

Getting correct results from imperfect Disks

2 RAID REDUNDANT ARRAY OF INEXPENSIVE DISKS-RAID 1,2

3 RAID REDUNDANT ARRAY OF INEXPENSIVE DISKS-RID 3,4,5

4 STABLE STORAGE DISK INTEGRITY

5 DISK INTEGRITY BAD SECTORS-SPARE SECTORS

6 TIMER CIRCUITS

Interrupts occur in the middle of an instruction, or the consequent context switch leaves too little time to process the interrupt.

7 PRECISE INTERRUPTS

8 INTERRUPTS, POLLING AND SOFT TIMERS

There are a large variety of Character Handling-terminals, intercomputer messages, printers, etc.

10 CHARACTER INPUTS TO TERMINAL DRIVER

11 UNIVERSAL ASYNCHRONOUS RECEIVER TRANSMITTER (UART)

Diverse Character oriented IO, need for buffers, and language to encompass diversity,

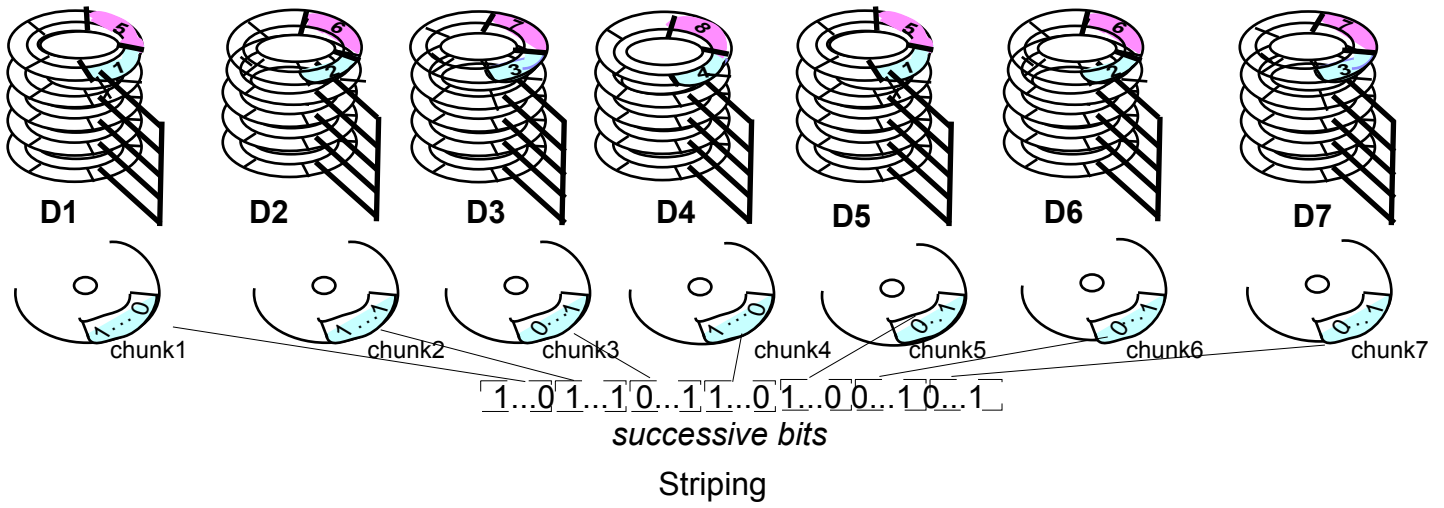
12 VIDEO MONITOR MODES

13 SIMPLE VIDEO MONITOR SCROLLING

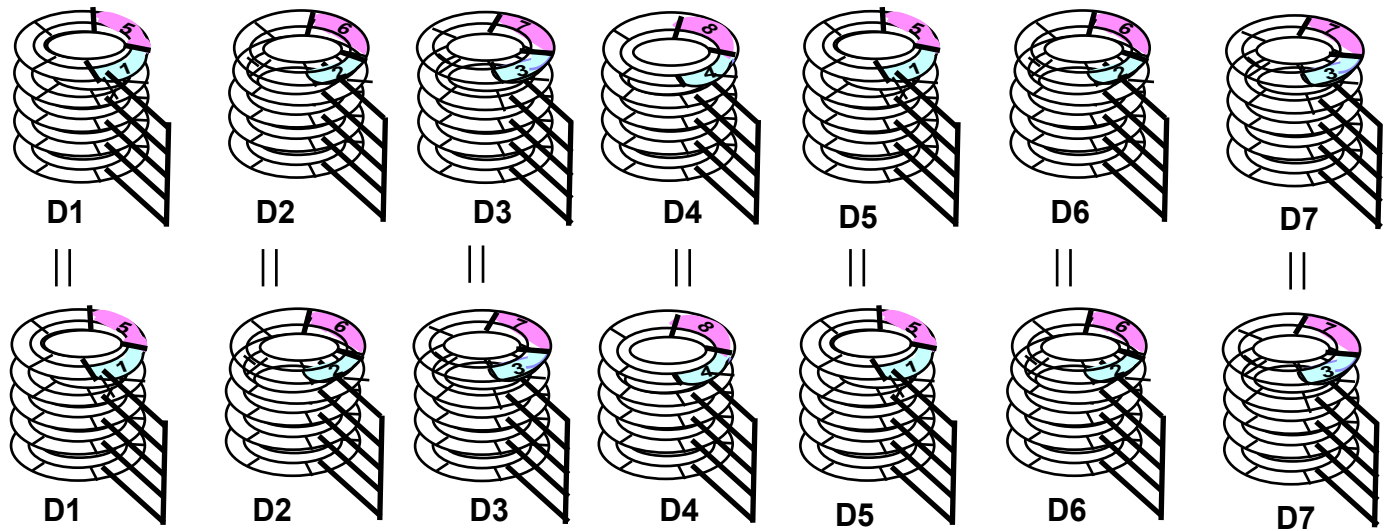
14 CD-ROM

15 POWER MANAGEMENT

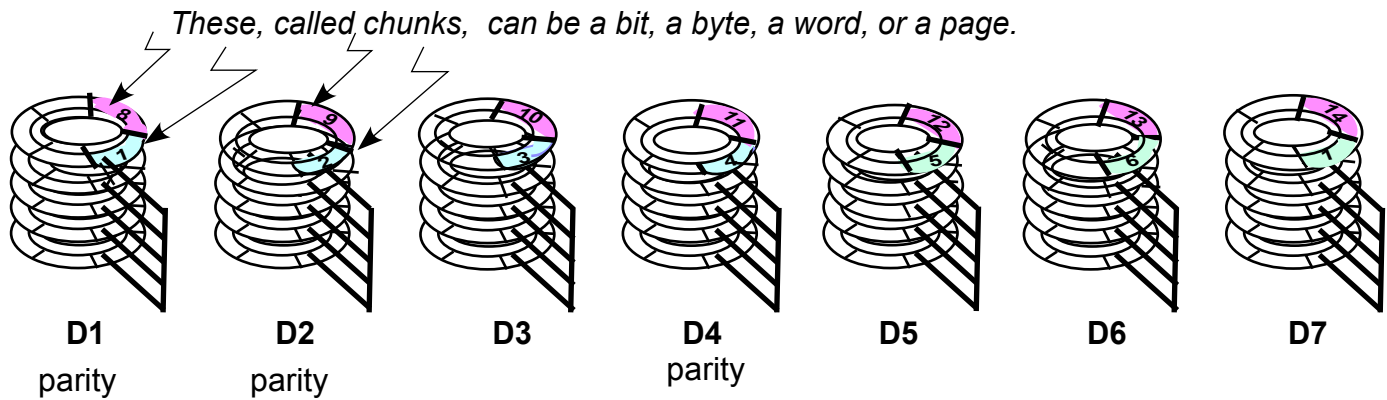
CONTENTS



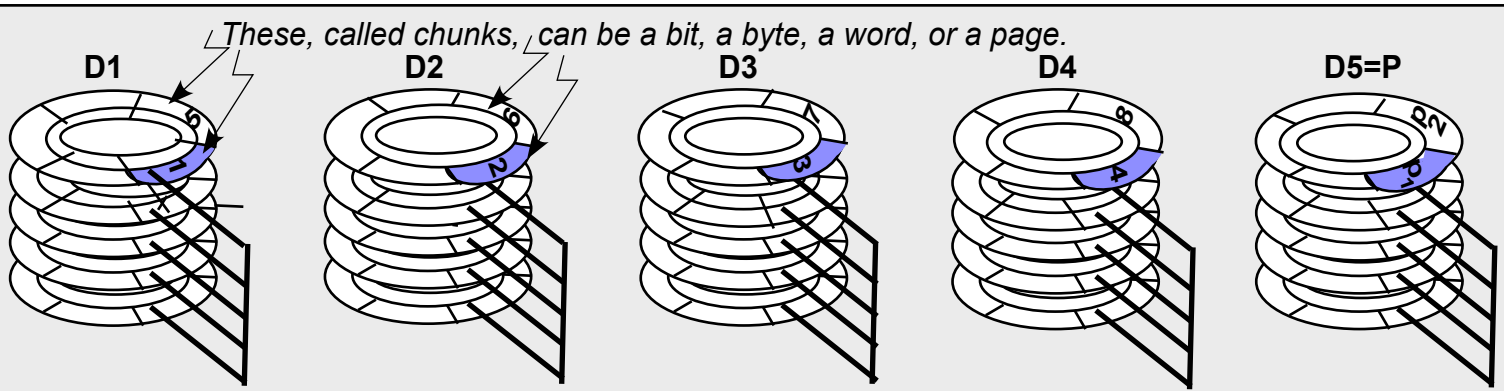
Parallel Operation



7 Info 7 Duplicates
RAID 1



HAMMING CODE
7 Bits 4 Info, 3 Check Bits Single Error Correcting
RAID 2



d_i are the corresponding bits on disks D_i , $i = 1$ to 5 . $d_5 = d_P$ is the parity bits. It is chosen so that:
 $d_1 \oplus d_2 \oplus d_3 \oplus d_4 \oplus d_5 = 0$ where \oplus is "exclusive or"

d_1 d_2 d_3 d_4 $d_5 = d_P$

0	1	1	0	0
1	0	1	1	1
0	0	0	1	1

example bit distribution

d_i	d_j	$d_i \oplus d_j$
0	0	0
0	1	1
1	0	1
1	1	0

exclusive or

$$d_i \oplus 0 = d_i$$

$$d_i \oplus 1 = \bar{d}_i$$

$$d_i \oplus d_i = 0$$

$$d_i \oplus d_j = d_j \oplus d_i$$

$$((d_i \oplus d_j) \oplus d_k) = (d_i \oplus (d_j \oplus d_k))$$

RECOVERY WHEN A SINGLE DISK FAILS

d_j , $j = 1$ to 5 are the striped bits on disks **D1** thru **D5**. If one disk fails, and we know which one, say **D3**, then we can determine the value of each bit because the unknown bit, ? must satisfy:

