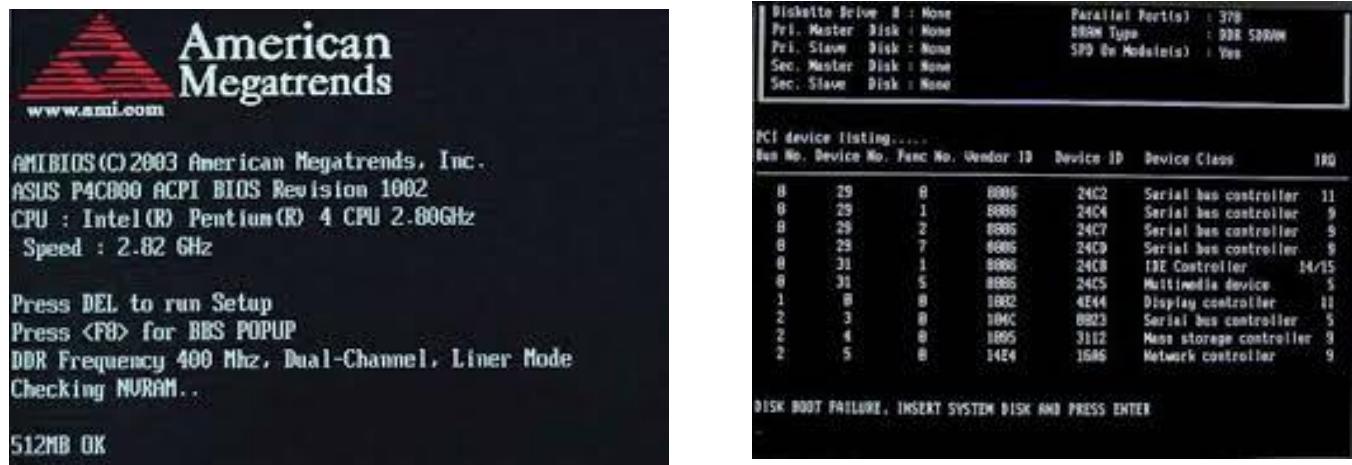


Figure 2. Display Screen during POST

when you first turn on your PC or laptop a few screens pop up. It might be a logo such as DELL or HP or ASUS, Tyan, AMI BIOS, AWARD BIOS etc. You might also see a memory count. This is all part of the POST (Power On Self Test).

POST is a test the BIOS runs before it hands over control to the OS (Operating System). It goes through several steps of checking, testing and initializing the hardware components. Before entering each step, the BIOS writes a 2 digit identifying code to an external address. This code is commonly referred to as a POST code.

5.0 POST CODES

The meaning of the codes varies widely. Most computer manufacturers use a BIOS supplied by a third party, the most common are Phoenix and AMI, but there are many others. The codes also vary depending on the motherboard and the manufacturer's requirements.

These codes are useful during the manufacturing process to help identify problems. To the consumer, they are useful in cases where the system won't boot up and the video screen does not work, or if you are attempting to make modifications.

If the BIOS detects a problem, it will stop on the problem, and the last code that was output will indicate what the failure is.

00 - 0F	Debug codes: Can be used by any PEIM/driver for debug	80 - 8F	Reserved for future use (new output console codes)
10 - 1F	Host Processors	90 - 9F	Input devices: Keyboard/ Mouse
20 - 2F	Memory/ Chipset	A0 - AF	Reserved for future use (new input console codes)
30 - 3F	Recovery	B0 - BF	Boot Devices: Includes fixed media and removable media
40 - 4F	Reserved for future use	C0 - CF	Reserved for future use
50 - 5F	I/O busses: PCI, USB, ISA,ATA, and so forth	D0 - DF	Boot device selection
60 - 6F	Not currently used	E0 - FF	E0 – EE: Miscellaneous codes F0 – FF : FF processor exception
70 - 7F	Output devices: All output consoles		

Figure 3. POST Code range

5.1 Post Code Reader

In order to see the codes being generated by your BIOS, you will need a Post Code Master Display card, which plugs into your PC and displays these codes as the BIOS performs its tests.

There are versions available for both ISA and PCI motherboard slots. The card displays HEX digits which consist of the numbers 0-9 and the letters A-F. The letters B and D are displayed as lower case (b & d).



Figure 4. POST Code Reader

5.2 Displaying POST codes

You can display POST codes using either of the following methods.

- POST card (PCI add-in card) The POST card decodes the port and displays the contents on an LED display. The POST card must be installed in PCI bus connector 1.
- Onboard POST code LED display Some Intel® Desktop Boards include an onboard LED to show POST codes

5.3 How Post Test Cards Work

Most POST test cards plug directly into expansion slots in the motherboard while a few others connect externally via a parallel or serial port. During the Power On Self Test, a two-digit code is produced and can usually be read on port 0x80. Some POST test cards include jumpers that let you modify which port to read the code from since some manufacturers use a different port. This code is created during each diagnostic step during boot up. After each piece of hardware is identified as working, the next component is checked. If an error is detected, the boot up process usually halts, and the POST test card shows the error code.

Eg. if the POST test card shows the error number 28, and Dell is the BIOS manufacturer, it means that the CMOS RAM battery has gone bad.

5.4 Port 80h POST code range

Typical port 80h POST sequence

Port 80h code values typically increase during the boot process. The early codes are for subsystems closer to the processor and the later codes are for peripherals. Generally, the order of initialization is **Processor -> Memory -> Busses -> Output/Input Devices -> Boot Devices**. The sequence of POST is system-specific.

Besides POST codes, beep codes and POST error messages are also used to detect the error prior to boot up.

Beep codes- Beep codes are audible error codes that serve a similar purpose to POST codes, but these errors don't require anything but a working internal speaker - no working screen or any need to open your computer to install and use a POST test card.

POST error messages - If the display is working, you may see a POST error message display on the screen. This is just a regular error message like what you'd expect to see at any stage of using your computer. This type of POST error code doesn't require a POST test card either.

Chapter-10

Boot process

1.0 How Does an Intel Processor Boot

When we switch on a computer, it goes through a series of steps before it is able to load the operating system. In this post we will see how a typical x86 processor boots. This is a very complex and involved process. We will only present a basic overall structure. Also what path is actually taken by the processor to reach a state where it can load an OS, is dependent on boot firmware. We will follow example of coreboot, an open source boot firmware.

1.1 Before Power is Applied

Let us start with BIOS chip, also known as boot ROM. BIOS chip is a piece of silicon on the motherboard of a computer and it can store bytes. It has two characteristics which are of interest to us. First, it (or a part of it) is memory mapped into the CPU's address space, which means that the CPU can access it in the same way it would access RAM. In particular, the CPU can point its instruction pointer to executed code inside BIOS chip. Second, the bytes that BIOS chip stores, represent the very first instructions that are executed by the CPU. BIOS chip also contains other pieces of code and data. A typical BIOS contains flash descriptor (a contents table for BIOS chip), BIOS region (the first instructions to be executed), Intel ME (Intel Management Engine) and GbE (gigabit ethernet). As you can see, BIOS chip is shared between several components of the system and not exclusive to CPU.

1.2 When power is applied

Modern Intel chips come with what is called Intel Management Engine. As soon as power is available – through battery or from mains – Intel ME comes on. It does its own initialisations which requires it to read BIOS's flash descriptor to find where Intel ME region is and then from Intel ME region of BIOS, read in code and config data. Next when we press power button on the computer, the CPU comes on. On a multiprocessor system, there is always a designated processor, called Bootstrap Processor (BSP), which comes on. In either case, the processor always comes on in what is called 16-bit Real Mode with instruction pointer pointing to address 0xffff.fff0, the reset vector.

EDIT: (thanks to [burfog](#) for indicating that this needs explanation)

You might be wondering how could a 16-bit system address 0xffff.fff0 which is clearly beyond 0xffff, the max 16-bit value? In 16-bit mode, physical address is calculated by left shifting code segment (CS) selector register by 4 bits and then adding instruction pointer (IP) address. On reset, IP contains value 0xffff0 and CS has value 0xf000 [1]. By the above formula the physical address should be:

$$CS \ll 4 + IP = 0x000f.0000 + 0xffff0 = 0x000f.fff0$$

which is still not what we expected. This is because on reset, the system is in a “special” Real Mode, where the first 12 address lines are asserted. So all addresses look like 0xffff.xxxx. This means in our case, we need to set the most significant 12 bits in the address we derived, which results in our expected address 0xffff.fff0. These 12 address lines remain asserted until a long JMP is executed, after which they are de-asserted and normal Real Mode addressing calculations resume.

The BIOS chip is also set up in such a way that first instruction to be executed from the BIOS is at physical address 0xffff.fff0 of the processor. Hence processor is able to execute the first instruction from BIOS region of the BIOS chip. This region contains what is called boot firmware. Examples of boot firmware are UEFI implementations, coreboot and the classic BIOS.

One of the first things that the boot firmware does is switch to 32-bit mode. It is also “protected mode”, i.e. segmentation is turned on and various segments of processor’s address space can be managed with different access permissions. Boot firmware however would have just one segment, effectively turning off segmentation. This is called flat mode.

2.0 Early Initialization

It is worth noting that at this point in boot process, DRAM is not available. DRAM Initialization is one of the main objectives of boot firmware. But before it can initialize DRAM, it needs to do some preparation.

Microcode patches are like patches for CPU to function correctly. Intel keeps publishing microcode patches for different CPUs. The boot firmware applies those patches very early on in boot process. Part of the processor is what is called south bridge or I/O controller hub (ICH) or peripheral controller hub (PCH). There are some initializations to be performed for ICH also. For example, ICH may contain a watchdog timer which can go off while DRAM is being initialized. That watchdog timer must be turned off first.

Of-course all of this is being done by firmware which is code written by someone. Now most of the code we know utilizes stack. But we have mentioned that DRAM hasn’t been initialized yet so there is no memory. So how is this code written and run? Answer is that this is stack-less code. Either it is hand written x86 assembly or, as in case of coreboot, it is written in C and compiled using special compiler called ROMCC which translates C to stackless assembly instructions. This of course comes with some restrictions so ROMCC compiled code is not how we want to execute everything. We need stack as soon as possible.

So, the next step is setting up what is called cache-as-RAM (CAR). Boot firmware basically sets up CPU caches so that they can be temporarily used as RAM. This way the firmware can run

code which is not stackless, but still restricted in terms of stack size and general amount of memory available.

3.0 Memory Initialization and Intel FSP

On Intel systems, memory initialization is performed using a blob called Intel Firmware Support Package (FSP). This is supplied by Intel in binary form. Intel FSP does a lot of heavy lifting when it comes to bootstrapping Intel processors and is not just limited to memory init. It is basically a three stage API. The way boot firmware interacts with FSP is set up some parameters and a return address, and jump into an FSP stage. The FSP stage would execute taking into account the parameters and then use the return address to jump back into boot firmware. This continues across these three FSP stages and in that order:

- **TempRamInit():** This performs some init for RAM and hand control back to boot firmware. Boot firmware can kick off some actions and then go on to next stage. This is because the next step performs chipset and memory initialization which may take some time. For example memory training is a time consuming operation. So this is an opportunity for boot firmware to kick off other initializations, like spinning up hard drive, which can take time to stabilize.
- **FspInitEntry():** This is where actual DRAM is achieved. This also performs other silicon init, like PCH and CPU itself. After this finishes, it passes control back to boot firmware. However, since this time, the memory has been initialized, the passing back of control and data is different from TempRamInit stage. After this stage, firmware does most of the rest of initializations – described in the next section ‘After Memory Init’ – before passing control to the next stage of FSP.
- **NotifyPhase():** This is where boot firmware would pass control back to FSP and set params which would tell FSP what sort of actions it needs to take before winding down. The types of things that FSP can do here are platform dependent but they include things like post PCI enumeration.

3.1 After Memory Init

Once DRAM is ready, it breathes a new life into boot process. First that the firmware does is copy itself into DRAM. This is done with help of “memory aliasing”, which means that reads and writes to addresses below 1MB are routed to and from DRAM. Then, firmware sets up the stack and transfer control to DRAM.

Next, some platform specific inits are done, such as GPIO configuration and re-enabling the watchdog timer in ICH which was disabled before memory init, paving the way for interrupts enabling. Local Advanced Programmable Interrupt Controller (LAPIC) sites inside each processor, i.e. it is local to each CPU in a multiprocessor system. LAPIC determines how each interrupt is delivered to that particular CPU. I/O APIC (IOxAPIC) lives inside ICH and there is one IOxAPIC for all processors. There can also be a Programmable Interrupt Controller (PIC) which is for use in Real Mode as is Interrupt Vector Table which contains 256 interrupt vectors –

pointers to handlers for corresponding interrupts. Interrupt Descriptor Table on the other hand, is used to hold interrupt vectors when in Protected Mode.

Firmware then sets up various timers depending upon platform and the firmware.

Programmable Interrupt Timer (PIT) is the system timer and sits on IRQ0. It lives inside ICH. High Precision Event Time (HPET) also sits inside ICH but boot firmware may not initialise it, letting the OS to set it up if needed. There is also a clock, the Real Time Clock (RTC) which too resides in ICH. There are other timers too, particularly LAPIC timer which is inside each CPU. Next, the firmware sets up memory caching. This basically means setting up different cache characteristics – write-back, uncached etc – for different ranges of memory.

3.2 Other Processors, I/O Devices and PCI

Finally, it is time to bring up other processors as all the work so far was being handled by the bootstrap processor. To find out about other application processors (AP) on the same package, BSP runs CPUID instruction. Then using its LAPIC, BSP sends an interrupt called SIPI, to each AP. Each SIPI points to the physical address at which the receiving AP should start executing. It is worth noting that each AP comes up in Real Mode, therefore the SIPI address must be less than 1MB, the maximum addressable in Real Mode. Usually soon after initialisation, each AP executes HLT instruction and gets into halt state, waiting for further instructions from BSP. However, just before OS gains control, APs are supposed to be in “waiting-for-SIPI” state. BSP achieves this by sending a couple of inter-processor interrupts to each AP.

Next come I/O devices like Embedded Controller (EC) and Super I/O, and after that PCI init. PCI init basically boils down to:

1. enumerating all PCI devices
2. allocating resources to each PCI device

This discussion here applies to PCIe also. PCI is a hierarchical bus system where for each bus, leaf is either a PCI device or a PCI bridge leading to another PCI bus. CPU communicates with PCI by reading and writing PCI registers. The resources needed by PCI devices are range inside memory address space, range inside I/O address space and IRQ assignment. CPU finds out about address ranges and their types (memory-mapped or I/O) by writing to and reading from Base Address Registers (BARs) of PCI devices. IRQs are usually set up based how the board is designed.

During PCI enumeration, firmware also reads Option ROM register. If that register is not empty then it contains address of Option ROM. This is ROM chip that is physically situated on the PCI device. For example the network card may contain Option ROM which holds iPXE firmware. When an Option ROM is encountered then it is read into DRAM and executed.

4.0 Handing Control to OS loader

Before handing over control to next stage loader which is usually an OS loader like GRUB2 or LILO, the firmware sets up some information inside memory which is later to be consumed by the OS. This information is things like Advanced Configuration and Power Interface (ACPI) tables and memory map itself. Memory map tells the OS what address ranges have been set up for what purposes. The regions can be general memory for OS use, ACPI related address ranges, reserved (i.e. not to be used by OS), IOAPIC (to be used by IOAPIC), LAPIC (to be used by LAPICs). Boot firmware also sets up handlers for System Management Mode (SMM) interrupts. SMM is an operating mode of Intel CPUs, just like Real, Protected and Long (64-bit) modes. A CPU enters SMM mode upon receipt of an SMM interrupt which can be triggered by a number of things like chip's temperature reaching a certain level. Before handing control to OS loader, the firmware also locks down some registers and CPU capability, so that it can't be changed afterwards by the OS.

Actual transfer of control to the OS loader usually takes form of a JMP to that part of memory. An OS loader like GRUB2 will perform actions based on its config and ultimately pass control to an operating system like Linux. For Linux, this will usually be a bzImage (big zImage, not bz compression). It is worth noting that the OS, like Linux would enumerate PCI devices again and may have other overlap with some of the final initialisations done by boot firmware. Linux usually picks up the system in 32-bit mode with paging turned off and performs its own initialisations which include setting up page tables, enabling paging and switching to long mode, i.e. 64-bit.

* How Does an Intel Processor Boot?

When we switch on a computer, it goes through a series of steps before it is able to load the operating system; in this post, we will see how a typical x86 processor boots. This is a very complex and involved process, but we will only present a basic overall structure. Also, what path is actually taken by the processor to reach a state where it can load an OS, is dependent on boot firmware. We will follow example of coreboot, an open source boot firmware.

* Before Power is Applied

Let us start with the BIOS chip, also known as boot ROM. The BIOS chip is a piece of silicon on the motherboard and it can store bytes. It has two characteristics which are of interest to us. First, it (or a part of it) is memory mapped into the CPU's address space, which means that the CPU can access it in the same way it would access RAM. In particular, the CPU can point its instruction pointer to executed code inside BIOS chip. Secondly, the bytes that the BIOS chip stores, represents the very first instructions that are executed by the CPU. The BIOS chip also contains other pieces of code and data. A typical BIOS contains a flash descriptor (a contents table for BIOS chip), BIOS region (the first instructions to be executed), Intel ME (Intel

Management Engine) and GbE (gigabit ethernet). As you can see, the BIOS chip is shared between several components of the system and is not exclusive to CPU.

* When power is applied

Modern Intel chips come with what is called Intel Management Engine. As soon as power is available, Intel ME comes on. It does its own initialisations which requires it to read the BIOS's flash descriptor to find where Intel ME region is and then from Intel ME region of BIOS, read the code and config data. Next, when we press power button on the computer, the CPU comes on. On a multiprocessor system, there is always a designated processor, called Bootstrap Processor (BSP), which comes on. In either case, the processor always comes on in what is called 16-bit Real Mode with the instruction pointer pointing to address 0xffff.fff0, the reset vector.

You might be wondering how could a 16-bit system address 0xffff.fff0 which is clearly beyond 0xffff, the max 16-bit value? In 16-bit mode, physical addresses are calculated by left shifting the code segment (CS) selector register by 4 bits and then adding instruction pointer (IP) address. On reset, IP contains value 0xffff0 and CS has value 0xf000. By the above formula the physical address should be:

$$CS \ll 4 + IP = 0x000f.0000 + 0xffff0 = 0x000f.fff0$$

which is still not what we expected. This is because on reset, the system is in a "special" Real Mode, where the first 12 address lines are asserted. So all addresses look like 0xffff.xxxx. This means in our case, we need to set the most significant 12 bits in the address we derived, which results in our expected address 0xffff.fff0. These 12 address lines remain asserted until a long JMP is executed, after which they are de-asserted and normal Real Mode addressing calculations resume.

The BIOS chip is also set up in such a way that the first instruction to be executed from the BIOS is at physical address 0xffff.fff0 of the processor. Hence the processor is able to execute the first instruction from the BIOS region of the BIOS chip; this region contains what is called boot firmware. Examples of boot firmware are UEFI implementations, coreboot and the classic BIOS.

One of the first things that the boot firmware does is switch to 32-bit mode. It is also "protected mode", i.e. segmentation is turned on and various segments of processor's address space can be managed with different access permissions. Boot firmware however would have just one segment, effectively turning off segmentation. This is called flat mode.

* Early Initialisations

It is worth noting that at this point in the boot process, DRAM is not available. DRAM

initialisation is one of the main objectives of boot firmware; but before it can initialise DRAM, it needs to do some preparation.

Microcode patches are like patches for the CPU to function correctly; Intel keeps publishing microcode patches for different CPUs. The boot firmware applies those patches very early on in boot process. Part of the processor is what is called south bridge or I/O controller hub (ICH) or peripheral controller hub (PCH). There are some initialisations to be performed for ICH also. For example, ICH may contain a watchdog timer which can go off while DRAM is being initialised. That watchdog timer must be turned off first.

Of course all of this is being done by firmware which is code written by someone, but most of the code we know utilizes the stack. However, we have mentioned that DRAM hasn't been initialised yet so there is no memory. So how is this code written and run? Answer is that this is stackless code. Either it is hand written x86 assembly or, as in case of coreboot, it is written in C and compiled using special compiler called ROMCC which translates C to stackless assembly instructions. This of course comes with some restrictions so ROMCC compiled code is not how we want to execute everything. We need the stack as soon as possible.

So, the next step is setting up what is called cache-as-RAM (CAR). Boot firmware basically sets up CPU caches so that they can be temporarily used as RAM. This way the firmware can run code which is not stackless, but still restricted in terms of stack size and general amount of memory available.

* Memory Initialisation and Intel FSP

On Intel systems, memory initialisation is performed using a blob called Intel Firmware Support Package (FSP). This is supplied by Intel in binary form. Intel FSP does a lot of heavy lifting when it comes to bootstrapping Intel processors and is not just limited to memory init. It is basically a three stage API. The way boot firmware interacts with FSP is to set up some parameters and a return address, and jump into an FSP stage. The FSP stage would execute, taking into account the parameters and then use the return address to jump back into boot firmware. This continues across these three FSP stages and in that order:

TempRamInit(): This performs some init for RAM and hands control back to boot firmware. Boot firmware can kick off some actions and then go on to next stage. This is because the next step performs chipset and memory initialisation which may take some time. For example memory training is a time consuming operation. So this is an opportunity for boot firmware to kick off other initialisations, like spinning up hard drive, which can take time to stabilise.

FspInitEntry(): This is where actual DRAM is achieved. This also performs other silicon init, like PCH and CPU itself. After this finishes, it passes control back to boot firmware. However, since this time, the memory has been initialised, the passing back of control and data is different from TempRamInit stage. After this stage, firmware does most of the rest of initialisations –

described in the next section ‘After Memory Init’ – before passing control to the next stage of FSP.

NotifyPhase(): This is where boot firmware would pass control back to FSP and set params which would tell FSP what sort of actions it needs to take before winding down. The types of things that FSP can do here are platform dependent but they include things like post PCI enumeration.

After Memory Init

Once DRAM is ready, it breathes a new life into boot process. First, the firmware copies itself into DRAM. This is done with help of “memory aliasing”, which means that reads and writes to addresses below 1MB are routed to and from DRAM. Then, firmware sets up the stack and transfer control to DRAM.

Next, some platform specific inits are done, such as GPIO configuration and re-enabling the watchdog timer in ICH which was disabled before memory init, paving the way for interrupts enabling. Local Advanced Programmable Interrupt Controller (LAPIC) sits inside each processor, i.e. it is local to each CPU in a multiprocessor system. LAPIC determines how each interrupt is delivered to that particular CPU core. I/O APIC (IOxAPIC) lives inside ICH and there is one IOxAPIC for all processors. There can also be a Programmable Interrupt Controller (PIC) which is for use in Real Mode as is Interrupt Vector Table which contains 256 interrupt vectors – pointers to handlers for corresponding interrupts. Interrupt Descriptor Tables, on the other hand, are used to hold interrupt vectors when in Protected Mode.

Firmware then sets up various timers depending upon platform and the firmware.

Programmable Interrupt Timer (PIT) is the system timer and sits on IRQ0. It lives inside ICH. High Precision Event Time (HPET) also sits inside ICH but boot firmware may not initialise it, leaving the OS to set it up if needed. There is also a clock, the Real Time Clock (RTC) which too resides in ICH. There are other timers as well, particularly LAPIC timer which is inside each CPU. Next, the firmware sets up memory caching. This basically means setting up different cache characteristics – write-back, uncached etc – for different ranges of memory.

* Other Processors, I/O Devices and PCI

Finally, it is time to bring up other processors as all the work so far was being handled by the bootstrap processor. To find out about other application processors (AP) on the same package, BSP runs CPUID instruction. Then using its LAPIC, BSP sends an interrupt called SIPI, to each AP. Each SIPI points to the physical address at which the receiving AP should start executing. It is worth noting that each AP comes up in Real Mode, therefore the SIPI address must be less than 1MB, the maximum addressable in Real Mode. Soon after initialisation, each AP executes HLT instruction and gets into halt state, waiting for further instructions from BSP. However, just before OS gains control, APs are supposed to be in a “waiting-for-SIPI” state. BSP achieves this by sending a couple of inter-processor interrupts to each AP.

Next comes I/O devices like Embedded Controller (EC) and Super I/O, and after that PCI init. PCI init basically boils down to:

1. enumerating all PCI devices
2. allocating resources to each PCI device

This discussion here applies to PCIe also. PCI is a hierarchical bus system where for each bus, leaf is either a PCI device or a PCI bridge leading to another PCI bus. CPU communicates with PCI by reading and writing PCI registers. The resources needed by PCI devices range inside memory address space, range inside I/O address space and IRQ assignment. The CPU finds out address ranges and their types (memory-mapped or I/O) by writing to and reading from Base Address Registers (BARs) of PCI devices. IRQs are usually set up based how the board is designed.

During PCI enumeration, firmware also reads Option ROM register. If that register is not empty then it contains address of Option ROM; this is a ROM chip that is physically situated on the PCI device. For example, the network card may contain Option ROM which holds iPXE firmware. When an Option ROM is encountered then it is read into DRAM and executed.

* Handing Control to OS loader

Before handing over control to the next stage loader, which is usually an OS loader like GRUB2 or LILO, the firmware sets up some information inside memory which is later to be consumed by the OS. This information is things like Advanced Configuration and Power Interface (ACPI) tables and memory map itself. Memory map tells the OS what address ranges have been set up for what purposes. The regions can be general memory for OS use, ACPI related address ranges, reserved (i.e. not to be used by OS), IOAPIC (to be used by IOAPIC), or LAPIC (to be used by LAPICs). Boot firmware also sets up handlers for System Management Mode (SMM) interrupts. SMM is an operating mode of Intel CPUs, just like Real, Protected and Long (64-bit) modes. A CPU enters SMM mode upon receipt of an SMM interrupt which can be triggered by a number of things like the chip's temperature reaching a certain level. Before handing control to the OS loader, the firmware also locks down some registers and CPU capability, so that it can't be changed afterwards by the OS.

Actual transfer of control to the OS loader usually takes form of a JMP to that part of memory. An OS loader like GRUB2 will perform actions based on its config and ultimately pass control to an operating system like Linux. For Linux, this will usually be a bzImage (big zImage, not bz compression). It is worth noting that an OS, like Linux, would enumerate PCI devices again and may have other overlap with some of the final initialisations done by boot firmware. Linux usually picks up the system in 32-bit mode with paging turned off and performs its own initialisations which include setting up page tables, enabling paging and switching to long mode, i.e. 64-bit.

5.0 Why Booting is required?

- Hardware doesn't know where the operating system resides and how to load it.
- Need a special program to do this job – **Bootstrap** loader.
 - E.g. BIOS – Boot Input Output System.
- Bootstrap loader locates the kernel, loads it into main memory and starts its execution.
- In some systems, a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel.

5.1 Boot Process Steps

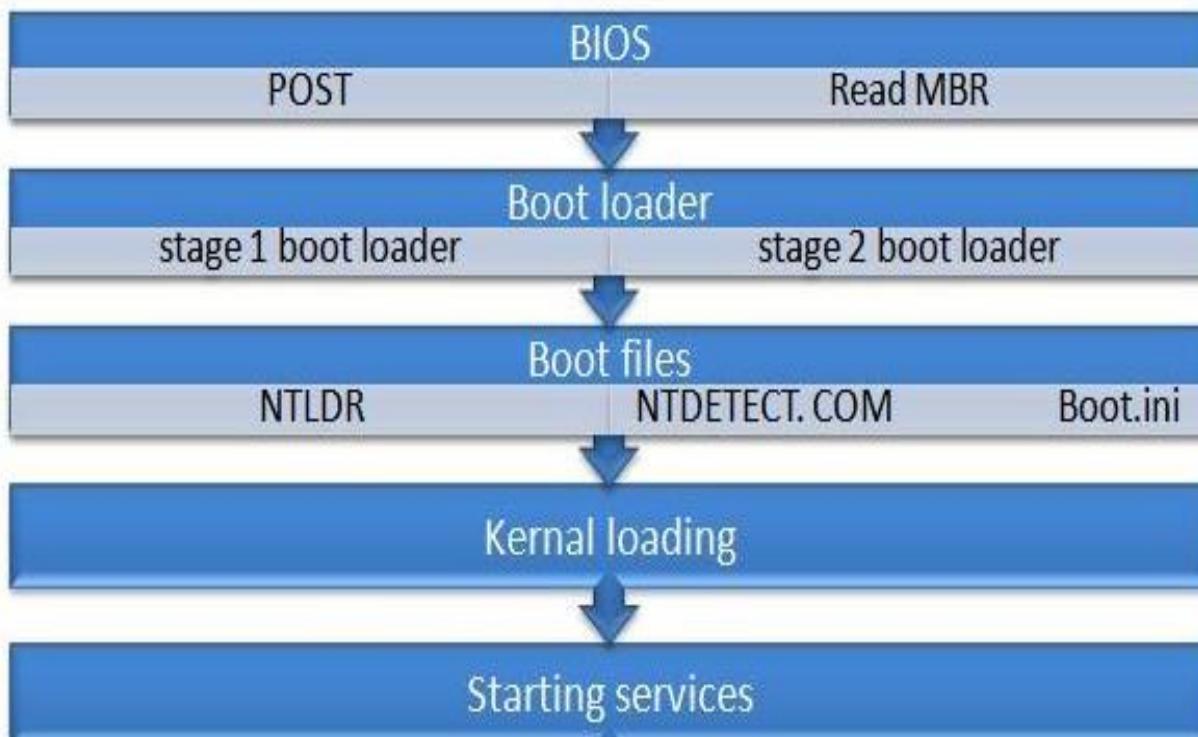


Fig-1: boot flow

- Execute code from a well-known location.
- i.e. ROM-BIOS Chip.
- Execute first-stage boot loader from MBR.
- Execute second-stage boot loader.
- Load the kernel.
- Load the first user space program.

6.0 POST (Power-On Self-Test)

- One of the first processes that a computer undergoes when booting.
- POST tests the computer to ensure that it is working as it is supposed to.
- POST can detect some errors with the processor, motherboard, RAM and other memory, as well as the video card.
- Most BIOS chips use a system of beep codes to indicate the POST status to the user and each BIOS chipset uses a different code.
- The IBM PC BIOS code standard, for example, uses one short beep to indicate a successful POST and two short beeps to indicate a POST error.

Stage 1: BIOS

- BIOS - Basic Input/output System.
- Boot firmware designed to run at start up.
- POST (Power-On Self-Test)
 - Identifies, tests, and initializes system devices
- Run-time services
 - Initial configuration
 - Selects which device to boot from
- Loads the MBR (Master Boot Record) to RAM.

Stage 2: MBR (Master Boot Record)

- After the POST the BIOS wants to boot up an operating system, which must be found somewhere: hard drives, CD-ROM drives, floppy disks, etc.
- The actual order in which the BIOS seeks a boot device is called Boot sequence and is user configurable.
- If there is no suitable boot device the BIOS halts with a complaint like “Non-System Disk or Disk Error.”
- The master boot record is always located at cylinder 0, head 0, and sector 0, the first sector on the disk
- The BIOS now reads the first 512-byte sector (sector zero) of the hard

- disk. This is called the Master Boot Record.

Stage 3: VBR (Volume Boot Record)

- Once the BIOS transfers control to the start of the MBR that was loaded into memory, the MBR code scans through its partition table and loads the Volume Boot Record for that partition.
- The Volume Boot Record is the first sector of a partition, as opposed to the first sector for the whole disk.
- The first block of the VBR identifies the partition type and size and contains an Instruction Program Loader that contains code to load additional blocks that comprise the second stage boot loader.
- On Windows NT-derived systems (e.g., Windows 2008, Windows 2012,
- Windows 7, Windows 8), the IPL loads a program called NTLDR, which then loads the operating system.

Bootloaders

- Program that allows the user to select
- which OS to boot, called from MBC
 - NTLDR (NT-Loader) comes with Windows
 - LILO, GRUB, LOADLIN from the Linux world
 - BootX from Apple

6.1 Windows Boot Sequence

- BIOS Power On Self Test
- Firmware checks hardware configuration
- Check media in boot sequence.
- For floppy drives, load the boot sector.
- For hard drives
 - Load the Master Boot Record.

- Find the active partition.
- Load boot sector.
- Load system hive(a place in which busily occupied)
- Load some drivers and services
- Initialize kernel and drivers that were loaded
- Load and initialize the rest of the drivers
- Create hardware list in registry using NTDETECT information
- Load and run Auto-check to check file systems
- Run the loaded boot code.
- Load and run NTLDR
- Load and run NTDETECT.COM
- Read BOOT.INI
- Ask which OS to start (if there is an option)
- Load and run Windows Kernel
- Load Hardware Adaptation Layer
- Set up paging
- Load services defined in the registry
- Win32 starts the logon process
- Run applications specified in the registry for this user.
- Run applications in startup directory.

