DYNACUBE OPERATING SYSTEM
An x86 based 32bit Protected mode GUI Operating System

PROJECT REPORT

SUBMITTED BY

J. Mohammed Hassan Shah (2KC60)
K.R Meenakshi (2KC59)

GUIDED BY

Mrs. G. Andal Jayalakshmi, B.E, M.S

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Thiagarajar College of Engineering

(An Autonomous Institution Affiliated to Madurai Kamaraj University**)**

MADURAI – 625 015.

April 2004

ACKNOWLEDGEMENT

We are grateful to our beloved Principal, **Dr. V. Abhai Kumar,** who is responsible for establishing an atmosphere conducive for completing our project successfully.

We thank our Head of the Department of Computer Science & Engineering **Dr. N. Ramaraj,** who boosted us to work hard in the project and guided in all academic fronts.

We are very grateful to our Guide, **Mrs. G. Andal Jayalakshmi, B.E., M.S.,** Senior Lecturer in Department of Computer Science & Engineering, who greatly helped us from the initial stage of this project.

Last but not the least we thank the almighty god who has guided us through difficult times and has helped us in completing this project.

I dedicate this project to my beloved parents and my dear brother …

J. Mohammed Hassan Shah

I dedicate this project to my beloved parents and my brother …

K.R. Meenakshi

CONTENTS

FIGURES

TABLES

1. SYNOPSIS

Dynacube operating system is a contemporary operating system that provides

- 32bit Protected mode operation
- TSS based multitasking
- Process and Memory management
- Inter-process communication support
- File System management
- Device management
- Graphical User Interface management
- POSIX 1003.1 compliancy

The Dynacube operating system was designed from scratch to provide an efficient, highly modular, secure and uniform system interface. This design decision has been upheld till the completion of the project. The main kernel takes care of managing processes, multitasking, memory management, and Inter-process communication. The other tasks like the File System Server, Disk Server and the GUI Server are high privilege tasks that service requests of client applications. The clients communicate with the Server using the Inter-process communication support provided by the kernel.

The kernel provides memory management functionality at the page level by using the BIMA page allocator. The variable chunk level dynamic memory request is handled at two entry points – one for the kernel and the other for the user applications.

The File System Server provides a uniform file system interface to the client irrespective of the underlying file system format. The Disk Server provides the direct disk manipulation interface to its clients. The GUI Server provides a cleaner way of screen handling to client applications. The clients are not directly allowed to access the video hardware but can request the GUI Server to do screen handling on their behalf. The GUI Server then services the request based on its validity.

2. OBJECTIVES

- To develop an x86 based 32bit protected mode operating system.
- To develop a full-fledged process handling and memory management subsystem.
- To provide Priority based round robin scheduling
- To provide a sophisticated message-passing interface for inter-process communication.
- To present an efficient, highly modular and uniform system interface.
- To provide a user-friendly Graphical User Interface.
- To comply with the POSIX 1003.1 standard.
- To create an Easy-to-use Dynacube Application Programming Interface (DAPI).

3. INTRODUCTION

"*An operating system is a system software that provides reasonably high-level services with unreasonably weird low-level hardware.*"

The art of designing and implementing an operating system is a highly complicated and rewarding mission. Our decision to develop a full-fledged 32bit protected mode, multitasking, POSIX compliant operating system with a user-friendly GUI, was purely made to understand system level programming and to experiment with the our computer's hardware. The fascination of making our computer run purely on our own operating system provided the impetus to make this project a reality, despite the many setbacks that we received during the initial stages of our project.

The prime aim of our Dynacube operating system is to provide a uniform and highly modular interface. We have also aimed at delivering a contemporary operating system.

The Dynacube operating system was designed to provide the following modules:

- Dynacube Kernel and Kernel level Device Drivers

  The kernel provides process manipulation, memory management, inter-process communication and low-level raw device drivers like the keyboard, mouse and floppy driver.

- Dynacube File Server

  The Dynacube File Server runs as a separate task and services file-system requests from the clients. This interaction is based upon the inter-process communication interface provided by our kernel.

- Dynacube Disk Server

  The Dynacube Disk Server runs as a separate task and services the direct disk manipulation requests from the clients. This interaction is based upon the inter-process communication interface provided by our kernel.

- Dynacube GUI Server

  The Dynacube GUI Server also runs as a separate task and services the GUI requests from client applications. Only the GUI Server controls the screen and clients are not allowed to access the video hardware directly. Thus a client sends a request to the GUI Server to perform screen handling on its behalf.

We chose the Linux operating system as our host platform for developing our operating system, as it provided our favorite C/C++ compilers namely the GCC and G++ compilers. It also provides a host of other useful utilities like the objdump and nm, which are very helpful in offline debugging. The chief advantage of using GCC is that it has strong support for inline assembly – this allowed us to embed assembly code within our C or C++ code.

We have used VMWare and Bochs as our System Simulators – This has been a great timesaver for our project. As the usage of system simulators helped us to continue development and testing of our operating system from within our host operating system without having to reboot our computer for test boot. After the completion of our project we have tested our Dynacube operating system on many real computers and have found our operating system to be stable for long operational periods.

Description	**Software**
Platform used	Redhat Linux 8.0
Assembler	NASM
Compilers	GCC, G++
Linker	LD
Utilities	objdump, nm, strip
Virtual System	Bochs, VMWare

4. GENERAL DESIGN

4.1 DESIGN GOALS

The Dynacube Operating System has the following goals

- To construct a highly scalable x86 based POSIX compliant operating system
- To provide a dynamic, highly modular and uniform system interface
- To provide multitasking support, ring based protection and virtualization of memory.
- To provide a user-friendly Graphical User Interface (GUI).

4.2 ARCHITECTURAL CHOICE

The targeted systems

- x86 Systems – 486 and above

The choice of architecture is purely based upon the widespread usage of Intel processors.

Minimum System Requirement

- Intel 486 or Above
- System RAM – 64MB or Above
- Video card – VESA VBE 2.0 compatible, with minimum 1 MB Video RAM with Linear Frame Buffer support.
- PS/2 mouse and keyboard
- NEC µPD765 or Intel 82072-7 Floppy Disk Controller

4.3 CONCEPTUAL DESIGN

The Dynacube Operating System has a highly modular design. The key components of Dynacube operating system are

1. Dynacube Kernel
2. File Server
3. Disk Server
4. DServer – GUI Server

4.3.1 Dynacube Kernel

- The kernel provides support for process handling and memory management.
- The kernel provides protection using the Intel processor's protected mode segmentation feature and virtualization of memory by paging mechanism.
- The kernel provides inter-process communication support and interrupt driven system call interface.

4.3.2 File Server

- The File Server is based on client-server architecture.
- It provides basic file system services to both the kernel and client applications.
- The client and the File Server interact using the inter-process communication interface provided by the kernel.

4.3.3 Disk Server

- The Disk Server is also based on client-server architecture.
- It provides direct disk manipulation to its clients.
- The File Server acts as a client to the Disk Server.
- The interaction between the Disk Server and the clients is based on the inter-process communication interface provided by the kernel.

4.4 Design Issues and Constraints

- The multitasking support is provided using 32bit Task State Segments (TSS).
- Virtualization of memory is provided using the Intel's paging mechanism.
- Remapping of Programmable Interrupt Controller (PIC) is done so as not to interfere with the Intel's reserved IRQ in protected mode. Thus the IRQ0-IRQ7 are remapped from interrupt 0x8-0xF to interrupt 0x20-0x28 and the IRQ8-IRQ15 are remapped from interrupt 0x46-0x4D to interrupt 0x29-0x2F. The system call IRQ is set as 0x30.
- Creation of virtual 8086 GPF monitor for performing BIOS and real mode operations while the system is in protected mode.

5. MODULE DESIGN AND IMPLEMENTATION

5.1 BOOTSTRAP DESIGN

5.1.1 Boot loading

Boot loading is one of the most important aspects of an operating system. The boot loader is responsible for loading the operating system from the host disk. The concept of bootstrap is ‘*pulling oneself using one’s own boot’s straps* ’. When the system boots up it checks for a boot media, which generally is the floppy disk or the hard disk. Once the system finds a boot media it searches for the boot partition. In our Dynacube operating system we use a floppy as our boot media. So when the system searches for the boot media it finds our floppy disk. The system then checks for boot signature (0xAA55) at the end of the first sector of the boot floppy. Once the system agrees with the boot signature it loads the first sector (512 bytes) of the floppy disk at the memory location 0x7C00 and passes control to it.