Of more formally: $d_1 \oplus d_2 \oplus ? \oplus d_4 \oplus d_5 = 0$

$d_5 = \text{parity bit}$

$$d_1 \oplus d_2 \oplus d_3 \oplus d_4 \oplus d_5 = 0$$

$$d_1 \oplus d_2 \oplus d_3 \oplus \underline{d_3} \oplus d_4 \oplus d_5 = 0 \oplus \underline{d_3}$$

$$d_1 \oplus d_2 \oplus d_4 \oplus d_5 = d_3$$

UPDATING PARITY DISK DEPENDS ON CHUNK SIZE

If the chunk is **small**, say a bit, then, when writing an entire word to disk, the chunk for each disk (assuming a word has no more bits than the number of disks) will be available at once. Therefore the parity bit for the last disk can be computed and written immediately.

If however each chunk is **large**, say a word, then when a word is read to disk the entire word will go to one disk. So computing the new parity that results seems to require reading all other non-parity disks. However one can do better.

$$d_1 \oplus d_2 \oplus d_{3_{new}} \oplus d_4 \oplus d_{5_{new}} = d_1 \oplus d_2 \oplus d_{3_{old}} \oplus d_4 \oplus d_{5_{old}} = 0$$

$$[d_1 \oplus d_2 \oplus d_4] \oplus d_1 \oplus d_2 \oplus d_{3_{new}} \oplus d_4 \oplus d_{5_{new}} = [d_1 \oplus d_2 \oplus d_4] \oplus d_1 \oplus d_2 \oplus d_{3_{old}} \oplus d_4 \oplus d_{5_{old}}$$

$$d_{3_{new}} \oplus d_{5_{new}} = d_{3_{old}} \oplus d_{5_{old}}$$

$$[d_{3_{old}} \oplus d_{3_{new}}] = [d_{5_{new}} \oplus d_{5_{old}}]$$

$$\Delta d_3 = \Delta d_5 \quad \Delta d_j = d_{j_{old}} \oplus d_{j_{new}} = d_{j_{new}} \oplus d_{j_{old}}$$

Note: This means that a bit in d_5 must change iff bit d_3 changed

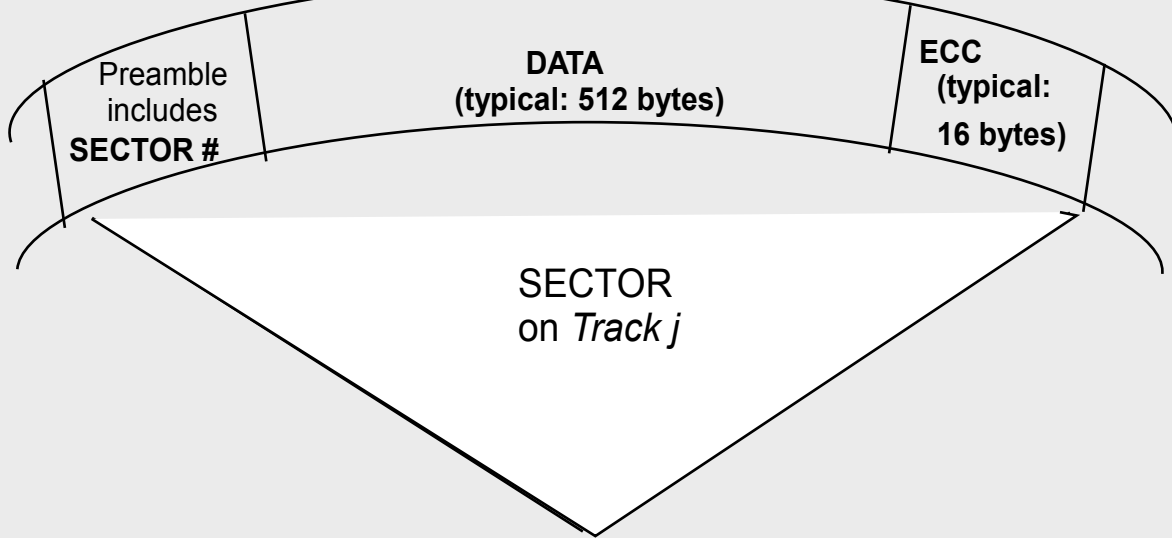
$d_5 = P = \text{parity bit}$

DISK USAGE

The parity disk $D_5 = P$ will be used every time there is a chunk written. The usage can be better distributed by using different disks for parity - for different tracks in a uniform distribution of tracks over disks .

RAID REDUNDANT ARRAY OF INEXPENSIVE DISKS

RAID 3,4,5



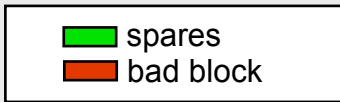
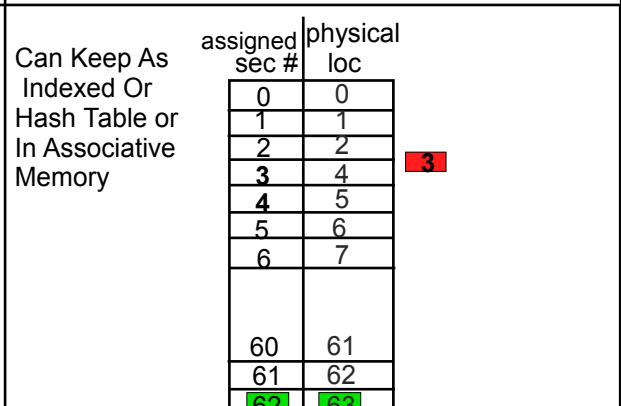
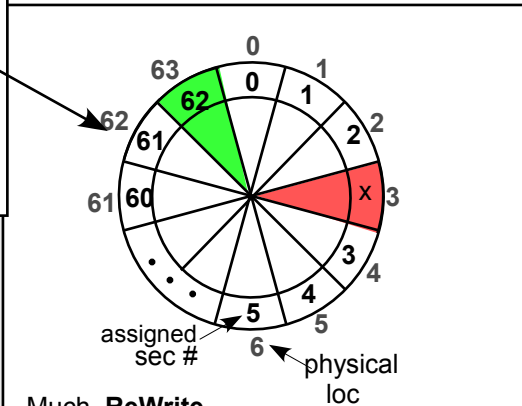
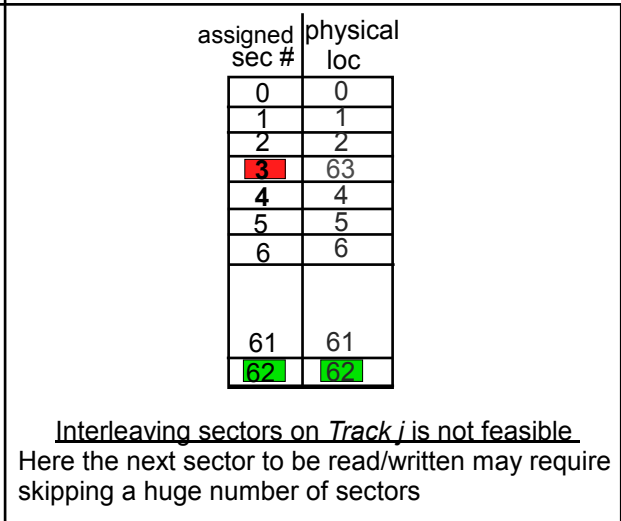
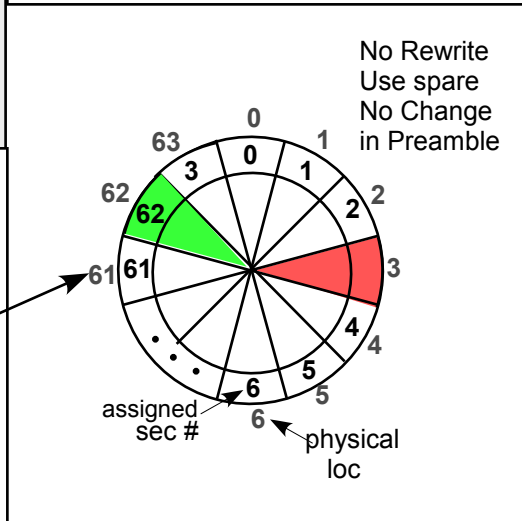
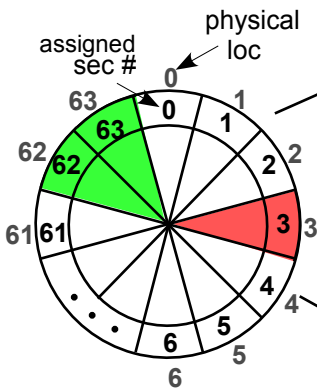
Constant rotation rate-sector numbers detected by reading Preamble under head

Usually there are a number of Bad Sectors on a disk and so there are a number of spare sectors provided