6.2 Boot Errors

- If the active partition does not contain the boot components (or if you left a data diskette in the floppy drive), you will see
 - **BOOT: Couldn't find NTLDR**
 - **Please insert another disk**
- If the boot.ini file points to the partition and directory that does not contain a copy of NT, you may
- see the messages:

- Windows could not start because the following file is missing or corrupt:
- \<winnt root>\system32\ntoskrnl.exe Please re-install a copy of the above file.

7.0 Processor Initialization

- Two pins, RESET and INIT, are used to reset the processor in different manners.
- A “cold” or “power on” RESET refers to the assertion of RESET while power is initially being applied to the processor.
- A “warm” RESET refers to the assertion of RESET or INIT while VCC and CLK remain within specified operating limits.
- CPU can't simply jump up and start fetching code from flash memory. When external power is first applied, the hardware platform must carry out a number of tasks before the processor can be brought out of its reset state.
- Once the processor reset line has been de-asserted, the processor begins fetching instructions. The location of these instructions is known as the reset vector. The reset vector may contain instructions or a pointer to the starting instructions in flash memory. The location of the vector is architecture-specific and usually in a fixed location, depending on the processor.
- The initial address must be a physical address, as the Memory Management Unit (MMU), if it exists, has not yet been enabled. The first fetching instructions start at 0xFFFF, FFFF.
- processor cache is not enabled by default.
- **A multiprocessor system does not truly enter multiprocessing operation until the OS takes over.**
- **While it is possible to do a limited amount of parallel processing during the UEFI boot phase, such as during memory initialization with multiple socket designs, any true multithreading activity would require changes to be made to the Driver Execution Environment (DXE) phase of the UEFI. Without obvious benefits, such changes are unlikely to be broadly adopted.**

7.1 Memory Initialization

- The initialization of the memory controller varies slightly depending on the DRAM technology and the capabilities of the memory controller itself.

- There is a very wide range of DRAM configuration parameters, including number of ranks, eight-bit or 16-bit addresses, overall memory size and constellation, soldered down or add-in module configurations, page-closing policy, and power management.
- If the platform supports add-in-modules for memory, it may use any of a number of standard form factors. The small-outline Dual In-Line Memory Module (DIMM) is often found in embedded systems. The DIMMs provide a serial EEPROM that contains DRAM configuration information known as Serial Presence Detect (SPD) data. The firmware reads the SPD data and subsequently configures the device. The serial EEPROM is connected via the System Management Bus (SMBus). This means the device must be available in the early initialization phase so the software can establish the memory devices on-board. It is also possible for memory-down motherboards to incorporate SPD EEPROMs to allow for multiple and updatable memory configurations that can be handled efficiently by a single BIOS algorithm. A hard-coded table in one of the MRC files could be used to implement an EEPROM-less design.

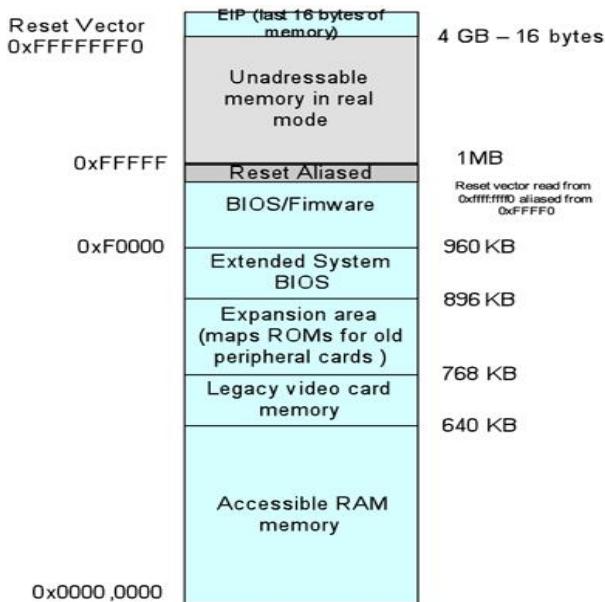


Fig 2: RAM Initialization

- The early initialization phase next readies the bootstrap processor and I/O peripherals' base address registers, which are needed to configure the memory controller.
- Once the memory controller has been initialized, a number of subsequent cleanup events take place, including tests to ensure that memory is operational.
- From the reset vector, execution starts directly from nonvolatile flash storage. This operating mode is known as execute-in-place. The read performance of nonvolatile storage is much slower than the read performance of DRAM. The performance of code running from flash is therefore much lower than code executed in RAM. Most firmware

is therefore copied from slower nonvolatile storage into RAM. The firmware is then executed in RAM in a process known as shadowing.

- Intel Architecture systems generally do not execute-in-place for anything but the initial boot steps before memory has been configured. The firmware is often compressed, allowing reduction of nonvolatile RAM requirements. Clearly, the processor cannot execute a compressed image in place. There is a trade-off between the size of data to be shadowed and the act of decompression. Pre-fetchers in the processor, if enabled, may speed up execution-in-place. Some SOCs have internal NVRAM cache buffers to assist in pipelining the data from the flash to the processor.
- Before memory is initialized, the data and code stacks are held in the processor cache. Once memory is initialized, the system must exit that special caching mode and flush the cache. The stack will be transferred to a new location in main memory and cache reconfigured.
- The stack must be set up before jumping into the shadowed portion of the BIOS that is now in memory. A memory location must be chosen for stack space. The stack will count down so the top of the stack must be entered and enough memory must be allocated for the maximum stack.
- If the system is in real mode, then set with the appropriate values. If protected mode is used then set to the correct memory location. This is where the code makes the jump into memory. If a memory test has not been performed before this point, the jump could very well be to garbage. System failures indicated by a Power-On Self Test (POST) code between "end of memory initialization" and the first following POST code almost always indicate a catastrophic memory initialization problem. If this is a new design, then chances are this is in the hardware and requires step-by-step debug.
- For legacy option ROMs and BIOS memory ranges, Intel chipsets usually come with memory aliasing capabilities that allow access to memory below 1 MB to be routed to or from DRAM or nonvolatile storage located just under 4 GB.
- For shadowing, if PAM registers remain at default values. Shadowing can be used to improve boot speed. This will direct reads to the flash device and writes to memory. Data can then be shadowed into memory by reading and writing the same address. Once BIOS code has been shadowed into memory, the enables can be changed to read-only mode so memory reads are directed to memory. This also prevents accidental overwriting of the image in memory.
- After memory is started, the remaining processors are initialized and left in a wait-for-SIPI state. To accomplish this, the system firmware must:
 - Find microcode and copy it to memory.

- Find the CPU code in the Serial Peripherals Interface (SPI) and copy it to memory — an important step to avoid execution-in-place for the remainder of the sequence.
- Send start-up inter-processor interrupts to all processors.
- Disable all NEM settings, if this has not already been done.
- Load microcode updates on all processors.
- Enable cache-on mode for all processors.

8.0 Option ROM

- Option ROMs (or OpROMs) are firmware run by the PC BIOS during platform initialization. They are usually stored on a plug-in card, though they can reside on the system board.
- Devices that typically require option ROMs are video cards, network adapters, and storage drivers for RAID modules. These option ROMs also typically provide firmware drivers to the PC.
- The Unified Extensible Firmware Interface (UEFI) has support for Legacy mode option ROMs.
- As per latest UEFI specification (currently at 2.3.1 Errata C – section 2.5.1.2), ISA (legacy) option ROMs are not a part of the UEFI Specification. For the purposes of this discussion, only PCI-based UEFI-compatible option ROMs will be considered.
- Option ROMs can be used when it's not be possible to embed a device's firmware in the PC firmware. When the option ROM carries the driver, the IHV can leverage that driver, and keep the driver and device in one place.

9.0 Graphics Enabling

video BIOS

- A video BIOS or VBIOS is the Basic Input Output System (BIOS) of a graphics card or the integrated graphics controller in a computer.
- The VBIOS provides a set of video-related functions that are used by programs to access the video hardware. An example of the video hardware is the integrated graphics controller within an Intel Chipset.
- When the computer is started, it usually displays the graphics card vendor, model, BIOS version and the amount of graphics memory.

Chapter-11

Operating Systems (OS)

Introduction

The operating system acts as an interface between the hardware and the programs requesting I/O. It is the most fundamental of all system software programs.

Responsibilities of the OS include:

- Hiding the complexities of hardware from the user.
- Managing between the hardware's resources that include the processors, memory, data storage and I/O devices.
- Handling "interrupts" generated by the I/O controllers.
- Sharing of I/O between many programs using the CPU.

There are two types of software:

- System Software - programs that manage the operation of a computer.
- Application Software - programs that help the user perform a particular task.

Need of Operating System

- OS as a platform for Application programs: Operating system provides a platform, on top of which, other programs, called application programs can run. These application programs help the users to perform a specific task easily. It acts as an interface between the computer and the user. It's designed in such a manner that it operates controls and executes various applications on the computer.
- Managing Input-Output unit: Operating System also allows the computer to manage its own resources such as memory, monitor, keyboard, printer etc. Management of these resources is required for an effective utilization. The operating system controls the various system input-output resources and allocates them to the users or programs as per their requirement.
- Consistent user interface: Operating System provides the user an easy-to-work user interface, so the user does not have to learn a different UI every time and can focus on the content and be productive as quickly as possible. Operating System provides templates, UI components to make the working of a computer, easy for the user.
- Multitasking: Operating System manages memory and allow multiple programs to run in their own space and even communicate with each other through shared memory. Multitasking gives users a good experience as they can perform several tasks on a computer at a time.

Functions of an Operating System

An operating system has variety of functions to perform. Some of the prominent functions of an operating system can be broadly outlined as:

- Processor Management:

This deals with management of the Central Processing Unit (CPU). The operating system takes care of the allotment of CPU time to different processes. When a process finishes its CPU processing after executing for the allotted time period, this is called scheduling. There are various type of scheduling techniques that are used by the operating systems:

1. Shortest Job First (SJF): Processes, which need the shortest CPU time are scheduled first.
2. Round Robin Scheduling: Each process is assigned a fixed CPU execution time in cyclic way.
3. Priority Based scheduling (Non Pre-emptive): In this scheduling, processes are scheduled according to their priorities, i.e., highest priority process is scheduled first. If priorities of two processes match, then schedule according to arrival time

- Device Management:

The Operating System communicates with hardware and the attached devices and maintains a balance between them and the CPU. This is all the more important because the CPU processing speed is much higher than that of I/O devices. In order to optimize the CPU time, the operating system employs two techniques – Buffering and Spooling.

- Buffering:

In this technique, input and output data is temporarily stored in Input Buffer and Output Buffer. Once the signal for input or output is sent to or from the CPU respectively, the operating system through the device controller moves the data from the input device to the input buffer and for the output device to the output buffer. In case of input, if the buffer is full, the operating system sends a signal to the program, which processes the data stored in the buffer. When the buffer becomes empty, the program informs the operating system, which reloads the buffer, and the input operation continues.

- Spooling (Simultaneous Peripheral Operation on Line):

This is a device management technique used for processing of different tasks on the same input/output device. When there are various users on a network sharing the same resource then it can be a possibility, that more than one user might give it a command at the same point of time. So, the operating system temporarily stores the data of every user on the hard disk of the computer to which the resource is attached. The individual user need not wait for the execution process to be completed. Instead, the operating system sends the data from the hard disk to the resource one by one.

Example: printer

- Memory management:

In a computer, both the CPU and the I/O devices interact with the memory. When a program needs to be executed, it is loaded onto the main memory till the execution is completed. Thereafter that memory space is freed and is available for other programs. The common memory management techniques used by the operating system are Partitioning and Virtual Memory.

- Partitioning:

The total memory is divided into various partitions of same size or different sizes. This helps to accommodate number of programs in the memory. The partition can be fixed i.e. remains same for all the programs in the memory or variable i.e. memory is allocated when

a program is loaded on to the memory. The later approach causes less wastage of memory but in due course of time, it may become fragmented.

- **Virtual Memory:**

This is a technique used by the operating systems which allow the user can load the programs which are larger than the main memory of the computer. In this technique the program is executed even if the complete program can not be loaded inside the main memory leading to efficient memory utilization.

- **File Management:**

The operating System manages the files, folders and directory systems on a computer. Any data on a computer is stored in the form of files and the operating system keeps information about all of them using File Allocation Table (FAT). The FAT stores general information about files like filename, type (text or binary), size, starting address and access mode (sequential/indexed sequential/direct/relative). The file manager of the operating system helps to create, edit, copy, allocate memory to the files and also updates the FAT. The operating system also takes care that files are opened with proper access rights to read or edit them.

OS Organization

- Layered
- Monolithic
- Micro kernel
- Hybrid Kernel (Distributed)

Layered

- The operating system is split into various layers in the layered operating system and each of the layers have different functionalities. This type of operating system was created as an improvement over the early monolithic systems.
- Layering provides a distinct advantage in an operating system. All the layers can be defined separately and interact with each other as required. In addition, it is easier to create, maintain and update the system if it is done in the form of layers. Change in one-layer specification does not affect the rest of the layers.
- Each of the layers in the operating system can only interact with the layers that are above and below it. The lowest layer handles the hardware and the uppermost layer deals with the user applications.

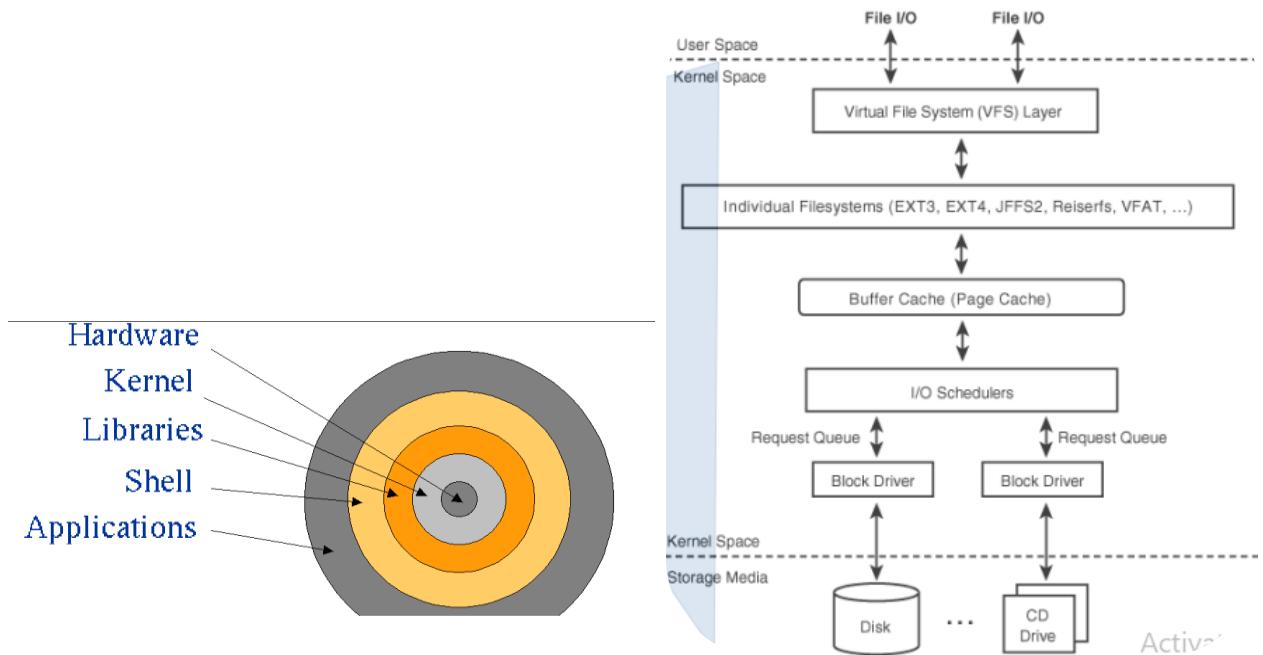


Figure 1: Layered Architecture

Monolithic

A monolithic kernel is an operating system architecture where the entire operating system is working in kernel space. The monolithic model differs from other operating system architectures (such as the microkernel architecture) in that it alone defines a high-level virtual interface over computer hardware. A set of primitives or system calls implement all operating system services such as process management, concurrency, and memory management. Device drivers can be added to the kernel as modules.

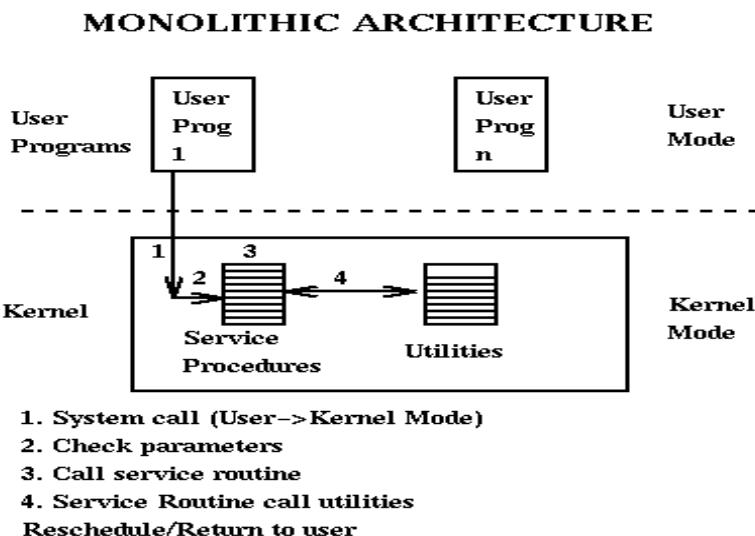
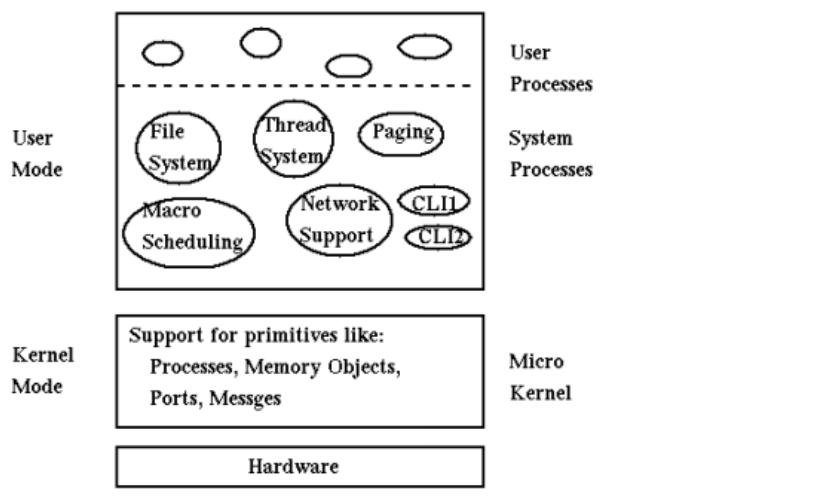


Figure2: Monolithic Architecture

Micro Kernel

Microkernel is one of the classification of the kernel. Being a kernel, it manages all system resources. However, in a microkernel, the user services and kernel services are implemented in different address space. The user services are kept in user address space, and kernel services are kept under kernel address space, thus also reduces the size of kernel and size of operating system as well.

It provides minimal services of process and memory management. The communication between client program/application and services running in user address space is established through message passing, reducing the speed of execution microkernel. The Operating System remains unaffected as user services and kernel services are isolated so if any user service fails it does not affect kernel service. Thus it adds to one of the advantages in a microkernel. It is easily extendable i.e. if any new services are to be added they are added to user address space and hence requires no modification in kernel space. It is also portable, secure and reliable.



Micro-Kernel Architecture

Minix Layered Micro Kernel Architecture

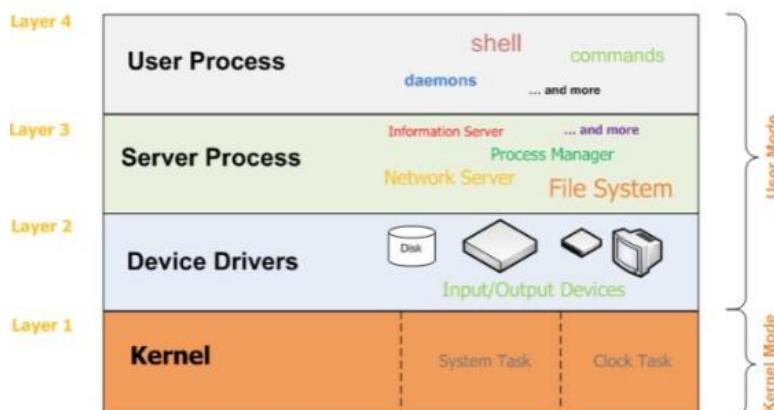
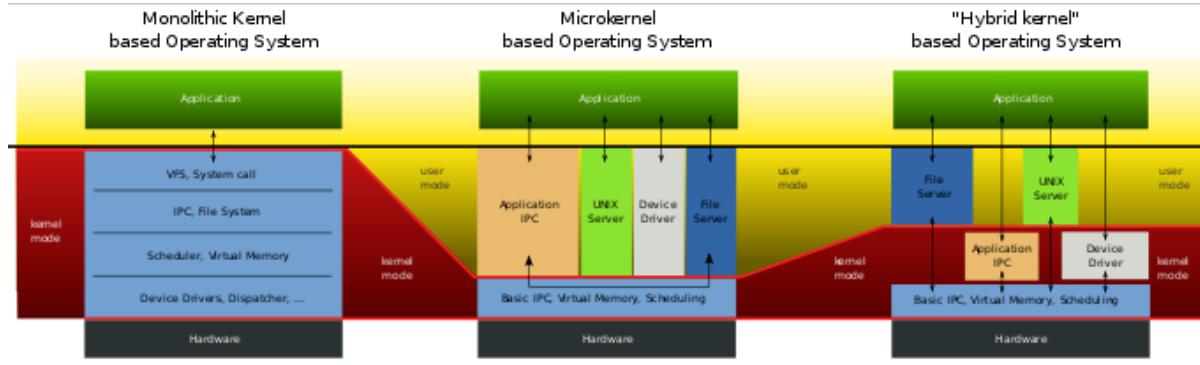


Figure 3: Micro Kernel Architecture

Hybrid Kernel

A hybrid kernel runs some services in the kernel space to reduce the performance overhead of a traditional microkernel, while still running kernel code as servers in the user space. For instance, a hybrid kernel design may keep the virtual files system and bus controllers inside the kernel and the file system drivers and storage drivers as user mode programs outside the kernel. Such a design keeps the performance and design principles of a monolithic kernel.



The Microsoft NT kernel is a well-known example of a hybrid kernel that powers Windows NT, Windows 2000, Windows XP, Windows Server 2003, Windows Vista, Windows Server 2008 and Windows 7. It is referred to as a monolithic kernel as emulation subsystems run in the user mode server processes. One of the most important thing about it is its structure, which is a collection of modules that communicate via well-known interfaces, with a small microkernel limited to core functions such as first-level interrupt handling, thread scheduling and synchronization primitives. This allows for the possibility of using either direct procedure calls or inter-process communication to communicate between modules, and hence for the potential location of modules in different address spaces.

Figure 4: Hybrid Architecture

Distributed Operating System

Distributed Operating System is a model where distributed applications are running on multiple computers linked by communications. A distributed operating system is an extension of the network operating system that supports higher levels of communication and integration of the machines on the network.

This system looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units (CPUs).

These systems are referred as *loosely coupled systems* where each processor has its own local memory and processors communicate with one another through various communication lines, such as high speed buses or telephone lines. By loosely coupled systems, we mean that such computers possess no hardware connections at the CPU - memory bus level, but are connected by external interfaces that run under the control of software.

The Distributed Os involves a collection of autonomous computer systems, capable of communicating and cooperating with each other through a LAN / WAN. A Distributed Os provides a virtual machine abstraction to its users and wide sharing of resources like as computational capacity, I/O and files etc.

The structure shown in fig contains a set of individual computer systems and workstations connected via communication systems, but by this structure we cannot say it is a distributed system because it is the software, not the hardware, that determines whether a system is distributed or not.

The users of a true distributed system should not know, on which machine their programs are running and where their files are stored. LOCUS and MICROS are the best examples of distributed operating systems.

Using LOCUS operating system it was possible to access local and distant files in uniform manner. This feature enabled a user to log on any node of the network and to utilize the resources in a network without the reference of his/her location. MICROS provided sharing of resources in an automatic manner. The jobs were assigned to different nodes of the whole system to balance the load on different nodes.

Below given are some of the examples of distributed operating systems:

1. IRIX operating system; is the implementation of UNIX System V, Release 3 for Silicon Graphics multiprocessor workstations.
2. DYNIX operating system running on Sequent Symmetry multiprocessor computers.
3. AIX operating system for IBM RS/6000 computers.
4. Solaris operating system for SUN multiprocessor workstations.
5. Mach/OS is a multithreading and multitasking UNIX compatible operating system;
6. OSF/1 operating system developed by Open Foundation Software: UNIX compatible.

Distributed systems provide the following advantages:

- 1 Sharing of resources.
- 2 Reliability.
- 3 Communication.
- 4 Computation speedup

Distributed systems are potentially more reliable than a central system because if a system has only one instance of some critical component, such as a CPU, disk, or network interface, and that component fails, the system will go down. When there are multiple instances, the system may be able to continue in spite of occasional failures. In addition to hardware failures, one can also consider software failures. Distributed systems allow both hardware and software errors to be dealt with.

A distributed system is a set of computers that communicate and collaborate each other using software and hardware interconnecting components. Multiprocessors (MIMD computers using

shared memory architecture), multicomputers connected through static or dynamic interconnection networks (MIMD computers using message passing architecture) and workstations connected through local area network are examples of such distributed systems.

A distributed system is managed by a distributed operating system. A distributed operating system manages the system shared resources used by multiple processes, the process scheduling activity (how processes are allocating on available processors), the communication and synchronization between running processes and so on. The software for parallel computers could be also tightly coupled or loosely coupled. The loosely coupled software allows computers and users of a distributed system to be independent each other but having a limited possibility to cooperate. An example of such a system is a group of computers connected through a local network. Every computer has its own memory, hard disk. There are some shared resources such files and [printers](#). If the interconnection network broke down, individual computers could be used but without some features like printing to a non-local printer.

Basic OS Architecture

A general purpose OS is composed of:

- Process manager
 - multiplexes the CPU time between the multiple execution units (processes)
- memory manager
 - controls, manages and multiplexes the access to physical and virtual memory
- Inter-process communication
 - Implements and handles mechanism for processes to communicate
- I/O manager
 - Manages communication with peripheral (keyboard/screen, disk, network)
- User interface
 - command line interpreter
 - GUI
- File System manager
 - manages and organizes data available on disks (file systems)
- Function calls
 - Programming functions that allow applications to use OS services (memory, disk, I/O)

Virtual Machine

A virtual machine (VM) is a software program or operating system that not only exhibits the behaviour of a separate computer, but is also capable of performing tasks such as running applications and programs like a separate computer. A virtual machine, usually known as a guest is created within another computing environment referred as a "host." Multiple virtual machines can exist within a single host at one time.

A virtual machine is also known as a guest.

Virtual machines are becoming more common with the evolution of virtualization technology. Virtual machines are often created to perform certain tasks that are different than tasks performed in a host environment.

Virtual machines are implemented by software emulation methods or hardware virtualization techniques. Depending on their use and level of correspondence to any physical computer, virtual machines can be divided into two categories:

1. System Virtual Machines: A system platform that supports the sharing of the host computer's physical resources between multiple virtual machines, each running with its own copy of the operating system. The virtualization technique is provided by a software layer known as a hypervisor, which can run either on bare hardware or on top of an operating system.
2. Process Virtual Machine: Designed to provide a platform-independent programming environment that masks the information of the underlying hardware or operating system and allows program execution to take place in the same way on any given platform.

Some of the advantages of a virtual machine include:

- Allows multiple operating system environments on a single physical computer without any intervention
- Virtual machines are widely available and are easy to manage and maintain.
- Offers application provisioning and disaster recovery options

Some of the drawbacks of virtual machines include:

- They are not as efficient as a physical computer because the hardware resources are distributed in an indirect way.
- Multiple VMs running on a single physical machine can deliver unstable performance

Requirements of memory management system

Memory management keeps track of the status of each memory location, whether it is allocated or free. It allocates the memory dynamically to the programs at their request and frees it for reuse when it is no longer needed. Memory management meant to satisfy some requirements that we should keep in mind.

These Requirements of memory management are:

Relocation – The available memory is generally shared among a number of processes in a multiprogramming system, so it is not possible to know in advance which other programs will be resident in main memory at the time of execution of his program. Swapping the active processes in and out of the main memory enables the operating system to have a larger pool of ready-to-execute process.

When a program gets swapped out to a disk memory, then it is not always possible that when it is swapped back into main memory then it occupies the previous memory location, since the location may still be occupied by another process. We may need to relocate the process to a different area of memory. Thus there is a possibility that program may be moved in main memory due to swapping.

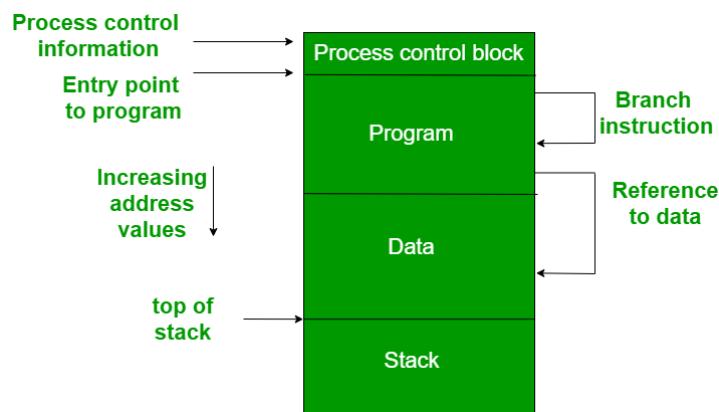


Figure: Memory layout

1. The figure depicts a process image. The process image is occupying a continuous region of main memory. The operating system will need to know many things including the location of process control information, the execution stack, and the code entry. Within a program, there are memory references in various instructions and these are called logical addresses.

After loading of the program into main memory, the processor and the operating system must be able to translate logical addresses into physical addresses. Branch instructions contain the address of the next instruction to be executed. Data reference instructions contain the address of byte or word of data referenced.

2. **Protection** – There is always a danger when we have multiple programs at the same time as one program may write to the address space of another program. So every process must be protected against unwanted interference when other process tries to write in a process whether accidental or incidental. Between relocation and protection requirement a trade-off occurs as the satisfaction of relocation requirement increases the difficulty of satisfying the protection requirement.

Prediction of the location of a program in main memory is not possible, that's why it is impossible to check the absolute address at compile time to assure protection. Most of

the programming language allows the dynamic calculation of address at run time. The memory protection requirement must be satisfied by the processor rather than the operating system because the operating system can hardly control a process when it occupies the processor. Thus it is possible to check the validity of memory references.

3. **Sharing** – A protection mechanism must have to allow several processes to access the same portion of main memory. Allowing each processes access to the same copy of the program rather than have their own separate copy has an advantage.

For example, multiple processes may use the same system file and it is natural to load one copy of the file in main memory and let it shared by those processes. It is the task of Memory management to allow controlled access to the shared areas of memory without compromising the protection. Mechanisms are used to support relocation supported sharing capabilities.

4. **Logical organization** – Main memory is organized as linear or it can be a one-dimensional address space which consists of a sequence of bytes or words. Most of the programs can be organized into modules, some of those are unmodifiable (read-only, execute only) and some of those contain data that can be modified. To effectively deal with a user program, the operating system and computer hardware must support a basic module to provide the required protection and sharing. It has the following advantages:

- Modules are written and compiled independently and all the references from one module to another module are resolved by 'the system at run time.
- Different modules are provided with different degrees of protection.
- There are mechanisms by which modules can be shared among processes. Sharing can be provided on a module level that lets the user specify the sharing that is desired.

5. **Physical organization** – The structure of computer memory has two levels referred to as main memory and secondary memory. Main memory is relatively very fast and costly as compared to the secondary memory. Main memory is volatile. Thus secondary memory is provided for storage of data on a long-term basis while the main memory holds currently used programs. The major system concern between main memory and secondary memory is the flow of information and it is impractical for programmers to understand this for two reasons:

- The programmer may engage in a practice known as overlaying when the main memory available for a program and its data may be insufficient. It allows different modules to be assigned to the same region of memory. One disadvantage is that it is time-consuming for the programmer.
- In a multiprogramming environment, the programmer does not know how much space will be available at the time of coding and where that space will be located inside the memory.

Basics of Windows Architecture

Let's dig deeper into the internal structure and the role each key operating system component plays. The figure below is more detailed diagram of the core Windows system

architecture and components. Note that it still does not show all components (networking in particular).