The bootstrap, which is working in the real mode environment, does the following:

- Disables hardware interrupts.
- Synchronizes the various segment registers.
- Enables the A20 line.
- Loads the kernel from the floppy disk to memory location 0x100000 (1MB).
- Sets up the protected mode bit in control register 0 (CR0) and loads a simple GDT in the GDTR. This GDT contains three segments – 1 code segment, 1 data segment and 1 stack segment each spanning an address from 0x00000000 to 0xFFFFFFFF (4GB).
- It flushes the 16bit instructions from the instruction prefetch pipe by jumping to a 32bit code. Now the system will automatically enter the protected mode.
- After this the bootstrap passes control to the kernel, which is currently loaded at the 0x100000.

5.1.2 Kernel Initialization

The kernel has to do self-initialization and environment initialization before the system can start functioning. The kernel after receiving the control from the bootstrap does the following initializations:

- The kernel disables hardware interrupts.
- The kernel initializes the Interrupt Descriptor Table (IDT) and loads it in the IDTR.
- Creates a configurable global descriptor table and loads the GDTR with the newly constructed GDT. In Dynacube the GDT contains the following segments.

Entry No	**Segment Description**	**Segment Base**	**Segment Limit**	**Protection Level**
1	NULL	0	0	0
2	NULL	0	0	0
3	Kernel Data	0x0	0xFFFFFFFF	0
4	Kernel Code	0x0	0xFFFFFFFF	0
5	Kernel Stack	0x0	0x1100000	0
6	File Server Stack	0x0	0x1000000	0
7	GUI Server Stack	0x0	0xF00000	0
8	Disk Server Stack	0x0	0xE00000	0
9	System LDT	0x0	0xF	0
10	User LDT	0x0	0xF	0
11	System TSS	&_system	sizeof(TSS)	0
12	TASK1 TSS	&_task[0]	sizeof(TSS)	0
13	TASK2 TSS	&_task[1]	sizeof(TSS)	0
14	NULL	0x0	0x0	0
.	NULL	0x0	0x0	0
.	NULL	0x0	0x0	0
255	NULL	0x0	0x0	0

Table 5.1 - Segments Present in GDT

The protection level has 0 as the highest protection level and 3 as the lowest protection level. So the kernel and kernel-mode device drivers operate at the level 0 and untrustable code like user applications and utility programs run at level 3. This helps the kernel to be secure from malicious user applications. Also paging of memory and the resulting virtualization of memory provides protection within level 3 processes from each other. Paging also provides protection with two levels – namely the Supervisor level and the User level.

The _system, _task[0] and _task[1] are storage variables for the TSSes used by the Dynacube operating system.

- The kernel then remaps the Programmable Interrupt Controller (PIC) and thus makes the IRQ 0 – 7 to be mapped to IRQ 0x20 – 0x27 and the IRQ 8-15 to be mapped to IRQ 0x28 – 0x2F. The kernel also disables the Timer, Keyboard and Mouse IRQ.

- The kernel initializes the Task State Segments. The TSS initialization consists of
  - Clearing the _system, _task[0] and _task[1] TSS structure.
  - Capturing the Systems internal state in the _system TSS.
  - Setting up the System LDT and patching up the GDT entry for _system, _task[0] and _task[1] TSS entries.
  - Loading the System LDT in the LDTR.

- The kernel then initializes Process structures that hold individual process' state. These structures are popularly known as PCBs (Process Control Block). In Dynacube we store the system's runtime state when the process was suspended from execution. Apart from the internal state of the machine we also store other information regarding a process in its PCB.

- The kernel then initializes the various queues it will use during its execution phase. The queues that are initialized are
    - Ready queue
    - Message queue
    - Timer queue
    - Interrupt queue
    - GUI queue
    - FS queue
    - Various Device queues

- Dynacube kernel then initializes the zorder structure which is used by the GUI Server for maintain window internal ordering.

- The kernel then initializes the hashmap and list structures, which are also used by the GUI Server for its internal processing.

- The kernel initializes the Video hardware to get an 800x600 resolution with a color depth of 16 bits/pixel using the VESA VBE BIOS functions.

- The kernel initializes the mouse driver, which in turn initializes the PS/2 mouse for stream mode operation.

- The kernel then initializes the kernel and user page directories. It then sets the control register 3 (CR3) with the address of the kernel page directory and maps the linear frame address of the SVGA controller within the kernel space using the kernel page directory. The setting up of cr3 causes the system to enter paged memory model.
- The kernel then initializes the File System Server structures.
- The kernel then forks out the NULL process. This process is forked out to remain as an idle process so as to keep the system running even when there are no other processes running in the system.
- The kernel then starts the GUI Server.
- The kernel then starts the Disk Server.

- The kernel then starts the File Server.
- The kernel then initializes the System timer, which is crucial for a multitasking system. The reason is that the timer is the used for producing interrupts at specific intervals of time. This causes the kernel to gain control of the system whenever a timer interrupt occurs. This makes the system pre-emptable.
- The kernel enables the Timer, Keyboard, Floppy and PS/2 Mouse IRQs.
- The kernel enables hardware interrupts.

The moment the kernel enables the hardware interrupts the Timer generates an interrupt that is passed as IRQ 0x20 from the PIC to the kernel. This sets the system in motion as the system starts to multitask.

5.1.3 Kernel Functioning

Once the system starts with a timer interrupt the system passes through the kernel ‘s interrupt handler. The handler finds out the source of interrupt (in this case it is the Timer) and dispatches the control to the scheduling function. The scheduling function’s job is to find the next process that will be allowed to run. The currently suspended process’ PCB is synchronized with the current internal state of the system. The chosen process’ PCB is loaded into the TSS and then the new process starts to execute until the next timer interrupt occurs (or) the process voluntarily yields control. The user processes can request services from the kernel by invoking the system call interface, which the kernel handles to provide the requested service to the client. The system-call interrupt request number for Dynacube operating system is 0x30.

5.2 PROCESS MODULE DESIGN

The Process module is one of the two modules that make the Dynacube kernel. The process module is basically used for process manipulation. The process' state information and the system's state information are stored in the process' PCB. The PCB or the Process Control Block has the following structure

```
typedef unsigned char          DB;     // 1 byte
typedef unsigned short         DW;     // 2 byte
typedef unsigned int           DD;     // 4 byte

typedef struct
 {
   boolean avl;
   DW ppid;
   DW pre_task_link;
   DD esp0;
   DW ss0;

   DD esp1;
   DW ss1;

   DD esp2;
   DW ss2;

    DD cr3;
    DD eip;
    DD eflags;
    DD eax;
    DD ecx;
    DD edx;
    DD ebx;
    DD esp;
    DD ebp;
    DD esi;
    DD edi;
    DW es;

   DW cs;
   DW ss;
   DW ds;
   DW fs;
   DW gs;

   DW ldt_sel;
   DW flags;              //T
```

```
  DW io_map;
  DW msg_q_delim;   //Size of q = 2^16
  DD recv_addr;        // To store the address from the recv call when blocked
  DD wait_int_num;   //The interrupt that the process wants to receive
  DD time_out;          //The timeout registered with the TIMER

 } PROC;

//The PROC structure is Dynacube's PCB structure
```

Table 5.2 – PROC Structure

As the structure of the PCB reveals, it contains the saved values of the general-purpose registers, segment registers and control registers (CR3) of the processor. It also contains the various privilege level stacks with their corresponding stack pointers that are used by the processor while switching between various protection levels. Apart from these the PROC (or) PCB structure also contains special kernel-usable information like the msg_q_delim. The usage of these fields will be discussed in the following sections.

The *avl* field is used to check whether a PCB is free to be used or is already in use by some other process. The *ppid* field is used to store the parent process id. The *ldt_sel* field is used for the LDT selector for this process when it will be loaded into a TSS for running.

The *flags* field is used to store TSS specific flags, which can be used to set the TRAP bit and thus enable the process to be debugged after each instruction. This is made to work by interrupting the process after every instruction by calling the DEBUG exception. Thus the kernel could use this flags field to debug user processes.

The *io_map* is used as a pointer to the I/O permission bitmap that is used for allowing (or) disallowing user processes to access specific I/O ports.

The *msg_q_delim* is used as the delimiter in the message queue structure that is used for inter-process communication.

The kernel uses the *recv_addr* when a process blocks for a message from its message queue when its message queue is empty. In this case the receiving address of the process is stored in the *recv_addr* and when a process sends a message to this blocked process we use this field to copy the message from its message queue.

Kernel saves the required IRQ value for which a process might want to be notified in the *wait_int_num.*

The kernel saves the timeout after which a requesting process must be woken up from its suspended state. This value is stored in *time_out*.