Two Ways To Handle Bad Sectors on Track j

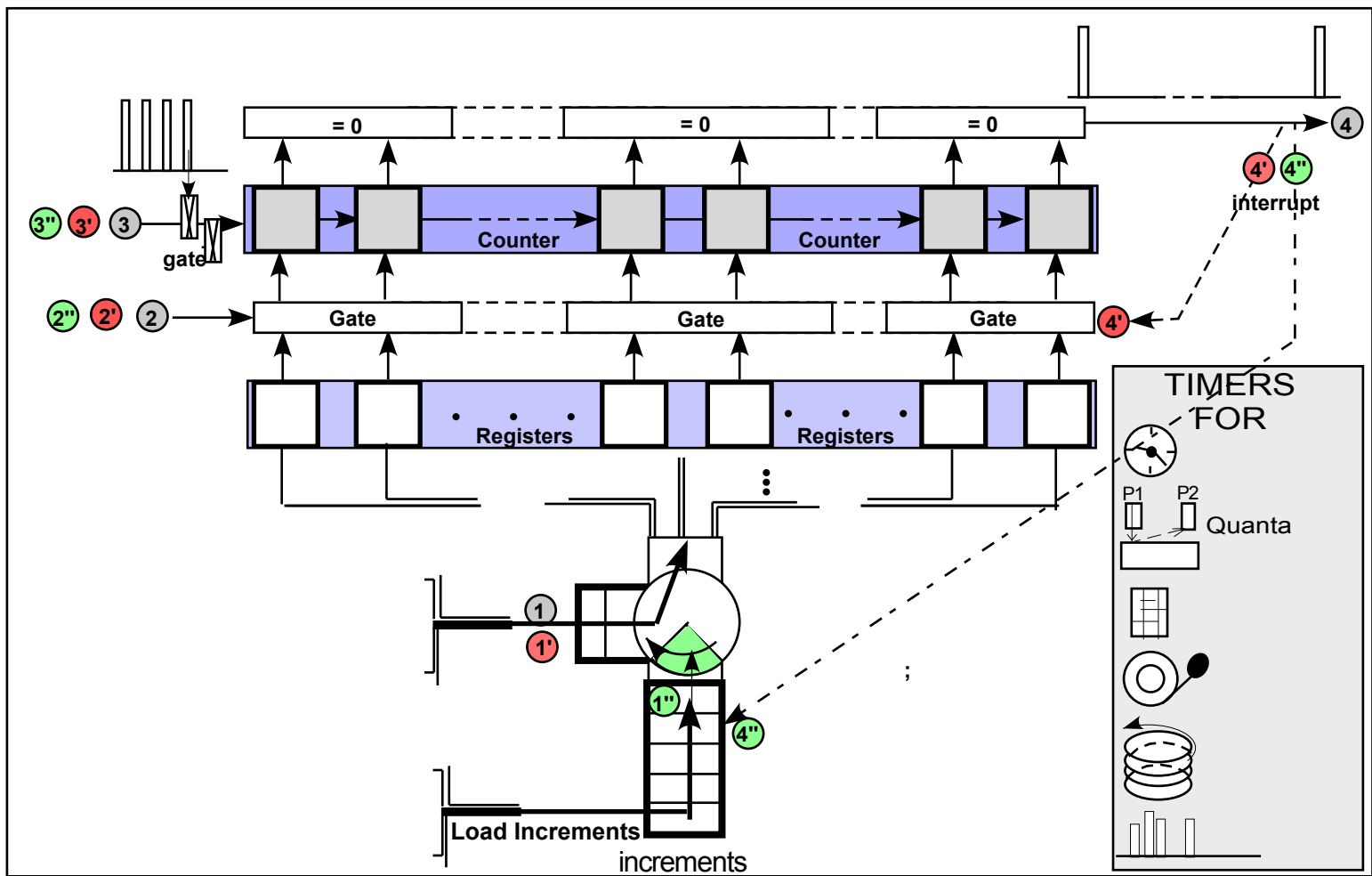
Corresponding Translation Tables In Controller Or in Disk Driver

Assume Sector 3 is Bad



Interleaving sectors is feasible.
For interleaving: sectors still occur in order around disk. However k interleaving would not always involve skipping k sectors-sometimes $k + \epsilon$.

DISK INTEGRITY BAD SECTORS-SPARE SECTORS



① CPU sends Interval time to Register
 ② Contents of register Gated to Counter
 ③ Oscillator pulses gated-causes counter to subtract 1 on each pulse
 ④ When Counter reaches 0 an interrupt is returned to CPU

One-Shot
One Time Interval

①' CPU sends Interval time to Register
 ②' Contents of register Gated to Counter
 ③' Oscillator pulses gated-causes counter to subtract 1 on each pulse
 ④' When Counter reaches 0 an interrupt is returned to CPU and

"Square-Wave"
Repeated Time Interval

Load Increments

①'' CPU sends top member of sequence of Interval increments times to Register, and shifts up subsequent members
 ②'' Contents of register Gated to Counter
 ③'' Oscillator pulses gated-causes counter to subtract 1 on each pulse
 ④'' When Counter reaches 0 an interrupt is returned to CPU and

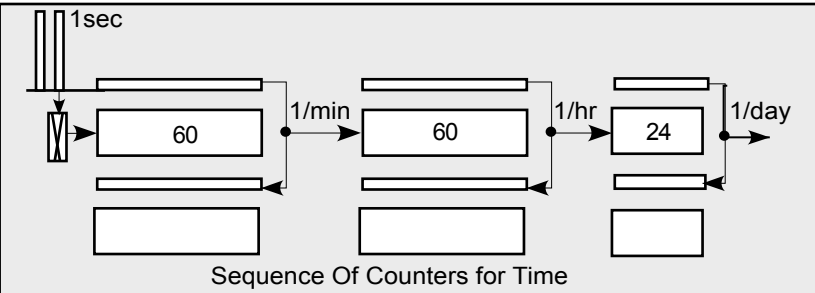
Incremented Intervals
Sequence Of Different Time Intervals

Times after 0 at which interrupts are to occur:

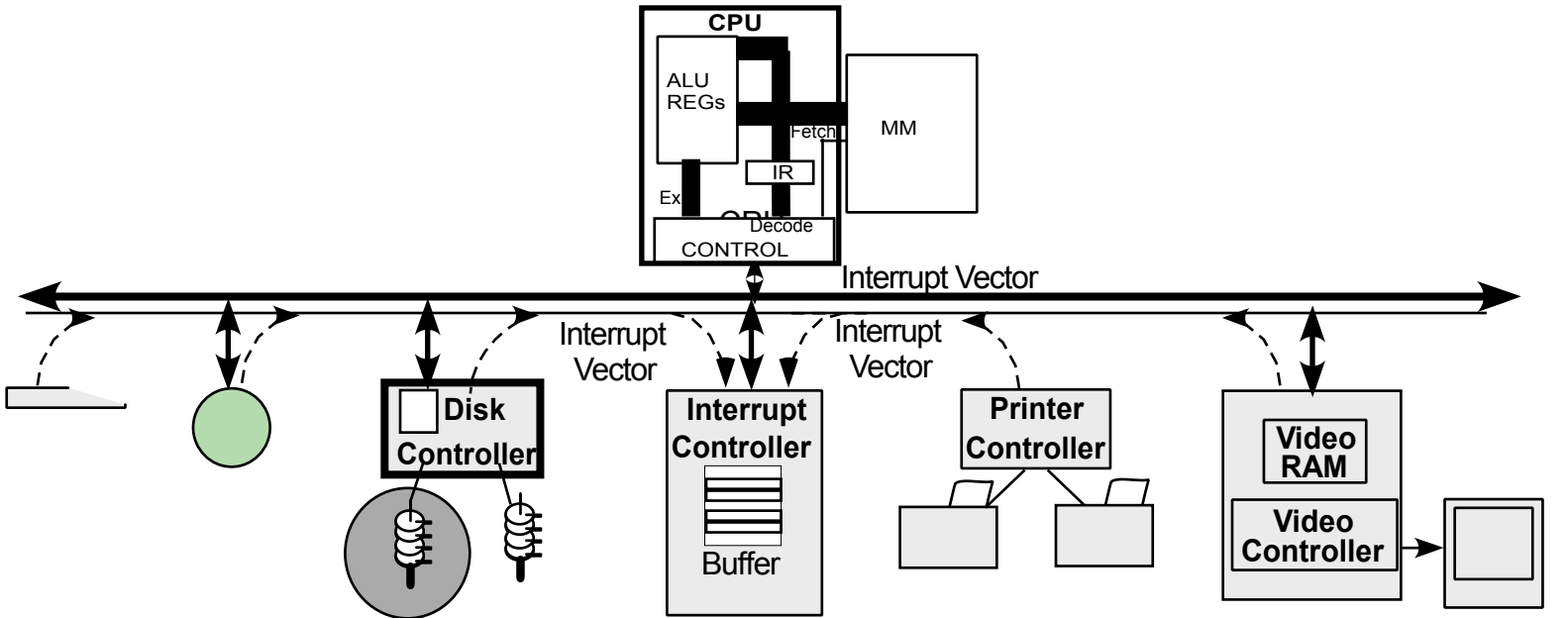
$$t_1, t_2, t_3, \dots, t_{n-1}, t_n$$

Corresponding Intervals sent as Incremented Intervals

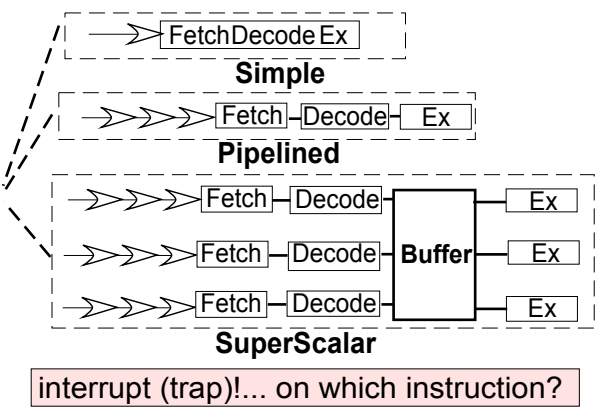
$$t_1, t_2 - t_1, t_3 - t_2, \dots, t_n - t_{n-1}$$



TIMER CIRCUITS



- 1 Device sends interrupt vector to Interrupt Controller-Buffered
 - 2 Interrupt Controller sends Interrupt Vector indicating device and interrupt type, and an interrupt alert to CPU
 - 3 CPU **polls** after each instruction. On detecting an interrupt it traps and indexes (based on interrupt vector) into an interrupt service routine-and after servicing sends acknowledgement to Interrupt Controller which can then send next buffered interrupt to CPU.
- Just before interrupt service starts the machine state must be recorded. This can be done by pushing all onto existing run-time stack (for procedure calls), and calling OS procedure, or doing context switch and loading Kernel Stack with machine state



In pipeline or superscalar when an interrupt is detected at the end of an instruction (or a few instructions) the partially executed instructions can be completed-no new instruction allowed to enter the pipe. If an instruction cannot be completed because of a page fault it must be possible to backup to the previous instruction and also maintain information of partially completed instructions in pipe (not a precise type of interrupt) or more likely undo subsequent instructions in the pipe. **This can be simplified if when an instruction which can page fault enters the pipe the following instructions are not allowed to enter till that danger passes.**

Precise Interrupt: (If all interrupts have this property then recovery is assured)

- 1 PC is saved in a known location, PCsave, and The execution state* of the PCsave instruction are known and give the correct information for the case that.
- 2 All instructions prior to PCsave instruction have been executed
- 3 None after PCsave instruction have been executed

To get a precise interrupt lots of hardware is necessary in pipelined and superscalar machines-or, if the hardware does not insure precision, a large amount of information must be dumped so the OS can provide the equivalent of a precise interrupt. In general a careful evaluation of the possible interrupts and the state of an executing instruction in the event of such interrupts is necessary.