Windows Architecture

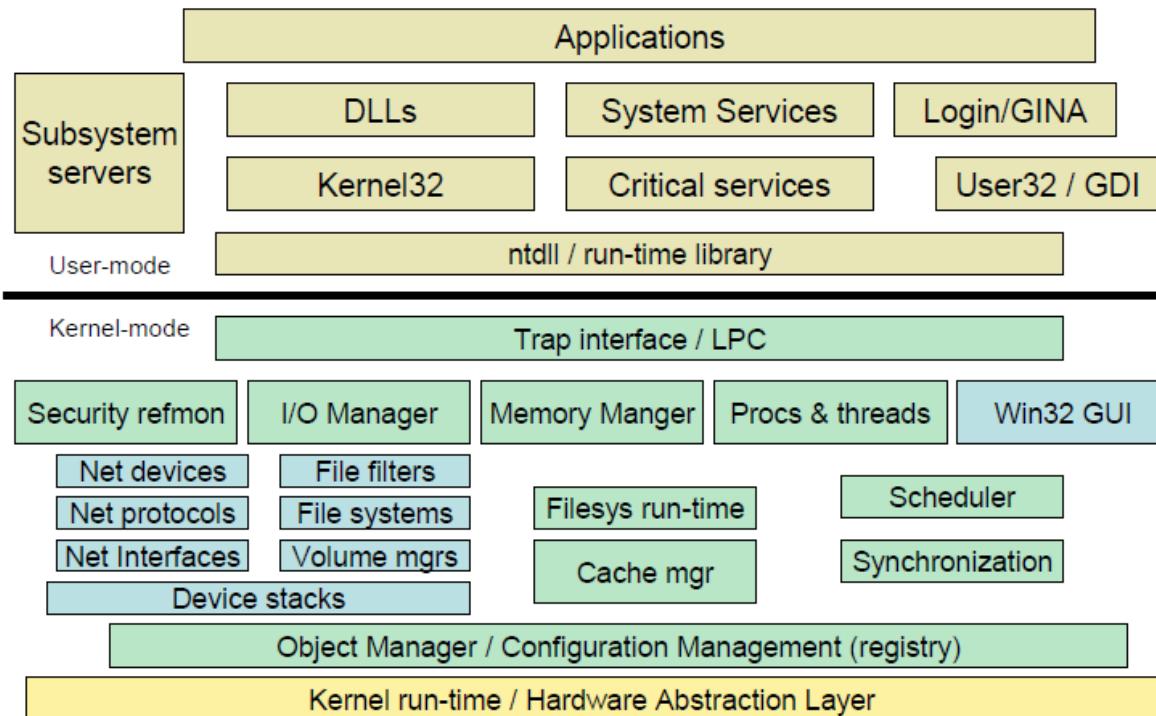


Figure 5: simplified windows Architecture

First notice the line dividing the user-mode and kernel-mode parts of the Windows operating system. The boxes above the line represent user-mode processes, and the components below the line are kernel-mode operating system services.

User and kernel modes are two processor access modes, where a kernel mode refers to a mode of execution privilege that grants access to system memory and all CPU instructions.

User mode is a less privileged processor mode than kernel mode. It uses well-defined operating system application program interfaces (APIs) to request system services. A **User mode** process:

- Have no direct access to hardware or kernel memory (Only kernel mode processes can access kernel resources as a way of protection).
- Is limited to an assigned address space.

- Can be paged out of physical memory into virtual RAM on a hard disk.
- Process at a lower priority than kernel mode components (OS components). Which means that the OS does not slow down or have to wait while an application finishes processing.
- Cannot access another user process address space (Unless opened a handle to the process, which means passing through security access check).

Before we continue let's just have a quick definition for virtual memory. Virtual memory, also known as virtual RAM, allows hard disk space to be used as if it were additional memory. In this manner, the user mode processes have access to more memory than is actually available to them.

Top right of the figure shows the environment subsystems. **Environment subsystems** provide exposed, documented interface between applications and Windows native APIs (undocumented APIs). The environment subsystem translates environment-specific instructions from an application into instructions that the Executive Services (First layer of the OS kernel) can carry out. Each subsystem can provide access to different subsets of the native services in Windows.

Environment subsystems include POSIX, OS/2 and Windows subsystems. The Windows subsystem major components consist of the environment subsystem process (Csrss.exe which you can see tuning in the task manager), the kernel-mode device driver (Win32k.sys), subsystem DLLs (such as Kernel32.dll, Advapi32.dll, User32.dll, and Gdi32.dll) and Graphics device drivers.

In the user mode and just above the line that divides the user and kernel modes is the Ntdll.dll. **Ntdll.dll** is a special system support library primarily for the use of subsystem DLLs.

Now let's move on to the kernel mode. Kernel mode is the privileged mode of operation in which the code has direct access to all hardware and all memory, including the address spaces of all user mode processes. **Kernel mode** components:

- Can access hardware directly.
- Can access all of the memory on the computer.
- Are not moved to the virtual memory page file on the hard disk.
- Process at a higher priority than user mode processes.

The kernel mode in Windows is comprised of the Windows Executive, which includes the Executive Services, the kernel, and the hardware abstraction layer (HAL).

The Windows executive is the upper layer of Ntoskrnl.exe. (The kernel is the lower layer.) The executive provides core OS services. The executive contains major components such as various modules that manage I/O, objects, security, processes, inter-process communications (IPC), virtual memory, and window and graphics management. It also includes device drivers (defined in previous post) functions.

The kernel consists of a set of functions in Ntoskrnl.exe provides the most basic operating system services, such as thread scheduling, first-level interrupt handling, and deferred procedure calls. The kernel resides between the Executive Services and HAL layers.

The other major job of the kernel is to abstract or isolate the executive and device drivers from variations between the hardware architectures supported by Windows.

One of the crucial elements of the Windows design is its portability across a variety of hardware platforms. **The hardware abstraction layer (HAL)** is a key part of making this portability possible. The HAL is a loadable kernel-mode module (Hal.dll) enables the same operating system to run on different platforms with different processors.

Also part of the kernel is the **device drivers**. Device drivers in Windows don't manipulate hardware directly, but rather they call functions in the HAL to interface with the hardware.

Windows OS Features

Extensibility: Code must be able to grow and change as market requirements change.

- **Portability:** The system must be able to run on multiple hardware architectures and must be able to move with relative ease to new ones as market demands dictate.
- **Dependability:** Protection against internal malfunction and external tampering.
- Applications should not be able to harm the OS or other running applications.
- **Compatibility:** User interface and APIs should be compatible with older versions of Windows as well as older operating systems such as MS-DOS.
- It should also interoperate well with UNIX, OS/2, and NetWare.
- **Performance:** Within the constraints of the other design goals, the system should be as fast and responsive as possible on each hardware platform.

Processes and Threads Basics

An application consists of one or more processes. A *process*, in the simplest terms, is an executing program. One or more threads run in the context of the process. A *thread* is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread.

A *job object* allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on the job object affect all processes associated with the job object.

A *thread pool* is a collection of worker threads that efficiently execute asynchronous callbacks on behalf of the application. The thread pool is primarily used to reduce the number of application threads and provide management of the worker threads.

A *fiber* is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them.

User-mode scheduling (UMS) is a lightweight mechanism that applications can use to schedule their own threads. UMS threads differ from fibers in that each UMS thread has its own thread context instead of sharing the thread context of a single thread.

For more Details and for more information, see the following topics:

- Multitasking
- Scheduling
- Multiple Threads
- Child Processes
- Thread Pools
- Job Objects
- User-Mode Scheduling
- Fibers

Process Table and Process Control Block (PCB)

While creating a process the operating system performs several operations. To identify these processes, it must identify each process, hence it assigns a process identification number (PID) to each process. As the operating system supports multi-programming, it needs to keep track of all the processes. For this task, the process control block (PCB) is used to track the process's execution status. Each block of memory contains information about the process state, program counter, stack pointer, status of opened files, scheduling algorithms, etc. All this information is required and must be saved when the process is switched from one state to another. When the process made transitions from one state to another, the operating system must update information in the process's PCB.

A process control block (PCB) contains information about the process, i.e. registers, quantum, priority, etc. The process table is an array of PCB's, that means logically contains a PCB for all of the current processes in the system.

Pointer – It is a stack pointer which is required to be saved when the process is switched from one state to another to retain the current position of the process.

Process state – It stores the respective state of the process.

Process number – Every process is assigned with a unique id known as process ID or PID which stores the process identifier.

Program counter – It stores the counter which contains the address of the next instruction that is to be executed for the process.

Register – These are the CPU registers which include: accumulator, base, registers and general purpose registers.

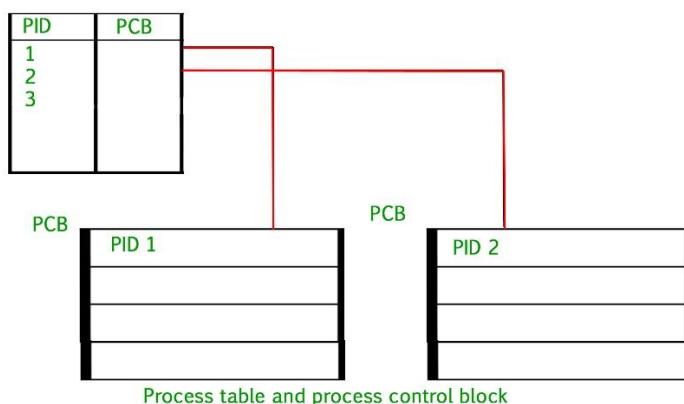
Memory limits – This field contains the information about memory management system used by operating system. This may include the page tables, segment tables etc.

Open files list – This information includes the list of files opened for a process.



Miscellaneous accounting and status data – This field includes information about the amount of CPU used, time constraints, jobs or process number, etc.

The process control block stores the register content also known as execution content of the processor when it was blocked from running. This execution content architecture enables the operating system to restore a process's execution context when the process returns to the running state. When the process made transitions from one state to another, the operating system update its information in the process's PCB. The operating system maintains pointers to each process's PCB in a process table so that it can access the PCB quickly.



Kernel-Mode Vs User-Mode

A processor in a computer running Windows has two different modes: *user mode* and *kernel mode*. The processor switches between the two modes depending on what type of code is running on the processor. Applications run in user mode, and core operating system components run in kernel mode. While many drivers run in kernel mode, some drivers may run in user mode.

When you start a user-mode application, Windows creates a *process* for the application. The process provides the application with a private *virtual address space* and a private *handle table*. Because an application's virtual address space is private, one application cannot alter data that belongs to another application. Each application runs in isolation, and if an application crashes, the crash is limited to that one application. Other applications and the operating system are not affected by the crash.

In addition to being private, the virtual address space of a user-mode application is limited. A processor running in user mode cannot access virtual addresses that are reserved for the operating system. Limiting the virtual address space of a user-mode application prevents the application from altering, and possibly damaging, critical operating system data.

All code that runs in kernel mode shares a single virtual address space. This means that a kernel-mode driver is not isolated from other drivers and the operating system itself. If a kernel-mode driver accidentally writes to the wrong virtual address, data that belongs to the operating system or another driver could be compromised. If a kernel-mode driver crashes, the entire operating system crashes.

This diagram illustrates communication between user-mode and kernel-mode components.

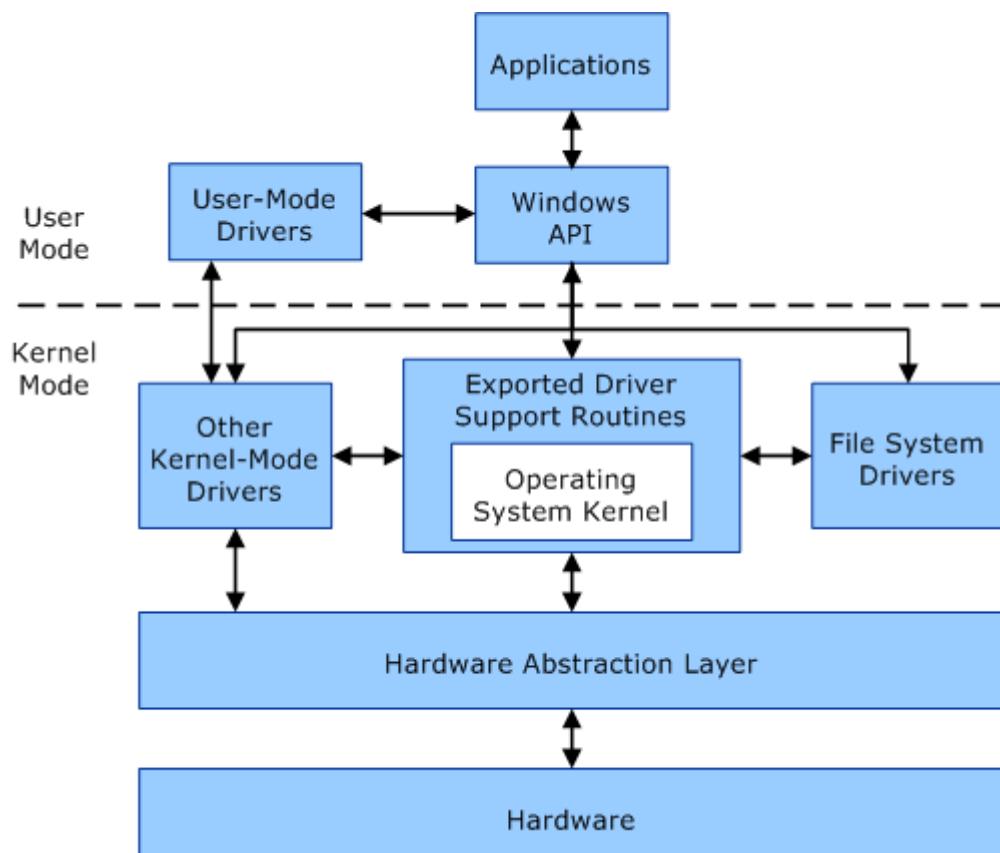


Figure 6: user-mode and kernel-mode components

Windows Services

Microsoft Windows services, formerly known as NT services, enable you to create long-running executable applications that run in their own Windows sessions. These services can be automatically started when the computer boots, can be paused and restarted, and do not show any user interface. These features make services ideal for use on a server or whenever you need long-running functionality that does not interfere with other users who are working on the same computer. You can also run services in the security context of a specific user account that is different from the logged-on user or the default computer account. For more information about services and Windows sessions, see the Windows SDK documentation.

You can easily create services by creating an application that is installed as a service. For example, suppose you want to monitor performance counter data and react to threshold values. You could write a Windows Service application that listens to the performance counter data, deploy the application, and begin collecting and analysing data.

You create your service as a Microsoft Visual Studio project, defining code within it that controls what commands can be sent to the service and what actions should be taken when those commands are received. Commands that can be sent to a service include starting, pausing, resuming, and stopping the service; you can also execute custom commands.

After you create and build the application, you can install it by running the command-line utility InstallUtil.exe and passing the path to the service's executable file. You can then use the **Services Control Manager** to start, stop, pause, resume, and configure your service. You can also accomplish many of these same tasks in the **Services** node in **Server Explorer** or by using the ServiceController class.

For more info

<https://docs.microsoft.com/en-us/dotnet/framework/windows-services/introduction-to-windows-service-applications>

Windows API

The Windows API is Microsoft's core set of application programming interfaces (APIs) available in the Microsoft Windows operating systems. It was formerly called WinAPI or Win32 API. Ex.

CreateProcess (&ProcHandle, Access, SectionHandle, DebugPort, ExceptionPort, ...)

CreateThread (&ThreadHandle, ProcHandle, Access, ThreadContext, bCreateSuspended, ...)

AllocateVirtualMemory (ProcHandle, Addr, Size, Type, Protection, ...)

MapViewOfSection (SectionHandle, ProcHandle, Addr, Size, Protection, ...)

ReadVirtualMemory (ProcHandle, Addr, Size, ...)

DuplicateObject (srcProcHandle, srcObjHandle, dstProcHandle, dstHandle, Access, Attributes, Options)

For more details

<https://docs.microsoft.com/en-us/windows/desktop/apiindex/windows-api-list>

Basics of System Calls

In computing, a **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. A system call is a way for programs to **interact with the operating system**. A computer program makes a system call when it makes a request to the operating system's kernel. System call **provides** the services of the operating system to the user programs via Application Program Interface(API). It provides an interface between a process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.

Services Provided by System Calls:

1. Process creation and management
2. Main memory management
3. File Access, Directory and File system management
4. Device handling(I/O)
5. Protection
6. Networking, etc.

Types of System Calls: There are 5 different categories of system calls –

1. **Process control:** end, abort, create, terminate, allocate and free memory.
2. **File management:** create, open, close, delete, read file etc.
3. Device management
4. Information maintenance
5. Communication
- 6.

Examples of Windows and Unix System Calls –

	WINDOWS	UNIX
Process Control	CreateProcess() ExitProcess()	fork() exit()
	WaitForSingleObject()	wait()

	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
File Manipulation	CloseHandle()	close()
	SetConsoleMode()	ioctl()
	ReadConsole()	read()
Device Manipulation	WriteConsole()	write()
	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
Information Maintenance	Sleep()	sleep()
	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
Communication	MapViewOfFile()	mmap()
	SetFileSecurity()	chmod()
	InitializeSecurityDescriptor()	umask()
Protection	SetSecurityDescriptorGroup()	chown()

MinWin

- **MinWin is first step at creating architectural partitions**
 - Can be built, booted and tested separately from the rest of the system
 - Higher layers can evolve independently
 - *An engineering process improvement, not a microkernel NT!*

- **MinWin was defined as set of components required to boot and access network**
 - Kernel, file system driver, TCP/IP stack, device drivers, services
 - No servicing, WMI, graphics, audio or shell, etc.
- **MinWin footprint:**
 - 150 binaries, 25MB on disk, 40MB in-memory

Virtual Memory

Virtual Memory is a storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program generated addresses are translated automatically to the corresponding machine addresses.

The size of virtual storage is limited by the addressing scheme of the computer system and amount of secondary memory is available not by the actual number of the main storage locations.

It is a technique that is implemented using both hardware and software. It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory.

1. All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process can be swapped in and out of main memory such that it occupies different places in main memory at different times during the course of execution.
2. A process may be broken into number of pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time address translation and use of page or segment table permits this.

If these characteristics are present then, it is not necessary that all the pages or segments are present in the main memory during execution. This means that the required pages need to be loaded into memory whenever required. Virtual memory is implemented using Demand Paging or Demand Segmentation.

Demand Paging:

The process of loading the page into memory on demand (whenever page fault occurs) is known as demand paging.

The process includes the following steps:

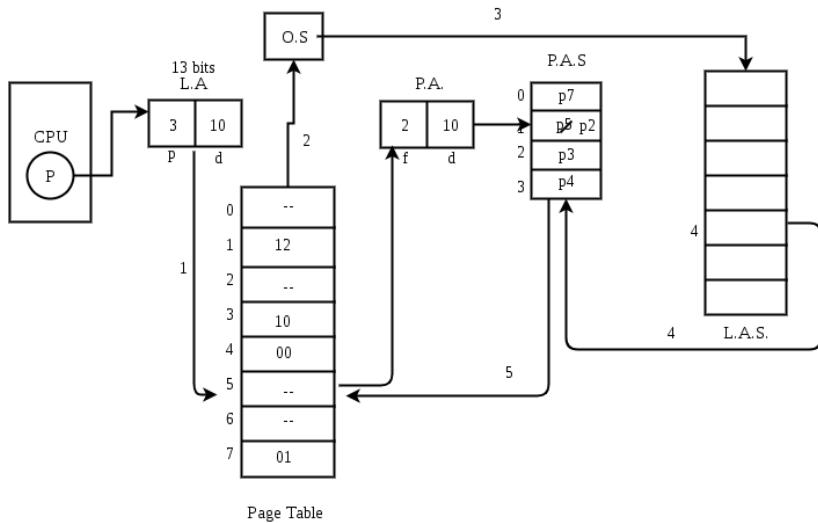


Figure 7: address Translation

1. If CPU try to refer a page that is currently not available in the main memory, it generates an interrupt indicating memory access fault.
2. The OS puts the interrupted process in a blocking state. For the execution to proceed the OS must bring the required page into the memory.
3. The OS will search for the required page in the logical address space.
4. The required page will be brought from logical address space to physical address space. The page replacement algorithms are used for the decision making of replacing the page in physical address space.
5. The page table will have updated accordingly.
6. The signal will be sent to the CPU to continue the program execution and it will place the process back into ready state.

Hence whenever a page fault occurs these steps are followed by the operating system and the required page is brought into memory.

Advantages:

- More processes may be maintained in the main memory: Because we are going to load only some of the pages of any particular process, there is room for more processes. This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in the ready state at any particular time.
- A process may be larger than all of main memory: One of the most fundamental restrictions in programming is lifted. A process larger than the main memory can be executed because of demand paging. The OS itself loads pages of a process in main memory as required.
- It allows greater multiprogramming levels by using less of the available (primary) memory for each process.

Page Fault Service Time:

The time taken to service the page fault is called as page fault service time. The page fault service time includes the time taken to perform all the above six steps.

Let Main memory access time is: m

Page fault service time is: s

Page fault rate is : p

Then, Effective memory access time = $(p*s) + (1-p)*m$

Swapping:

Swapping a process out means removing all of its pages from memory, or marking them so that they will be removed by the normal page replacement process. Suspending a process ensures that it is not runnable while it is swapped out. At some later time, the system swaps back the process from the secondary storage to main memory. When a process is busy swapping pages in and out then this situation is called thrashing.

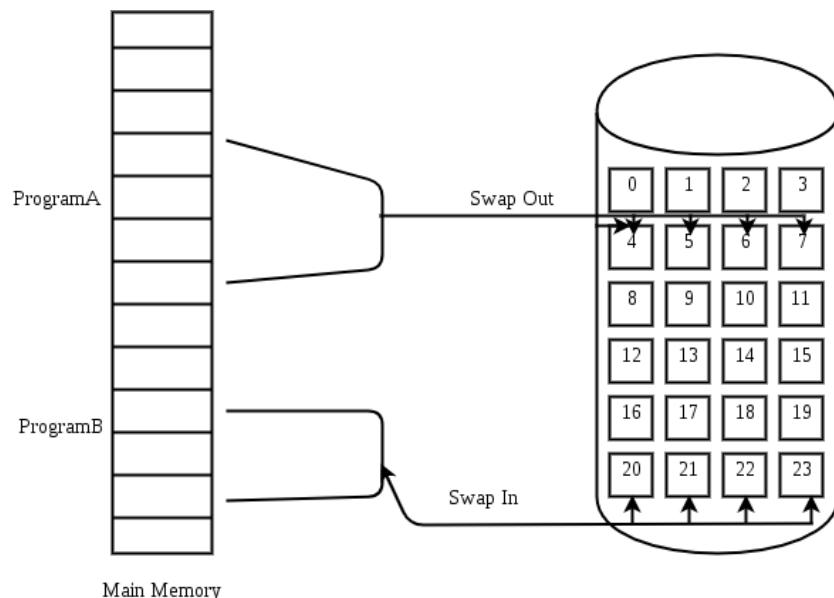


Figure 8: Page swapping

Thrashing:

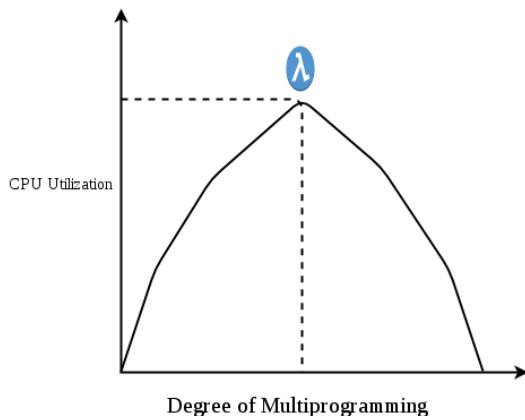


Figure 9: Thrashing

At any given time, only few pages of any process are in main memory and therefore more processes can be maintained in memory. Furthermore, time is saved because unused pages are not swapped in and out of memory. However, the OS must be clever about how it manages this scheme. In the steady state practically, all of main memory will be occupied with process's pages, so that the processor and OS has direct access to as many processes as possible. Thus when the OS brings one page in, it must throw another out. If it throws out a page just before it is used, then it will just have to get that page again almost immediately. Too much of this leads to a condition called Thrashing. The system spends most of its time swapping pages rather than executing instructions. So a good page replacement algorithm is required.

In the given diagram, initial degree of multi programming upto some extent of point(lamda), the CPU utilization is very high and the system resources are utilized 100%. But if we further increase the degree of multi programming the CPU utilization will drastically fall down and the system will spend more time only in the page replacement and the time taken to complete the execution of the process will increase. This situation in the system is called as thrashing.

Causes of Thrashing :

1. High degree of multiprogramming:

If the number of processes keeps on increasing in the memory than number of frames allocated to each process will be decreased. So, less number of frames will be available to each process. Due to this, page fault will occur more frequently and more CPU time will be wasted in just swapping in and out of pages and the utilization will keep on decreasing.

For example:

Let free frames = 400

Case 1: Number of process = 100

Then, each process will get 4 frames.

Case 2: Number of process = 400

Each process will get 1 frame.

Case 2 is a condition of thrashing, as the number of processes are increased, frames per process are decreased. Hence CPU time will be consumed in just swapping pages.

2. **Lacks of Frames:** If a process has less number of frames then less pages of that process will be able to reside in memory and hence more frequent swapping in and out will be required. This may lead to thrashing. Hence sufficient amount of frames must be allocated to each process in order to prevent thrashing.

Recovery of Thrashing:

- Do not allow the system to go into thrashing by instructing the long term scheduler not to bring the processes into memory after the threshold.
- If the system is already in thrashing, then instruct the midterm scheduler to suspend some of the processes so that we can recover the system from thrashing.

Segmentation:

A process is divided into Segments. The chunks that a program is divided into which are not necessarily all of the same sizes are called segments. Segmentation gives user's view of the process which paging does not give. Here the user's view is mapped to physical memory.

There are types of segmentation:

Virtual memory segmentation – Each process is divided into a number of segments, not all of which are resident at any one point in time.

Simple segmentation – Each process is divided into a number of segments, all of which are loaded into memory at run time, though not necessarily contiguously.

There is no simple relationship between logical addresses and physical addresses in segmentation. A table stores the information about all such segments and is called Segment Table.

Segment Table – It maps two-dimensional Logical address into one-dimensional Physical address. It's each table entry has:

Base Address: It contains the starting physical address where the segments reside in memory.

Limit: It specifies the length of the segment.

Address generated by the CPU is divided into:

- **Segment number (s):** Number of bits required to represent the segment.
- **Segment offset (d):** Number of bits required to represent the size of the segment.

Advantages of Segmentation –

- No Internal fragmentation.
- Segment Table consumes less space in comparison to Page table in paging.

Disadvantage of Segmentation –

- As processes are loaded and removed from the memory, the free memory space is broken into little pieces, causing External fragmentation.

Working Set

The working set of a process is the set of pages in the virtual address space of the process that are currently resident in physical memory. The working set contains only page-able memory allocations; non-page-able memory allocations such as Address Windowing Extensions (AWE) or large page allocations are not included in the working set.

When a process references page-able memory that is not currently in its working set, a page fault occurs. The system page fault handler attempts to resolve the page fault and, if it succeeds, the page is added to the working set. (Accessing AWE or large page allocations never causes a page fault, because these allocations are not page-able.)

A *hard page fault* must be resolved by reading page contents from the page's *backing store*, which is either the system paging file or a memory-mapped file created by the process. A *soft page fault* can be resolved without accessing the backing store. A soft page fault occurs when:

- The page is in the working set of some other process, so it is already resident in memory.
- The page is in transition, because it either has been removed from the working sets of all processes that were using the page and has not yet been repurposed, or it is already resident as a result of a memory manager pre-fetch operation.
- A process references an allocated virtual page for the first time (sometimes called a *demand-zero fault*).

Pages can be removed from a process working set as a result of the following actions:

- The process reduces or empties the working set by calling the **SetProcessWorkingSetSize**, **SetProcessWorkingSetSizeEx** or **EmptyWorkingSet** function.
- The process calls the **VirtualUnlock** function on a memory range that is not locked.
- The process unmaps a mapped view of a file using the **UnmapViewOfFile** function.
- The memory manager trims pages from the working set to create more available memory.
- The memory manager must remove a page from the working set to make room for a new page (for example, because the working set is at its maximum size).

If several processes share a page, removing the page from the working set of one process does not affect other processes. After a page is removed from the working sets of all processes that were using it, the page becomes a *transition page*. Transition pages remain cached in RAM until the page is either referenced again by some process or repurposed (for example, filled with zeros and given to another process). If a transition page has been modified since it was last written to disk (that is, if the page is "dirty"), then the page must be written to its backing store before it can be repurposed. The system may start writing dirty transition pages to their backing store as soon as such pages become available.

Each process has a minimum and maximum working set size that affect the virtual memory paging behavior of the process. To obtain the current size of the working set of a specified

process, use the `GetProcessMemoryInfo` function. To obtain or change the minimum and maximum working set sizes, use the `GetProcessWorkingSetSizeEx` and `SetProcessWorkingSetSizeEx` functions. The process status application programming interface (PSAPI) provides a number of functions that return detailed information about the working set of a process. For details, see Working Set Information.

Page fault

A page fault occurs when a program attempts to access data or code that is in its address space, but is not currently located in the system RAM. So when page fault occurs then following sequence of events happens:

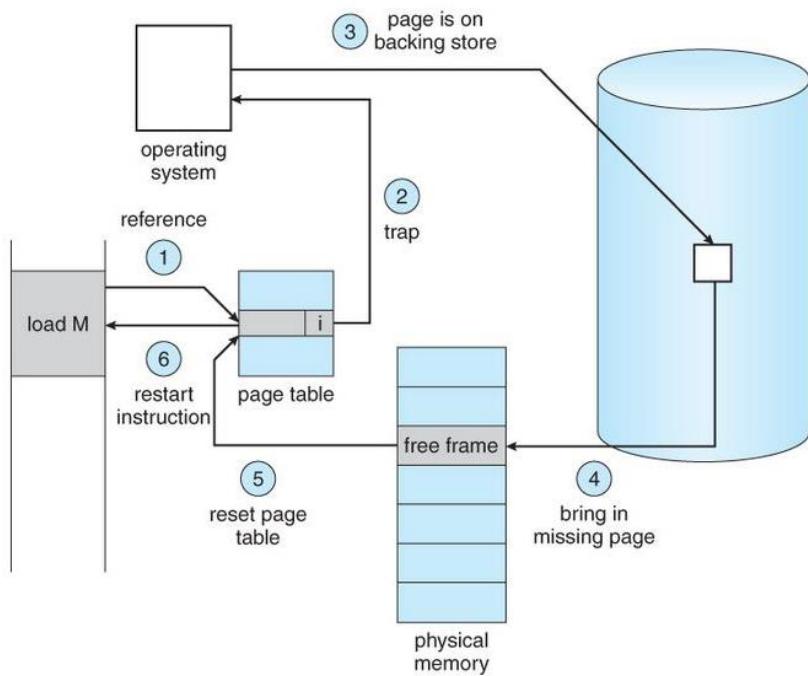


Figure 10: Page fault

- The computer hardware traps to the kernel and program counter (PC) is saved on the stack. Current instruction state information is saved in CPU registers.
- An assembly program is started to save the general registers and other volatile information to keep the OS from destroying it.
- Operating system finds that a page fault has occurred and tries to find out which virtual page is needed. Sometimes hardware register contains this required information. If not, the operating system must retrieve PC, fetch instruction and find out what it was doing when the fault occurred.
- Once virtual address caused page fault is known, system checks to see if address is valid and checks if there is no protection access problem.
- If the virtual address is valid, the system checks to see if a page frame is free. If no frames are free, the page replacement algorithm is run to remove a page.

- If frame selected is dirty, page is scheduled for transfer to disk, context switch takes place, fault process is suspended and another process is made to run until disk transfer is completed.
- As soon as page frame is clean, operating system looks up disk address where needed page is, schedules disk operation to bring it in.
- When disk interrupt indicates page has arrived, page tables are updated to reflect its position, and frame marked as being in normal state.
- Faulting instruction is backed up to state it had when it began and PC is reset. Faulting is scheduled, operating system returns to routine that called it.
- Assembly Routine reloads register and other state information, returns to user space to continue execution.

Multithreading

A **thread** is a path which is followed during a program's execution. Majority of programs written now a days run as a single thread. Let's say, for example a program is not capable of reading keystrokes while making drawings. These tasks cannot be executed by the program at the same time. This problem can be solved through multitasking so that two or more tasks can be executed simultaneously.

Multitasking is of two types: Processor based and thread based. Processor based multitasking is totally managed by the OS, however multitasking through multithreading can be controlled by the programmer to some extent.