5.2.1 Forking

The kernel forks out a process on request. The process of forking out can be described as a series of the following steps:

- The kernel checks out the PCB array for a free PCB that can be used for the new process. If it finds an empty slot it proceeds to the next step. Otherwise the kernel returns FORK_FAILURE error to the invoking process.
- The kernel then allocates free pages to the process by calling the findpage function till the necessary number of pages has been allotted. If the findpage returns valid pages till the last request the kernel proceeds to the next step, otherwise it return FORK_FAILURE to the invoking process. The findpage function returns the page number when it has a free page (or) returns PAGE_NOT_FOUND when it has run out of free pages. The size of each page is 4KB. Thus a program whose binary image size is 200KB needs at least 50 pages.
- The base address of each page is mapped into the process' page directory entries. So that the process will be thinking that it is running in a memory address spanning from 0x0 to MAX_PROC_SIZE. However internally these addresses will be resolved to altogether different memory locations by the MMU (Memory Management Unit) by the using the page directory as translation table.

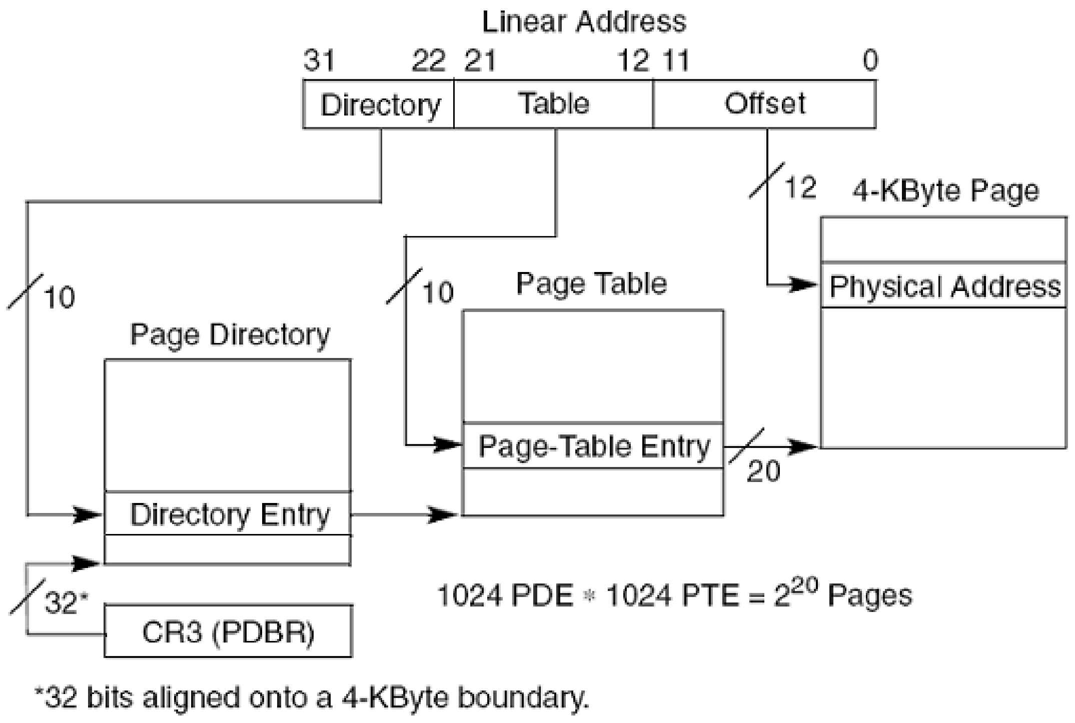


Figure 5.1 – Address Translation

- The kernel sets up the LDT for the newly forked out process and then initializes the segment registers in its PCB so that they point to the entries in the LDT rather than the GDT, which is performed by setting up the Table Indicator bit (TI) in the Segment selectors. This provides better abstraction and protection. The RPL is set to 0x3 so as to run the process in the protection level 3.

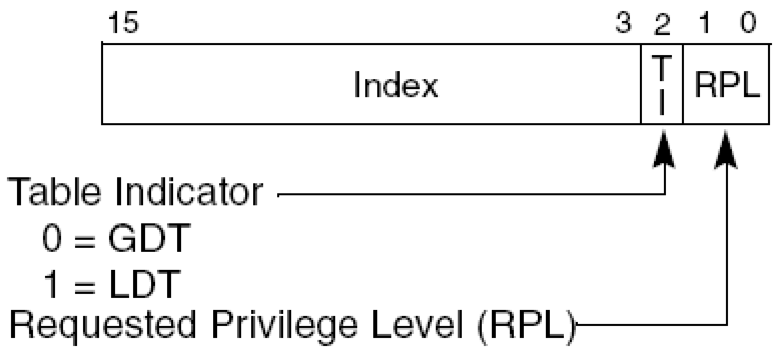


Figure 5.2 – Segment Selector

- The kernel then sets up the Stack frame for the forked out process and initializes the instruction counter. The kernel also initializes the eflags to 0x2|(1<<9). This is done to ensure that the eflags conforms to the Intel specification of the reserved bit and IF flag is set.
- The kernel then loads program’s binary image to the allocated memory by sending a request to the File Server to load the binary file. The File Server then loads the image into memory with the help of the Disk Server. Once the loading is complete the process is removed from the FS queue and added to the ready queue. However if there is a failure then the process is killed and the space allotted to it and its PCB entries are freed so that they can be used in future.

5.2.2 Task Design

The kernel contains three TSS structures – One for the kernel and two for multitasking purposes. The task structure has the following composition:

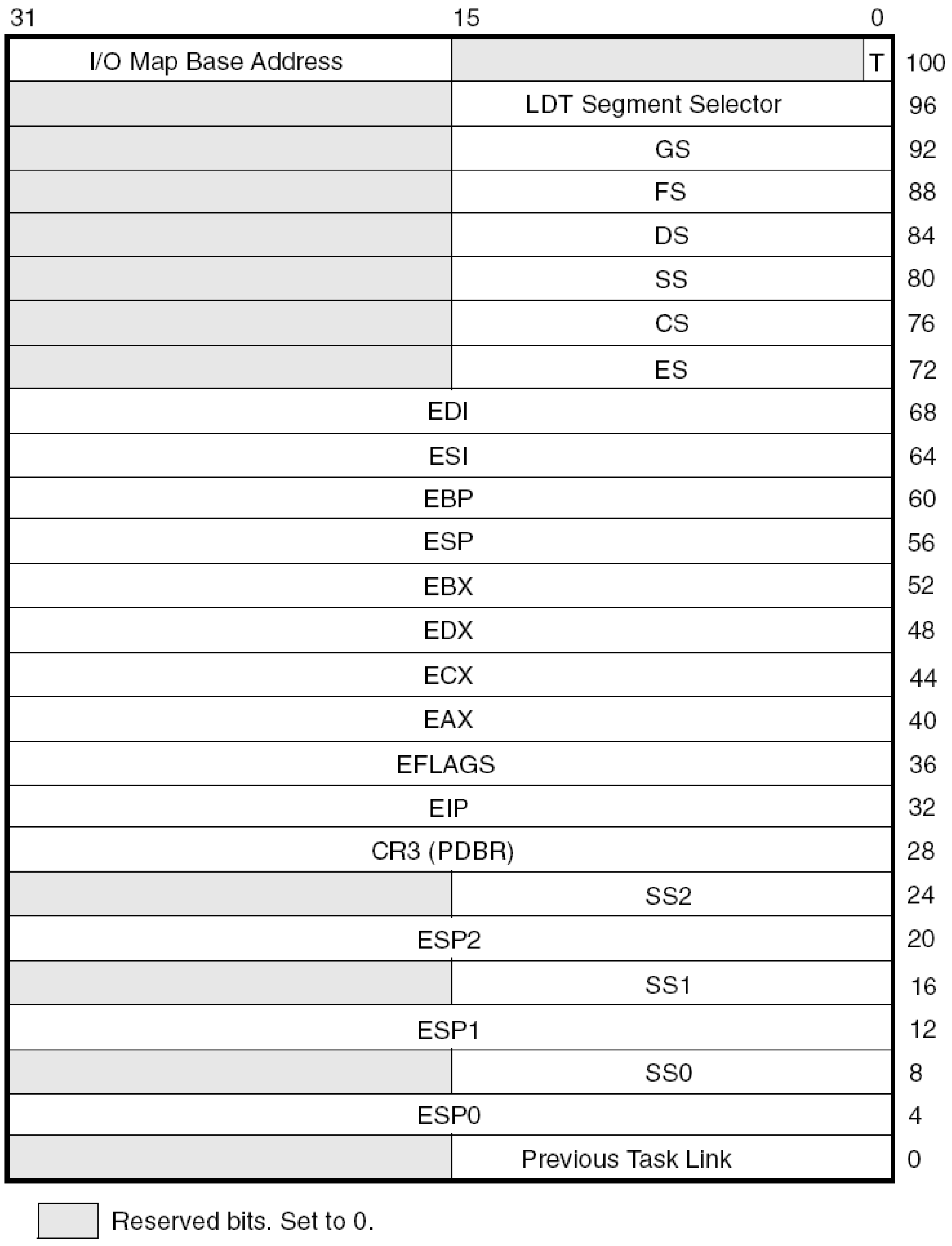


Figure 5.3 – 32-bit Task-State-Segment (TSS)

The processor updates the dynamic fields when a task is suspended during a task switch. The following are dynamic fields:

General-purpose register fields

State of the EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI registers prior to the task switch.