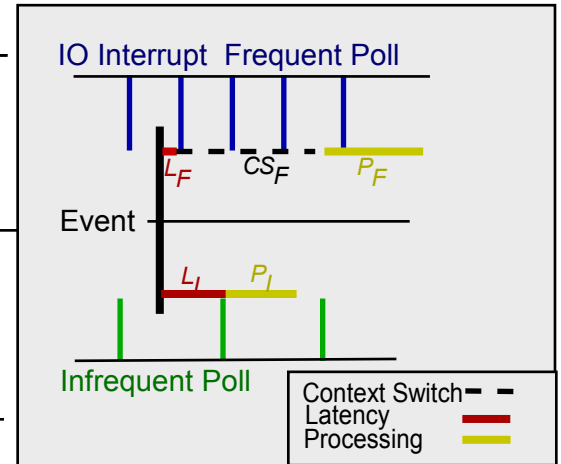
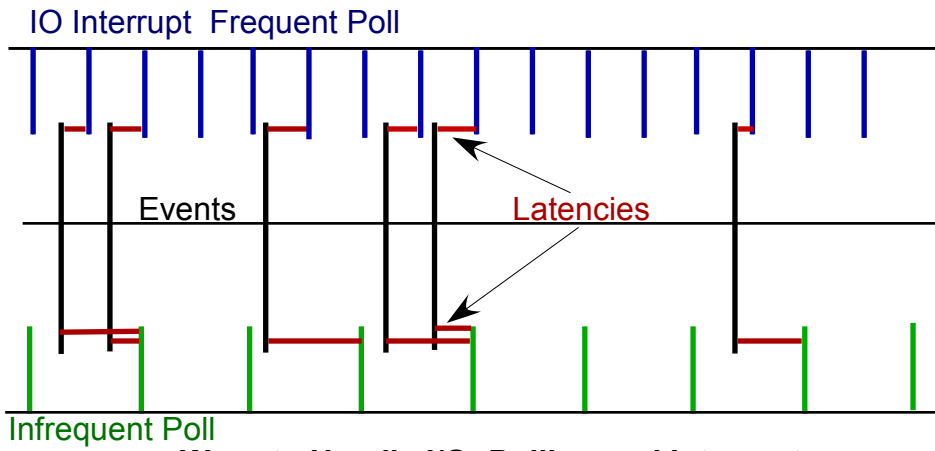
*Note that this state must provide enough information to be able to backup to the start of that instruction. So for example if the instruction *instr @ k: mem1-->mem2.* is invalidated by an interrupt there was
 Possible state ex. 1 fault on mem2 *instr in IR, mem1 contents in reg_k fault on mem2- reg_k --> mem2.*
 2 fault on mem1 *instr in IR, fault on mem1.*

MORE DETAIL ON INTERRUPTS-PRECISE INTERRUPTS

	Latency	Overhead	
IO Interrupts (Interrupt = frequent polling)	low latency	context switch (Involves Pipeline, TLB, Cache)	Possible After Each Instruction
Polling BY User (Kernel) For Event	high latency average: 1/2 Poll Rate	no context switch	Scheduled by Application (Op Sys)

Latency: The Interval of Time between an Event occurrence and the OS being aware of it

- 1) Interrupt: Interval between Instructions instruction time before known
- 2) Polling: Interval between I/O requests (and others requiring Kernel)

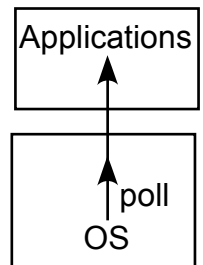


Ways to Handle I/O: Polling and Interrupts

Some Events have to be monitored very often. Ex., Packets for a single message **must be sent at a high rate**. If time required for a context-switch intervenes the rate may not be sustainable. **A Soft Timer** eliminates context-switch time-instead it employs Polling By The OpSys Kernel.

Just before leaving the Kernel a check is made of the real time clock to determine whether a soft time for an event has expired. If it has then the event is handled before leaving the kernel. A number of times can be checked, handled and timer reset before leaving the Kernel. This avoids context switches.

However for it to be useful the **system must be in the Kernel often**. The Kernel is called on for System Calls, TLB-Cache Misses, Page Faults, I/O interrupts, Timing interrupts, etc. In tests on Compute Bound programs the Kernel entry interval varied from 2 to 18 μ secs. About half were system calls. So an event that needed say 12 μ sec intervals could probably be handled with occasional misses by this technique.



SCENARIO

Send message in Users Process Buffer to Kernel Buffer. Kernel forms a Packet from its Buffer + additional info. Kernel gives to sending hardware. An acknowledgement interrupt is expected when packet has been received. (A timer could be set to determine time in which to expect interrupt) This acknowledgement interrupt can be handled either a) after an instruction or b) detected by OS on leaving Kernel

a) After instruction poll for acknowledgement execute a context switch from user to OS (slow). OS sends out next packet (if available).

Alternatively:

b) On leaving Kernel poll for acknowledgement (could first check timer). If completion is indicated send next packet if available.

Language:

IO is handled as though each IO device is a file-namely a Special File. These files have names just like other files. All are given the family name */dev-*so a disk may be */dev/hdl*, a printer */dev/p*. Such files can be opened like a regular file-which results in a file descriptor being returned (representing more than a table index in most cases) which then represents that device in *send or write*, and *receive or read* commands for transferring information to and from that device.

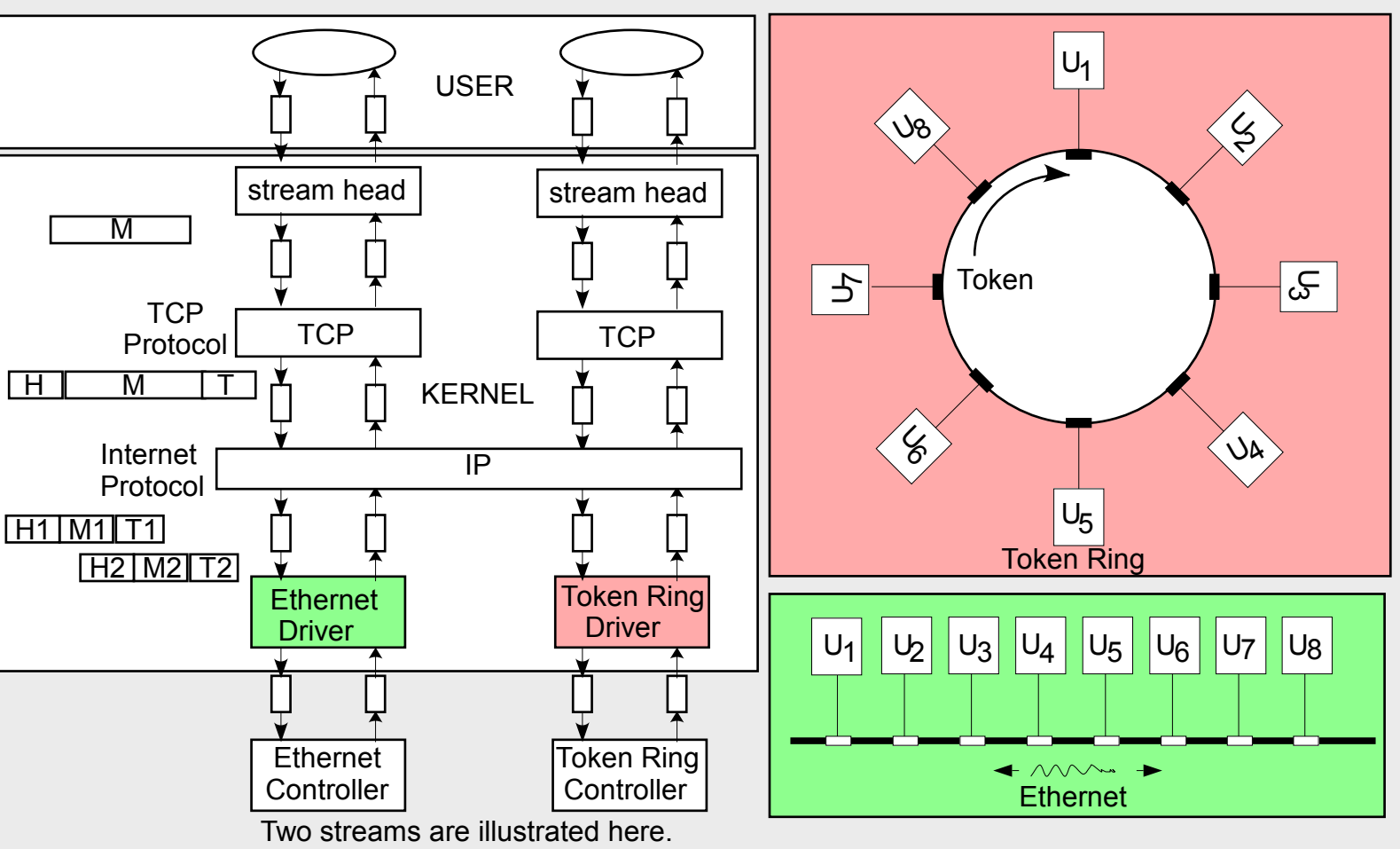
There are two types of Special Files.