The concept of **multi-threading** needs proper understanding of these two terms – **a process and a thread**. A process is a program being executed. A process can be further divided into independent units known as threads.

A thread is like a small light-weight process within a process. Or we can say a collection of threads is what is known as a process.

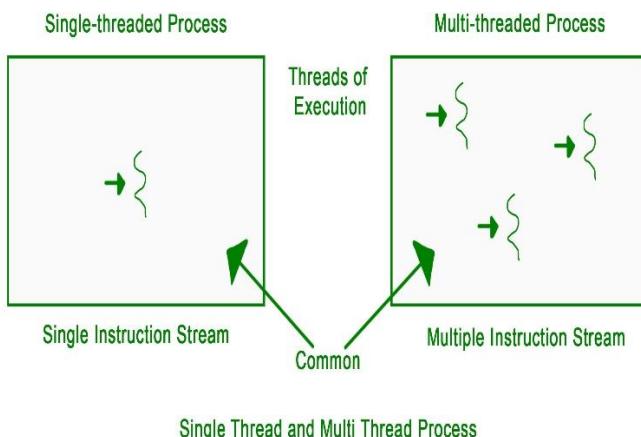


Figure 11: Single thread vs Multithread

Applications:

Threading is used widely in almost every field. Most widely it is seen over the internet now days where we are using transaction processing of every type like recharges, online transfer, banking etc. Threading is a segment which divide the code into small parts that are of very light weight and has less burden on CPU memory so that it can be easily worked out and can achieve goal in desired field. The concept of threading is designed due to the problem of fast and regular changes in technology and less the work in different areas due to less application. Then as says “need is the generation of creation or innovation” hence by following this approach human mind develop the concept of thread to enhance the capability of programming.

Benefits of Multithreading

Prerequisite – Operating-System-Thread The benefits of multi-threaded programming can be broken down into four major categories:

Responsiveness:

Multithreading in an interactive application may allow a program to continue running even if a part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. In a non-multi-threaded environment, a server listens to the port for some request and when the request comes, it processes the request and then resume listening to another request. The time taken while processing of request makes other users wait unnecessarily. Instead a better approach would be to pass the request to a worker thread and continue listening to port.

For example, a multi-threaded web browser allows user interaction in one thread while a video is being loaded in another thread. So instead of waiting for the whole web-page to load the user can continue viewing some portion of the web-page.

Resource Sharing:

Processes may share resources only through techniques such as-

- Message Passing
- Shared Memory

Such techniques must be explicitly organized by programmer. However, threads share the memory and the resources of the process to which they belong by default.

The benefit of sharing code and data is that it allows an application to have several threads of activity within same address space.

Economy:

Allocating memory and resources for process creation is a costly job in terms of time and space.

Since, threads share memory with the process it belongs, it is more economical to create and context switch threads. Generally, much more time is consumed in creating and managing processes than in threads.

In Solaris, for example, creating process is 30 times slower than creating threads and context switching is 5 times slower.

Scalability:

The benefits of multi-programming greatly increase in case of multiprocessor architecture, where threads may be running parallel on multiple processors. If there is only one thread, then it is not possible to divide the processes into smaller tasks that different processors can perform.

Single threaded process can run only on one processor regardless of how many processors are available.

Multi-threading on a multiple CPU machine increases parallelism.

Difference between multitasking, multithreading and multiprocessing

1. **Multiprogramming** – A computer running more than one program at a time (like running Excel and Firefox simultaneously).
2. **Multiprocessing** – A computer using more than one CPU at a time.
3. **Multitasking** – Tasks sharing a common resource (like 1 CPU).
4. **Multithreading** is an extension of multitasking.

1. Multi programming:

In a modern computing system, there are usually several concurrent application processes which want to execute. Now it is the responsibility of the Operating System to manage all the processes effectively and efficiently.

One of the most important aspects of an Operating System is to multi program. In a computer system, there are multiple processes waiting to be executed, i.e. they are waiting when the CPU will be allocated to them and they begin their execution. These processes are also known as jobs. Now the main memory is too small to accommodate all of these processes or jobs into it. Thus, these processes are initially kept in an area called job pool. This job pool consists of all those processes awaiting allocation of main memory and CPU. CPU selects one job out of all these waiting jobs, brings it from the job pool to main memory and starts executing it. The processor executes one job until it is interrupted by some external factor or it goes for an I/O task.

Non-multi programmed system's working:

In a non multi programmed system, as soon as one job leaves the CPU and goes for some other task (say I/O), the CPU becomes idle. The CPU keeps waiting and waiting until this job (which was executing earlier) comes back and resumes its execution with the CPU. So CPU remains free for all this while.

Now it has a drawback that the CPU remains idle for a very long period of time. Also, other jobs which are waiting to be executed might not get a chance to execute because the CPU is still allocated to the earlier job.

This poses a very serious problem that even though other jobs are ready to execute, CPU is not allocated to them as the CPU is allocated to a job which is not even utilizing it (as it is busy in I/O tasks).

It cannot happen that one job is using the CPU for say 1 hour while the others have been waiting in the queue for 5 hours. To avoid situations like this and come up with efficient utilization of CPU, the concept of multi programming came up.

The main idea of multi programming is to maximize the CPU time.

Multi programmed system's working:

In a multi-programmed system, as soon as one job goes for an I/O task, the Operating System interrupts that job, chooses another job from the job pool (waiting queue), gives CPU to this new job and starts its execution. The previous job keeps doing its I/O operation while this new job does CPU bound tasks. Now say the second job also goes for an I/O task, the CPU chooses a third job and starts executing it. As soon as a job completes its I/O operation and comes back for CPU tasks, the CPU is allocated to it.

In this way, no CPU time is wasted by the system waiting for the I/O task to be completed.

Therefore, the ultimate goal of multi programming is to keep the CPU busy as long as there are processes ready to execute. This way, multiple programs can be executed on a single processor by executing a part of a program at one time, a part of another program after this, then a part of another program and so on, hence executing multiple programs. Hence, the CPU never remains idle.

In the image below, program A runs for some time and then goes to waiting state. In the meantime, program B begins its execution. So the CPU does not waste its resources and gives program B an opportunity to run.

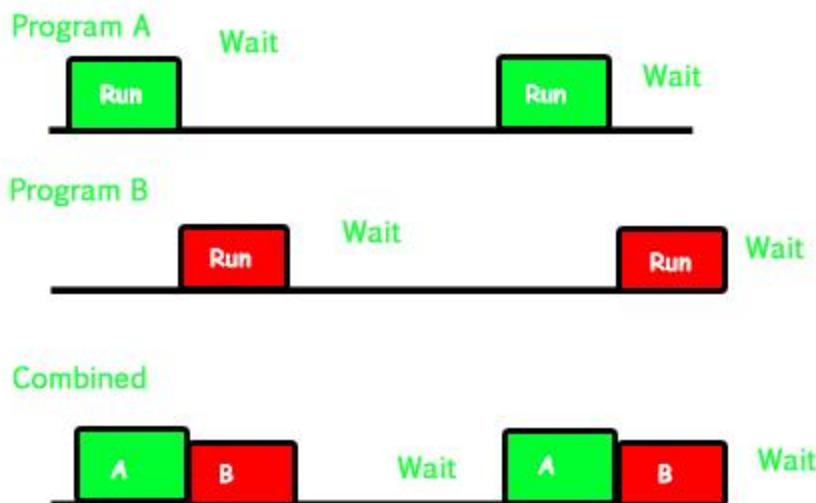


Figure 12: Multi programmed system's working

2. Multiprocessing:

In a uni-processor system, only one process executes at a time. Multiprocessing is the use of two or more CPUs (processors) within a single Computer system. The term also refers to the ability of a system to support more than one processor within a single computer system. Now since there are multiple processors available, multiple processes can be executed at a time. These multi processors share the computer bus, sometimes the clock, memory and peripheral devices also.

Multi-processing system's working:

With the help of multiprocessing, many processes can be executed simultaneously. Say processes P1, P2, P3 and P4 are waiting for execution. Now in a single processor system, firstly one process will execute, then the other, then the other and so on.

But with multiprocessing, each process can be assigned to a different processor for its execution. If it's a dual-core processor (2 processors), two processes can be executed simultaneously and thus will be two times faster, similarly a quad core processor will be four times as fast as a single processor.

Why use multi-processing:

The main advantage of multiprocessor system is to get more work done in a shorter period of time. These types of systems are used when very high speed is required to process a large volume of data. Multi-processing systems can save money in comparison to single processor systems because the processors can share peripherals and power supplies.

It also provides increased reliability in the sense that if one processor fails, the work does not halt, it only slows down. e.g. if we have 10 processors and 1 fails, then the work does not halt, rather the remaining 9 processors can share the work of the 10th processor. Thus the whole system runs only 10 percent slower, rather than failing altogether.

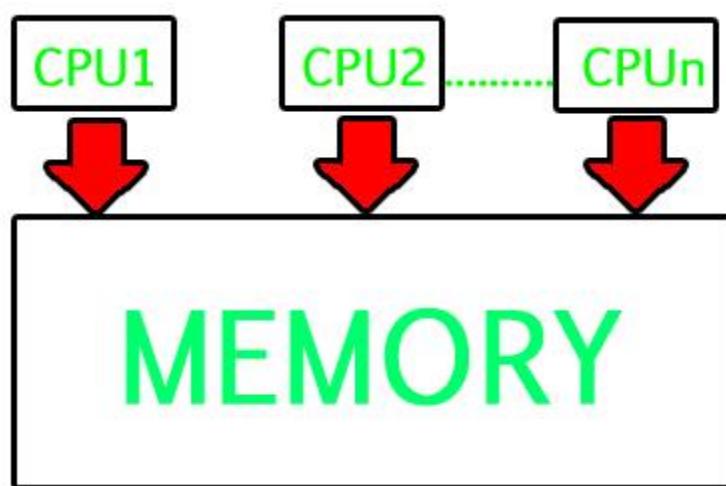


Figure 13: Multi-processing system

Multiprocessing refers to the hardware (i.e., the CPU units) rather than the software (i.e., running processes). If the underlying hardware provides more than one processor, then that is multiprocessing. It is the ability of the system to leverage multiple processors' computing power.

Difference between Multi programming and Multi processing:

- A System can be both multi programmed by having multiple programs running at the same time and multiprocessing by having more than one physical processor. The difference between multiprocessing and multi programming is that Multiprocessing is basically executing multiple processes at the same time on multiple processors, whereas multi programming is keeping several programs in main memory and executing them concurrently using a single CPU only.
- Multiprocessing occurs by means of parallel processing whereas Multi programming occurs by switching from one process to other (phenomenon called as context switching).

3. Multitasking:

As the name itself suggests, multi-tasking refers to execution of multiple tasks (say processes, programs, threads etc.) at a time. In the modern operating systems, we are able to play MP3 music, edit documents in Microsoft Word, surf the Google Chrome all simultaneously, this is accomplished by means of multi-tasking.

Multitasking is a logical extension of multi programming. The major way in which multitasking differs from multi programming is that multi programming works solely on the concept of context switching whereas multitasking is based on time sharing alongside the concept of context switching.

Multi-tasking system's working:

- In a time sharing system, each process is assigned some specific quantum of time for which a process is meant to execute. Say there are 4 processes P1, P2, P3, P4 ready to execute. So each of them are assigned some time quantum for which they will execute e.g time quantum of 5 nanoseconds (5 ns). As one process begins execution (say P2), it executes for that quantum of time (5 ns). After 5 ns the CPU starts the execution of the other process (say P3) for the specified quantum of time.
- Thus the CPU makes the processes to share time slices between them and execute accordingly. As soon as time quantum of one process expires, another process begins its execution.
- Here also basically a context switch is occurring but it is occurring so fast that the user is able to interact with each program separately while it is running. This way, the user is given the illusion that multiple processes/ tasks are executing simultaneously. But actually only one process/ task is executing at a particular instant of time. In multitasking, time sharing is best manifested because each running process takes only a fair quantum of the CPU time.

In a more general sense, multitasking refers to having multiple programs, processes, tasks, threads running at the same time. This term is used in modern operating systems when multiple tasks share a common processing resource (e.g., CPU and Memory).

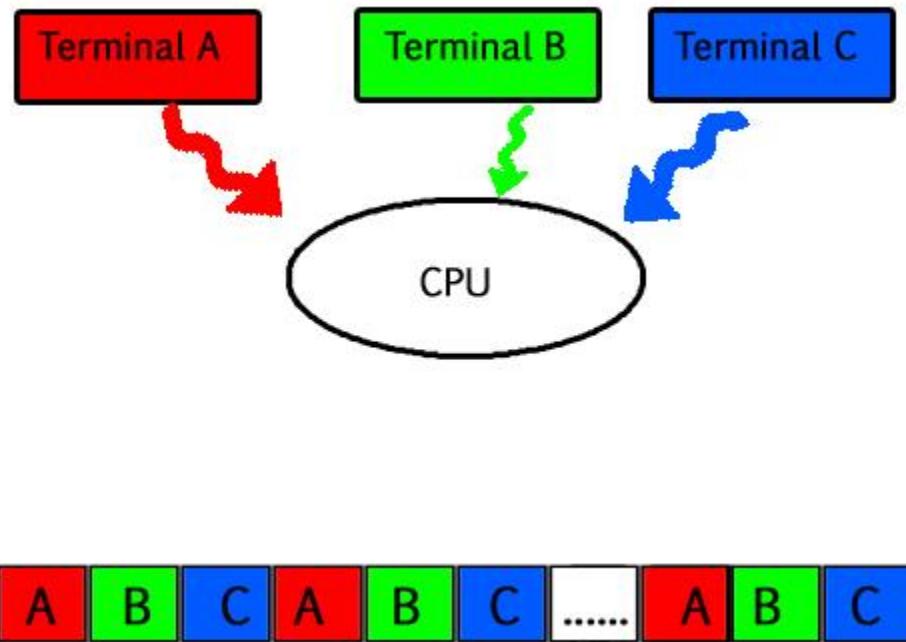


Figure 14: Multi-tasking system

- As depicted in the above image, at any time the CPU is executing only one task while other tasks are waiting for their turn. The illusion of parallelism is achieved when the CPU is reassigned to another task. i.e. all the three tasks A, B and C are appearing to occur simultaneously because of time sharing.
- So for multitasking to take place, firstly there should be multiprogramming i.e. presence of multiple programs ready for execution. And secondly the concept of time sharing.

4. Multi-threading:

A thread is a basic unit of CPU utilization. Multi-threading is an execution model that allows a single process to have multiple code segments (i.e., threads) running concurrently within the “context” of that process.

e.g. VLC media player, where one thread is used for opening the VLC media player, one thread for playing a particular song and another thread for adding new songs to the playlist.

Multi-threading is the ability of a process to manage its use by more than one user at a time and to manage multiple requests by the same user without having to have multiple copies of the program.

Multi-threading system's working –

Example 1 –

Say there is a web server which processes client requests. Now if it executes as a single threaded process, then it will not be able to process multiple requests at a time. Firstly one client will make its request and finish its execution and only then, the server will be able to process another client request. This is really costly, time consuming and tiring task. To avoid this, multi-threading can be made use of.

Now, whenever a new client request comes in, the web server simply creates a new thread for processing this request and resumes its execution to hear more client requests. So the web server has the task of listening to new client requests and creating threads for each individual request. Each newly created thread processes one client request, thus reducing the burden on web server.

Example 2 –

We can think of threads as child processes that share the parent process resources but execute independently. Now take the case of a GUI. Say we are performing a calculation on the GUI (which is taking very long time to finish). Now we cannot interact with the rest of the GUI until this command finishes its execution. To be able to interact with the rest of the GUI, this command of calculation should be assigned to a separate thread. So at this point of time, 2 threads will be executing i.e. one for calculation, and one for the rest of the GUI. Hence here in a single process, we used multiple threads for multiple functionality.

The image below completely describes the VLC player example:

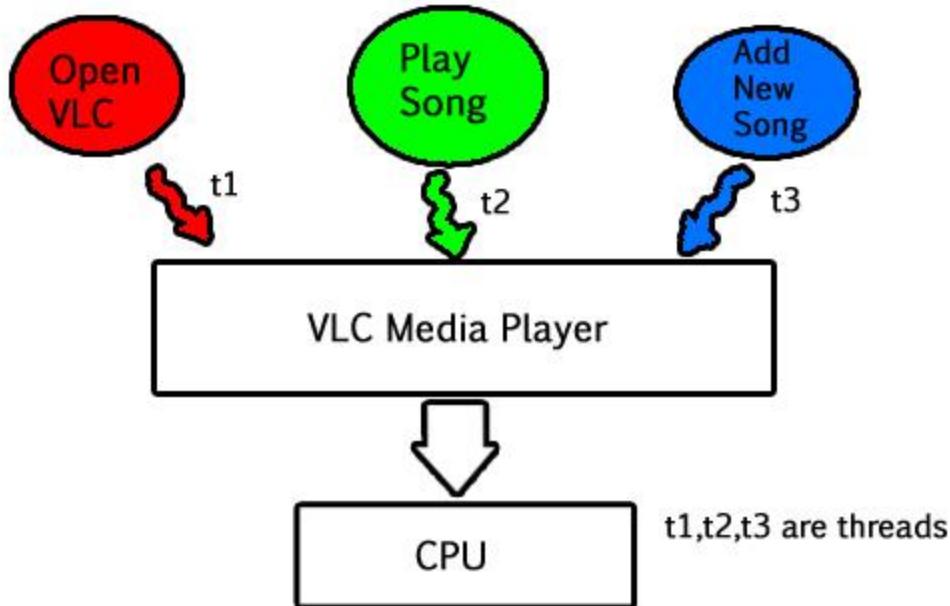


Figure 15: Multi-threading system

Advantages of Multi-threading:

Benefits of Multi-threading include increased responsiveness. Since there are multiple threads in a program, so if one thread is taking too long to execute or if it gets blocked, the rest of the

threads keep executing without any problem. Thus the whole program remains responsive to the user by means of remaining threads.

Another advantage of multi-threading is that it is less costly. Creating brand new processes and allocating resources is a time consuming task, but since threads share resources of the parent process, creating threads and switching between them is comparatively easy. Hence multi-threading is the need of modern Operating Systems.

Process Synchronization

On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process:** Execution of one process does not affects the execution of other processes.
- **Cooperative Process:** Execution of one process affects the execution of other processes.

Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

Race Condition

Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place.

Critical Section Problem

Critical section is a code segment that can be accessed by only one process at a time. Critical section contains shared variables which need to be synchronized to maintain consistency of data variables.

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (TRUE);
```

In the entry section, the process requests for entry in the **Critical Section**.

Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion:** If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress:** If no process is executing in the critical section and other processes are waiting outside the critical sections, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection cannot be postponed indefinitely.

- **Bounded Waiting:** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Peterson's Solution

Peterson's Solution is a classical software based solution to the critical section problem.

In Peterson's solution, we have two shared variables:

- Boolean flag[i]: Initialized to FALSE, initially no one is interested in entering the critical section
- int turn: The process whose turn is to enter the critical section.

```

do {

    flag[i] = TRUE ;
    turn = j ;
    while (flag[j] && turn == j) ;

        critical section

        flag[i] = FALSE ;