Segment selector fields

Segment selectors stored in the ES, CS, SS, DS, FS, and GS registers prior to the task switch.

EFLAGS register field

State of the EFAGS register prior to the task switch.

EIP (instruction pointer) field

State of the EIP register prior to the task switch.

Previous task link field

Contains the segment selector for the TSS of the previous task. The processor reads the static fields, but does not normally change them. These fields are set up when a task is created. The following are static fields:

LDT segment selector field

Contains the segment selector for the task’s LDT.

CR3 control register field

Contains the base physical address of the page directory to be used by the task.

Privilege level-0, -1, and -2 stack pointer fields

These stack pointers consist of a logical address made up of the segment
selector for the stack segment (SS0, SS1, and SS2) and an offset into the stack
(ESP0, ESP1, and ESP2).

T (debug trap) flag (byte 100, bit 0)

When set, the T flag causes the processor to raise a debug exception when a
task switch to this task occurs.

I/O map base address field

Contains a 16-bit offset from the base of the TSS to the I/O permission bit map
and interrupt redirection bitmap. When present, these maps are stored in the
TSS at higher addresses. The I/O map base address points to the beginning of
the I/O permission bit map and the end of the interrupt redirection bit map.

5.2.3 Twin-TSS based Multitasking

In Dynacube operating system, multitasking is achieved by the usage of Twin-TSS scheduling method and the System Timer. The system timer is responsible for generating timer interrupts at pre-specified intervals of time. The timed interrupts allow the system to transfer the flow control to the kernel's interrupt handling code. This allows the kernel to run multiple processes for fixed time slices in a gyratory fashion – This creates the illusion of concurrent execution of processes.

The switching between processes is especially interesting as we use only two TSS segments in the GDT for multitasking as compared to the other contemporary kernels that use as many TSS as there are processes running under them. Multitasking can be explained as a series of the following steps:

1. The kernel loads the Task Register with the kernels TSS during the system initialization phase.
2. During subsequent calls to the schedule function generally caused by the Timer interrupt the schedule function first synchronizes the PCB of the currently suspended process with the system's state.
3. It then finds the next candidate from the ready queue that is eligible to run. The selected process' PCB is copied to the TSS that was used by the penultimate process. The reason for doing this lies in the fact that a suspended TSS cannot be called recursively. So we use the second TSS and clear the Busy Bit of the first TSS segment.

This cycle between step 2 and 3 continues till the system is shutdown. This concept gives a clean way of implementing scheduling.

```
 which_task = !which_task;

 //Clearing Busy Bit
  t = (SEG_DESC*)(GDT_BASE+SYS_TSS_SEL*8);
  t->flags_8_15 &= ~0x2;
  asm("ltr %%ax"::"a"(SYS_TSS_SEL*8));


if(which_task)
{
 t = (SEG_DESC*)(GDT_BASE+TASK1_TSS_SEL*8);
 t->flags_8_15 &= ~0x2;
 asm("lcall %0,$0"::"i"(TASK1_TSS_SEL*8));
}
else
{
 t = (SEG_DESC*)(GDT_BASE+TASK2_TSS_SEL*8);
 t->flags_8_15 &= ~0x2;
 asm("lcall %0,$0"::"i"(TASK2_TSS_SEL*8));
}
```

Table 5.3 - Code Snippet showing the logic behind Twin-TSS based multitasking

5.2.4 Scheduling Policy

The scheduling routine is responsible for choosing the next candidate from the pool of candidate processes available in the ready queue. The scheduling algorithm used in Dynacube operating system is called the "***Priority based round robin***" algorithm that gives fixed quantity time slices to each of the processes in the ready queue in an order that is dependent on their priority levels. This makes higher priority processes to get the opportunity to run with reduced waiting time as compared to other less privileged processes.

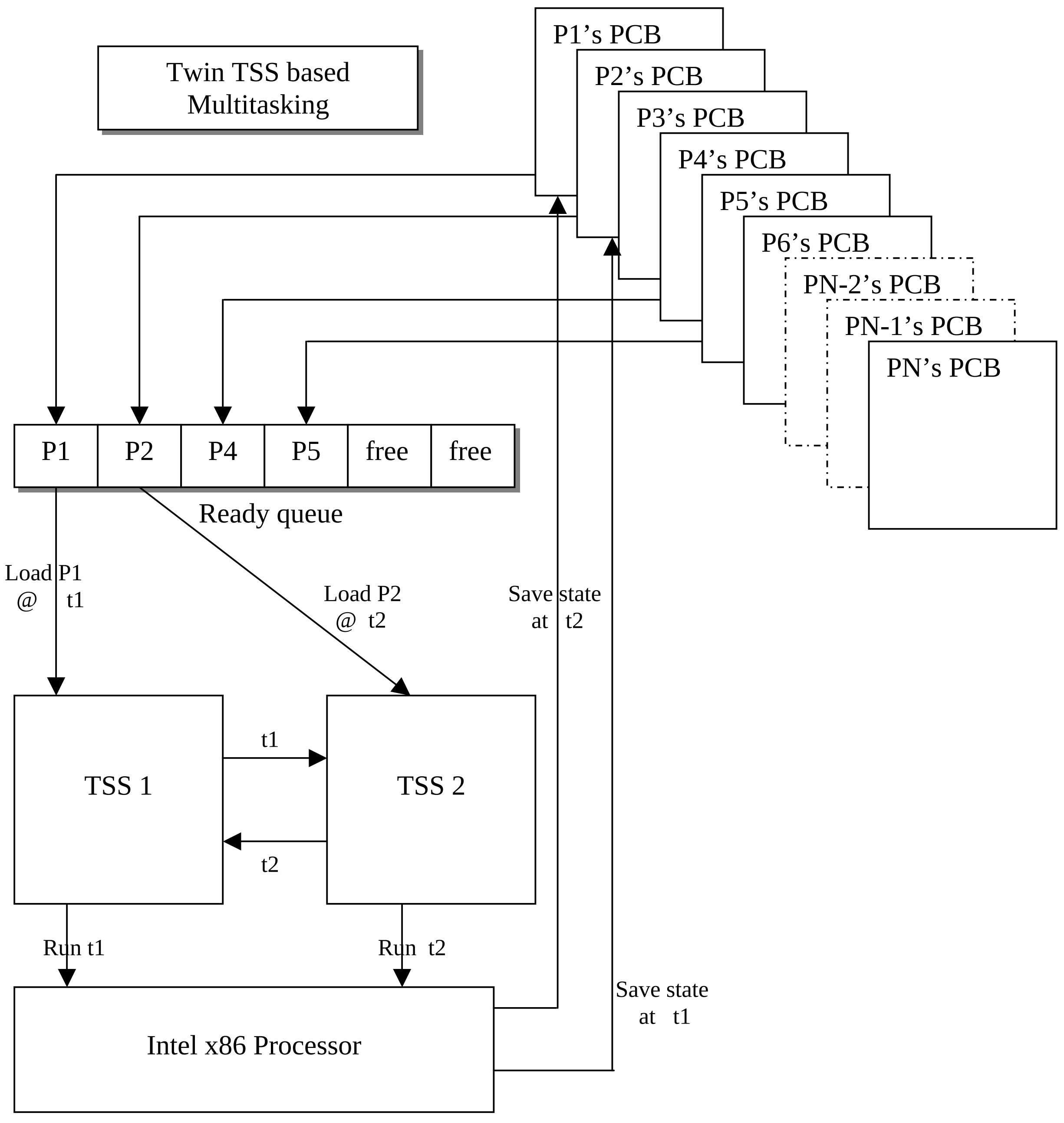


Figure 5.4 Twin-TSS based Multitasking

5.2.5. Queue Design

Dynacube uses the following queues for process manipulation:

- Ready queue – This queue is used to store the process ids of processes that are ready to run, and are waiting for their time-slice.
- Message queue – This queue is used for processes that want to receive messages from other processes when there is no message for this process. In this case the kernel simply suspends the requesting process and removes it from the ready queue and enqueues it in the message queue.
- Timer queue – This queue is used for storing the process ids of processes that request the kernel to be woken up after a given timeout. The kernel removes the process from the ready queue and places it in the timer queue.
- Interrupt queue – This queue is used for process ids of processes that request the kernel to be notified when a specific interrupt occurs that is indicated by the *wait_int_num* field in the PCB structure. The kernel removes it from the ready queue and places it in the timer queue.
- GUI queue – This queue is used for storing process ids from the ready queue for processes that send messages to the GUI Server. This is done primarily to stop the process from doing anything till the GUI Server has completed its previous requests.
- FS queue – This queue is used for storing process ids that send requests to the File Server. The kernel removes the pid from the ready queue and enqueues it in the FS queue. The restoration of the pid to ready queue occurs when the FS Server has completed the request.
- Device Queues – These queues are used for queuing process that request service of certain devices – In Dynacube we use it for blocking processes that request service from the Disk Server, which uses the floppy device.