1 Block, A file in consisting of a sequence of numbered blocks each individually randomly addressable and accessible. ex. Disk.

2 Character Devices that handle character streams such as keyboards, printers, networks, mice etc. These are more heterogenous and require a number of different modes of communication. There are system calls which allow the specification of a variety of means for accessing character special files.

A driver is associated with an IO device. The driver communicates directly with the device controller. The driver may handle one or more similar devices. A driver is identified by a major number giving the type of device it handles and if, it handles a number of similar devices (ex.disks), a minor device number to identify the individual device.

For character devices there is often considerable processing in a sequence of transformations required after the information has been formed by the User process and before it can control the Driver-or in the opposite direction between the Drivers receipt of the information from the controller and the arrival in the users buffer. Information progresses by being passed from buffer to buffer. This sequence is called **a stream**. The same transformations may be used within *streams* for different devices and therefore it is advantageous to set up these streams dynamically. UNIX system V provides language for doing this. The language essentially allows one to specify configuration designs for handling a great variety of character special files.



Two streams are illustrated here.

CHARACTER IO, STREAMS

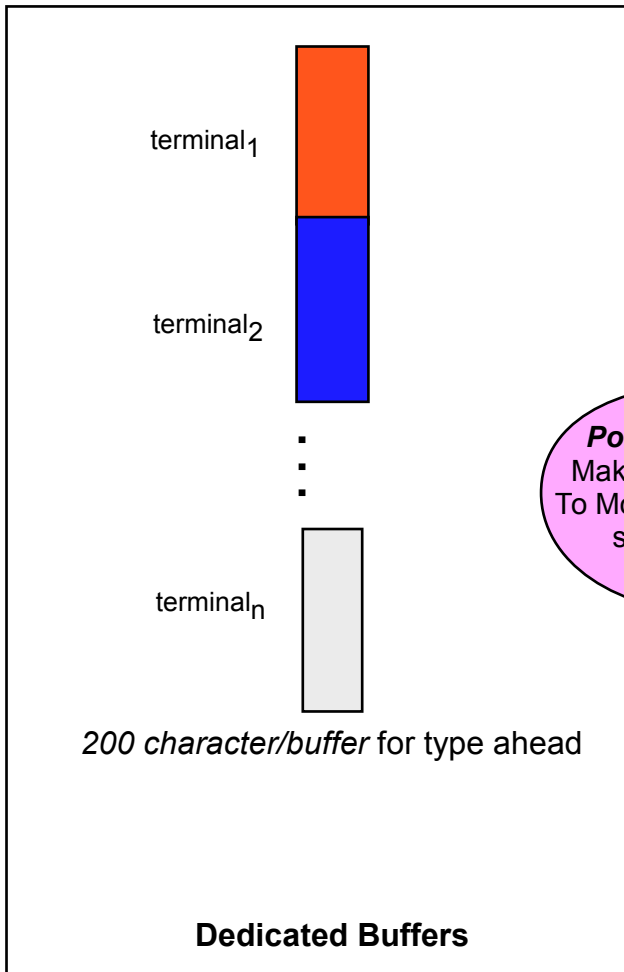
raw or canonical mode ex. "dste <<< ate"
 Choice: Buffer an entire line or Each Keystroke
 where keystrokes be given special meaning
 all keystrokes will be important to editing
 (ex. emacs)
 Pass on every key stroke or Buffer keystroke
 to pass on a complete line

cooked mode or noncanonical date

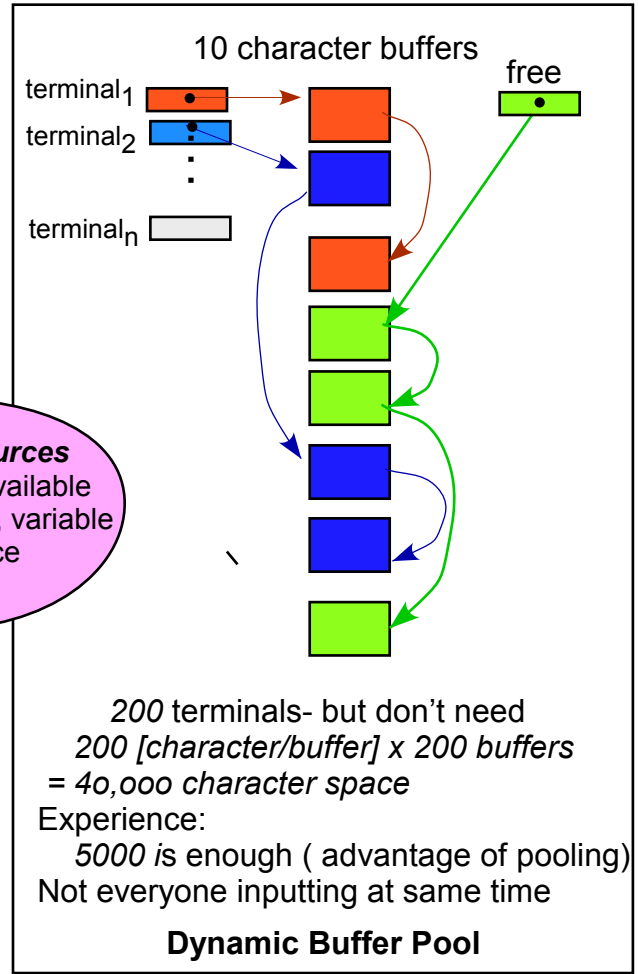
Must Buffer an entire line

Key Strokes

What Does the Driver Give To The Application



Pooling Resources
 Making Space Available
 To More than one, variable
 sized, resource