        remainder section

    } while (TRUE) ;

```

Peterson's Solution preserves all three conditions:

- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not blocks other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

Disadvantages of Peterson's Solution

- It involves Busy waiting
- It is limited to 2 processes.

Deadlock Introduction

A process in operating systems uses different resources and uses resources in following way.

- 1) Requests a resource
- 2) Use the resource
- 2) Releases the resource

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait

for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.

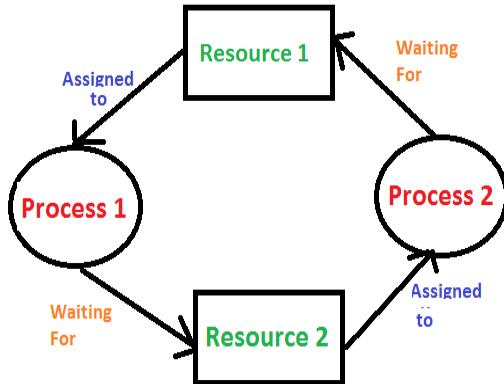


Figure 16: Deadlock

Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)

Mutual Exclusion: One or more than one resource are non-sharable (Only one process can use at a time)

Hold and Wait: A process is holding at least one resource and waiting for resources.

No Preemption: A resource cannot be taken from a process unless the process releases the resource.

Circular Wait: A set of processes are waiting for each other in circular form.

Methods for handling deadlock

There are three ways to handle deadlock

1) Deadlock prevention or avoidance: The idea is to not let the system into deadlock state.

One can zoom into each category individually, Prevention is done by negating one of above mentioned necessary conditions for deadlock.

Avoidance is kind of futuristic in nature. By using strategy of "Avoidance", we have to make an assumption. We need to ensure that all information about resources which process WILL need are known to us prior to execution of the process. We use Banker's algorithm (Which is in-turn a gift from Dijkstra) in order to avoid deadlock.

2) Deadlock detection and recovery: Let deadlock occur, then do preemption to handle it once occurred.

3) Ignore the problem all together: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

Process Scheduler

There are three types of process scheduler.

1. Long Term or job scheduler It brings the new process to the ‘Ready State’. It controls Degree of Multi-programming, i.e., number of process present in ready state at any point of time. It is important that the long-term scheduler make a careful selection of both IO and CPU bound process.

2. Short term or CPU scheduler: It is responsible for selecting one process from ready state for scheduling it on the running state. Note: Short-term scheduler only selects the process to schedule it doesn’t load the process on running. Dispatcher is responsible for loading the process selected by Short-term scheduler on the CPU (Ready to Running State) Context switching is done by dispatcher only. A dispatcher does the following:

- Switching context.
- Switching to user mode.
- Jumping to the proper location in the newly loaded program.

3. Medium-term scheduler It is responsible for suspending and resuming the process. It mainly does swapping (moving processes from main memory to disk and vice versa). Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

Chapter-12

ACPI and SMBIOS

SMM

- Most privileged x86 processor operating mode
- intended for use only by system firmware
- Runs transparent to the operating system
- When the processor enters SMM, all other running tasks are suspended
- SMM can be invoked *only* by a System Management Interrupt (SMI) and exited *only* by the RSM (resume) instruction
- Intended use is to provide an isolated operating environment for
- Power/Battery management
- Controlling system hardware
- Running proprietary OEM code
- etc. (anything that should run privileged and uninterrupted)

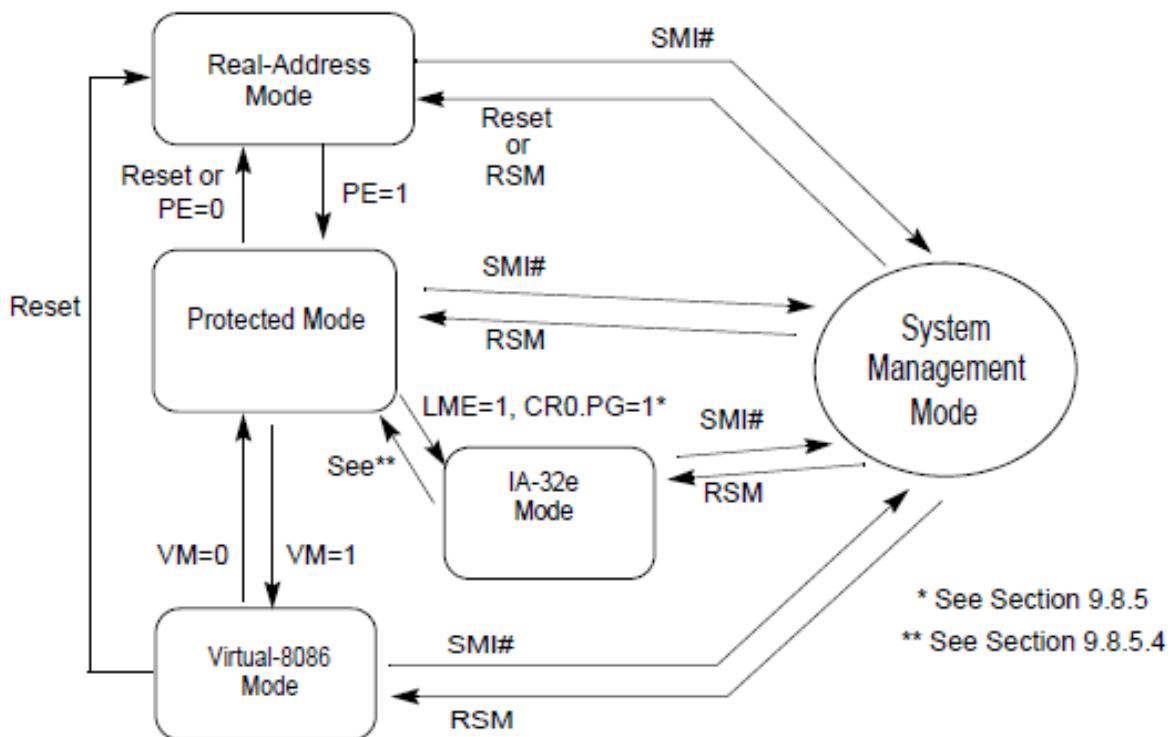


Fig-1 Mode transition diagram

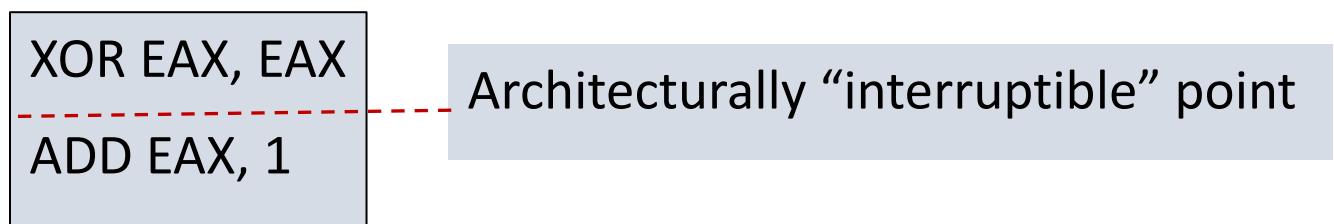
System Management Interrupt (SMI)

- SMM can only be invoked by signaling a System Management Interrupt (SMI)
- SMI's can be received via the SMI# pin on the processor or through the APIC bus

- SMI's cannot be masked like normal interrupts (e.g. with the “cli” instruction, or clearing the IF bit in EFLAGS)
- SMI's are independent from the normal processor interrupt and exception-handling mechanisms
- SMI's take precedence over **non-maskable** and **maskable interrupts**
 - Including debug exceptions and external interrupts
- If a SMI and NMI occur at the same time, only the SMI will be handled
 - On a system like a laptop, SMM will likely be called frequently to check the battery/power status
 - Should be an SMI# in that case
 - But on a desktop, it may be called much less frequently

Entering SMM: SMI

- When receiving an SMI, the processor waits for all instructions to complete and stores to complete
- SMI interrupts are handled on an architecturally defined “interruptible” point in program execution
 - Like an instruction boundary:
 - Processor saves the context in SMRAM and begins executing the SMI Handler
- In a multi-core processor, no SMI handler code is executed until all cores have performed the above and entered SMM



Exiting SMM: RSM

- The only way to exit SMM is through the RSM instruction
 - Or system reset/shut down
- Returns control to the application program or operating-system procedure that was interrupted by the SMI
- The processor's state is restored from the save state area within SMRAM. If the processor detects invalid state information during state restoration, it enters the shutdown state.
- The operating mode the processor was in at the time of the SMI is restored.

- RSM can only be executed from within SMM
- Executing RSM while not in SMRAM generates an invalid opcode exception

SMRAM

- SMRAM is the address space where the processor switches to upon entering SMM
- This address space contains the SMI handler code and data
- The processor's pre-SMI register context is saved at a pre-defined location in SMRAM (fixed offset from SMBASE)
- SMBASE is the base address of SMRAM and is located in a reserved portion of main RAM
 - Thus access control mechanisms must be based in the memory controller (MCH or CPU)

Address Space Layout

- **SMRAM** refers to the entire range (or ranges) where the SMI handler code and data is located
- **SMBASE** is a private CPU-internal register that holds the address denoting the base address of SMRAM for a processor (or core)
 - Each core will have its own SMBASE
- The state save area(s) and entry point(s) are fixed offsets from SMBASE
- SMBASE is also found as a field stored in the state save area within SMRAM
 - The stored value is always at the same offset from SMBASE (FFE8h)
 - A 32-bit value containing the physical address of SMRAM (SMBASE)
 - Even in x64 architecture
- Therefore SMRAM is relocatable by changing the saved value of SMBASE, stored in the SMRAM save state area upon SMI

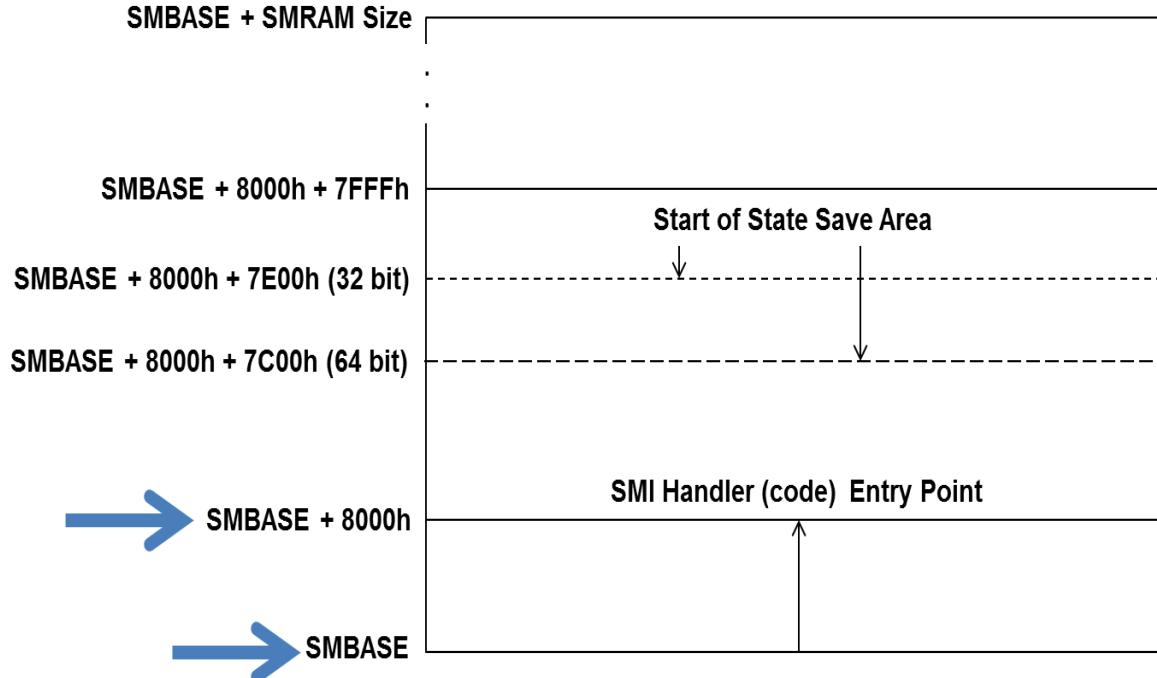
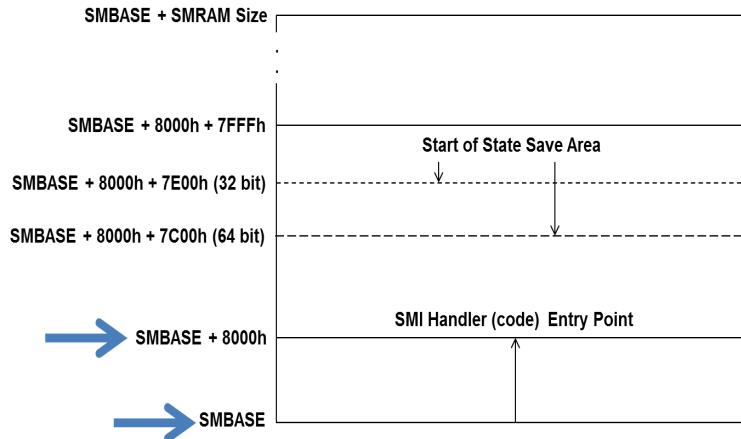


Fig-2 SMRAM BASE

- Default SMBASE on startup is 30000h, but can be relocated
- SMI Handler Executable code entry point is always at SMBASE + 8000h
 - CPU always begins executing at SMBASE + 8000h
- Multi-core systems will typically have their SMBASE offset by N bytes from each other. For example:
 - Core 0 defines SMBASE as A_0000h, will enter SMI handler at A_8000h
 - Core 1 defines SMBASE as A_1000h, will enter SMI handler at A_9000h
- State save address starts at SMBASE + 8000h + 7FFFh
- For 32-bit CPU's, the state-save area is 200h bytes
- State save area extends down to SMBASE + 8000h + 7E00h



- State save address starts at SMBASE + 8000h + 7FFFh
- For 64-bit CPU's, the state-save area is 400h bytes
- The state-save extends down to SMBASE + 8000h + 7C00h
- The remaining area is free for use as SMI handler code and data
- Total size of SMRAM region is defined by the BIOS when it configures SMM

SMRAM Location

- SMRAM can be located anywhere in the 4GB memory address space
- SMBASE can be overwritten by the SMI handler
- Typically SMRAM is relocated at least once:
 - On system startup, the first time the system enters SMM, SMBASE is at 0x30000h
 - SMI handler starts executing at 0x38000h
 - There is no reason it needs to stay at that address
- Intel defines a few locations for SMRAM
 - But it really is a flexible system and can be put anywhere
 - guidelines are provided to make configuration easier for the BIOS developers and to avoid areas where SMRAM may overlap with other regions
 - This is all part of building that memory map

SMBIOS

- It is the premier standard for delivering management information via system firmware
- addresses how motherboard and system vendors present management information about their products in a standard format by extending the BIOS interface on processor architecture systems.

- In OS-present, OS-absent, and pre-OS environments, SMBIOS offers motherboard and system vendors a standard format to present management information about their products.
- Information include *system manufacturer, model name, serial number, BIOS version, asset tag, processors, ports and device memory installed*

SMBIOS Table convention

- The only access method defined for the SMBIOS structures is a table-based method
- It provides the SMBIOS structures as a packed list of data referenced by a table entry point.
- The table convention allows the SMBIOS structures to be accessed under 32-bit and 64-bit protected mode operating systems such as Microsoft® Windows XP®, Microsoft® Windows Server®, or Linux®.
- This convention provides a searchable entry-point structure that contains a pointer to the packed SMBIOS structures.
- The original SMBIOS 2.1 (32-bit) entry point allows the SMBIOS structure table to reside anywhere in 32-bit physical address space
- The SMBIOS 3.0 (64-bit) entry point allows the SMBIOS structure table to reside anywhere in 64-bit memory
- If an implementation provides both a 32-bit and a 64-bit entry point and they point to distinct SMBIOS structure tables, the 32-bit table must be a consistent subset of the 64-bit table
- The 64-bit table may contain structure types not found in the 32-bit table

SMBIOS Entry Point

	UEFI	Non UEFI
32 bit	<p>can be located by looking in the EFI Configuration Table for the SMBIOS GUID (SMBIOS_TABLE_GUID) and using the associated pointer</p>	<p>can be located by application software by searching for the anchor-string on paragraph (16-byte) boundaries within the physical memory address range 000F0000h to 000FFFFh.</p> <p>This entry point encapsulates an intermediate anchor string that is used some existing DMI browsers.</p>

64 bit	can be located by looking in the EFI Configuration Table for the SMBIOS 3.x GUID (SMBIOS3_TABLE_GUID) and using the associated pointer	can be located by application software by searching for the anchor-string on paragraph (16-byte) boundaries within the physical memory address range 000F0000h to 000FFFFh.
---------------	--	---

Name and types

Structure Name and type	Data Requirements
BIOS Information (Type 0)	One and only one structure is present in the structure-table. <i>BIOS Version</i> and <i>BIOS Release Date</i> strings are non-null; the date field uses a 4-digit year (for example, 1999). All other fields reflect full BIOS support information..
System Information (Type 1)	<i>Manufacturer</i> and <i>Product Name</i> strings are non-null. <i>UUID</i> field identifies the system's non-zero UUID value. <i>Wake-up Type</i> field identifies the wakeup source and cannot be Unknown. One and only one structure is present in the structure-table
System Enclosure (Type 3)	<i>Manufacturer</i> string is non-null; the <i>Type</i> field identifies the type of enclosure. (Unknown is disallowed.)
Processor Information (Type 4)	One structure is required for each system processor. The presence of two structures with the <i>Processor Type</i> field set to <i>Central Processor</i> , for instance, identifies that the system is capable of dual-processor operations. <i>Socket Designation</i> string is non-null. <i>Processor Type</i> , <i>Max Speed</i> , and <i>Processor Upgrade</i> fields are all set to "known" values



Advanced Configurations and Power Interface

- Open industry specification
 - Co-developed by Intel, HP, MS, Phoenix, Toshiba
- Establishes industry-standard interfaces
 - OS-directed configuration Power management
 - Thermal management
 - Across devices: mobile, desktop, and server platforms
- When first published in 1996, ACPI evolved an existing collection of
 - Power management BIOS code,
 - Advanced Power Management (APM)
 - Application Programming Interfaces (APIs),
 - PNPBIOS APIs, and
 - Multi-Processor Specification (MPS)

Power Management History

- BIOS based system power management
- OS used BIOS interfaces for power management and device discovery and configuration but had following consequences
 - The behavior of OS applications could be negatively affected by the BIOS-configured power management settings
 - the power management interface was proprietary on each system which required developers to learn how to configure power management for each individual system
 - the default settings for various devices could also conflict with each other

What is ACPI Specification?

- Enables independent evolution of PM technology, co-working of:
 - OS
 - Hardware
- OS- directed motherboard device configuration and power management of both:

- Devices
- Entire system
- It is an architecture-independent power management and configuration framework that forms a subsystem within the host OS
- Has a hardware register set to define power states (sleep , hibernate, wake etc.) that can accommodate operations on dedicated h/w and general purpose h/w.
- It acts as an interface between OS and system firmware

Acpi structure

- ACPI defines 2 types of data structure that acts as primary communication mechanism between OS and firmware
 - Data tables –store raw data and are consumed by device drivers
 - Definition blocks - consist of byte code that is executable by an interpreter.
- Upon initialization, the AML interpreter extracts the byte code in the definition objects as enumerable objects
- This collection of enumerable objects forms the OS construct called the ACPI namespace
- Objects can have directly defined values or must be evaluated and interpreted by the AML interpreted
- AML interpreter interfaces with the system hardware to perform necessary operations with the help of these objects

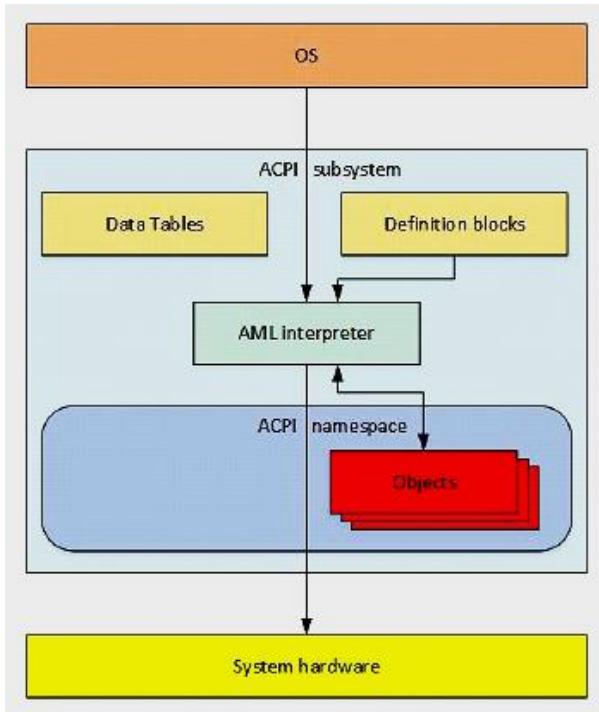


Fig-3 ACPI structure

ASL and AML

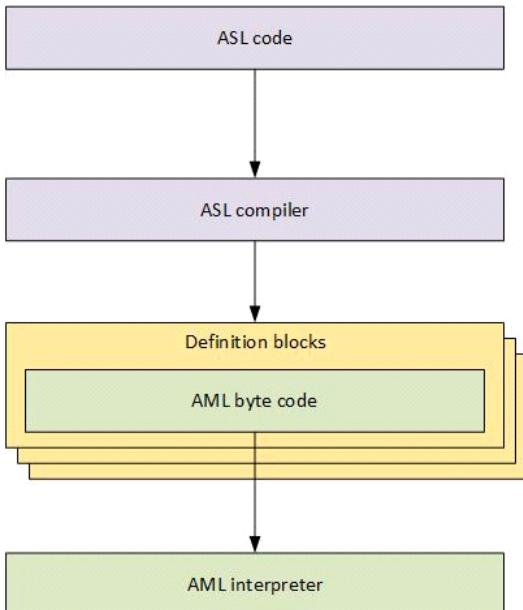


Fig-4 ASL to AML conversion

- Definition block byte code is compiled from ACPI Source Language (ASL)
- ASL is the language used to define ACPI objects and to write control methods
- The ASL compiler translates ASL into ACPI Machine Language (AML) byte code contained within the ACPI definition blocks

- Definition blocks consist of an identifying table header and byte code that is executable by an AML interpreter
- Executes byte code to perform loop constructs, access defined address space and perform other operations that application requires
- Has read/write access
- Objects
- Are the entry points for AML interpreter to access the address space
- These can be directly defined values or must be evaluated and interpreted by the AML interpreter
- ACPI namespace
- Collection of enumerable objects
- It is an OS construct
- Hierarchical representation of the ACPI devices on system

ACPI initialization

- When a system is powered ON, system firmware completes its setup, initialization and self test
- System firmware updates the ACPI tables before passing the bootstrap loader
- The XSDT (extended root system description table) is the first table used by the ACPI subsystem and contains the addresses of most of the other ACPI tables on the system and points to the FADT, SSDT, and other Major ACPI Tables
- The FADT(fixed ACPI description table) directs the ACPI subsystem to the DSDT (differentiated system description table) , which is the beginning of the namespace by virtue of being the first table that contains a definition block
- The ACPI subsystem then consumes the DSDT and begins building the ACPI namespace from the definition blocks.
- The XSDT also points to the SSDTs (secondary system description tables) and adds them to the namespace

Global System State Definitions (G States)

- Global system states are defined by six principal criteria:

- 1.Does application software run?
- 2.What is the latency from external events to application response?
- 3.What is the power consumption?
- 4.Is an OS reboot required to return to a working state?

5. Is it safe to disassemble the computer?

6. Can the state be entered and exited electronically?

Global System State Definitions (G States)

Global system state	Software runs	Latency	Power consumption	OS restart required	Safe to disassemble computer	Exit state electronically
G0 Working	Yes	0	Large	No	No	Yes
G1 Sleeping	No	>0, varies with sleep state	Smaller	No	No	Yes
G2/S5 Soft Off	No	Long	Very near 0	Yes	No	Yes
G3 Mechanical Off	No	Long	RTC battery	Yes	Yes	No

Device Power State Definitions (D States)

- Device states are generally defined in terms of four principal criteria:
 1. Power consumption--How much power the device uses.
 2. Device context--How much of the context of the device is retained by the hardware. The OS is responsible for restoring any lost device context (this may be done by resetting the device)
 3. Device driver--What the device driver must do to restore the device to full on.
 4. Restore time--How long it takes to restore the device to full on.

Device State	Power Consumption	Device Context Retained	Driver Restoration
D0 - Fully-On	As needed for operation	All	None
D1	D0>D1>D2> D3hot>D3	>D2	<D2
D2	D0>D1>D2> D3hot>D3	<D1	>D1
D3hot	D0>D1>D2>D3hot>D3	Optional	None <->Full initialization and load
D3 - Off	0	None	Full initialization and load

Sleeping and Soft-off State Definitions (S States)

- **S1 Sleeping State**
 - low wake latency sleeping state
 - no system context is lost (CPU or chip set)
 - hardware maintains all system context.

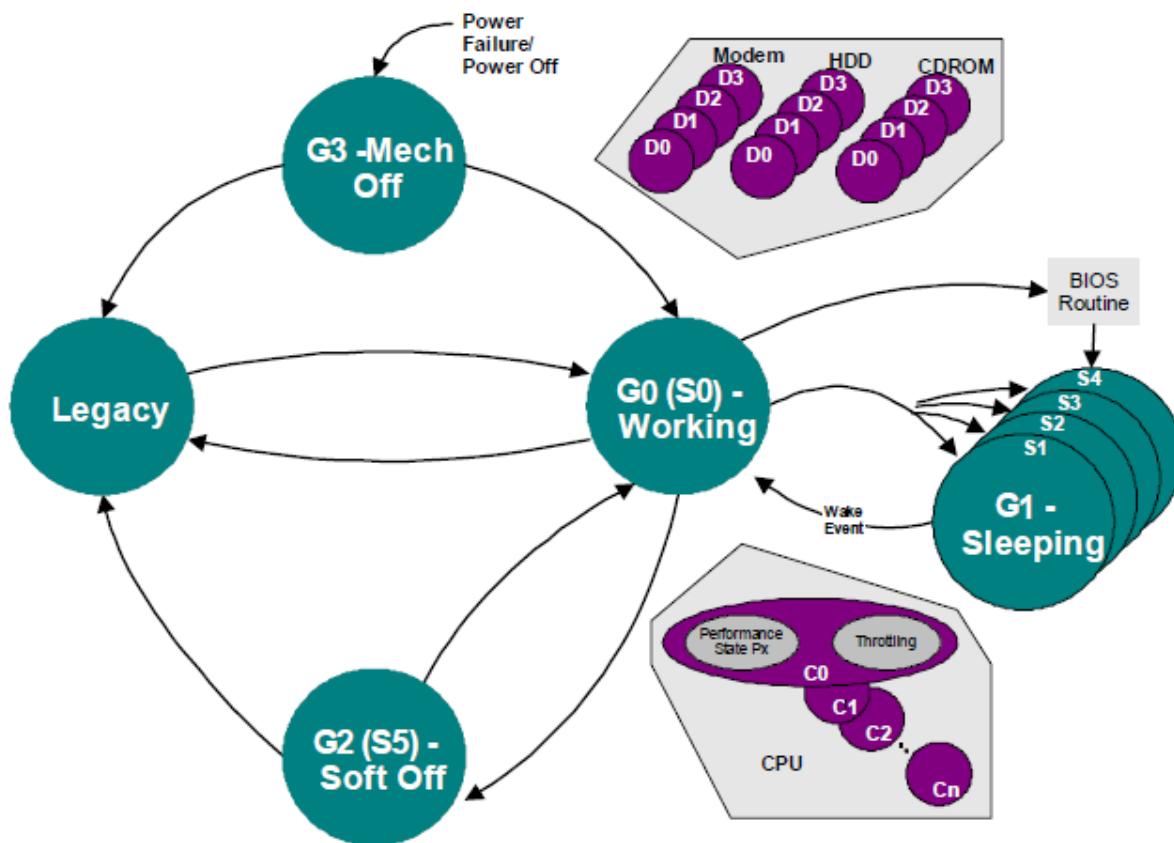
- **S2 Sleeping State**
 - low wake latency sleeping state
 - similar to the S1 except that the CPU and system cache context is lost
 - Control starts from the processor's reset vector after the wake event
- **S3 Sleeping State**
 - low wake latency sleeping state
 - all system context is lost except system memory.
 - Control starts from the processor's reset vector after the wake event.
- **S4 Sleeping State**
 - lowest power, longest wake latency
 - hardware platform has powered off all devices.
 - Platform context is maintained.
- **S5 Soft Off State**
 - S5 state is similar to the S4 state except that the OS does not save any context
 - requires a complete boot when it wakes

Processor Power State Definitions (C States)

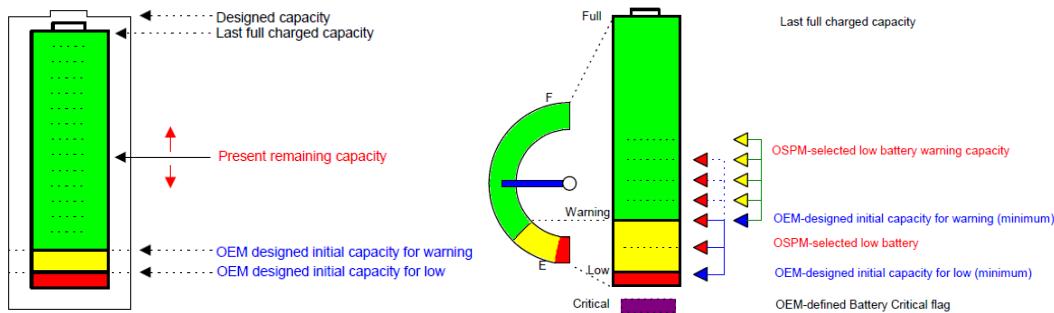
- **C0 Processor Power State**
 - processor is in this state, it executes instructions
- **C1 Processor Power State**
 - has the lowest latency
 - Aside from putting the processor in a non-executing power state, this state has no other software-visible effects
- **C2 Processor Power State**
 - offers improved power savings over the C1 state
 - The worst-case hardware latency for this state is provided via the ACPI system firmware
- **C3 Processor Power State**
 - C3 state offers improved power savings over the C1 and C2 states
 - While in the C3 state, the processor's caches maintain state but ignore any snoops.
 - The operating software is responsible for ensuring that the caches maintain coherency.

Device and Processor Performance State Definitions (P States)

- **P0 Performance State**
 - uses maximum performance capability and may consume maximum power.
- **P1 Performance State**
 - performance capability is limited below its maximum and consumes less than maximum power.
- **Pn Performance State**
 - minimum level and consumes minimal power while remaining in an active state
 - State n is a maximum number and is processor or device dependent.
 - number of performance states not to exceed 255.



Battery Capacity



Thermal Management

- ACPI defines two cooling modes to reduce the temperature of the machine :
 - **Passive cooling**
 - OS reduces the power consumption of devices
 - At the cost of system performance.
 - **Active cooling**
 - OS increases the power consumption of the system
 - for example, by turning on a fan
- Performance vs Energy Consumption
 - OSPM allows end user to choose a preference
 - Performance -> Greater Active cooling
 - Energy -> Greater Passive cooling

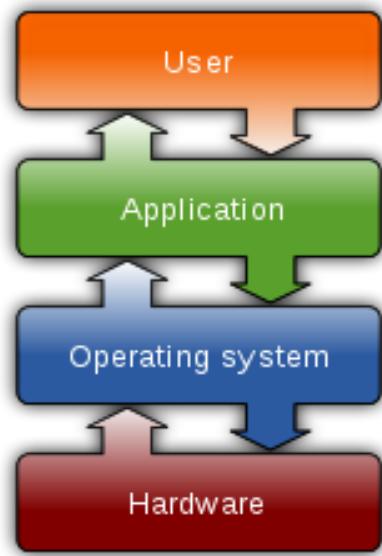
Chapter-13

Operating system Installation and Troubleshooting

An operating system (OS) is system software that manages computer hardware and software resources and provides common services for computer programs.

The operating system manages a computer's hardware resources, including:

- Input devices such as a keyboard and mouse
- Output devices such as display monitors, printers and scanners
- Network devices such as modems, routers and network connections
- Storage devices such as internal and external drives



For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary between programs and the computer hardware, although the application code is usually executed directly by the hardware and frequently makes system calls to an OS function or is interrupted by it. Operating systems are found on many devices that contain a computer – from cellular phones and video game consoles to web servers and supercomputers

Types of operating systems

Single- and multi-tasking:

A single-tasking system can only run one program at a time, while a multi-tasking operating system allows more than one program to be running in concurrency. This is achieved by time-sharing, where the available processor time is divided between multiple processes. These processes are each interrupted repeatedly in time slices by a task-scheduling subsystem of the operating system. Multi-tasking may be characterized in preemptive and co-operative types. In preemptive multitasking, the operating system slices the CPU time and dedicates a slot to each of the programs. Unix-like operating systems, such as Solaris and Linux—as well as non-Unix-like, such as AmigaOS—support preemptive multitasking.

Single- and multi-user:

Single-user operating systems have no facilities to distinguish users, but may allow multiple programs to run in tandem.^[6] A multi-user operating system extends the basic concept of multi-tasking with facilities that identify processes and resources, such as disk space, belonging to multiple users, and the system permits multiple users to interact with the system at the same time. Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting software for cost allocation of processor time, mass storage, printing, and other resources to multiple users.

Distributed:

A distributed operating system manages a group of distinct computers and makes them appear to be a single computer. The development of networked computers that could be linked and communicate with each other gave rise to distributed computing. Distributed computations are carried out on more than one machine. When computers in a group work in cooperation, they form a distributed system.

Templated:

In an OS, distributed and cloud computing context, templating refers to creating a single virtual machine image as a guest operating system, then saving it as a tool for multiple running virtual machines. The technique is used both in virtualization and cloud computing management, and is common in large server warehouses.

Embedded:

Embedded operating systems are designed to be used in embedded computer systems. They are designed to operate on small machines like PDAs with less autonomy. They are able to operate with a limited number of resources. They are very compact and extremely efficient by design. Windows CE and Minix 3 are some examples of embedded operating systems.

Real-time:

A real-time operating system is an operating system that guarantees to process events or data by a specific moment in time. A real-time operating system may be single- or multi-tasking, but when multitasking, it uses specialized scheduling algorithms so that a deterministic nature of

behavior is achieved. An event-driven system switches between tasks based on their priorities or external events while time-sharing operating systems switch tasks based on clock interrupts.

Library:

A library operating system is one in which the services that a typical operating system provides, such as networking, are provided in the form of libraries and composed with the application and configuration code to construct a unikernel: a specialized, single address space, machine image that can be deployed to cloud or embedded environments.

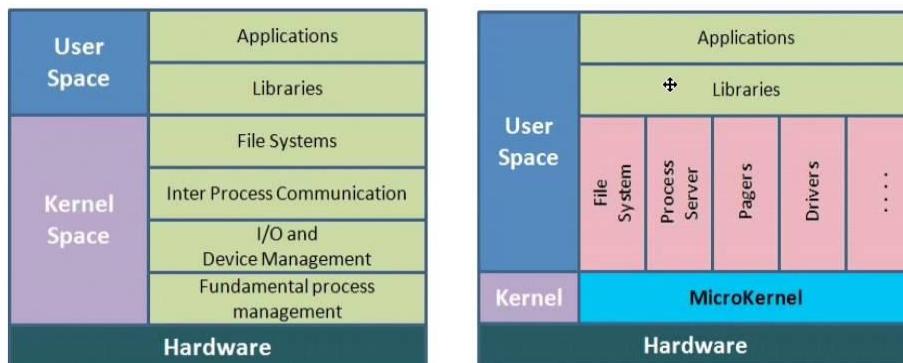
Kernel

- A kernel is the central part of an operating system. It manages the operations of the computer and the hardware - most notably memory and CPU time.

There are two types of kernels:

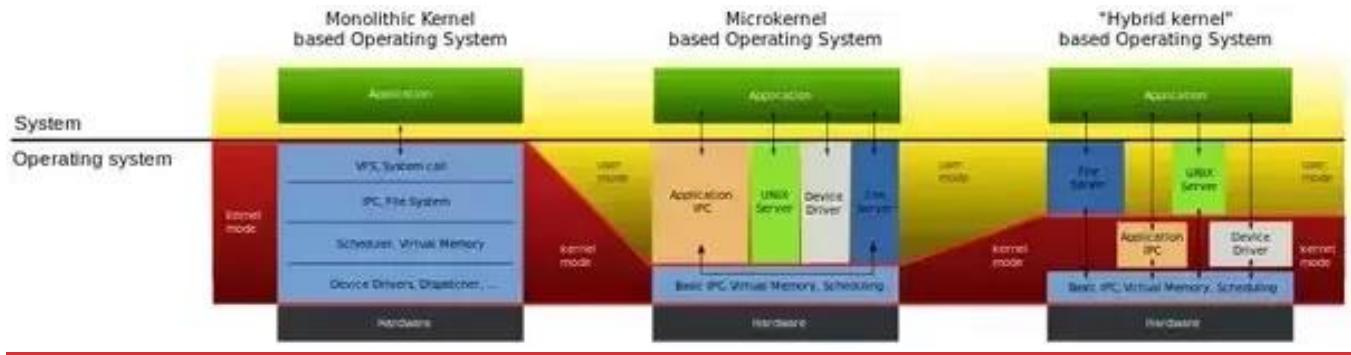
- A micro kernel, which only contains basic functionality
- A monolithic kernel, which contains many device drivers

Monolithic Kernel vs Microkernel

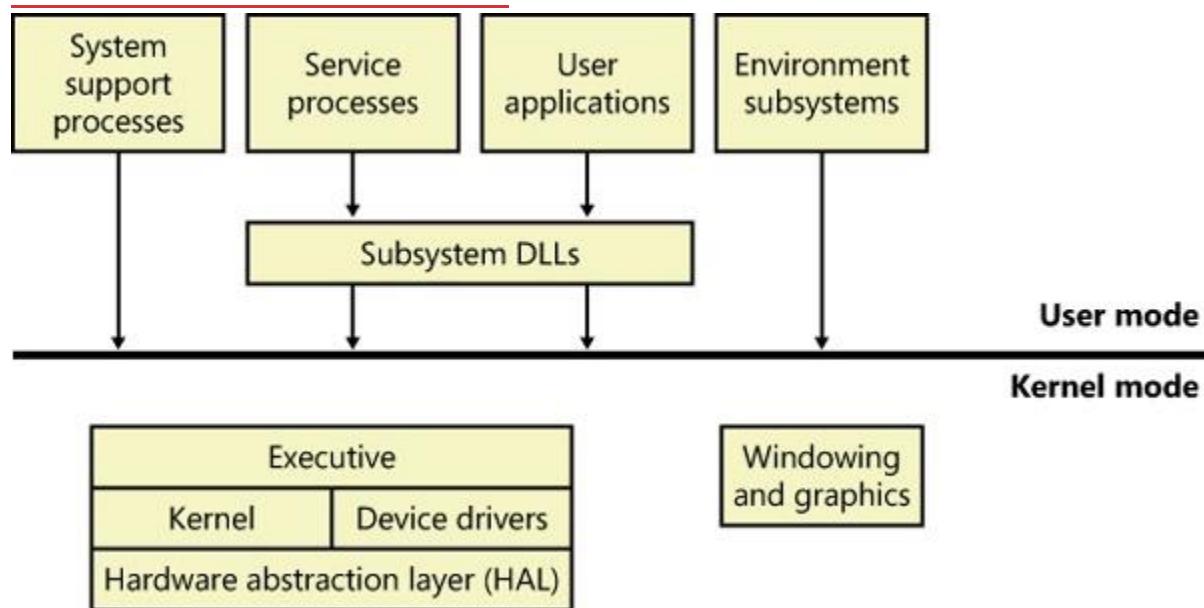


Hybrid kernel

A hybrid kernel is an operating system kernel architecture that attempts to combine aspects and benefits of microkernel and monolithic kernel architectures used in computer operating systems.



Simplified Windows Architecture



File system

- A file system is a process that manages how and where data on a storage disk, typically a hard disk drive (HDD), is stored, accessed and managed
- Commonly used file systems
 - File Allocation Table (FAT)
 - New Technology File System (NTFS)
 - Hierarchical File System (HFS)

OS and File System Compatibility

Operating System	FAT16	FAT32	NTFS
Windows XP	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Windows 2000	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Windows NT	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>
Windows 95, 98, ME	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
Windows 95	<input checked="" type="checkbox"/>		
MS-DOS	<input checked="" type="checkbox"/>		

OS installation

- Insert the installation disk or flash drive. To install a new operating system on Windows, you must have the operating system's install tool on a DVD or flash drive, and the disk or flash drive must be inserted into your computer. If you don't have a version of the operating system you want to download, you can download one for the following operating systems:
 - Windows 10
 - Windows 8
 - Windows 7



Restart your computer. Press and hold your computer's Power button to turn it off, wait for a few seconds, and then press the Power button again to turn the computer back on.

Press and hold Del or F2 to enter the BIOS page. The key you're prompted to press might also be different; if so, use that key instead.



- **Locate the "Boot Order" section**

- This section is normally on the main page of the BIOS, though you may need to use the arrow keys to navigate over to the "Boot" or "Advanced" tab.



- **Select the location from which you want to start your computer.**

- In this case, you'll be starting your computer either from the disk drive or from an external drive, such as a flash drive.
- **Move your selected location to the top of the list .**

-Typically, you'll press the + key with the boot location you want to use selected until it's at the top of the "Boot Order" list.

Save your settings and exit the BIOS

-There should be a "Save and Exit" key listed in the BIOS' key legend; press the key listed there to save your settings and exit the BIOS.

Restart your computer

Follow the OS user configuration and setup the OS.



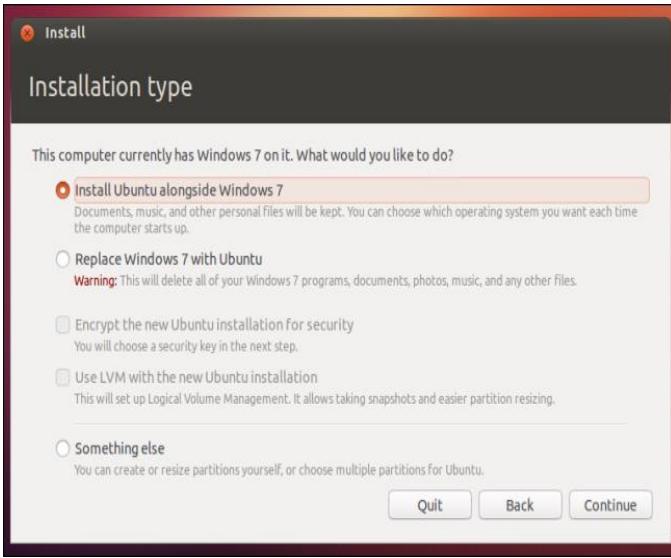
Multiboot Operating system

How Dual-Booting Works

- Your computer's operating system is generally installed on its internal hard drive. When you boot your computer, the BIOS loads the boot loader from the hard drive and the boot loader boots the installed operating system.
- There's no limit to the number of operating systems you can install — you're not just limited to a single one. You could put a second hard drive into your computer and install an operating system to it, choosing which hard drive to boot in your BIOS or boot menu. You could also boot an operating system — like a live Linux system or a Windows To Go USB drive — from external storage media.

Setting Up a Dual-Boot System

Dual Boot Windows and Linux: Install Windows first if there's no operating system installed on your PC. Create Linux installation media, boot into the Linux installer, and select the option to install Linux alongside Windows



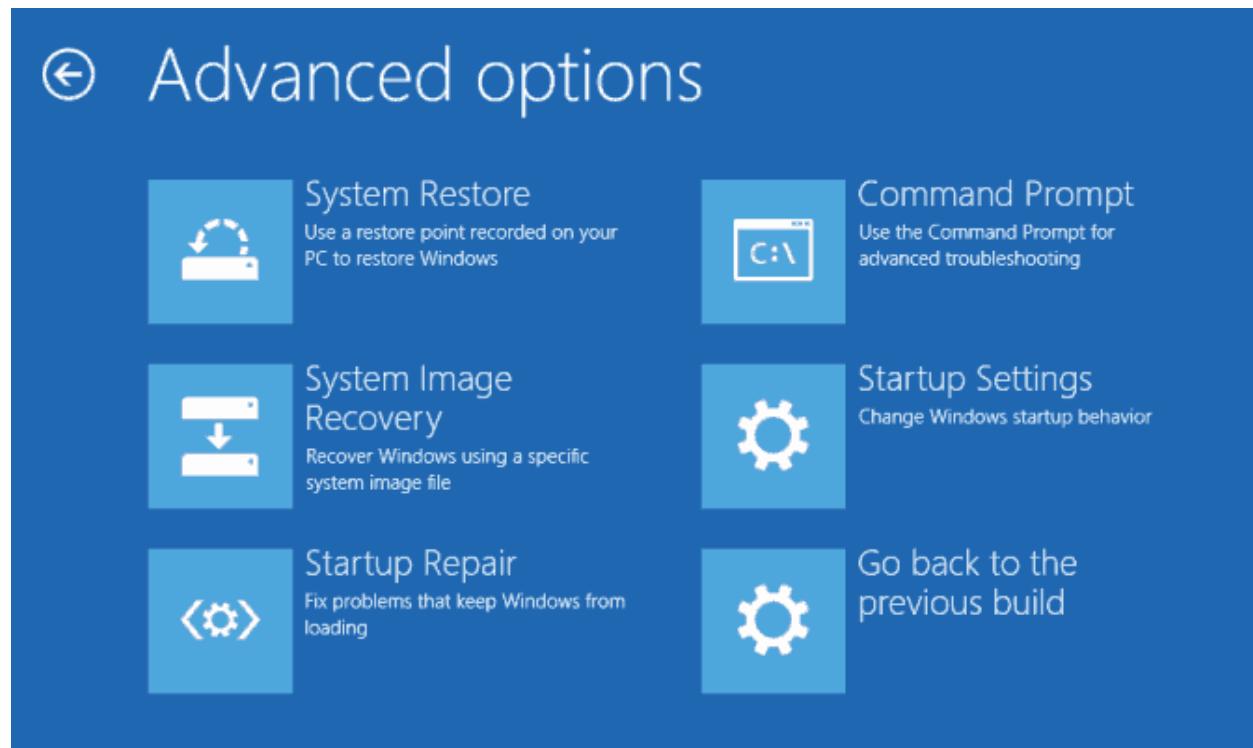
Other multiboot combination

- **Dual Boot Windows and Another Windows:** Shrink your current Windows partition from inside Windows and create a new partition for the other version of Windows. Boot into the other Windows installer and select the partition you created. Read more about dual-booting two versions of Windows.
- **Dual Boot Linux and Another Linux:** You should be able to dual-boot two Linux distributions by installing one first and then installing the other. Choose to install the new Linux system alongside your old Linux system. Resize your old Linux partitions in the installer and create new ones to make space if the installer won't do this automatically.
- **Dual Boot Mac OS X and Windows:** The Boot Camp utility included with Mac OS X allows you to easily set up a Windows dual-boot system on your Mac.
- **Dual Boot Mac OS X and Linux:** Boot Camp doesn't allow you to set up a dual-boot Linux system, so you'll need to do a bit more footwork here. Follow our guide to installing Linux on a Mac for more details.

BOOT REPAIR

- **Boot-Repair** is a simple tool to repair frequent boot issues you may encounter in Ubuntu like when you can't boot Ubuntu after installing Windows or another Linux distribution, or when you can't boot Windows after installing Ubuntu, or when GRUB is not displayed anymore, some upgrade breaks GRUB, etc...
- **Boot-Repair** lets you fix these issues with a simple click, which (generally reinstalls GRUB and) restores access to the operating systems you had installed before the issue.

Boot-Repair also has advanced options to back up table partitions, back up bootsectors, create a Boot-Info (to get help by email or forum), or change the default repair parameters: configure GRUB, add kernel options (acpi=off ...), purge GRUB, change the default OS, restore a Windows-compatible MBR, repair a broken filesystem, specify the disk where GRUB should be installed, etc



Backup

- You can backup the MBR since it's stored on the hard disk and you restore it later, when needed.
- You only need one command to backup the MBR:

```
dd if=/dev/sda of=/path-to-save/mbr-backup bs=512 count=1
```

In the above command, /path-to-save/mbr-backup is the path where you want to save mbr-backup

Restore

You can restore the saved Master boot record using this command:

```
dd if=/path-to-get/mbr-backup of=/dev/sda bs=512 count=1
```

- Operating System not found
- Operating System not found
- No bootable medium found
- Reboot and select proper boot device

Fix the MBR with Easy Recovery Essentials

- Easy Recovery Essentials, our bootable recovery and repair CD/USB, is guaranteed to repair most damages done to the Master boot record (MBR) using its **Automated Repair** feature for Windows XP, Vista, 7 or 8:
1. Download Easy Recovery Essentials. Choose your Windows version (XP, Vista, 7 or 8) before downloading.
 2. Burn the image. Follow these instructions on how to burn the bootable ISO image very carefully, as making a bootable CD can be tricky!
 3. Boot Easy Recovery Essentials
 4. Choose **Automated Repair** option and click Continue
 5. Choose your Windows installation drive's letter (usually it's c:/ and click on **Automated Repair**.
 6. Once the process is complete, click Restart

Restoring the grub

Recovering Ubuntu After Installing Windows

- Using the Ubuntu CD (Recommended)
1. Insert your Ubuntu CD, reboot your computer and set it to boot from CD in the BIOS and boot into a live session. You can also use a LiveUSB if you have created one in the past.
 2. Install and run Boot-Repair
 3. Click "Recommended Repair".
 4. Now reboot your system. The usual GRUB boot menu should appear. If it does not, hold Left Shift while booting. You will be able to choose between Ubuntu and Windows

Windows firewall

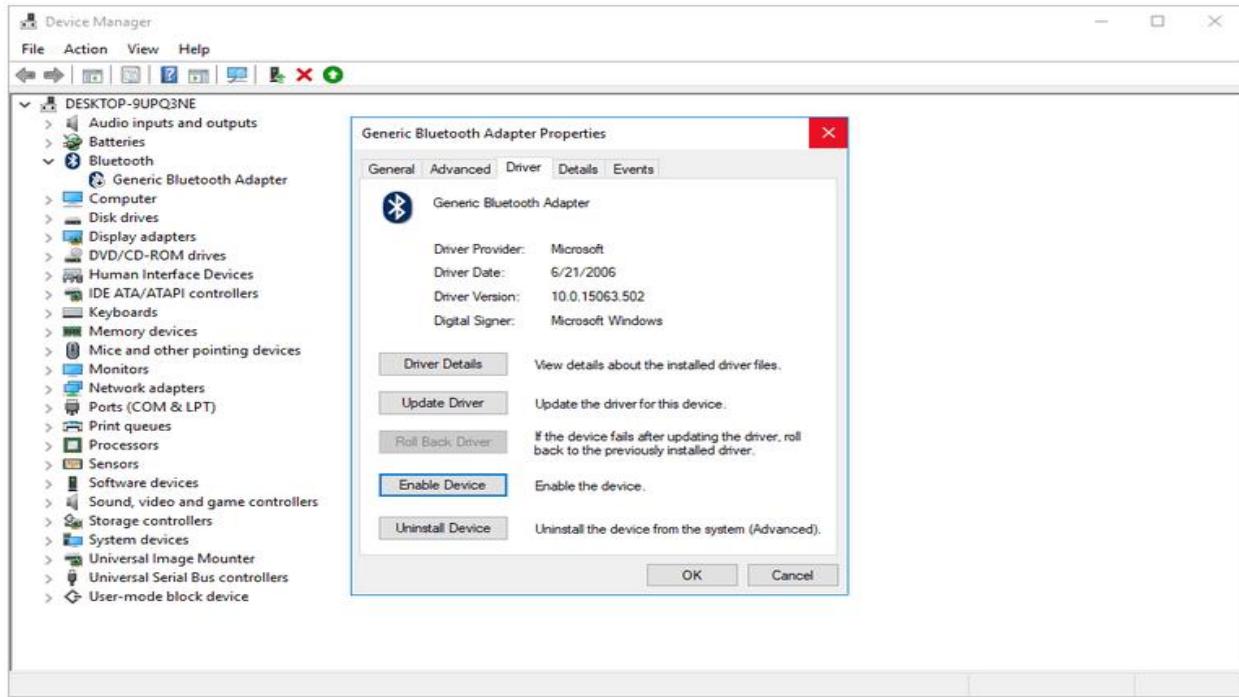
- A firewall, at its most basic level, permits or denies communications between computers, between networks, or between computers and networks.
- Only connections are explicitly allowed, using firewall rules, are permitted. Windows Firewall, by default, allows all *outbound* connections, and permits only *established inbound* connections.
- To turn Windows Defender Firewall on or off, select the **Start** button, and then select **Settings > Update & Security > Windows Security > Firewall & network protection**. Choose a network profile, and then under **Windows Defender Firewall**, switch the setting to **On** or **Off**.

WINDOWS DEFENDER

- Windows Defender is Microsoft's free antispyware program that is included in Windows 7. Like Windows Firewall, Windows Defender is enabled by default, and no configuration is required for it to begin protecting your computer.
- There are some Windows Defender settings you may want to customize, such as how often and when your computer is automatically scanned, what Windows Defender does when spyware is detected, and what you want scanned or excluded (such as certain programs, files, folders, or e-mail).

Rollback Drivers

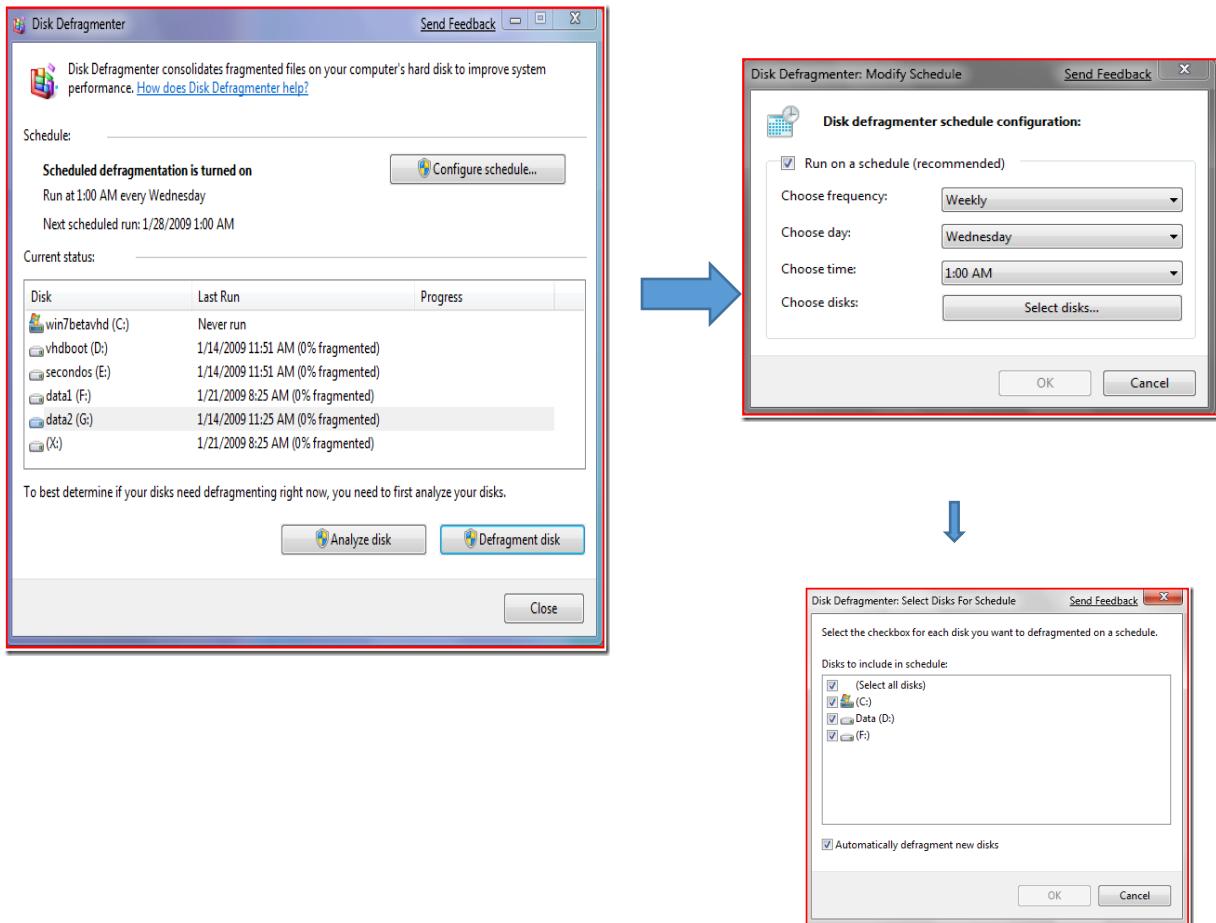
- The *Roll Back Driver* feature, available within Device Manager in all versions of Windows, is used to uninstall the current driver for a hardware device and then automatically install the previously installed driver.
- Think of rolling back a driver as a quick and easy way to uninstall the latest driver, and then reinstall the previous one, all in one simple step.
- The most common reason to use the driver roll back feature in Windows is to "reverse" a driver update that didn't go so well. Maybe it didn't fix the problem that the driver update was supposed to fix, or maybe the update actually *caused* a problem.



- Unfortunately, the Driver Roll Back feature is not available for printer drivers, as handy as that would be. Driver Roll Back is only available for hardware that's managed within Device Manager.
- Additionally, Driver Roll Back only allows you to roll back a driver once. In other words, Windows only keeps a copy of the very last driver installed. It does not keep an archive of all the previously installed drivers for the device.
- If there's no driver to roll back to, but you know there's a previous version available that you'd like to install, just "update" the driver with the older version

Disk defrag

- Defragmentation is the process of locating the noncontiguous fragments of data into which a computer file may be divided as it is stored on a hard disk, and rearranging the fragments and restoring them into fewer fragments or into the whole file.
- Defragmentation reduces data access time and allows storage to be used more efficiently. Some operating systems automatically defragment storage periodically; others require that the user occasionally use a special utility for this purpose.
- "Defrag" is a short form of the verb *to defragment* and sometimes the name of the utility used for defragmenting, which is also called a *defragmenter*.



Formatting the disk

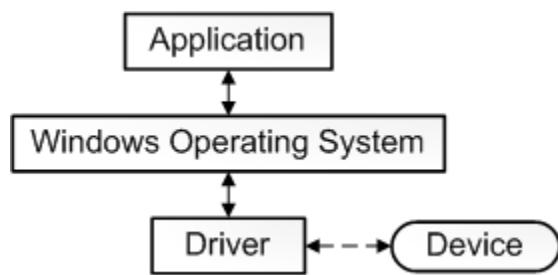
- **Disk formatting** is the process of preparing a data storage device such as a hard disk drive, solid-state drive, floppy disk or USB flash drive for initial use.
- In some cases, the formatting operation may also create one or more new file systems.
- The first part of the formatting process that performs basic medium preparation is often referred to as "low-level formatting".
- Partitioning is the common term for the second part of the process, making the data storage device visible to an operating system. The third part of the process, usually termed "high-level formatting" most often refers to the process of generating a new file system.

Windows Aero

- It is a design language introduced in the Windows Vista operating system.
- The changes made in the Aero interface affected many elements of the Windows interface, including the incorporation of a new look, along with changes in interface guidelines reflecting appearance, layout, and the phrasing and tone of instructions and other text in applications.
- Windows Aero provides users with visually appealing effects, appearance and access to applications. Examples of these are the translucent title bars, stylish color schemes and smooth, rounded edges. Windows Aero provides a smoother desktop experience along with dynamic window animations.

Device drivers

- a driver is a software component that lets the operating system and a device communicate with each other.

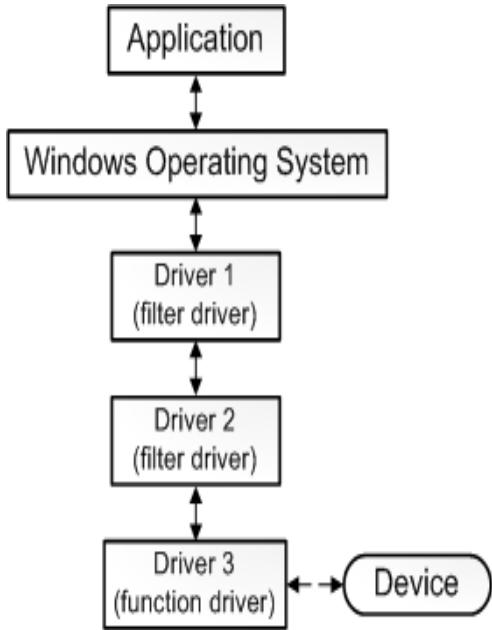


For example, suppose an application needs to read some data from a device. The application calls a function implemented by the operating system, and the operating system calls a function implemented by the driver. The driver, which was written by the same company that designed and manufactured the device, knows how to communicate with the device hardware to get the data. After the driver gets the data from the device, it returns the data to the operating system, which returns it to the application.

function driver and filter drivers.

The one driver in the stack that communicates directly with the device is called the *function driver*; the drivers that perform auxiliary processing are called *filter drivers*.

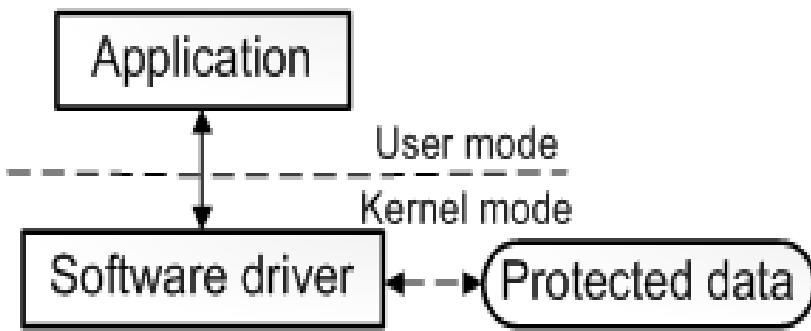
We could expand our definition of *driver* by saying that a driver is any software component that observes or participates in the communication between the operating system and a device.



Software drivers

- The component that runs in user mode is called an application, and
- the component that runs in kernel mode is called a *software driver*.

A software driver is not associated with a hardware device



Universal driver

- Universal Windows drivers enable developers to create a single driver package that runs across multiple different device types, from embedded systems to tablets and desktop PCs.
- A Universal Windows driver package contains an INF file and binaries that install and run on Universal Windows Platform (UWP) based editions of Windows 10 as well as other Windows 10 editions that share a common set of interfaces.
- The following are required when writing a universal driver package:

- Create a universal INF file for your driver:
 - Review the list of INF sections and directives that are valid in universal driver packages.
 - Use the InfVerifier tool to verify that your driver package's INF file is universal.
- Use the ApiValidator tool to verify that the APIs your binaries call are valid for a universal driver package.

Validating Universal Windows drivers

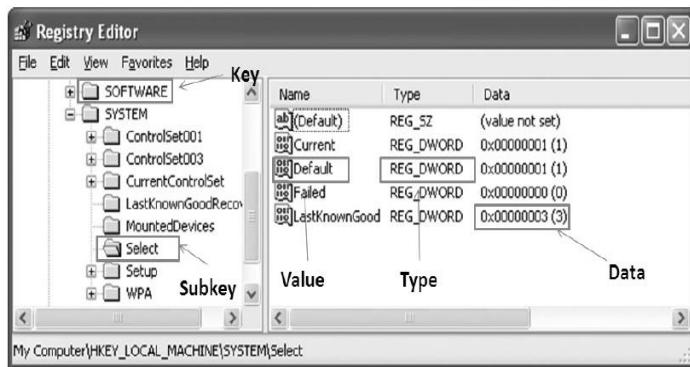
- You can use the ApiValidator.exe tool to verify that the APIs that your binaries call are valid for a Universal Windows driver. The tool returns an error if your binaries call an API that is outside the set of valid APIs for Universal Windows drivers. This tool is part of the Windows Driver Kit (WDK) for Windows 10.

Running ApiValidator from the Command Prompt

- navigate to C:\Program Files (x86)\Windows Kits\10\bin\<arch>.
- **Apivalidator.exe -DriverPackagePath:<driver folder path> -SupportedApiXmlFiles:<path to XML files containing supported APIs for universal drivers>**

What is the Registry?

- The Windows Registry is used to store much of the information and settings for software programs, hardware devices, user preferences, operating system configurations, and more.
- In many ways, the registry can be thought of as a kind of DNA for the Windows operating system.
- It is a database of artifacts for a forensic



The Registry Hives

1. HKEY CLASSES_ROOT

- Contains the necessary information for Windows to know what to do when you ask it to do something.
- View a drive
- Open files
- Contains file extension association information
- Program ID's

2. HKEY CURRENT USER

- Contains configuration information for Windows and software specific configurations for the currently logged in user.
- User level settings like:
- Installed/Preferred printers
- Wallpaper
- Display settings
- EnvVariables
- Keyboard layout
- Mapped drives

3. HKEY LOCAL MACHINE

- Contains configuration
- data for software installed, as well as the operating system .
- Information about connected devices
- Contains boot configuration

4. HKEY USERS

- Contains configuration information
- corresponding to each system user.

- Each user has their own key
- Each user has their own security identified, SID
- Associated with Current User Key

5.HKEY CURRENT CONFIG

- Contains keys that are
- pointers to information about the hardware profiles
- currently in use.
- Really just for convenience

Registry Structure

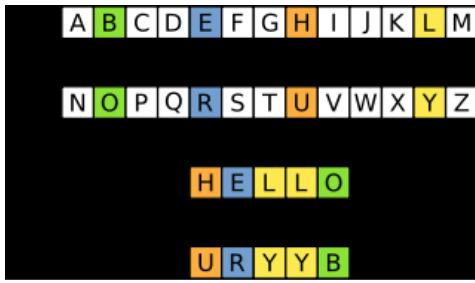
- Registry is maintained within several files on the system.
- Hives that are persistent on the system: %SYSTEMROOT%\system32\config folder.
 - HKEY_LOCAL_MACHINE\System (System folder).
 - Specific user configuration:
 - Documents and Settings folder.
 - NTUSER.DAT file.

Registry as a Log File

- Registry keys have a value indicate “LastWrite” time. FILETIME.
- Indicates Registry key last modification.

What's in the Registry?

- Useful forensic information within the Registry àDepends on investigator.
- Registry string search little data.
- -Data may be kept in binary.
- -ROT-13 encryption.



Tools to Decode



RegRigger Extracts and Decrypts



How Registry Data is Stored?

- Most data needs to be converted into human readable format

- Binary

- XML

- ### • Rot-13 Encoded

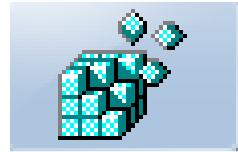
- ## • User Assist Data

- Key-Values are tagged with Last Update time (64 bit Windows datatime) but can

Page | 359

- not be viewed in the Registry Viewer. Need tools to extract the data and reconstruct it.

Tools to View the Registry



Windows supplied RegEdit

Lacks display of LastUpdate

Doesn't parse data



RegRipper

Extensible through plugins

Windows Registry Reviewer



Browsing and extract tool

Autostart Locations

- Allow applications to be launched without any direct user interaction.

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run

- Most popular: "Run" key.

- Used by many pieces of malware.

- Many other Registry keys launch applications and depend on actions:
 - –System starts up.
 - – User logs in.
 - Example 1: The key

software\Microsoft\CommandProcessor\AutoRun

- Found in both the HKEY_LOCAL_MACHINE and HKEY_CURRENT_USER hives
- launch application whenever cmd.exe is launched

USB Removable Storage

Mounted Devices

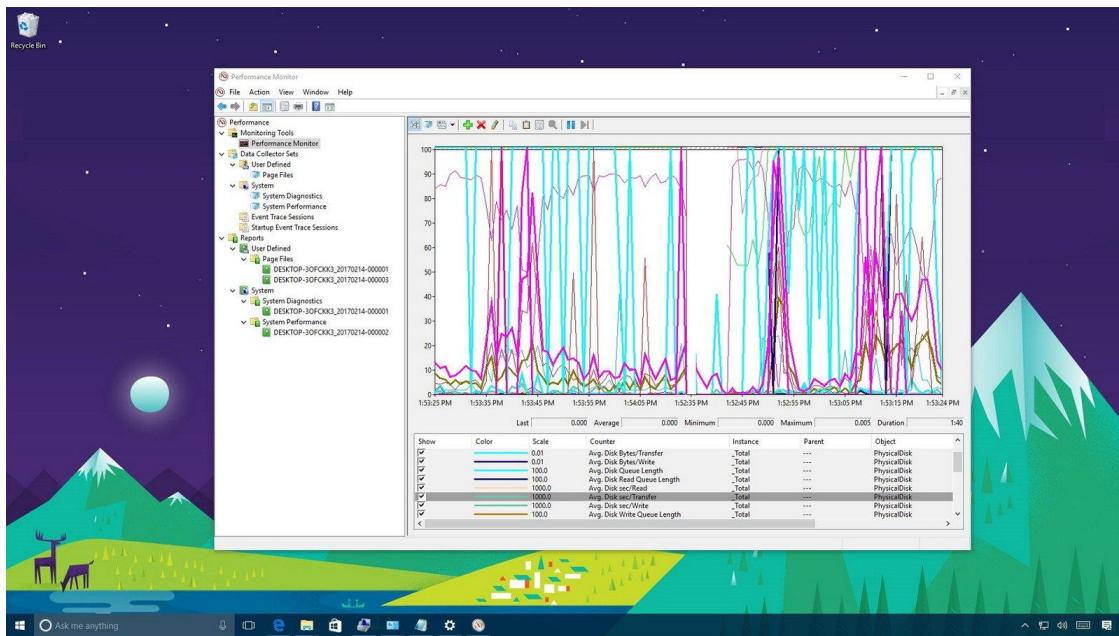
- HKEY LOCAL MACHINE\System\MountedDevices
 - View each drive associated with the system.
 - Stores a database of mounted volumes used by NTFS.
- The investigator may indicate that the user removed the drive if:
 - a device is in the list of MountedDevices and it is not physically in the system.

Chapter-14

OS Environmental Tools

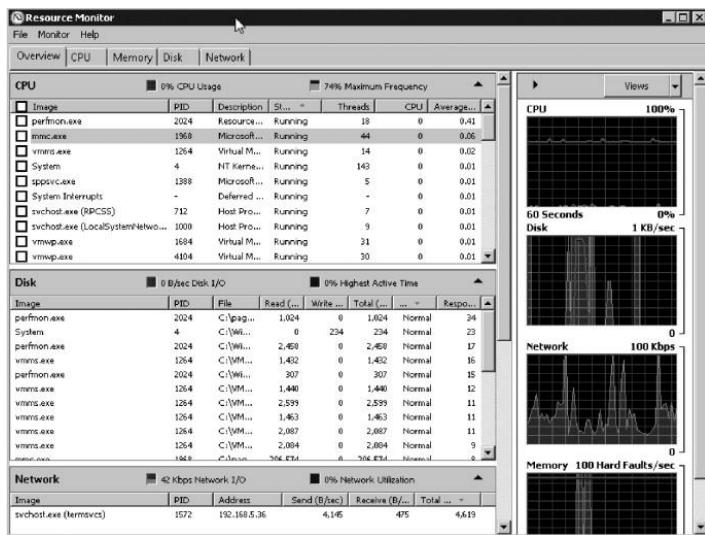
Performance monitor

- when your computer suddenly isn't fast as usual, or keeps freezing up, you're likely dealing with performance issues that could be caused by a number of reasons. It could be as a result of problems with the hardware, apps or drivers poorly designed, the operating system using excessive system resources, or even be a malware that infected your device.



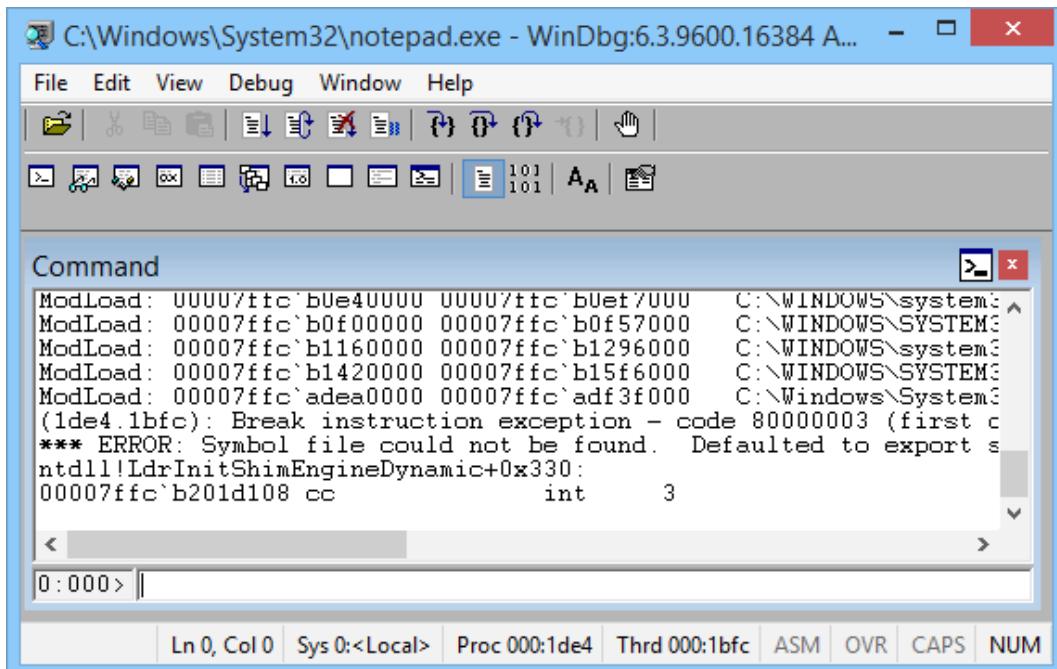
Resource monitor

- Identifying high processor usage using resource monitor Resource Monitor is a tool built into Windows and Windows Server that provides more detail about processor usage than Task Manager.



WinDbg

- The Windows Debugger (WinDbg) can be used to debug kernel-mode and user-mode code, to analyze crash dumps, and to examine the CPU registers while the code executes.
- WinDbg** is a multipurpose debugger for the Microsoft Windows computer operating system, distributed by Microsoft



Crash dump analysis

- Extracting information from a memory dump after a OS crash is an important part of root cause analysis.
- **Tools Required:**
 - **WinDbg:**
 - WinDbg is the main program for debugging code and analyzing crash dumps. This software is provided by Microsoft as part of the Windows SDK
 - **Debug Symbols:**
 - debug symbols provide additional metadata about the program you're debugging to make it easier to read and understand the details of what's in the dump

SDK

- Windows SDK are software development kits developed by Microsoft Corp. They and other SDK like the Platform SDK and .NET Framework SDK contains samples, documentation, libraries and header files and tools using which developers can develop applications for Microsoft Windows and .NET Framework. The difference between these software development kits or SDK lies in their area of specialization.
- The Visual Studio SDK helps you extend Visual Studio features or integrate new features into Visual Studio.
- The Visual Studio SDK is an optional feature in Visual Studio setup. You can also install the VS SDK later on.

WDK

- The **Windows Driver Kit** (WDK) is a software toolset from Microsoft that enables the development of device drivers for the Microsoft Windows platform.
- It includes documentation, samples, build environments, and tools for driver developers.
- Tools under WDK:
 - Poolmon
 - Pwrtest
 - PNPCPU

- PNPUtil
- Usbview etc...

Sysinternal Tools

- The Sysinternals Troubleshooting Utilities have been rolled up into a single Suite of tools. This file contains the individual troubleshooting tools and help files.
- SYStools contains tools on following:
 - File and disk utility
 - Networking utility
 - Process utility
 - System information
 - Miscellaneous

Chapter-15

Testing Basics

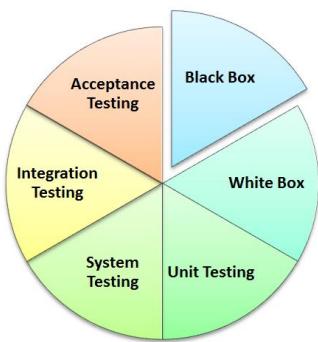
Manual Testing

What is Manual testing?

- Manual Testing is a type of Software Testing where Testers manually execute test cases without using any automation tools.
 - Basically, this testing checks the quality of the system and delivers bug-free product to the customer.
- One of the Software Testing Fundamental is "**100% Automation is not possible**".

Types of Manual Testing:

- Black Box Testing
- White Box Testing
- Unit Testing
- System Testing
- Integration Testing
- Acceptance Testing

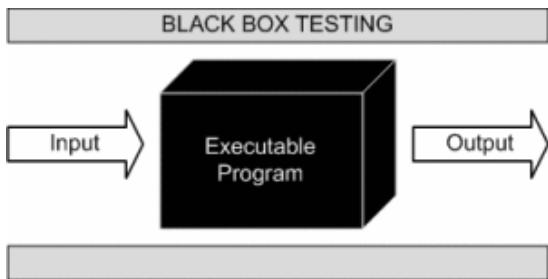


BLACK BOX testing

Testing, either functional or non-functional, without reference to the internal structure of the component or system.

EXAMPLE:

A tester, without knowledge of the internal structures of a hardware /software, tests the model by using a component; providing inputs (clicks, keystrokes) and verifying the outputs against the expected outcome.



Advantages of BBT

- Tests are done from a user's point of view and will help in exposing discrepancies in the specifications.
- Tester need not know programming languages or how the software has been implemented.

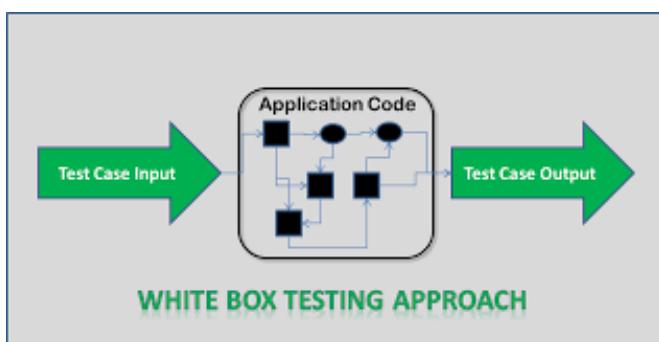
Disadvantages of BBT

- Only a small number of possible inputs can be tested and many program paths will be left untested.
- Without clear specifications, which is the situation in many projects, test cases will be difficult to design.

White box testing

- Is also known as Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester.
- Testing based on an analysis of the internal structure of the component or system.
- **Example:**

White Box Testing is like the work of a mechanic who examines the engine to see why the car is not moving.



Advantages of white box testing

- Testing can be commenced at an earlier stage. One need not wait for the GUI to be available.
- Testing is more thorough, with the possibility of covering most paths

Disadvantages of White Box Testing

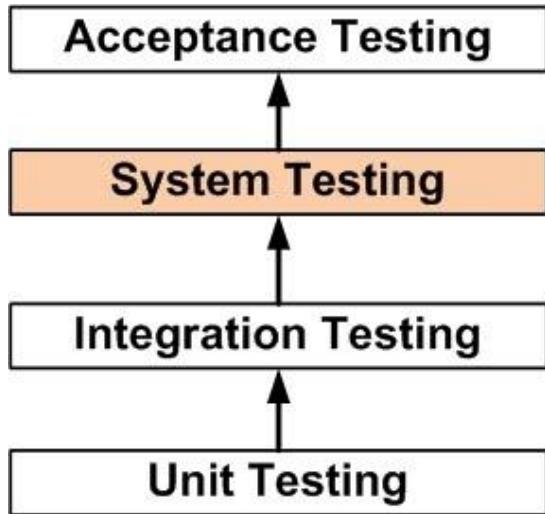
- Since tests can be very complex, highly skilled resources are required, with a thorough knowledge of programming and implementation.
- Test script maintenance can be a burden if the implementation changes too frequently.

Unit testing

- It is performed by using the White_Box_Testing method.
- It is normally performed by software developers themselves or their peers. In rare cases, it may also be performed by independent software testers.
- **UNIT TESTING** is a level of software testing where individual units/ components of a software are tested. The purpose is to validate that each unit of the software performs as designed.
- Unit Testing is the first level of software testing and is performed prior to Integration Testing.

System testing

- **SYSTEM TESTING** is a level of software testing where a complete and integrated software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.
- The process of testing an integrated system to verify that it meets specified requirements.
- Usually, Black Box Testing method is used.



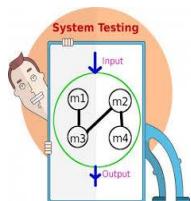
System testing

When is it performed?

- System Testing is the third level of software testing performed after Integration Testing and before Acceptance Testing.

Who performs it?

- Normally, independent Testers perform System Testing.



Integration testing

- **INTEGRATION TESTING** is a level of software testing where individual units are combined and tested as a group.
- The purpose of this level of testing is to expose faults in the interaction between integrated units.
- Test drivers and test stubs are used to assist in Integration Testing.
- Any of Black Box Testing, White Box Testing and Gray Box Testing methods can be used. Normally, the method depends on your definition of 'unit'.

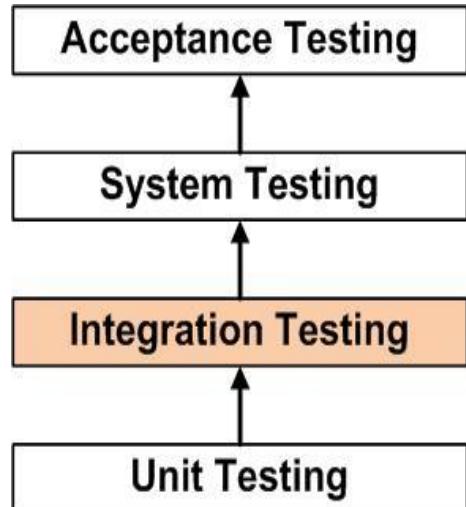
Integration testing

When is Integration Testing performed?

- Integration Testing is the second level of testing performed after Unit Testing and before System Testing.

Who performs Integration Testing?

- Developers themselves or independent testers perform Integration Testing.



Acceptance test

- **ACCEPTANCE TESTING** is a level of software testing where a system is tested for acceptability.
- The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.
- Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user.

Software process

- A software process (also known as software methodology) is a set of related activities that leads to the production of the software. These activities may involve the development of the software from the scratch, or modifying an existing system.
- **Any software process must include the following four activities**
 - **Software specification**
 - **Software design and implementation**

- Software verification and validation
- Software evolution

SDLC (software development life cycle)

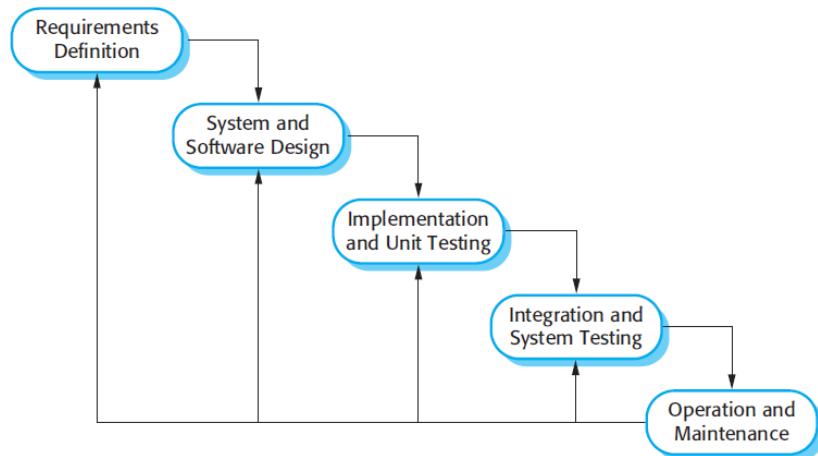
- Some methodologies are sometimes known as **software development life cycle (SDLC)** methodologies, though this term could also be used more generally to refer to any methodology.

SDLC models:

- Waterfall model
- Iterative model
- Spiral model
- V-shaped model
- Agile model

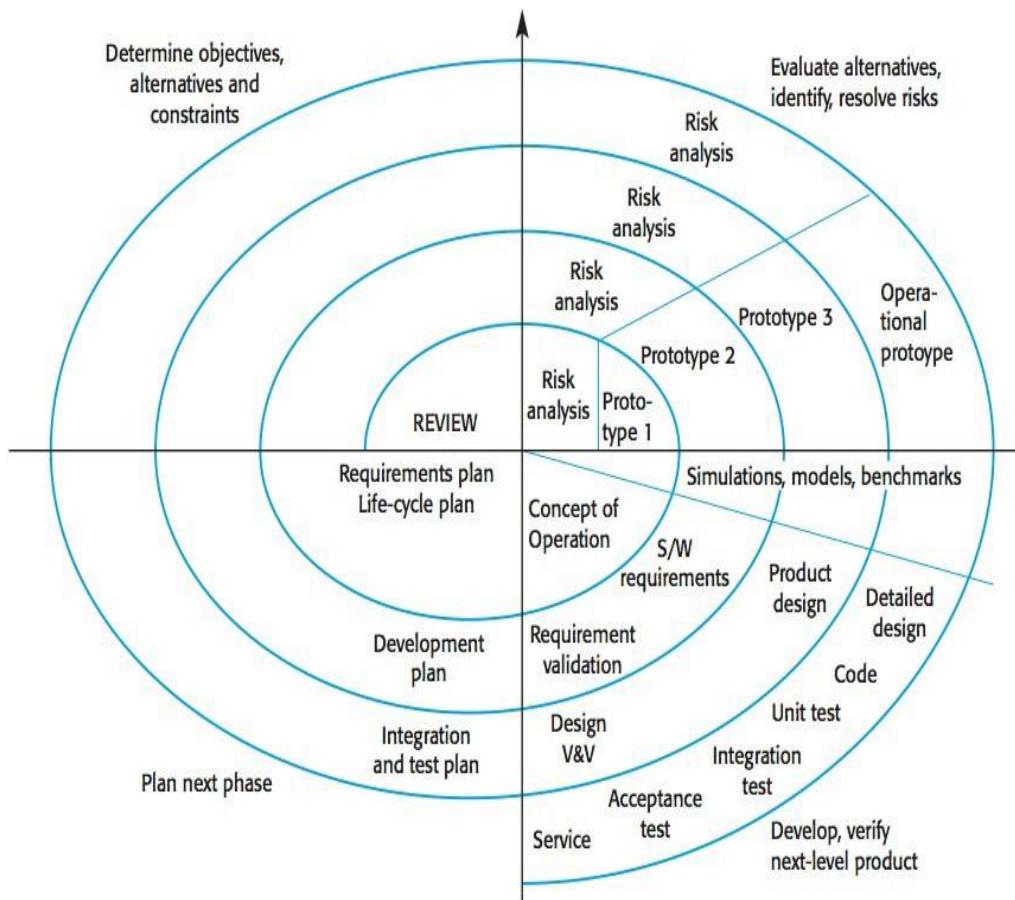
Waterfall model

- The waterfall model is a sequential approach, where each fundamental activity of a process represented as a separate phase, arranged in linear order.



Spiral model

- The spiral model is a risk-driven where the process is represented as spiral rather than a sequence of activities.
- It was designed to include the best features from the waterfall and prototyping models, and introduces a new component; risk-assessment.



How to perform Manual Testing

- Read and understand the software project documentation/guides. Also, study the Application Under Test (AUT) if available.
- Draft Test cases that cover all the requirements mentioned in the documentation.
- Review and baseline the test cases with Team Lead, Client (as applicable)
- Execute the test cases on the AUT
- Report bugs.
- Once bugs are fixed, again execute the failing test cases to verify they pass.

Myths of Manual Testing

- **Myth: Anyone can do manual testing**
 - **Fact:** Testing requires many skill sets

- **Myth: Testing ensures 100% Defect free product**
 - **Fact:** Testing attempts to find as many defects as possible. Identifying all possible defects is impossible.
- **Myth: Automated testing is more powerful than manual testing**
 - **Fact:** 100% test automation cannot be done. Manual Testing is also essential.
- **Myth: Testing is easy**
 - **Fact:** Testing can be extremely challenging. Testing an application for possible use cases with minimum test cases requires high analytical skills

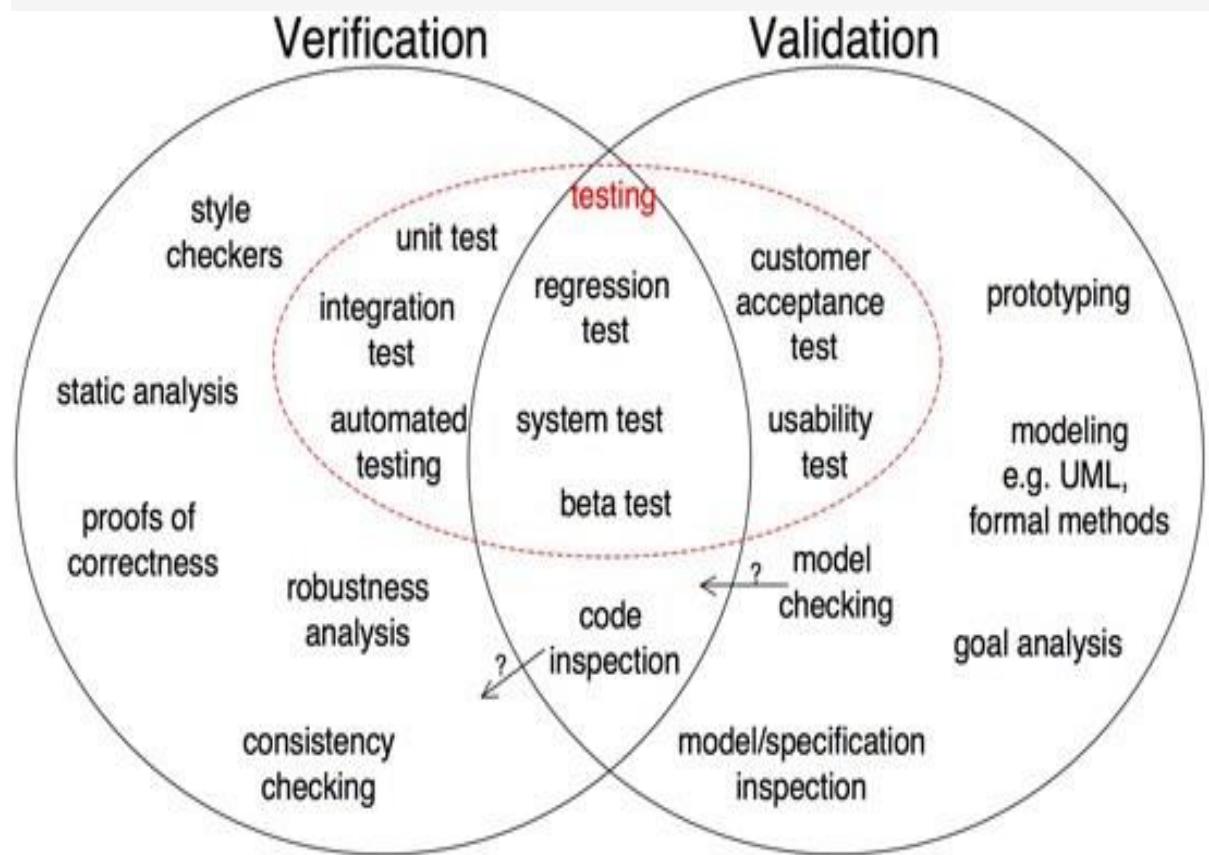
Role of testing in SDLC

- There are major **Importance of testing** in the part of SDLC and it is better to introduce testing in the early stage of SDLC phases so it help to identify the defects in the early stage & try to avoid the bugs finding & get resolve in the last critical stage.
- In the today's competitive market only the quality product stays longtime firmly, so to make sure the produce the good quality product the testing of application is key factor in SDLC.
- As it not possible makes it software application is defect free but testing will be necessary.
- Most important thing of testing is the development environment is different than the Testing environment and the testing done on testing environment is similar to the Production environment.
- In the software development life cycle(SDLC) the Testing is plays an importance role, which helps to improve the quality, reliability & performance of the system with all check what all functions software supposed to do & also check that Software is not doing what he not supposed to do.

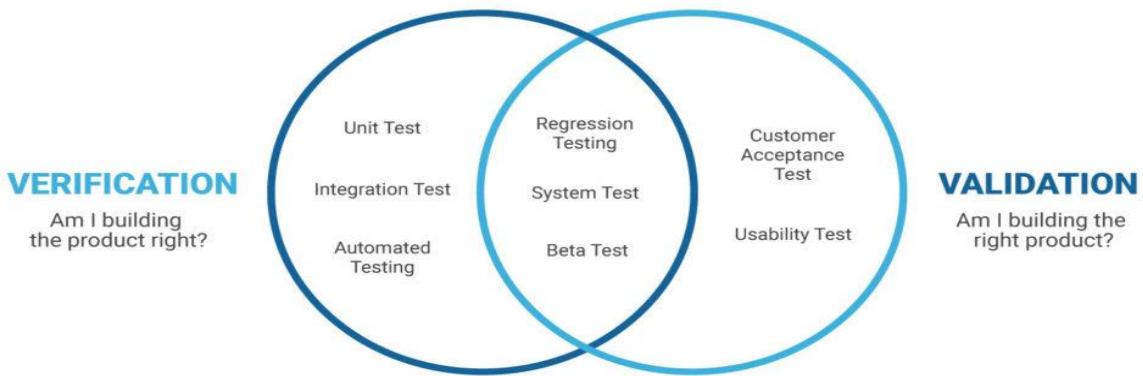
Verification and validation

Criteria	Verification	Validation
<i>Definition</i>	The process of evaluating work-products (not the actual final product) of a development phase to determine whether they meet the specified requirements for that phase.	The process of evaluating software during or at the end of the development process to determine whether it satisfies specified business requirements.

<i>Objective</i>	To ensure that the product is being built according to the requirements and design specifications. In other words, to ensure that work products meet their specified requirements.	To ensure that the product actually meets the user's needs and that the specifications were correct in the first place. In other words, to demonstrate that the product fulfills its intended use when placed in its intended environment.
<i>Question</i>	Are we building the product <i>right</i> ?	Are we building the <i>right</i> product?
<i>Evaluation Items</i>	Plans, Requirement Specs, Design Specs, Code, Test Cases	The actual product/software.
<i>Activities</i>		
<ul style="list-style-type: none"> • Reviews • Walkthroughs • Inspections • Testing 		



- Following items are evaluated during **Verification**: Plans, Requirement Specifications, Design Specifications, Code, Test Cases etc,
- Following item is evaluated during **Validation**: Actual product or Software under test.



Test planning

- A **TEST PLAN** is a document describing software testing scope and activities. It is the basis for formally testing any software/product in a project.
- **master test plan**: A test plan that typically addresses multiple test levels.
- **phase test plan**: A test plan that typically addresses one test phase.

Test Plan Types

- **Master Test Plan**: A single high-level test plan for a project/product that unifies all other test plans.
- **Testing Level Specific Test Plans**: Plans for each level of testing.
 - Unit Test Plan
 - Integration Test Plan
 - System Test Plan
 - Acceptance Test Plan
- **Testing Type Specific Test Plans**: Plans for major types of testing like Performance Test Plan and Security Test Plan.

Test preparation

Test Plan Guidelines

- Make the plan concise. Avoid redundancy and superfluousness.
- Be specific.
- Make use of lists and tables wherever possible. Avoid lengthy paragraphs.
- Have the test plan reviewed a number of times prior to baselining it or sending it for approval.
- Update the plan as and when necessary.

Test Plan Template

- *The format and content of a software test plan vary depending on the processes, standards, and test management tools being implemented. Nevertheless, the following format, which is based on IEEE standard for software test documentation, provides a summary of what a test plan can/should contain.*
- **Test Plan Identifier**
- **Introduction**
- **References**
- **Test Items**
- **Features to be Tested**
- **Features Not to Be Tested**
- **Approach**
- **Item Pass/Fail Criteria**
- **Suspension Criteria and Resumption Requirements:**
- **Test Deliverables:**
- **Test Environment:**
- **Estimate**
- **Schedule**
- **Staffing and Training Needs:**
- **Responsibilities:**

- **Risks**
- **Assumptions and Dependencies:**
- **Approvals**

Execution and reporting

		SUMMARY					
Name:	XYZ Systems	Total Test Cases		3			
Project ID:	W10978	Executed		2			
Application ID/Name:	XYZ Accounting	Pass		1			
From Report Date:	11-Apr-15	Fail		1			
Report Date:	17-Apr-15	Not executed		1			
Complete By (Milestone):	08-May-15						
Manager:	Ram Ray						
QA manager:	Shyam Das						
FUNCTIONAL TESTING							
Test Case ID	Description	Pass/Fail/Not Executed	Test Date	Responsible Developer	Responsible Tester	Comment	Additional Comment (other than QA team)
01	Valid Login	Pass	13-Apr-2015	Developer 1	Tester 1	Login successful	
02	Login Error on invalid Login	Fail	13-Apr-2015	Developer 1	Tester 2	Incorrect error message on failure	
03	Forget Password	Not Executed	13-Apr-2015	Developer 3	Tester 2	NA	

Automation testing

- Automation Testing means using an automation tool to execute your test case suite.
- This can be achieved by writing test scripts or using any automation testing tool.
- Test automation is used to automate repetitive tasks and other testing tasks which are difficult to perform manually.

When should we automate?

- Scripting of automation test cases should be started when the application is stable and frequent changes in the application (or component of the application to be automated) are not anticipated.



Advantages of automation testing

- Automation testing **reduces the overall test execution time**.
- It **reduces the cost and resource requirement** of the project, as the script created once can be made to run any number of times, as long as there is no change in the application.
- Helps in **working with large set of input** which won't be feasible with manual testing.
- Helps in creating **Continuous Integration** environment where in, after each code push, automatic test suite execution takes place with the new build.

Disadvantages of automation testing

- Skills and expertise of an automation tool and\or programming language is required to create robust scripts.
- Success of automation testing heavily depends on the test cases chosen for automation. Incorrectly chosen tests lead to wastage of resources and time invested in automation.
- For applications with frequent changes, parallel updation is required in the test scripts.
- Inefficiently written test scripts e.g. scripts using incorrect or limited validation can lead to false positive test results

Automation is suited to following types of testing with specific intentions:

- **1) Functional Testing** – on operations which perform as per the expectations.
- **2) Regression Testing** – on the behavior of the system which has not been changed.
- **3) Exception or Negative Testing** – thereby forcing error conditions in the system.
- **4) Stress Testing** – to determine the absolute capacities of the application and operational infrastructure.
- **5) Performance Testing** – to provide assurance that the performance of the system will be adequate for both batch runs and online transactions in relation to business projections and requirements.
- **6) Load Testing** – to determine the points at which the capacity and performance of the system become degraded to the situation that hardware or software upgrades would be required.

Chapter-16

Hardware Lab Kit (HLK)

What is Testing????

- **Testing** is a process, to evaluate the functionality of a software application with an intent to find whether the developed software met the specified requirements or not and to identify the defects to ensure that the product is defect free in order to produce the quality product.



verification and validation

What is Verification?

- *The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.*
- **Methods of Verification : Static Testing**
- *Walkthrough*
- *Inspection*
- *Review*

What is a Benchmark?

- Bench Mark means a set standard that helps to determine the quality of the product or service.
-

What is Benchmark Testing?

- It is a process used to compare the performance of software or hardware system also known as SUT (System Under Test).

* Benchmarking is not easy and often involves several iterative rounds in order to arrive at predictable, useful conclusions. Interpretation of benchmarking data is also extraordinarily difficult

Windows logo kit

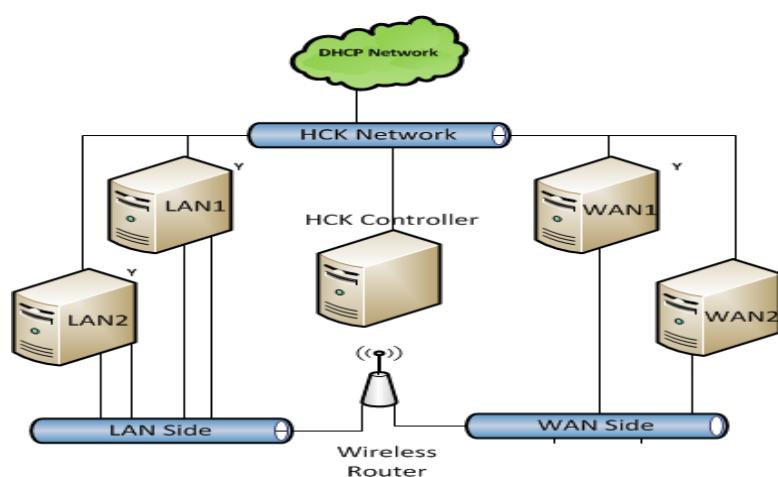
- The Windows Logo Kit (WLK) contains everything needed to certify devices for Windows Server 2003 and Windows Server 2008 — Driver Test Manager (DTM), the required tests, and the documentation to help set up DTM and execute the tests.

Supported Operating System

- Windows Server 2008
- Windows Server 2008 for the Kit Controller, Windows Server 2003 and Windows Server 2008 for Test Clients

Hardware certification kit

- The Windows Hardware Certification Kit enables developers, ISVs, IHVs, and OEMs to certify their hardware devices, systems, and filter drivers for the latest Windows operating systems. It contains all the tools and documentation that you need to certify your hardware and filter drivers for these operating systems
- The Windows Hardware Certification Kit replaces the Windows Logo Kit and is a part of the Windows Certification Program.
- **Platforms**
 - Windows 8
 - Windows Server 2012
 - Windows 7
 - Windows Server 2008 R2



HCK best practice

- These steps outline the certification test process.
 1. Set up test environment.
 2. Create a project.
 3. Create one or more machine pools.
 4. Select features to validate.
 5. Select and run tests against those features.
 6. View test results.
 7. Submit your package.

Windows ADK

- The Windows Assessment and Deployment Kit (Windows ADK) has the tools you need to customize Windows images for large-scale deployment, and to test the quality and performance of your system, its added components, and the applications running on it.
- **The Windows ADK includes**
 - The Windows Assessment Toolkit and the Windows Performance Toolkit to assess the quality and performance of systems or components.
 - Deployment tools such as WinPE, Sysprep, and other tools that you can use to customize and deploy Windows 10 images.

Windows Lab Kit

- The Windows Hardware Lab Kit (Windows HLK) is a test framework used to test hardware devices for Windows 10. To qualify for the Windows Hardware Compatibility Program, your product must pass certain tests using the Windows HLK.
- It provides automated scheduling and execution of the driver tests that hardware vendors are required to pass in order to qualify to use the Microsoft Designed for Windows Logo. It also enables users to automate driver tests they have created themselves

New for 1809:

- Certify using the Virtual HLK (VHLK)! The VHLK is the entire Hardware Lab Kit pre-installed and pre-configured on a VHDX, ready to boot as a virtual machine. Use the VHLK to save setup time, quickly stand up a controller, and run Windows Hardware Certification from a virtual machine.

Steps to install HLK

1. Install HLK controller

Warning

- The Windows Hardware Lab Kit should only be installed on machines that are dedicated solely for testing purposes. Do not install any HLK component on a machine that is outside of a dedicated testing environment.
- you install Windows HLK software on the designated test server. The setup program installs the Windows HLK Controller and Studio, in addition to other resources

2. Setup HLK proxy system

- Before proceeding, be sure that the credentials of the user on the proxy system match those of an administrator on your HLK controller.
- Local administrator account on proxy: <ProxyMachineName>\TestAdmin
- Local administrator account on controller: <HLKControllerMachineName>\TestAdmin
- Both with the same password

3. Create a machine pool

- A *machine pool* is a logical grouping of one or more test systems. Before you can work with a test system, you must move it to a working machine pool.
- Every project needs a machine pool. A machine pool can be used for multiple projects. A project can reference multiple machine pools.

4. Create a project

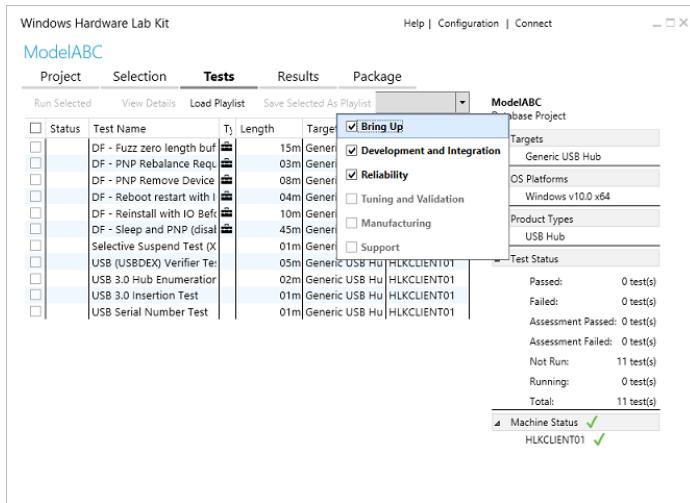
- After you setup your test server and clients, you're ready to create a project. A project defines what you want to test. The most common project corresponds to the device you want to submit for certification.
- To create a project
 - In Windows HLK Studio, choose the Project tab, and then choose Create project.
 - Replace the default project name with the name of your project, and then press Enter.

5. Select target to test

- Windows HLK Studio detects all features that a device implements. An individually testable feature is called a *target*. A device may contain multiple targets, represented by one or more hardware IDs.
- **On the Selection tab, you can filter what you want to test by using the following views:**
 - **Systems**
 - **devices and printers**
 - **device manager**
 - **software device**

6. Select and run tests

- The **Tests** tab displays all of the tests that are associated with the features found on your device. In the HLK, test categories replace the level classifications previously used with the HCK.



7: View test results and log files

The Results tab displays detailed information about each test. As each test completes, the status column updates with the result--pass or fail.

Status	Test Name	Target	Machine(s)
X	AutoMemoryBenchmark 02/16/2015 09:53:26 (Machine(s)) ↳ ✓ HLK Config Library Tasks I ↳ ✓ Copy PerfX2 locally ↳ ✓ Copy PerfX2 OS dependent ↳ ✓ Copy D3D Support (OS ar... ↳ ✗ Copy Perl ↳ ✗ Register TestX.man ↳ ✗ MemoryBenchmark Setup ↳ ✗ Set optimal display test et... ↳ ✗ Get Perf_DX ↳ ✗ Add Perf_DX ↳ ✗ Launch PerfX2 Tests ✗ verifier /reset ✓ DEL Perl.exe ✓ DEL Perl512.dll ✓ DEL Pri Perf_Memory_EX.c ✓ DEL Pri Perf_DX.dll	HLKCLIENT01	HLKCLIENT01
✓	02/16/2015 09:45:41 (Machine(s)) ↳ ✓ HL K Config Library Tasks I		

Troubleshooting Windows HLK environment

Cannot get a client computer out of Debug state

- **Problem:** The Client does not change from the Debug state to the **Ready** state, even after changing the state to **Reset** in Windows HLK Manager.
 - **Solution:**
 - Reinstall the Client using the steps in Remove and reinstall the Windows HLK Client.

Cannot select tests after previous test fails

- After a test fails, the checkboxes for the list of tests are dimmed and you cannot select additional tests.
- If you select a target but cannot select any of the tests, the computer is not in a ready state.
 - **Solution:**
 - Go to the **Configuration** menu and put the computer in a ready state.

Client computer goes to sleep during a test

- An always on, always connected (AOAC) computer that is not an ARM machine can go to sleep during a test. To prevent this, run the following command from an elevated command prompt on each Windows HLK Client computer:

```
powercfg /setacvalueindex scheme_current sub_video videoidle 0 & Powercfg /setdcvalueindex scheme_current sub_video videoidle 0 & Powercfg /s scheme_current
```

Client computer is locked by an LLU account

- If a client computer is locked by a LLU account or is unresponsive, you must log on by using the correct credentials for the LLU account. The two main LLU accounts on a client computer are:

Client computers cannot communicate with the Windows HLK Controller

- If a client computer cannot communicate with the Windows HLK Controller, review the following problems and solutions:

Troubleshooting Windows HLK Test Failures

1. Install latest updates and filters
2. Review the Status icons
3. Make sure that the test was run correctly
4. Check for configuration changes
5. Identify the failure category

Windows HLK compatibility program

- To qualify for the Windows Hardware Compatibility Program, your product must pass certain tests using the Windows HLK. For guidance on which kit to use for compatibility certification.
- Windows Hardware Compatibility playlist so you can run the Windows HLK tests that ensure your hardware meets the requirements for compatibility with Windows 10.
- The Windows Hardware Compatibility Program leverages the tests in the Windows Hardware Lab Kit, a test framework used to test hardware devices for Windows 10.

HLK compatibility playlist

- The Windows Hardware Compatibility Program uses an official playlist to determine which devices meet the requirements for compatibility with Windows 10.
- Compatibility playlist is a XML files on different compatibility test that need to run in-order to ensure the compatibility of hardware with operating system.

Note: always download the latest compatibility list from the website.

HLK Filters

- When there is a problem in either a Windows HLK test or in the operating system itself that causes certification tests to fail incorrectly, we create an errata that allows partners to bypass the problematic test.
- Most errata use filters to automatically filter the failure from the submission results. Filters are applied within Windows HLK Studio.

- Note:
 - Hlk filters are xml files provided by Microsoft.
 - Download the latest one in order to get the better result.

Chapter-17

Python Basics

What is python...?

- Python is a powerful high-level, object-oriented programming language created by Guido van Rossum.
- Features of python
 - 1. Simple Elegant Syntax