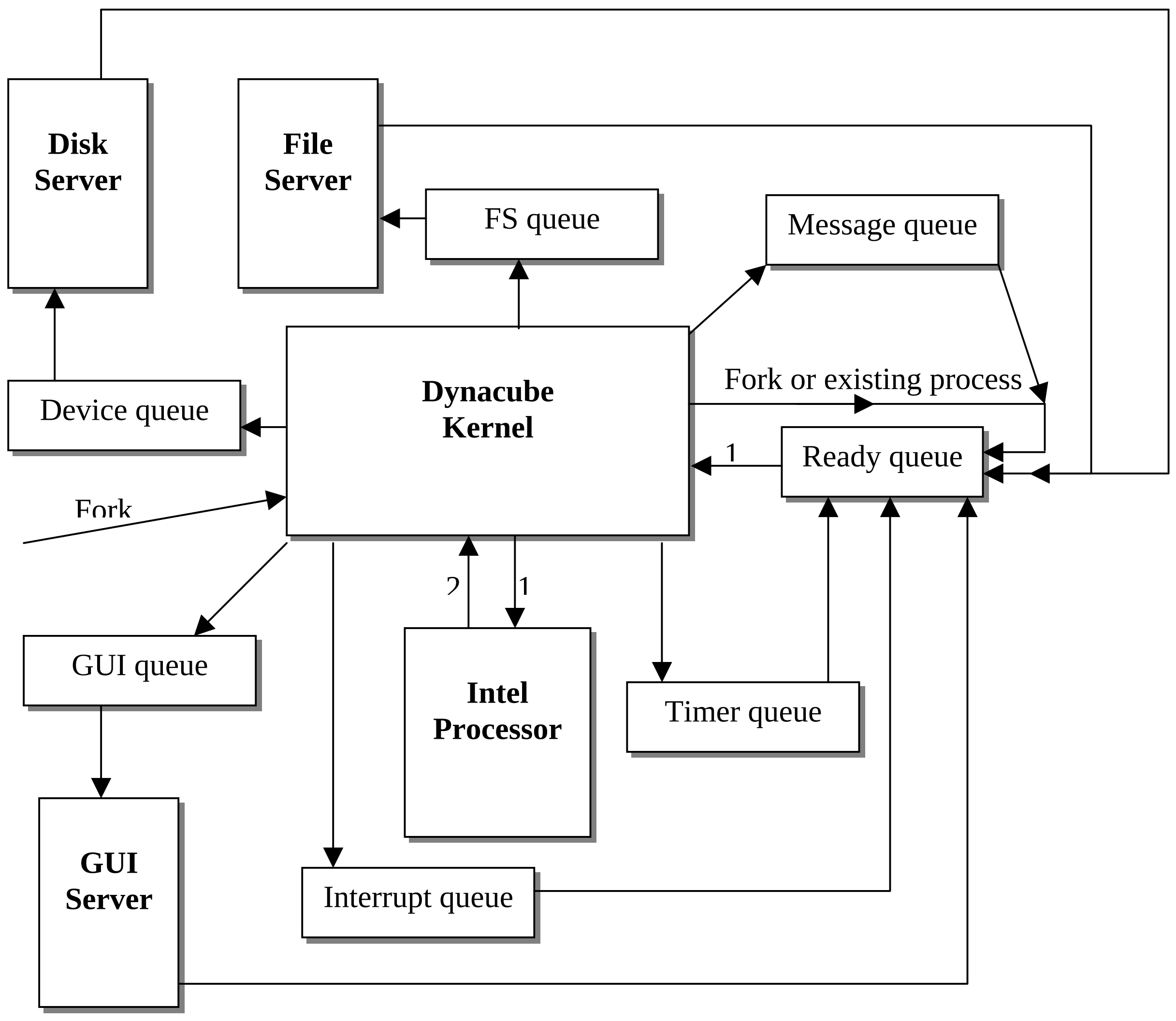


Figure 5.5 - Queue Design

5.2.6 Message Passing Interface

The message-passing interface is one of the most important interfaces that Dynacube kernel provides which is used for inter-process communication. The message-passing interface follows the following rules

1. Asynchronous send - A process can send messages to another process without having to block.
2. Synchronous receive – A process can receive message by using the system call recv. The call is immediately returned if there is message for the requesting process from other processes. However if the message buffer is empty then the requesting process is removed from ready queue and enqueued to the message queue.

The message format: The process must conform to the following format while sending messages.

```
#define MAX_MSG_BUF 64

typedef struct  {
    DW from_pid;
    DW type;
    DW sub_type;
    DD length;
    DB msg_buf[MAX_MSG_BUF];
  }MSG;
```

Table 5.4 – Message Format

The *from_pid* field is to store the process from which the message originated. The kernel however overwrites this value by the sending process' pid. This is done to thwart any attempt by the sending process to fake identity. The *type* and *sub_type* fields have message specific meaning. The *length* field informs the kernel about the size of *msg_buf* utilized. Thus the kernel has to copy only the used part of the message into its address space instead of the entire chunk. This boosts performance.

The kernel actually buffers the received messages of every process till the process does a *recv* to receive the message. The kernel message buffer resides in the ring 0 protected segment. Thus we can effectively safeguard messages from prying processes.

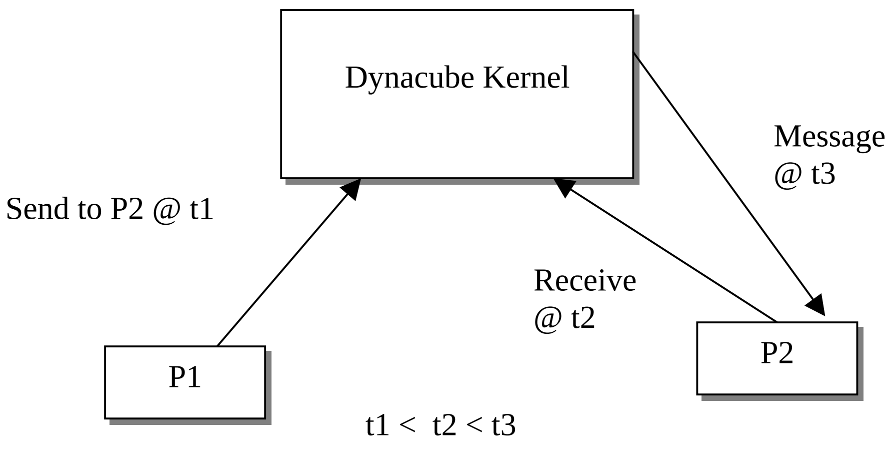


Figure 5.6 - Message Passing – Scenario 1

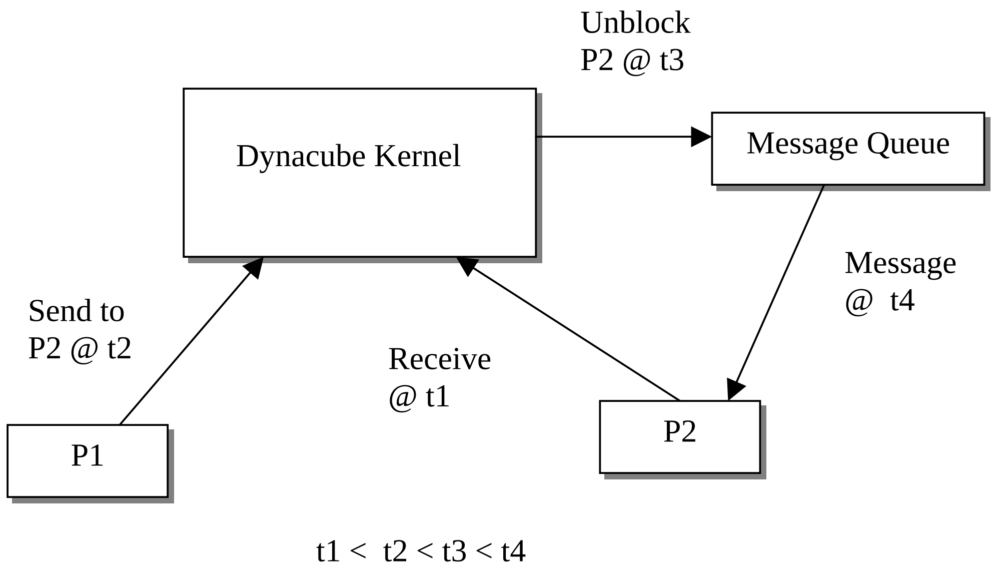


Figure 5.7 - Message Passing – Scenario 2

5.2.7 System Call Interface

Dynacube operating system provides its services via the system call interface which can be accessed by using INT 0x30**.** The system calls provided by Dynacube are