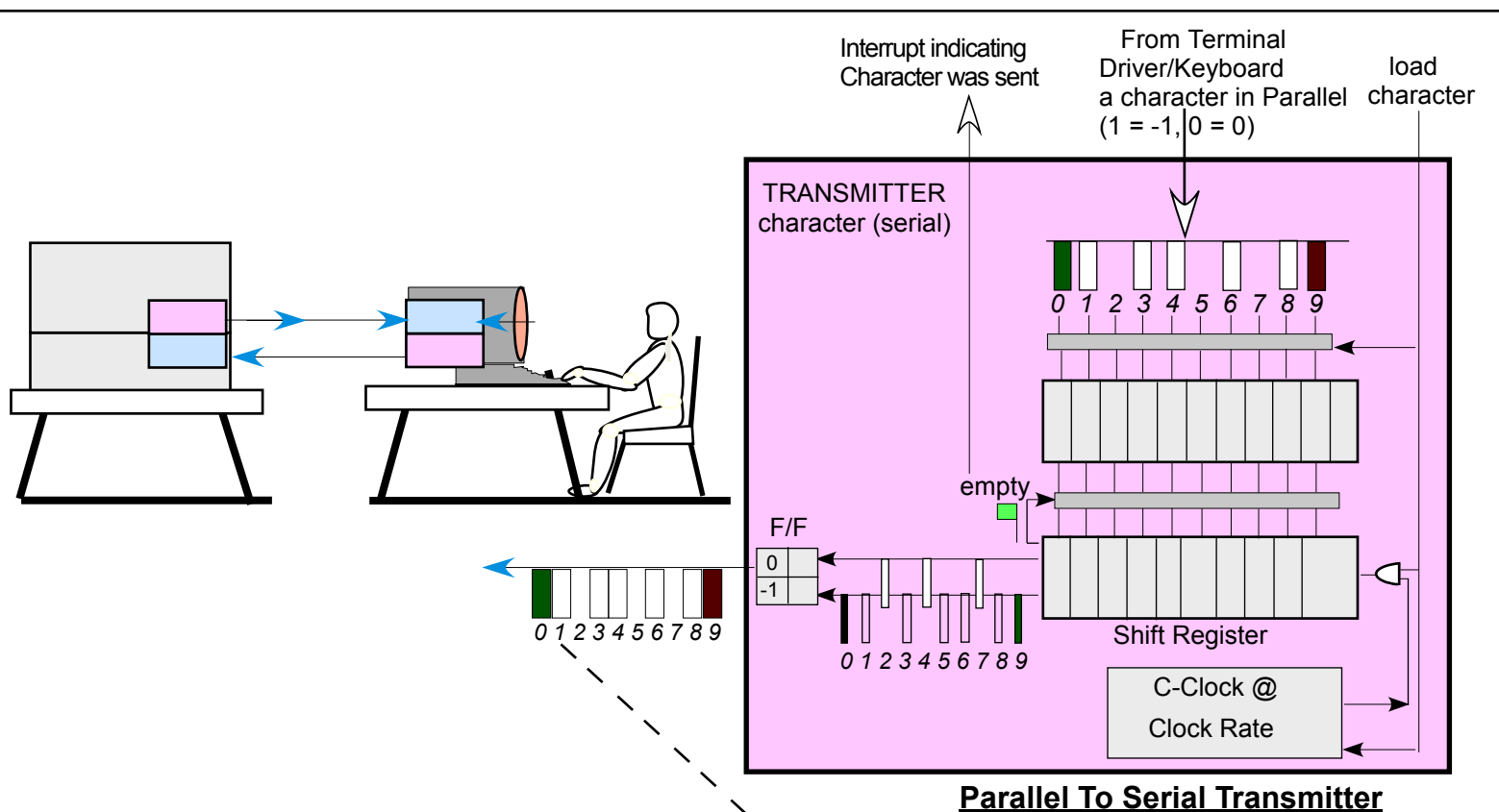


Buffered Key Strokes With Many Terminals

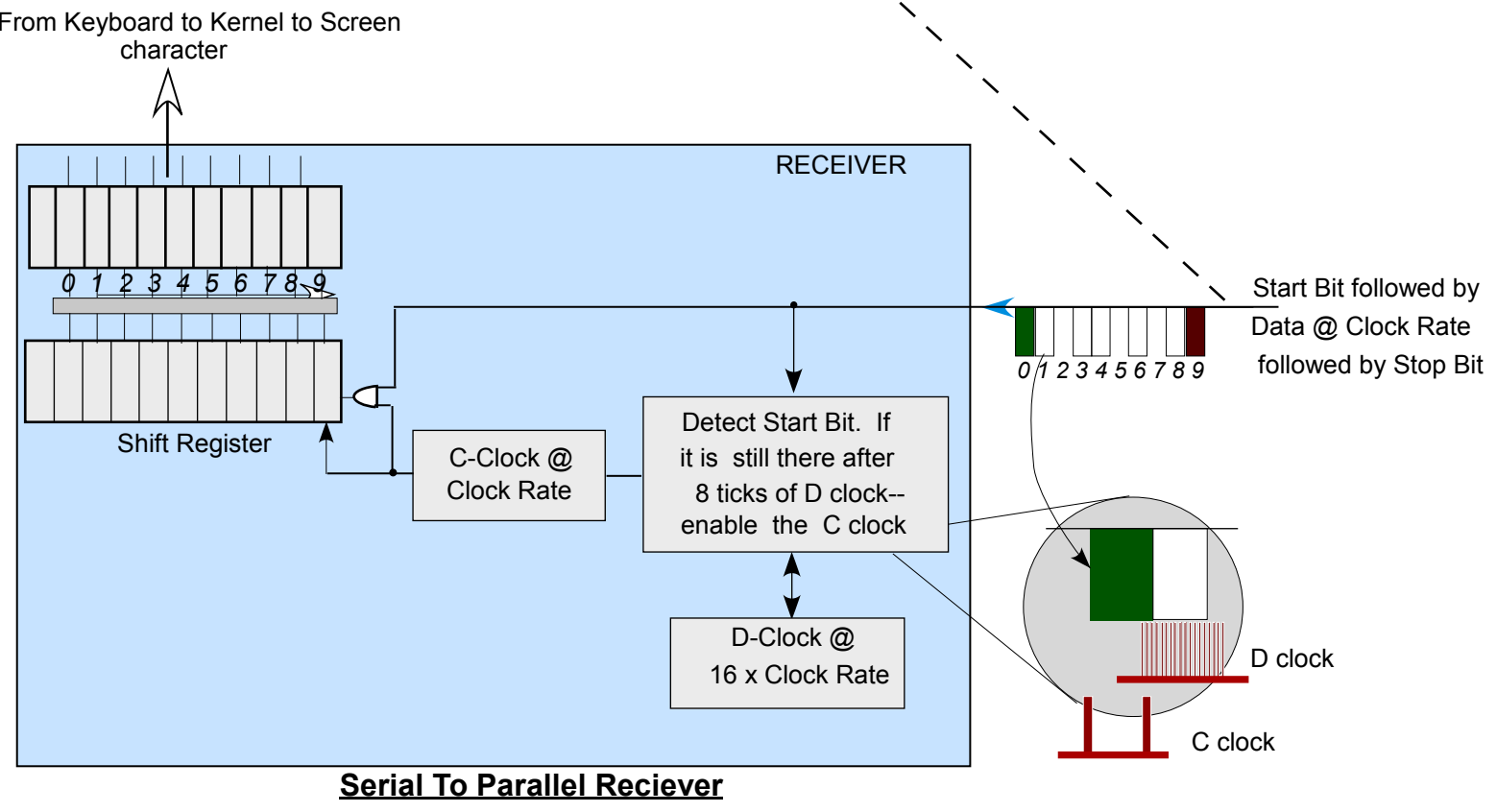
Displaying Typed characters on Screen. Usually Done By Software in the Driver or an Editor for example. So *passwords not echoed*. Older method used hardware to automatically echo. Hardware which couldn't distinguish between passwords and other text for example.
 Problems Remain ; Avoid interference with other output to screen

Echoing:

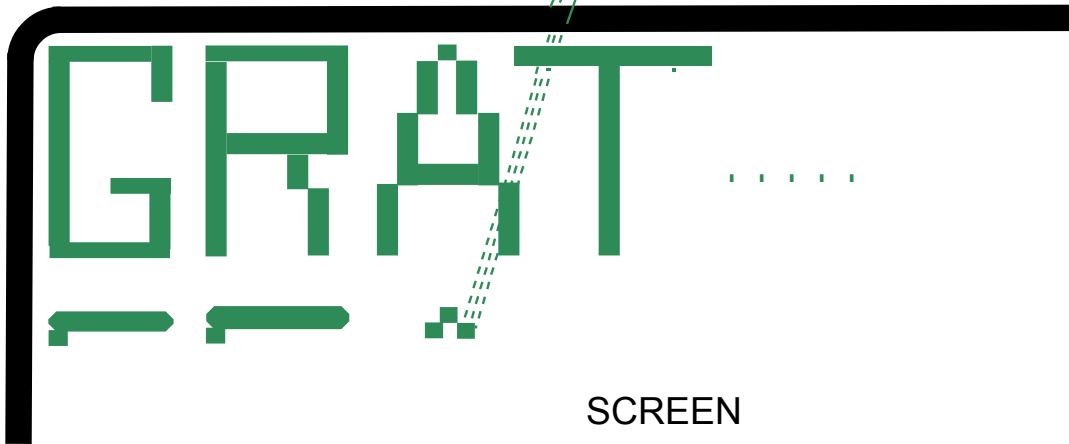
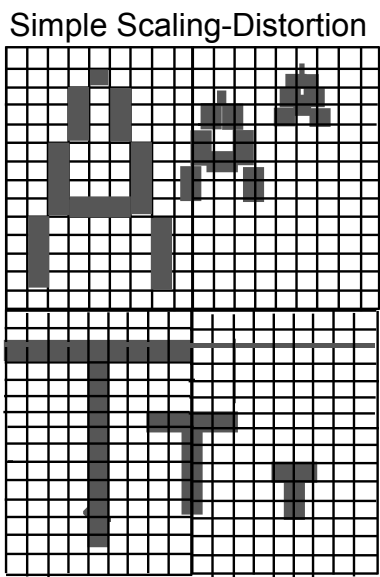
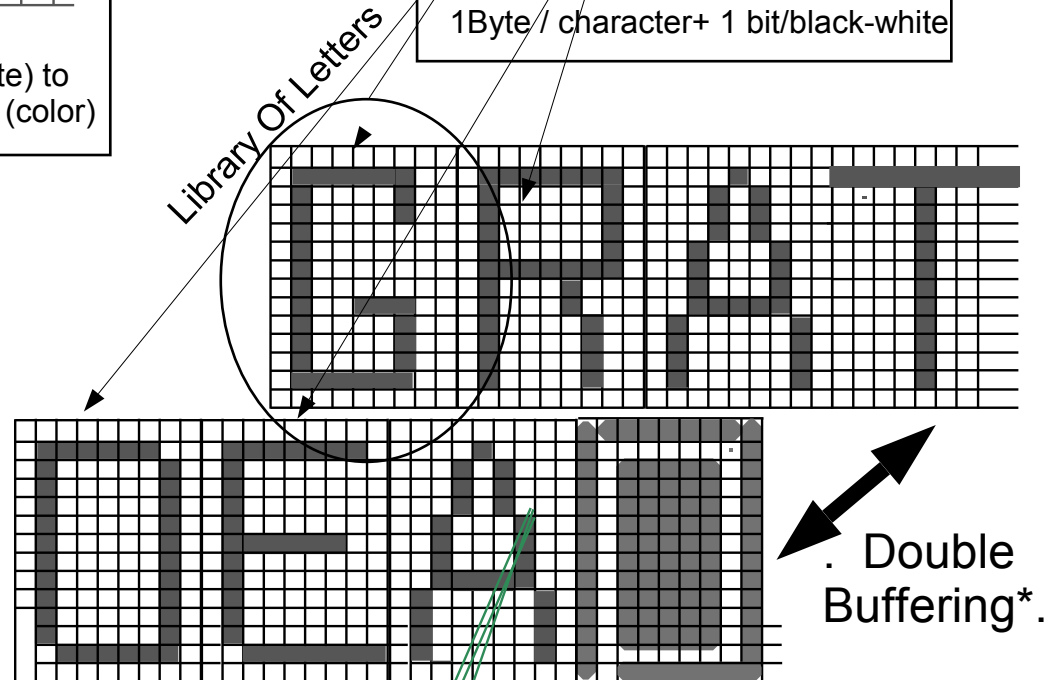
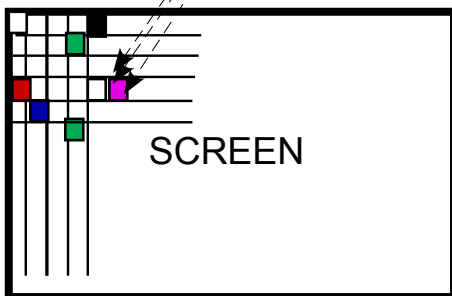
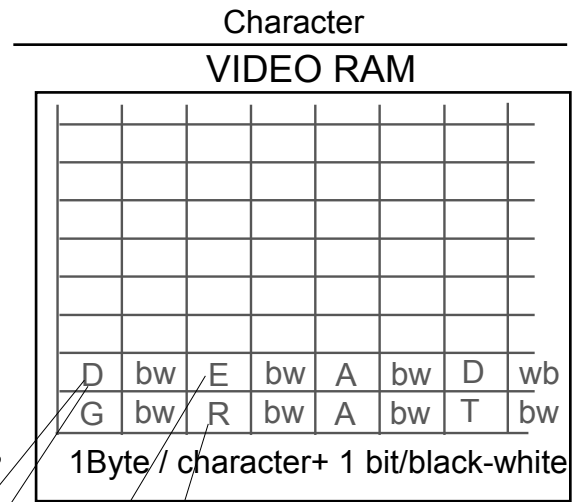
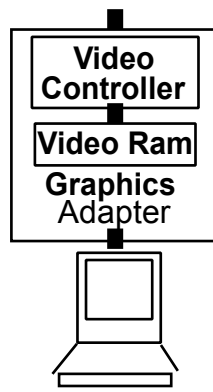
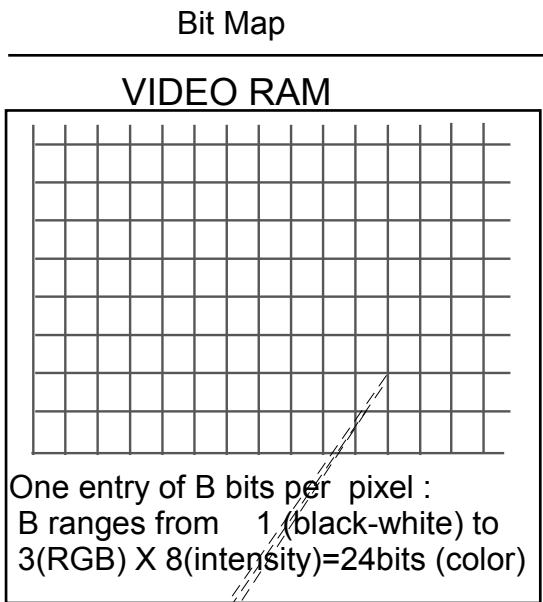
CHARACTER INPUTS TO TERMINAL DRIVER



1 for 1 clock pulse = 1
 0 for 1 clock pulse = 0



UNIVERSAL ASYNCHRONOUS RECEIVER TRANSMITTER (UART)



*The Rate at which Video ram is read = rate at which Double Buffer is read
But the Form of the Data is changed.