```
a = 2
```

```
b = 3
```

```
sum = a + b
```

```
print(sum)
```

- 2. Not overly strict

You don't need to define the type of a variable in Python. Also, it's not necessary to add

semicolon at the end of the statement. It enforces you to follow proper indentation.

- 3. Interpreted language

Python is an interpreted language i.e. interpreter executes the code line by line at a time. This makes debugging easy and thus suitable for beginners.

- 4. Cross-platform Language

Python can run equally on different platforms such as Windows, Linux, Unix and Macintosh etc. So, we can say that Python is a portable language.

- 5. Extensible

It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our python code.

Applications of python

1. **GUI-Based Desktop Applications:**

- Image Processing and Graphic Design Applications.
- Scientific and Computational Applications.
- Games.

2. Web Frameworks and Web Applications.

3. Operating system

Python is often an integral part of Linux distributions. For instance, Ubuntu's Ubiquity Installer, and Fedora's and Red Hat Enterprise Linux's Anaconda Installer are written in Python.

4. Language Development

Python's design and module architecture has influenced development of numerous languages. Boo language uses an object model, syntax and indentation, similar to Python. Further, syntax of languages like Apple's Swift, CoffeeScript, Cobra, and OCaml all share similarity with Python.

5. Artificial Intelligence

Pycharm

PyCharm is a dedicated Python and Django IDE providing a wide range of essential tools for Python developers, tightly integrated together to create a convenient environment for productive Python development and Web development.

PyCharm is available in three editions: **Professional**, **Community**, and **Educational (Edu)**. The **Community** and **Edu** editions are open-source projects and they are free, but they have less features. PyCharm Edu provides courses and helps you learn programming with Python. The **Professional** edition is commercial, and provides an outstanding set of tools and features.



Python is Interpreted Language

A high-level language is one which is understandable by us humans. It contains words and phrases from the English (or other) language. But a computer does not understand high-level language. It only understands program written in 0's and 1's in binary, called the machine code. It contains words and phrases from the English (or other) language. A program written in high-level language is called a source code. We need to convert the source code into machine code and this is accomplished by compilers and interpreters. Hence, a compiler or an interpreter is a

program that converts program written in high-level language into machine code understood by the computer.

COMPILER

A program that converts instructions into a machine-code or lower-level form so that they can be read and executed by a computer.

INTERPRETER

The interpreter transforms the high-level program into an intermediate language that it then executes, or it could parse the high-level source code and then performs the commands directly, which is done line by line or statement by statement.

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyse the source code but the overall execution time is slower.	It takes large amount of time to analyse the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.

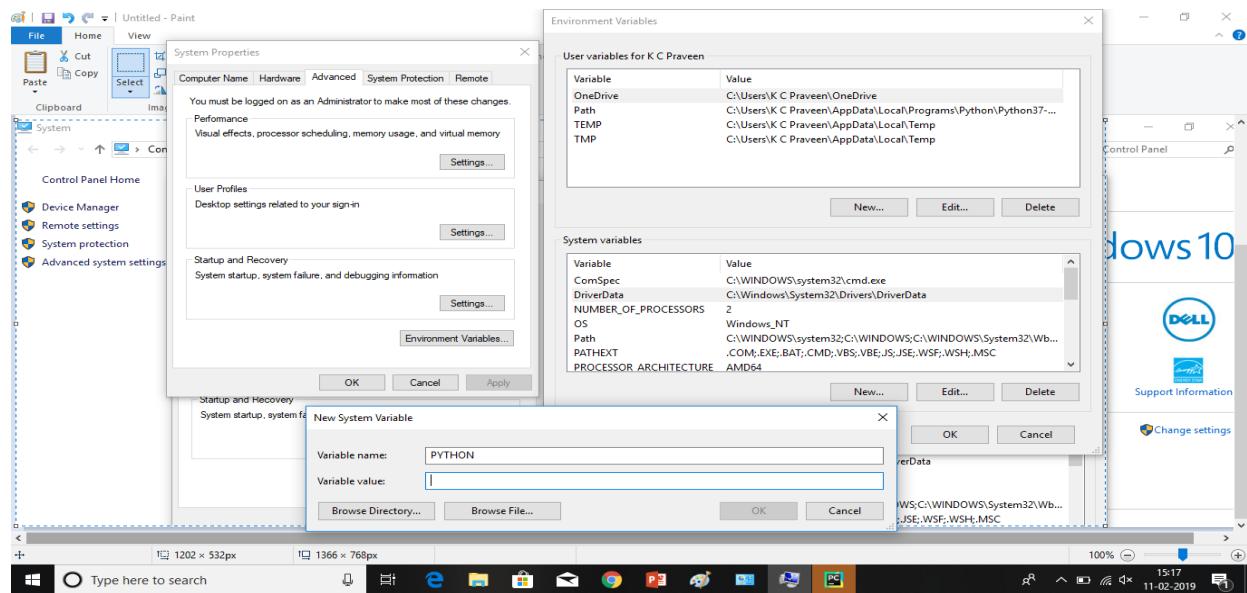
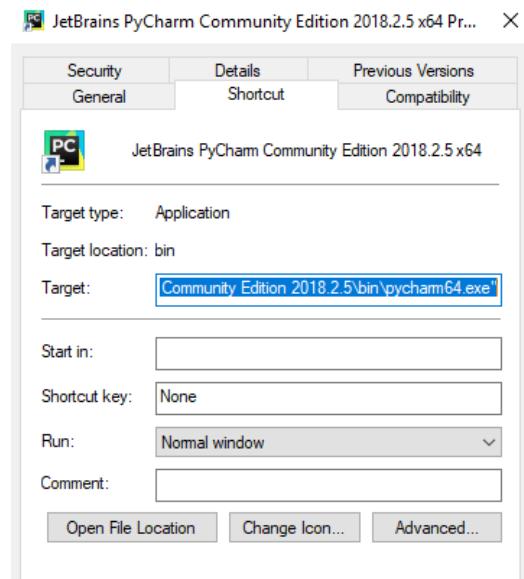
Programming language like Python, Ruby use interpreters.

Programming language like C, C++ use compilers.

Accessing Pycharm from command prompt

Add pycharm to environment variables

When you have located the shortcut file, right-click it, click Properties, select the Shortcut tab, and look in the field called Target: to see the location of the Pycharm executable.



What is PIP

One of the best tools to install and manage Python packages is called Pip. Pip has earned its fame by the number of applications using this tool. Used for its capabilities in handling binary packages over the easy installed packaged manager, Pip enables 3rd party package installations. Though the newest versions of Python come with pip installed as a default.

To check whether pip is installed or not , type `pip -V`

Once you've confirmed that Python is correctly installed, you can proceed with installing Pip.

1. Download [get-pip.py](#) to a folder on your computer.
2. Open a command prompt and navigate to the folder containing get-pip.py.
3. Run the following command:
`python get-pip.py`
4. Pip is now installed!

You can verify that Pip was installed correctly by opening a command prompt and entering the following command:

```
pip -V
```

You should see output similar to the following:

```
pip 18.0 from
c:\users\administrator\appdata\local\programs\python\python37\lib\site-
```

Powershell

Windows PowerShell is object-oriented automation engine and scripting language. It is designed mainly for the system administrators. It helps IT, professionals, to control & automate the administration of the Window OS and other applications.

Power shell Commands

- `Get-CimInstance Win32_OperatingSystem | Select-Object Caption, InstallDate, ServicePackMajorVersion, OSArchitecture, BootDevice, BuildNumber, CSName | FL`
- `mspaint.exe (open a application)`
- `Get-ComputerInfo`
- `Get-ComputerInfo -Property "os*"`

```
PS C:\WINDOWS\system32> mspaint.exe

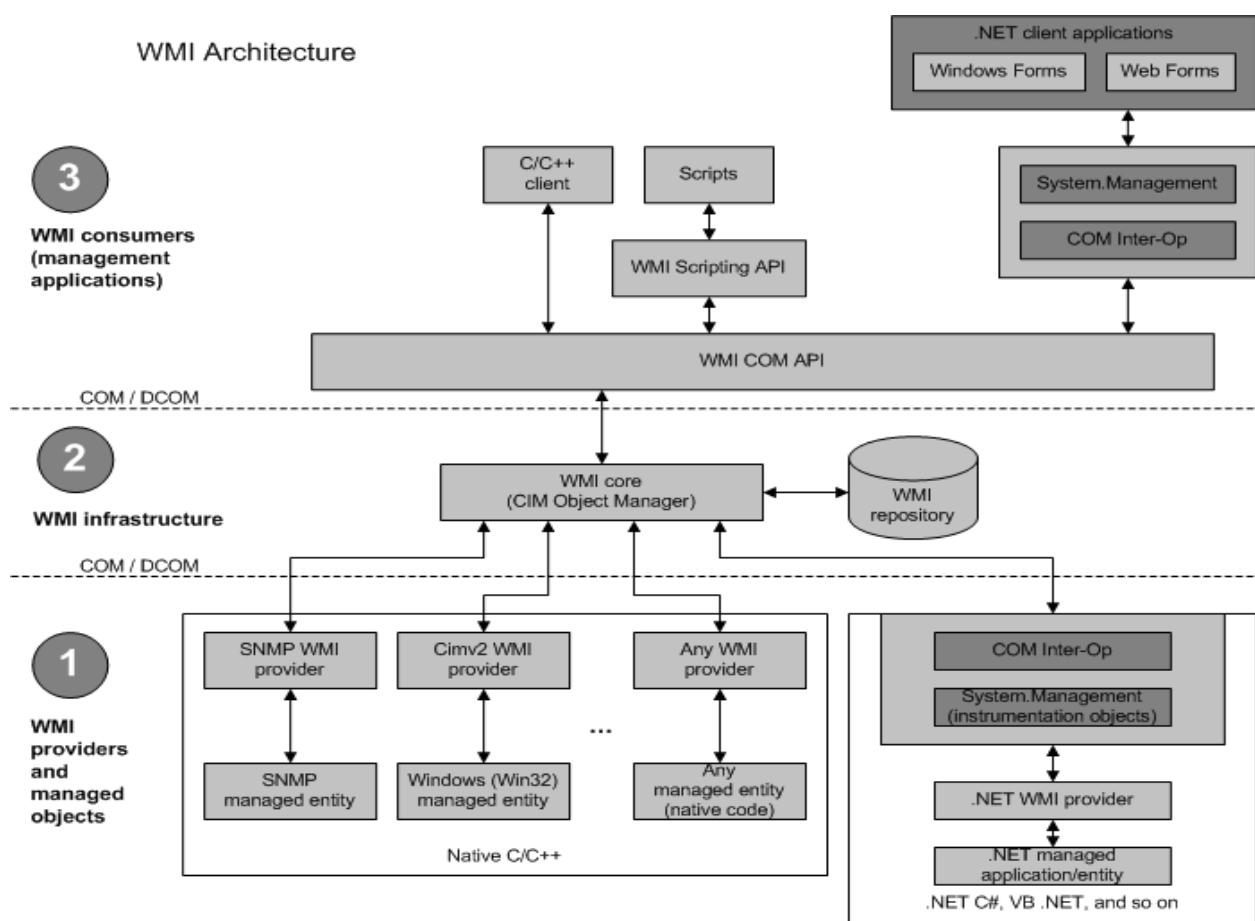
PS C:\WINDOWS\system32> Get-ComputerInfo

WindowsBuildLabEx : 17134.1.amd64fre.rs4_release.180410-1804
WindowsCurrentVersion : 6.3
WindowsEditionId : CoresingleLanguage
WindowsInstallationType : client
WindowsInstallDateFromRegistry : 31-10-2018 09:22:22
WindowsProductId : 00327-35000-00000-AAOEM
WindowsProductName : Windows 10 Home Single Language
WindowsRegisteredOrganization :
WindowsRegisteredOwner :
WindowsSystemRoot : C:\WINDOWS
WindowsVersion : 1803
BiosCharacteristics :
BiosBIOSVersion :
BiosBuildNumber :
BiosCaption :
BiosVendor :
```

What is WMI

- Windows Management Instrumentation (WMI) is the infrastructure for management data and operations on Windows-based operating systems.
- You can write WMI scripts or applications to automate administrative tasks on remote computers
- WMI can be used in all Windows-based applications, and is most useful in enterprise applications and administrative scripts.

WMI Architecture



Chapter-18

Introduction to JTAG

J-tag

- **JTAG** (named after the **Joint Test Action Group** which codified it) is an industry standard for verifying designs and testing printed circuit boards after manufacture.
- The interface connects to an on-chip Test Access Port (TAP) that implements a stateful protocol to access a set of test registers that present chip logic levels and device capabilities of various parts.
- Although JTAG's early applications targeted board level testing, here the JTAG standard was designed to assist with device, board, and system testing, diagnosis, and fault isolation

The connector pins are:

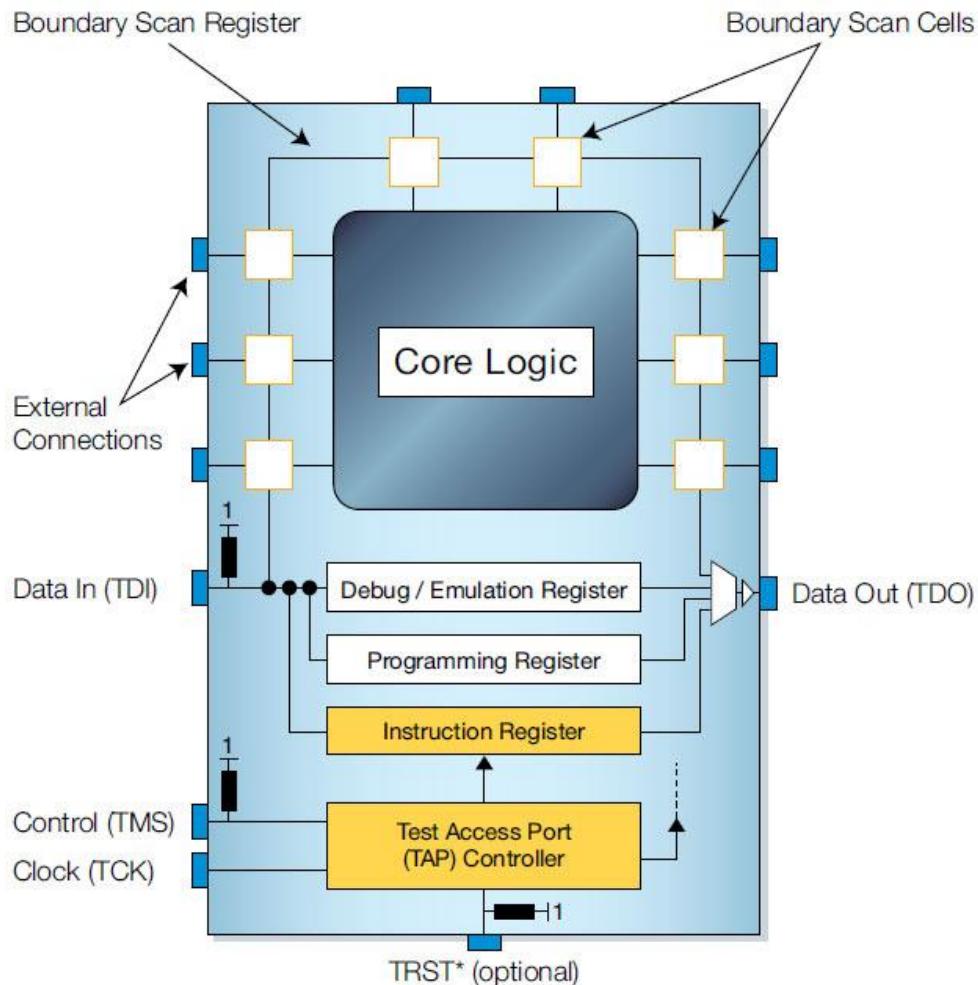
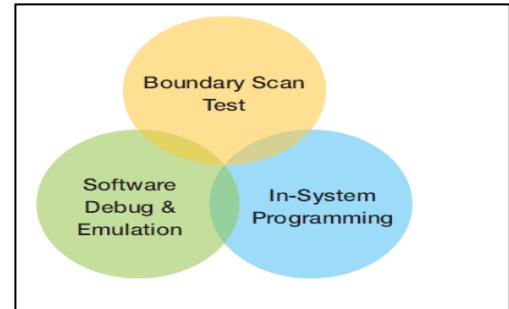
- **TDI** (Test Data In)
- **TDO** (Test Data Out)
- **TCK** (Test Clock)
- **TMS** (Test Mode Select)
- **TRST** (Test Reset) optional

VTref	1	●	● 2	NC
nTRST	3	●	● 4	GND
TDI	5	●	● 6	GND
TMS	7	●	● 8	GND
TCK	9	●	● 10	GND
RTCK	11	●	● 12	GND
TDO	13	●	● 14	GND*
RESET	15	●	● 16	GND*
DBGREQ	17	●	● 18	GND*
5V-Supply	19	●	● 20	GND*

J-tag boundary scan

J-tag Applications

1. Software debugging and emulation
2. Boundary scan
3. In system programming



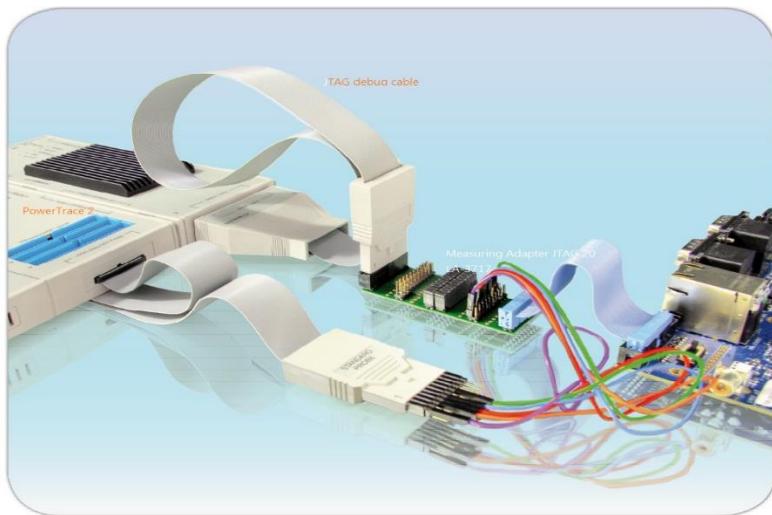
J-tag on ARM

ARM 20-PIN Interface

VCC	1	<input type="checkbox"/>	<input type="checkbox"/>	2	VCC (optional)
TRST	3	<input type="checkbox"/>	<input type="checkbox"/>	4	GND
TDI	5	<input type="checkbox"/>	<input type="checkbox"/>	6	GND
TMS	7	<input type="checkbox"/>	<input type="checkbox"/>	8	GND
TCLK	9	<input type="checkbox"/>	<input type="checkbox"/>	10	GND
RTCK	11	<input type="checkbox"/>	<input type="checkbox"/>	12	GND
TDO	13	<input type="checkbox"/>	<input type="checkbox"/>	14	GND
RESET	15	<input type="checkbox"/>	<input type="checkbox"/>	16	GND
N/C	17	<input type="checkbox"/>	<input type="checkbox"/>	18	GND
N/C	19	<input type="checkbox"/>	<input type="checkbox"/>	20	GND

FS2 10-PIN Interface

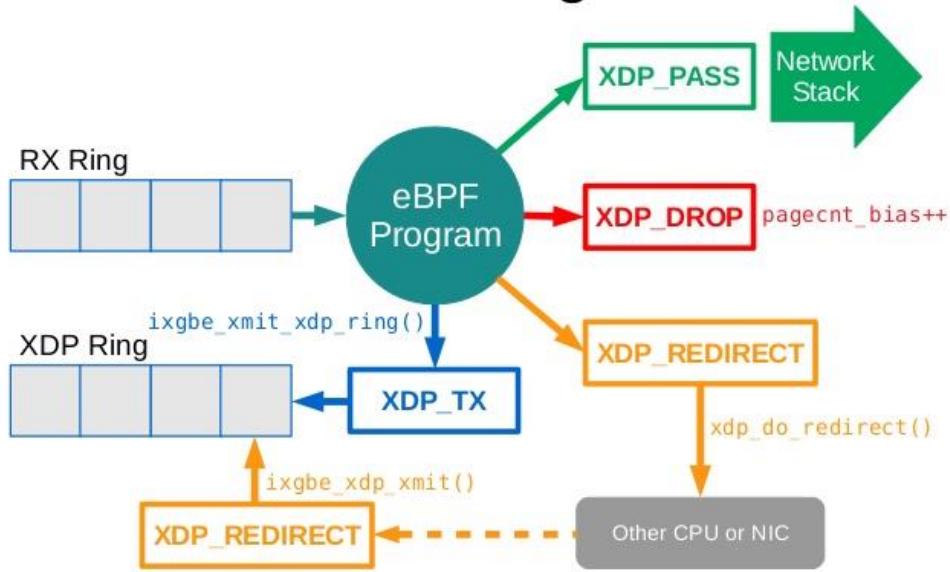
TCK	1	<input type="checkbox"/>	<input type="checkbox"/>	2	GND
TDO	3	<input type="checkbox"/>	<input type="checkbox"/>	4	VCC
TMS	5	<input type="checkbox"/>	<input type="checkbox"/>	6	VIO
TRIG	7	<input type="checkbox"/>	<input type="checkbox"/>	8	RST
TDI	9	<input type="checkbox"/>	<input type="checkbox"/>	10	GND

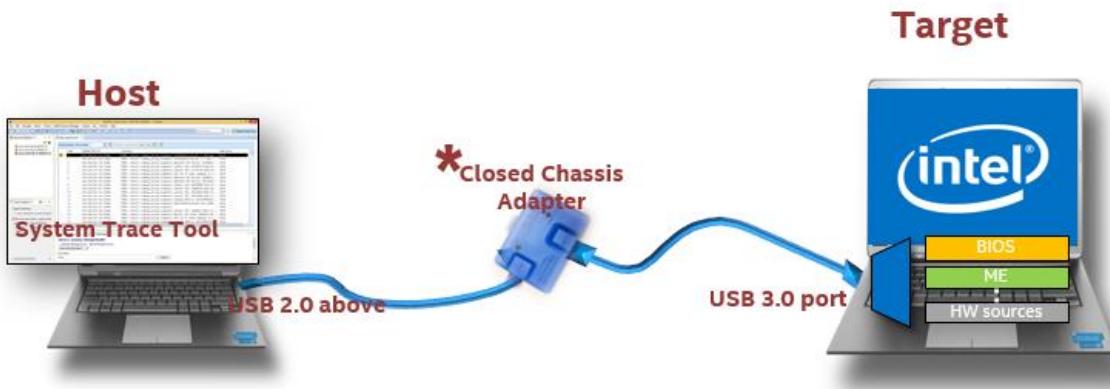


- J-link
- U-link
- Olimex

Jtag on intel

XDP in ixgbe





J-tag through DCI

- Intel latest processor support DCI interface for 3.0 debug usb cable.
- Intel system studio gives the software support for the debugging.



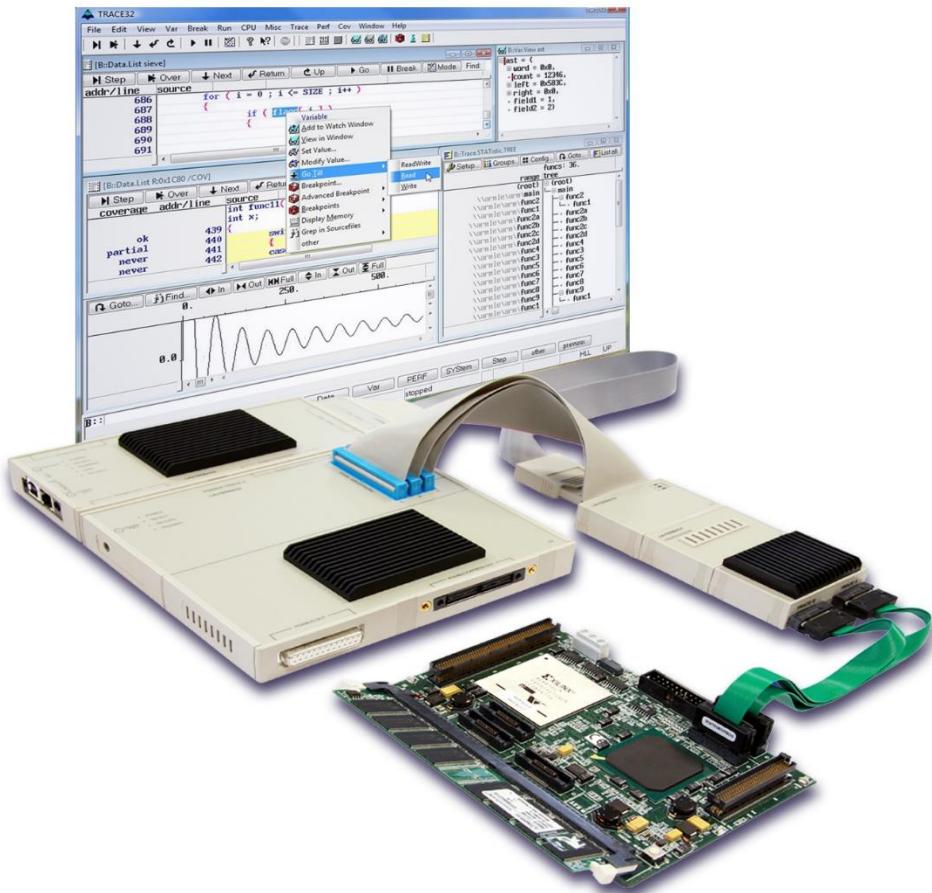
Lauterbach

- Many embedded developers would like to see a higher level of standardization of the on-chip debug and trace logic as well as a reduction in pin count without any performance loss.

- The **TRACE32** In-Circuit Emulators feature dual-port emulation memory which can be accessed by both the emulation CPU, as well as the system control CPU.



Lauterbach



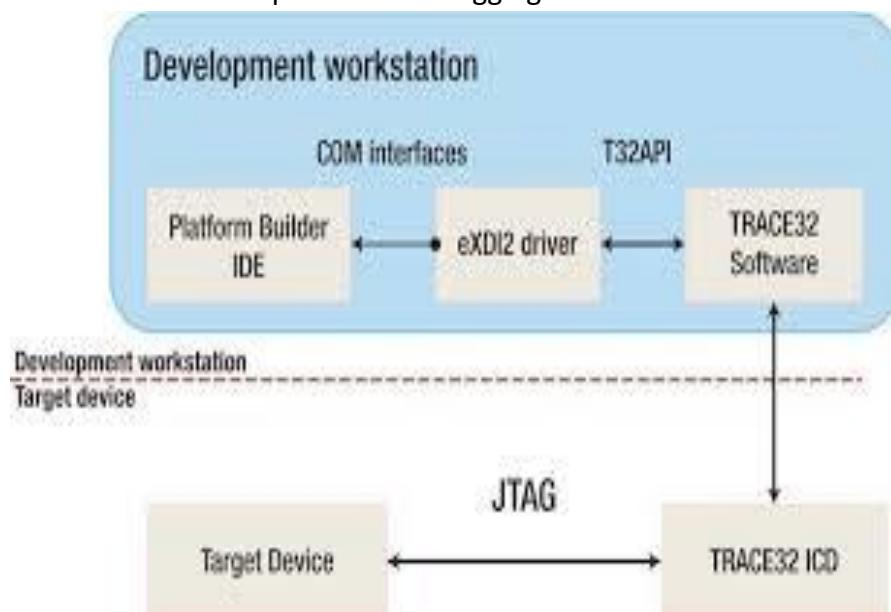
The on-chip debug system provides the following basic features:

- Read/write memory
- Read/write CPU register

- Single step and real time execution
- Hardware breakpoints and trigger features

TRACE32-ICD

- TRACE32-ICD are microprocessor development tools based on the debug and trace logic (BDM, JTAG, ETM, OCDS, NEXUS) integrated on the chip.
- The In-Circuit Debugger TRACE32-ICD uses these basic features of the on-chip debug system to provide a powerful debug tool that offers
 - Easy high-level and assembler debugging
 - Display of internal and external peripherals on a logical level
 - On-chip break and trigger support
 - RTOS awareness
 - Flash programming
 - Powerful script language
 - Multiprocessor debugging

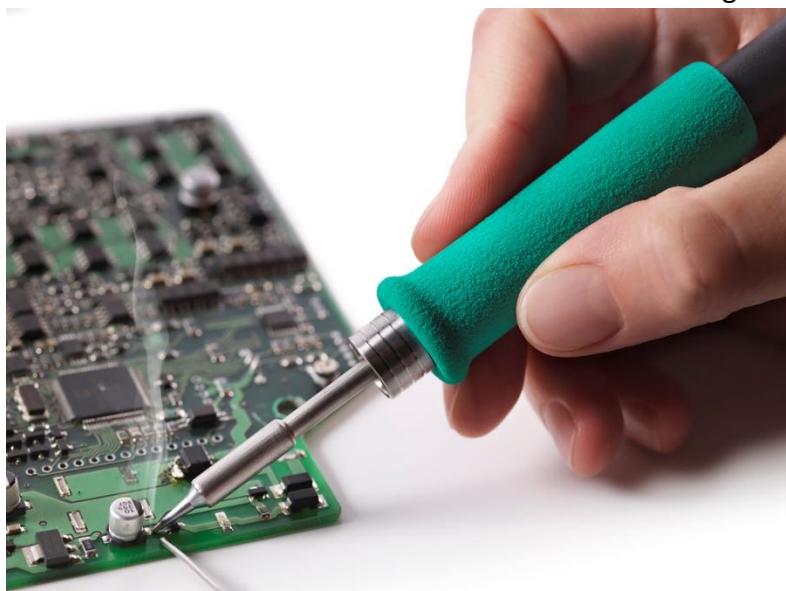


Chapter-19

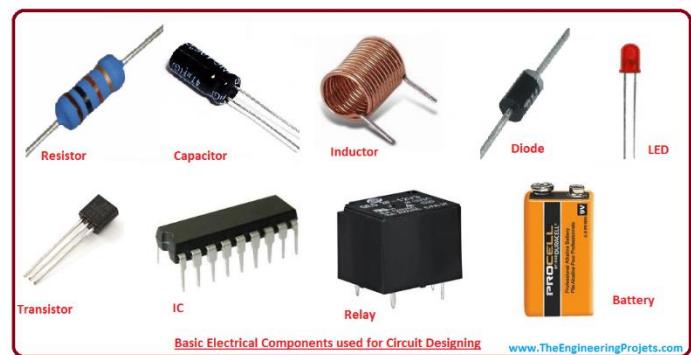
Soldering Basics

What is soldering?

- Formation of “metal to metal” joint using solder.
- The joint is made by alloy formation of base metal and solder.
- In soldering, there are four key elements: Iron, Solder, Flux, And Component; they all are important.
- Flux cored solder wire is used for hand soldering



Electronic Component



- Electronic components are basic electronic element or electronic parts usually packaged in a discrete form with two or more connecting leads or metallic pads.
- There are main three types of Electronic components –

1. Active Component

2. Passive Component

3. Electromechanica

Classification of Parts

Active Component: Those devices or components which required external source to their operation is called Active Components.

- For Example: Diode, Transistors, SCR etc...

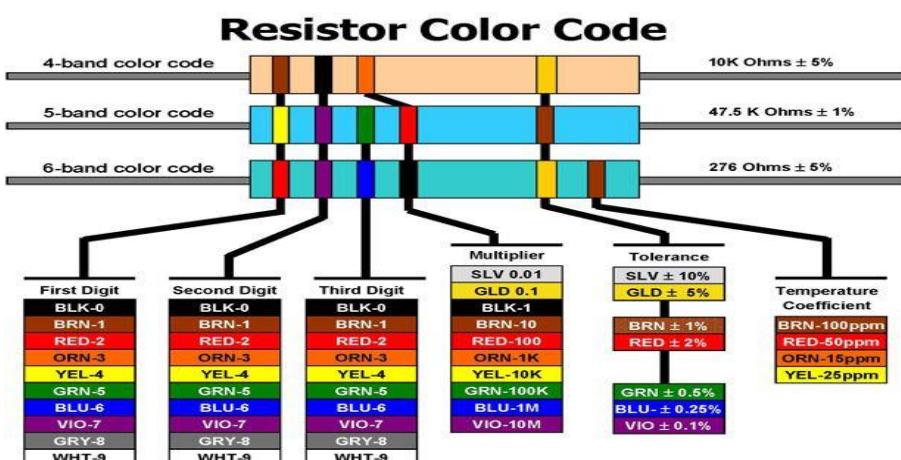
Passive Components: Those devices or components which do not required external source to their operation is called Passive Components.

- For Example: Resistor, Capacitor, Inductor etc...

Component Identification

Color Codes		4 Band Resistors	5 Band Resistors	6 Band Resistors
0	Black	EXAMPLE	EXAMPLE	EXAMPLE
1	Brown	$\pm 1\%$	$\pm 1\%$	$\pm 1\%$
2	Red	$\pm 2\%$	$\pm 2\%$	$\pm 2\%$
3	Orange	$\pm 5\%$	$\pm 5\%$	$\pm 5\%$
4	Yellow	$\pm 10\%$	$\pm 10\%$	$\pm 10\%$
5	Green			
6	Blue			
7	Purple			
8	Grey			
9	White			
$\pm 1\%$	Brown			
$\pm 2\%$	Red			
$\pm 5\%$	Gold			
$\pm 10\%$	Silver			

comparison	Metric code	Imperial code	comparison
0.1x0.1 mm	0402	01005	0.01x0.01 in (10x10 mils)
	0603	0201	
	1005	0402	
	1608	0603	
1x1mm	2012	0805	0.1x0.1 in (100x100 mils)
	2520	1008	
	3216	1206	
	3225	1210	
	4516	1806	
	4532	1812	
	5025	2010	
	6332	2512	



Flux Function

- ❑ During soldering chemical reaction takes place. The flux removes all the surface tarnish leaving clean metal underneath.
- ❑ As cleaning agents, **fluxes** facilitate **soldering**, brazing, and welding by removing oxidation from the metals to be joined



Solder Wire

- Solder is a fusible metal alloy used to create a permanent bond between metal and workpieces.
- You should be aware of following while choosing solder wire – Lead (Pb) Content Flux Content Wire Size



- Bevel Tip: It can be used for solder jobs that require preloading the iron with solder. The large flat surface can hold more solder than most other tips, and it's helpful when

soldering small-gauge wires together or dragging solder across surface-mount chips to solder multiple pins at once.

- Chisel Tip: With its broad tip, the chisel tip helps to evenly deliver heat to component leads and pads.
- This tip is great for soldering wires, through-hole components, large surface-mount components, and for desoldering as well.



Correct Posture when soldering