Function No.	Description
0	Exit system call – To kill the invoking process.
1	Fork system call – Used to fork out a new process
3	Send system call – Used to send messages to other processes
4	Receive system call – Used to receive messages from other processes
5	Sleep system call – To request the kernel to wake up the invoking process to be woken up after the specified timeout.
6	Wait for interrupt system call – To get notified by the kernel when a specified interrupt occurs.
7	Wait for an interrupt and also wait for a Timer interrupt system call – To get notified by the kernel when a specified interrupt and the timer interrupt occurs.
8	Read floppy sector system call
9	Write floppy sector system call
11	GUI processing completion system call – Used to remove a blocked process from the gui queue and add it to the ready queue.
12	FS processing completion system call – Used to remove a blocked process from the fs queue and add it to the ready queue.
13	File opening system call.
14	Closes an already open file.
15	Read bytes from an opened file.
16	Write bytes into an opened file.
17	Create a file
18	Open directory
19	Create Directory
20	Close Directory
21	Read Directory

22	Remove file
23	Rename
30	Get System Date
31	Set System Date
32	Get System Time
33	Set System Time

Table 5.5 – System Calls in Dynacube

5.3. Memory Module Design

The main work of the memory module is to provide consistent and protected access to physical memory.

The memory model used by Dynacube operating system is called the *Segmented Paged memory model*. The advantage of this model lies in the fact that it allows complete virtualization of memory along with the protection offered by segmentation. The client process is made to believe that it is executing in a memory space that spans from 0x0 to MAX_PROC_SIZE. In reality this logical address upon translation to linear address, which upon further translation to physical memory gives the real address, which is not from 0x0 to MAX_PROC_SIZE but some other address range.

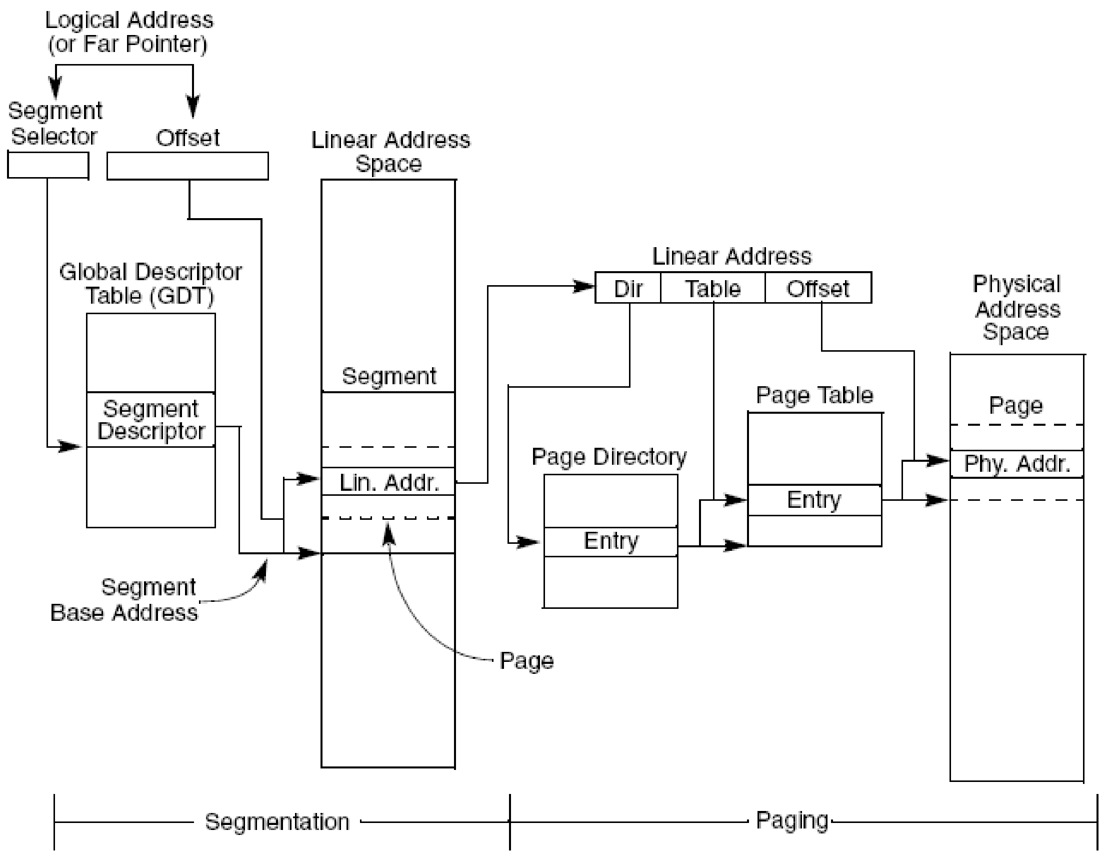


Figure 5.8 – Segmentation and Paging

This illusion is necessary for non-relocatable programs that by default assume that it is running at some predefined location (like 0x0 for plain binary or 0x100 for COM programs). As symbol resolution is impossible in plain binary code it is better to provide the same address in which the program thinks it is executing. However this again leads to another problem – At any given instant only one process can be allowed to occupy that address. This will mean that for every task switch we have to reload the image files to that location, which will be a great performance hindrance. To overcome this obstacle we perform virtualization of memory, which is accomplished by the usage of paging mechanism. The concept is pretty simple we just make the client program work by resolving all memory references by the program and then correctly mapping it to physical memory. The MMU or the Memory Management Unit, which takes care of the address translation, provides this functionality. All that we have to do is load the correct translation tables. These translation tables are called page directories.

In Dynacube we use two page directories. The first page directory is the kernel's page directory that provides a one to one mapping of the entire memory. This is necessary for the kernel to be able to access the entire memory space without generating page faults (#PG). The second page directory is used for client processes – In this page directory we map the kernel's memory space as a one-to-one mapping. However the remaining entries are dynamically mapped using the BIMA Page Allocator. Thus each user process can get a total of 4MB space as we reserve one directory entry for each process.

In general a page directory contains 1024 entries and each entry has in the directory contains the pointer to page tables each of which has 1024 entries. Each entry in the page table contains the address that points to the real base address of a page. Thus a system contains 1024 * 1024 pages which comes to 2^20 pages. As each page is of 4KB size the total memory that can be addressed by this scheme = 4KB * 2^20 = 2^32 or 4GB memory space.

5.3.1 BIMA – Bitmap Memory Allocator

The Bitmap Memory Allocator is the page allocator for Dynacube operating system. The BIMA uses a bitmap representation of the physical memory. In this bitmap a single bit represents a 4KB page. Thus for a given system RAM of N MB the bitmap size is only (N * 32) bytes. Thus for a 256 MB RAM our BIMA requires only 8192 bytes (or) just two 4KB pages.

The BIMA uses the bitmap and a counter called the *pindex,* which is used to point to the recently used byte in the bitmap. The BIMA uses the technique of *Divide and Conquer* to find free pages. It first searches for a DWORD starting from the DWORD pointed by the *pindex* till a DWORD is found that on ANDing with 0xFFFFFFFF doesn't give 0xFFFFFFFF. This means that DWORD has a free page. We try to find the page by first ANDing the lower word with 0xFFFF if that returns 0xFFFF then that word has no free pages. We do this till we get the free page.

BIMA Deallocation

The BIMA's deallocation algorithm is simple. Whenever a request to deallocate a page comes to the BIMA it just clears the corresponding bit in the bitmap. This way it provides a faster way of deallocation as compared to other linked-list based memory management techniques.