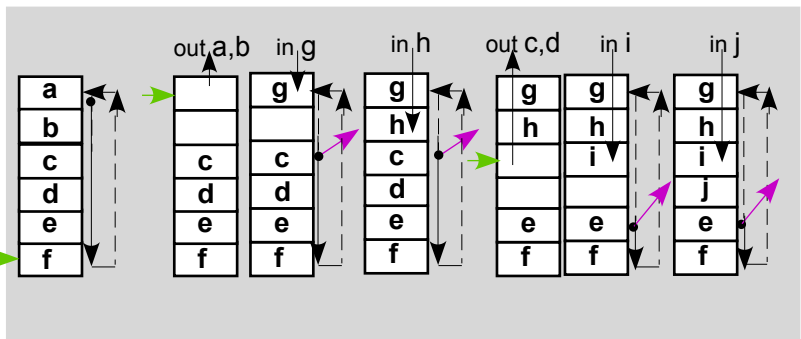
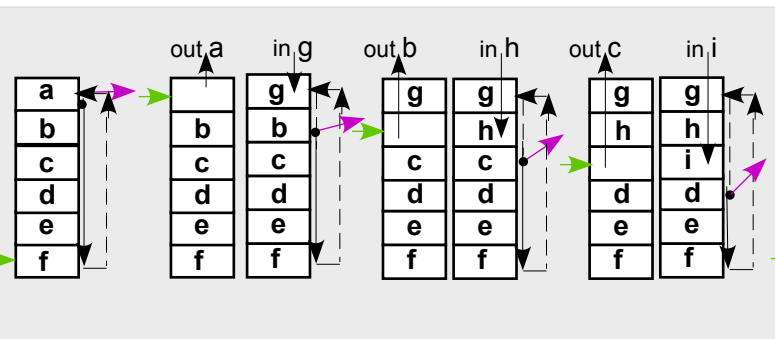
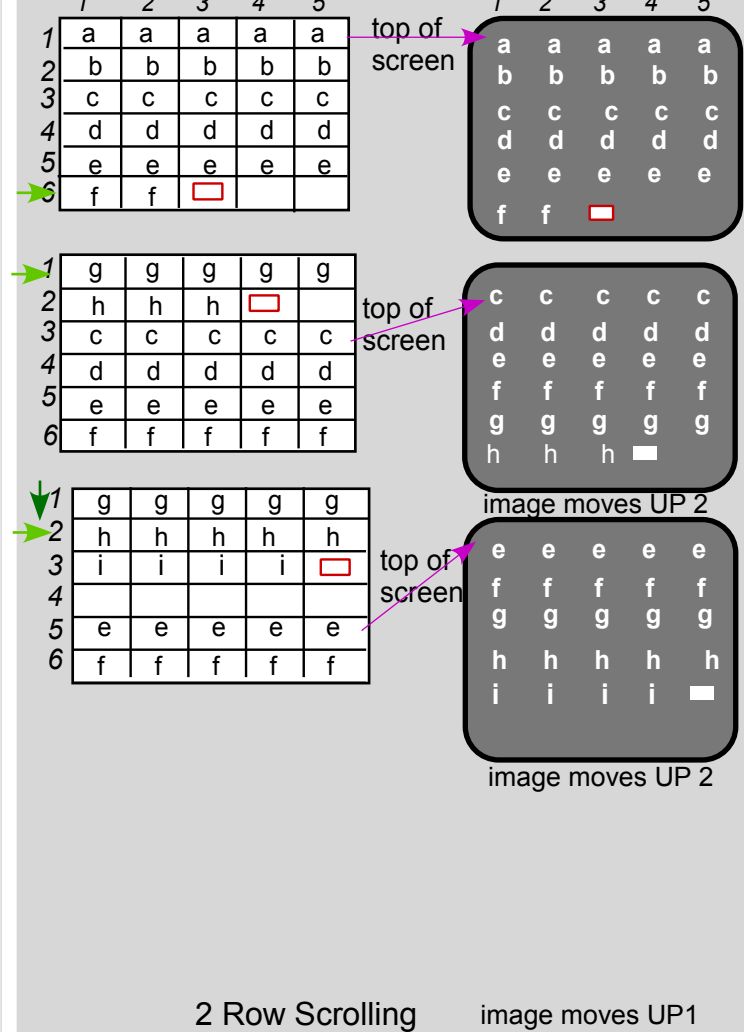
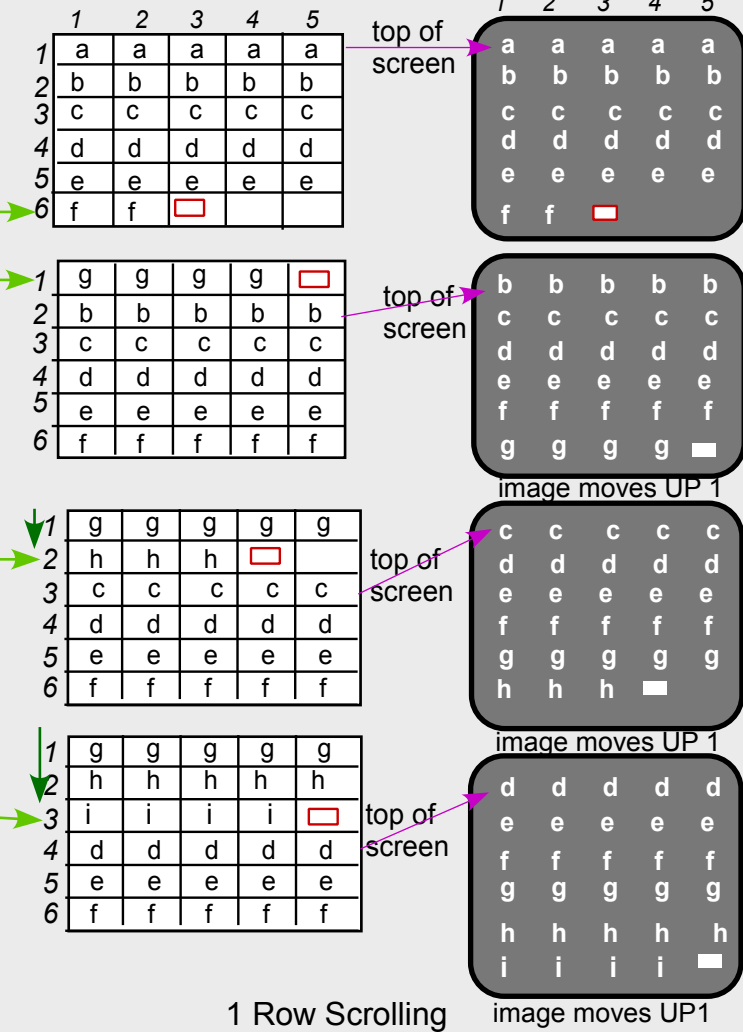
VIDEO MONITOR MODES

VIDEO RAM

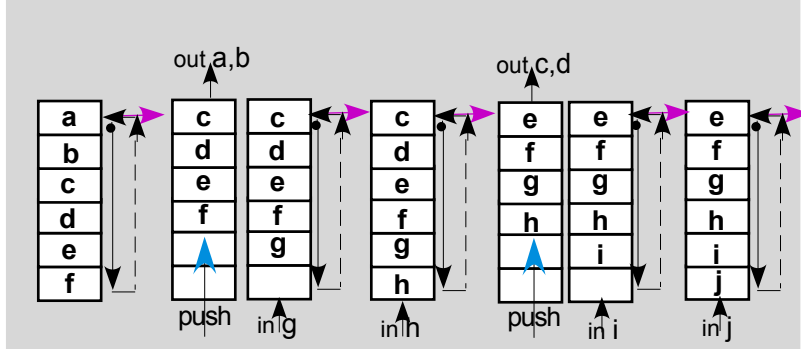
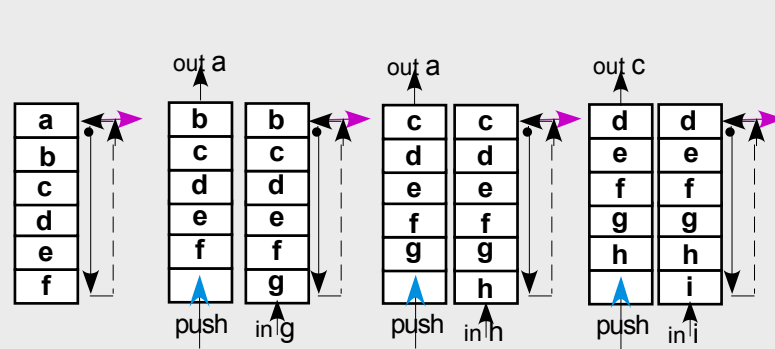
SCREEN

VIDEO RAM

SCREEN



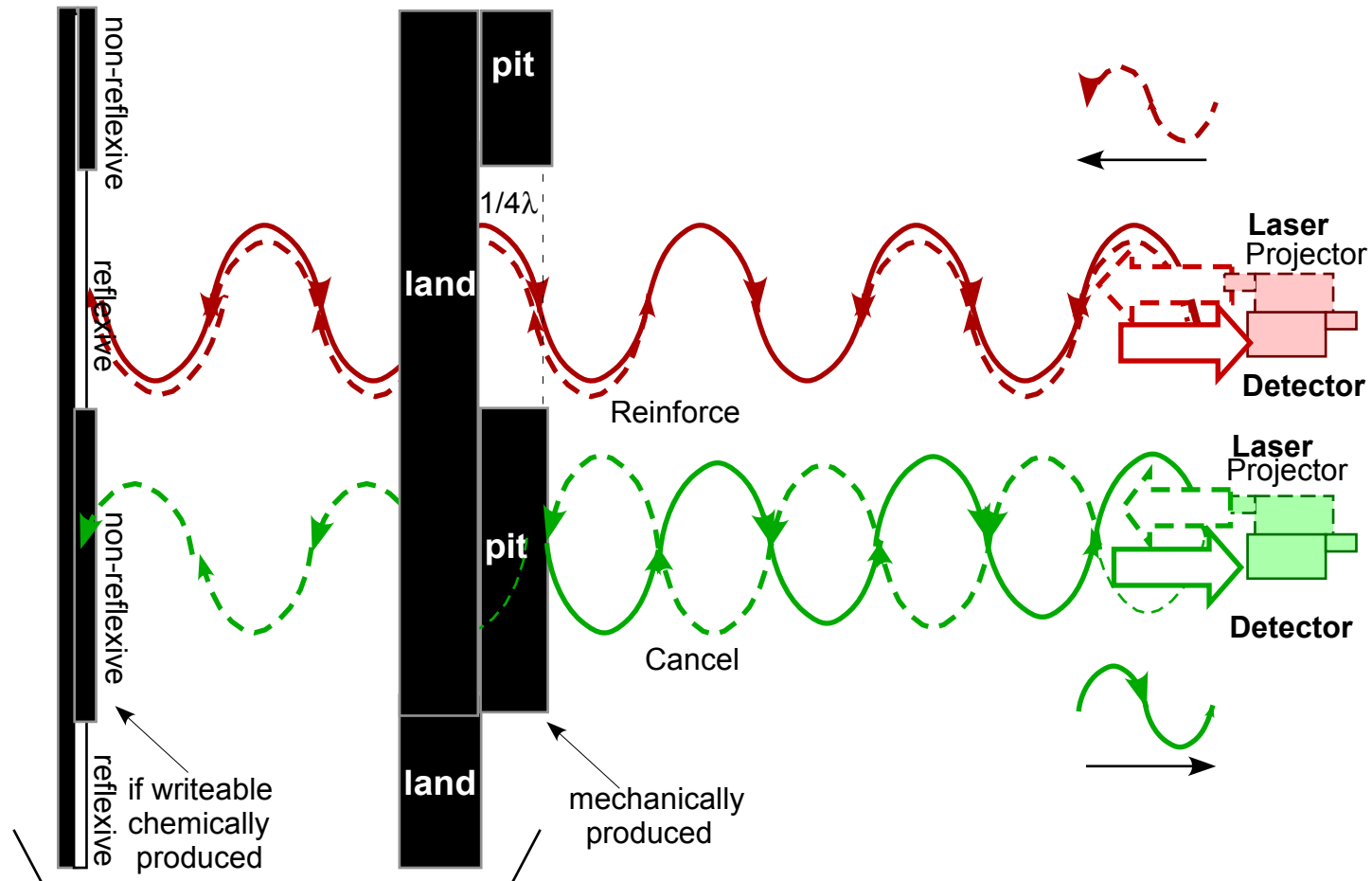
Looking at a single column Algorithm is FIFO by pointers: → To Top of Screen ↓ row for new entry



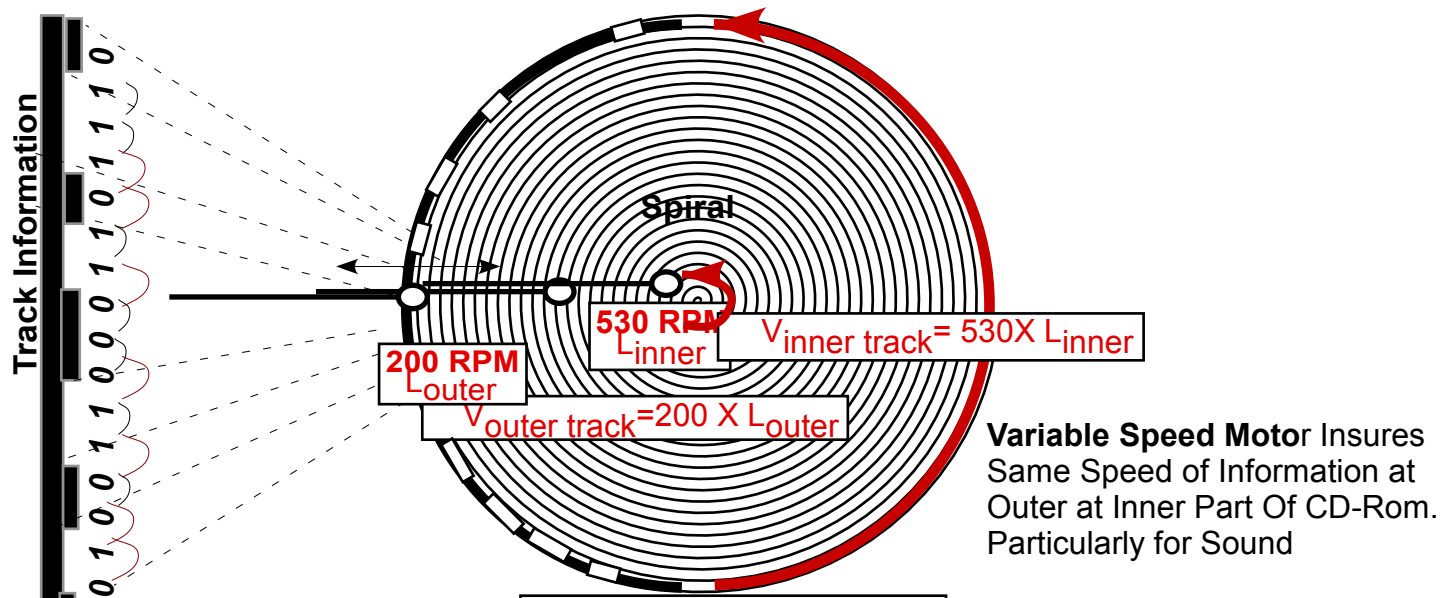
Equivalent To FIFO by Pushing Screen Appearance For Scrolling

SIMPLE VIDEO MONITOR SCROLLING

Reference Notes 4c, pg 6



Two Different Surfaces INTERFERENCE PATTERNS



Variable Speed Motor Insures Same Speed of Information at Outer at Inner Part Of CD-Rom. Particularly for Sound

$V_{outer\ track} = V_{inner\ track}$
CONSTANT VELOCITY

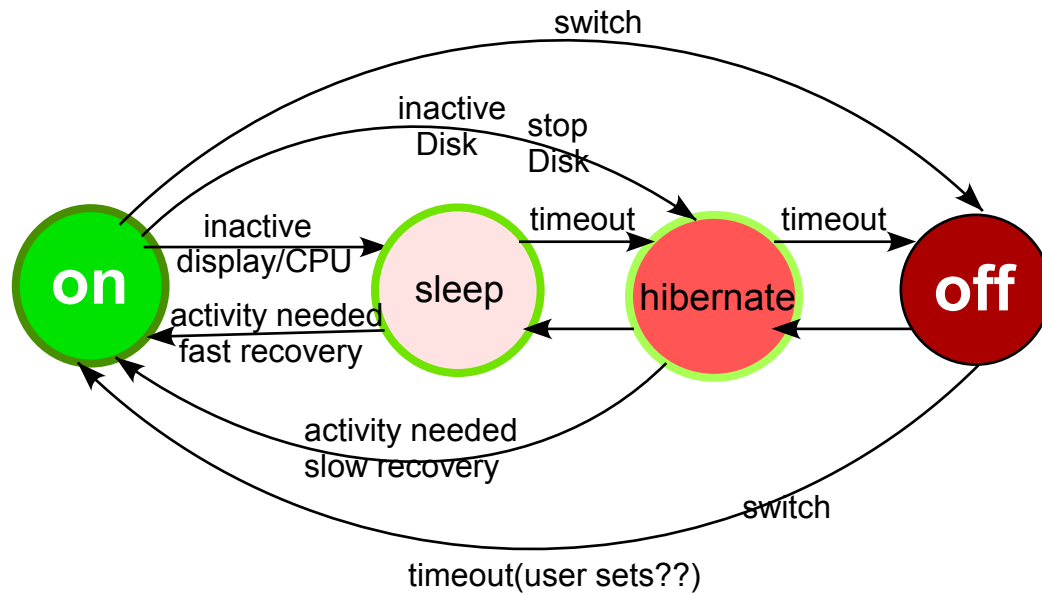
$RPM \times L(engh) = Distance/Minute = Velocity$

- 1 For Audio CD (Phillips)
- 2 For Data-Files CD-ROM (Simulate Sector, Preamble etc. Error Correcting)) Contiguous Files
- 3 For Movies DVD

Change to represent 1
 No Change to represent 0
Alternative: Detect Changes
 [More Reliable due to wobble. and other inaccuracies,)

CD, CD--ROM, DVD

Power Use Considerations:
 Costs-Immediate (User Pays) -Long Term(Every one Pays)
 Total Energy Of all computers Running at any time is huge.
 Individual Organization with large Data Bases, Many active Screens
 Battery Life is also a factor.



Power Saving States

Power Users:	Display >	CPU <>	Hard Disk >>> MM
Recovery Time:	Small		Large
Inactive State	Sleeping	Sleeping	Hibernating
Partial Inactivity Alternatives	2 Zones	Slow Down*	

*CPU: $K_1 \times (\text{Voltage})^2 \longrightarrow \text{Power}$
 $K_2 \times \text{Voltage} \longrightarrow \text{Speed}$
If voltage is cut in half :
 CPU: $K_1 \times (\text{Voltage}/2)^2 \longrightarrow \text{Power}/4$
 $K_2 \times \text{Voltage}/2 \longrightarrow \text{Speed}/2$
Trade off Of Power For Speed

POWER MANAGEMENT

Next Entry To Video Ram

Row Of Video Ram To Top Of Screen

Next Entry To Video Ram

Row Of Video Ram To Top Of Screen