The following code snippet shows the single-line deallocation routine: We call it the freepage routine.

```
frmlist[index/32] ^= (1<<index%32);
```

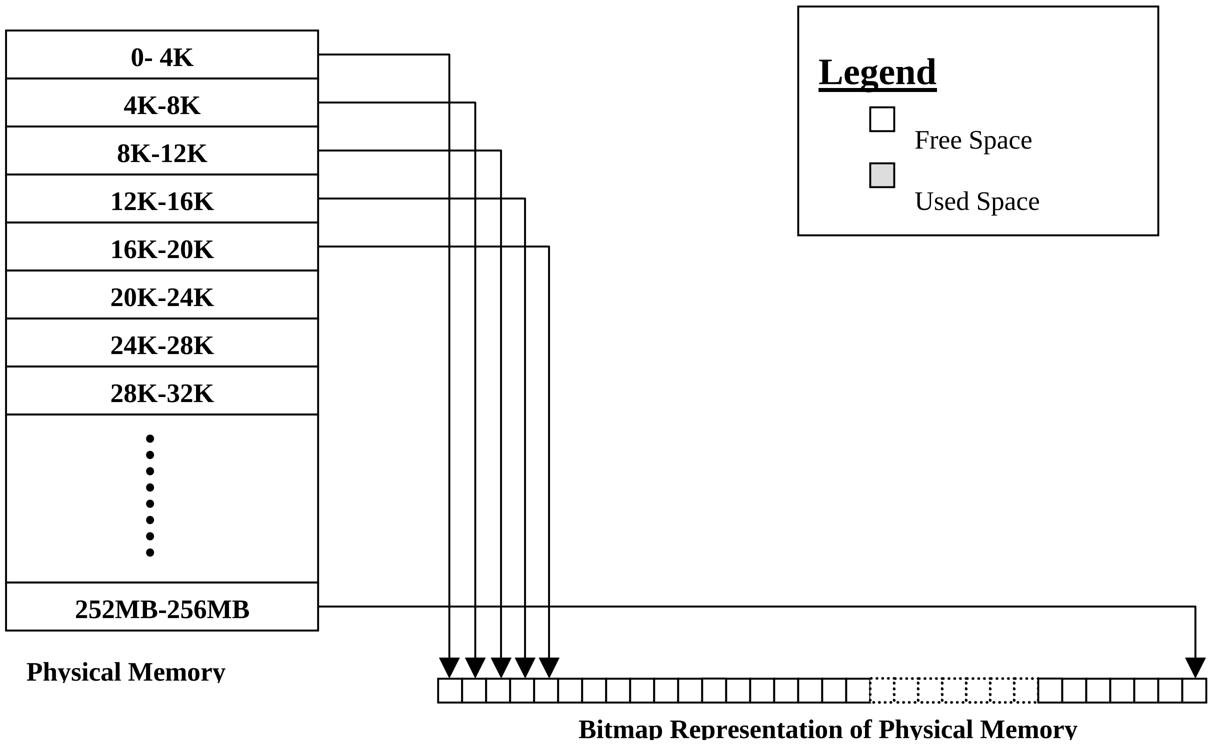


Figure 5.9 - BIMA (BItmap Memory Allocator)

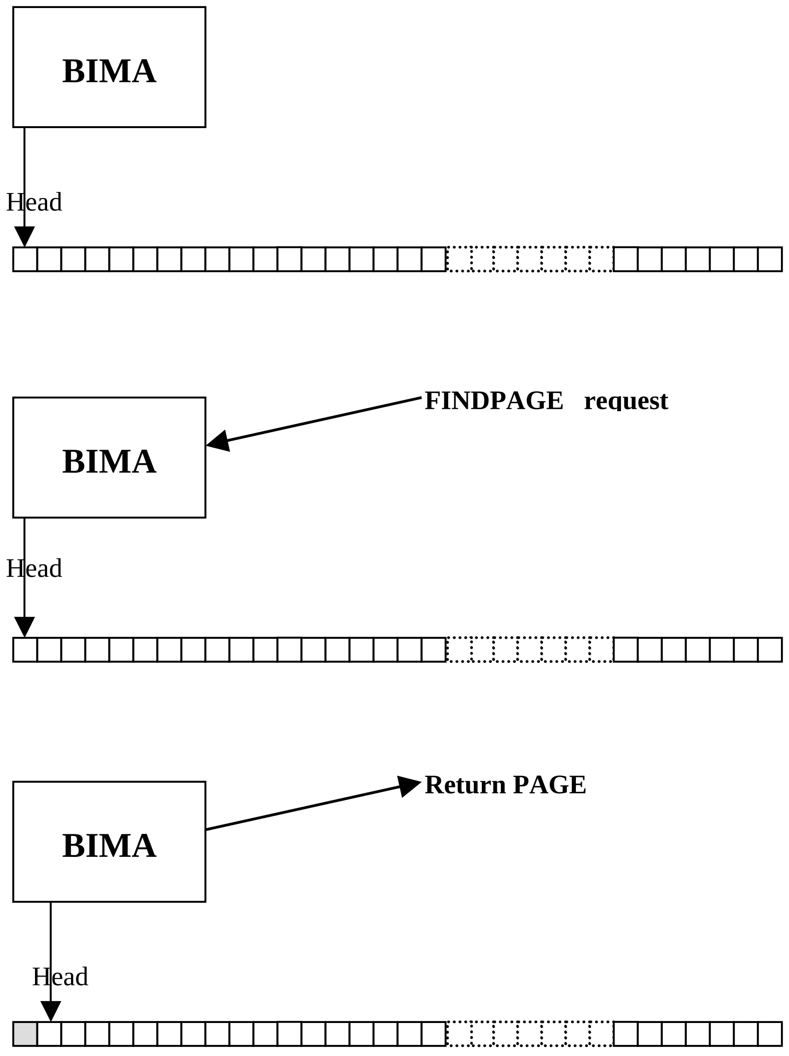


Figure 5.10 BIMA Allocation Scheme

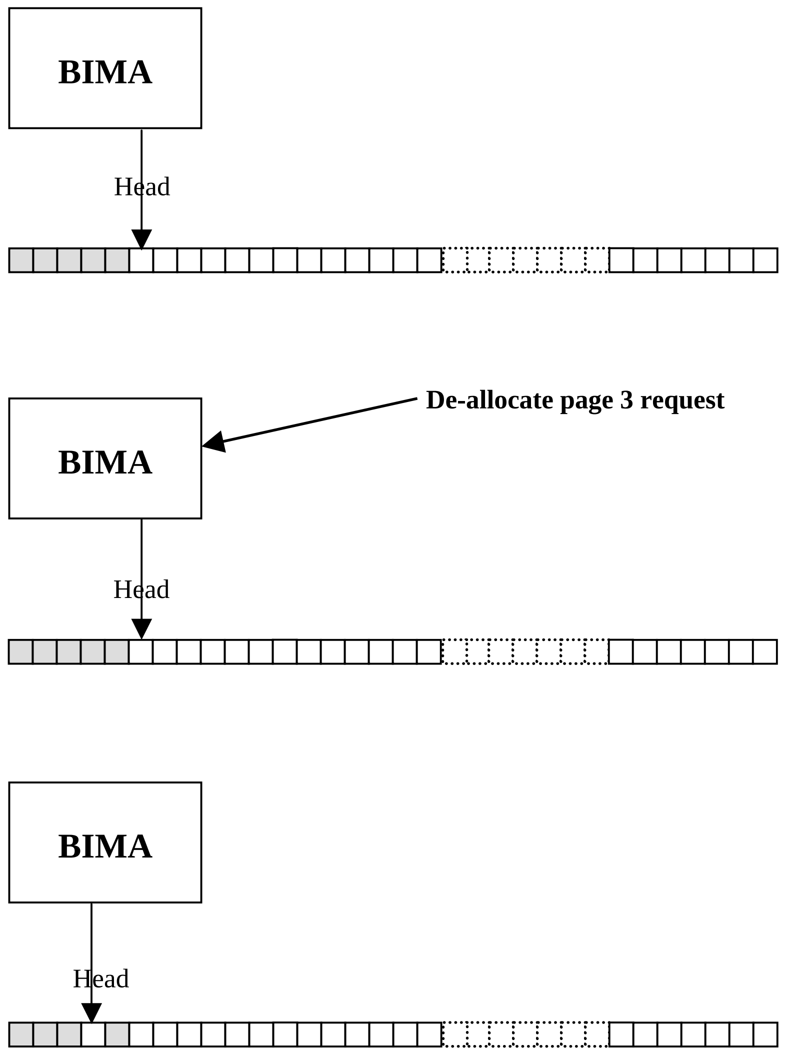


Figure 5.11 BIMA De-allocation Scheme

5.3.2 Variable Memory Chunk Allocator Design

The variable memory chunk allocator has two principal parts – kernel mode memory allocator and user mode memory allocator. The kernel mode memory allocator provides memory by getting free pages from the findpage function. Then it sets the Supervisor bit in the page entry.

In the user mode allocation the memory is obtained from the findpage allocator but the User bit is set in the page entry. And the page is mapped into the address space of the process. This ensures that the client application gets memory within its own addressable space with protection level 3.

5.4 Device Driver Module

5.4.1 Introduction

A device driver is a software layer that lies between the applications and the actual device. A Device Driver is the user's window to access a device. The device driver is developed for the following reasons

- Providing abstraction to the Operating System of the device being used.
- To manage simultaneous access of applications on a device.
- To provide a clean standard interface to a device.

Character devices

A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the *open*, *close*, *read*, and *write* system calls.

Eg: Video driver, keyboard driver, and mouse driver

Block devices

A block device is something that can host a file system, such as a disk. The block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a file system node and the difference between them is transparent to the user.

Eg: Floppy Driver

5.4.2 PS/2 Keyboard Driver

Keyboards consist of a large matrix of keys, all of which are monitored by an on-board processor (called the "keyboard encoder".) The specific processor varies from keyboard-to-keyboard but they all basically do the same thing: Monitor which key(s) are being pressed/released and send the appropriate data to the host. The processor takes care of all the debouncing and buffers any data in its 16-byte buffer, if needed. The motherboard contains a "keyboard controller" that is in charge of decoding all of the data received from the keyboard and informing your software of what's going on. All communication between the host and the keyboard uses an IBM protocol.

Reset:

At power-on or software reset (see the "Reset" command) the keyboard performs a diagnostic self-test referred to as BAT (Basic Assurance Test) and loads the following default values:

- Typematic delay 500 ms.
- Typematic rate 10.9 cps.
- Scan code set 2.
- Set all keys typematic/make/break.

When entering BAT, the keyboard enables its three LED indicators, and turns them off when BAT has completed. At this time, a BAT completion code of either 0xAA (BAT successful) or 0xFC (Error) is sent to the host. This BAT completion code must be sent 500~750 milliseconds after power-on.

Many of the keyboards ignore their CLOCK and DATA lines until *after* the BAT completion code has been sent. Therefore, an "Inhibit" condition (CLOCK line low) may not prevent the keyboard from sending its BAT completion code.

Make Codes, Break Codes, and Typematic Repeat

The *make code* is the code that is sent to the computer when a key is pressed. And a *Break code* is sent to the computer when the key is released. When a key is pressed and held down, that key becomes *typematic*, which means the keyboard will keep sending that key's make code until the key is released or another key is pressed.

Command Set:

A few notes regarding commands the host can issue to the keyboard:

- The keyboard clears its output buffer when it receives any command.
- If the keyboard receives an invalid command or argument, it must respond with "resend" (0xFE).
- The keyboard must not send any scancodes while processing a command.
- If the keyboard is waiting for an argument byte and it instead receives a command, it should discard the previous command and process this new one.

Below are all the commands the host may send to the keyboard:

- 0xFF (Reset) - Keyboard responds with "ack" (0xFA), then enters "Reset" mode. (See "Reset" section.)
- 0xFE (Resend) - Keyboard responds by resending the last-sent byte. The exception to this is if the last-sent byte was "resend" (0xFE). If this is the case, the keyboard resends the last non-0xFE byte.

The next six commands can be issued when the keyboard is in any mode, but it only effects the behavior of the keyboard when in "mode 3" (ie, set to scan code set 3.)

- 0xFD (Set Key Type Make) - Disable break codes and typematic repeat for specified keys. Keyboard responds with "ack" (0xFA), then disables scanning (if enabled) and reads a list of keys from the host. These keys are specified by their set 3 make codes. Keyboard responds to each make code with "ack". Host terminates this list by sending an invalid set 3 make code (eg, a valid command.) The keyboard then re-enables scanning (if previously disabled).
- 0xFC (Set Key Type Make/Break) - Similar to previous command, except this one only disables typematic repeat.
- 0xFB (Set Key Type Typematic) - Similar to previous two, except this one only disables break codes.

- 0xFA (Set All Keys Typematic/Make/Break) - Keyboard responds with "ack" (0xFA). Sets all keys to their normal setting (generate scan codes on make, break, and typematic repeat)
- 0xF9 (Set All Keys Make) - Keyboard responds with "ack" (0xFA). Similar to 0xFD, except applies to all keys.
- 0xF8 (Set All Keys Make/Break) - Keyboard responds with "ack" (0xFA). Similar to 0xFC, except applies to all keys.
- 0xF7 (Set All Keys Typematic) - Keyboard responds with "ack" (0xFA). Similar to 0xFB, except applies to all keys.
- 0xF6 (Set Default) - Load default typematic rate/delay (10.9cps / 500ms), key types (all keys typematic/make/break), and scan code set (2).
- 0xF5 (Disable) - Keyboard stops scanning, loads default values (see "Set Default" command), and waits further instructions.
- 0xF4 (Enable) - Re-enables keyboard after disabled using previous command.
- 0xF3 (Set Typematic Rate/Delay) - Host follows this command with one argument byte that defines the typematic rate and delay as follows:

The Keyboard Controller

The 8042 chip acts as a keyboard controller and the mouse controller in the PS/2 compatible mode.

The 8042 contains the following registers:

- A one-byte input buffer - contains byte read from keyboard; read-only
- A one-byte output buffer - contains byte to-be-written to keyboard; write-only
- A one-byte status register - 8 status flags; read-only
- A one-byte control register - 7 control flags; read/write

The first three registers (input, output, status) are directly accessible via ports 0x60 and 0x64. The last register (control) is read using the "Read Command Byte" command, and written using the "Write Command Byte" command.

The following table shows how the peripheral ports are used to interface the 8042:

Port	Read / Write	Function
0x60	Read	Read Input Buffer
0x60	Write	Write Output Buffer
0x64	Read	Read Status Register
0x64	Write	Send Command

Table 5.6 - 8042 ports and functions

Writing to port 0x64 doesn't write to any specific register, but sends a command for the 8042 to interpret. If the command accepts a parameter, this parameter is sent to port 0x60. Likewise, any results returned by the command may be read from port 0x60.

Status Register:

The 8042's status flags are read from port 0x64. They contain error information, status information, and indicate whether or not data is present in the input and output buffers. The flags are defined as follows:

PS/2-compatible mode:

PERR	TO	MOBF	INH	A2	SYS	IBF	OBF

Figure 5.12 8042's status register

- OBF (Output Buffer Full) - Indicates when it's okay to write to output buffer.
  0: Output buffer empty
  1: Output buffer full
- IBF (Input Buffer Full) - Indicates when input is available in the input buffer.
  0: Input buffer empty - No unread input at port 0x60
  1: Input buffer full - New input can be read from port 0x60

- SYS (System flag) - Post reads this to determine if power-on reset, or software reset.

  0: Power-up value - System is in power-on reset.

  1: BAT code received - System has already been initialized.
- A2 (Address line A2) - Used internally by the keyboard controller

  0: A2 = 0 - Port 0x60 was last written to

  1: A2 = 1 - Port 0x64 was last written to
- INH (Inhibit flag) - Indicates whether or not keyboard communication is inhibited.

  0: Keyboard Clock = 0 - Keyboard is inhibited

  1: Keyboard Clock = 1 - Keyboard is not inhibited
- TxTO (Transmit Timeout) - Indicates keyboard isn't accepting input (kbd may not be plugged in).

  0: No Error - Keyboard accepted the last byte written to it.

  1: Timeout error - Keyboard didn't generate clock signals within 15 ms of "request-to-send".
- RxTO (Receive Timeout) - Indicates keyboard didn't respond to a command (kbd probably broke)

  0: No Error - Keyboard responded to last byte.

  1: Timeout error - Keyboard didn't generate clock signals within 20 ms of command reception.
- PERR (Parity Error) - Indicates communication error with keyboard (possibly noisy/loose connection)

  0: No Error - Odd parity received and proper command response received.

  1: Parity Error - Even parity received or 0xFE received as command response.
- MOBF (Mouse Output Buffer Full) - Similar to OBF, except for PS/2 mouse.

  0: Output buffer empty - Okay to write to auxiliary device's output buffer

  1: Output buffer full - Don't write to port auxiliary device's output buffer
- TO (General Timeout) - Indicates timeout during command write or response. (Same as TxTO + RxTO.)

  0: No Error - Keyboard received and responded to last command.

  1: Timeout Error - See TxTO and RxTO for more information.

Keyboard Controller Commands:

Commands are sent to the keyboard controller by writing to port 0x64. Command parameters are written to port 0x60 after the command is sent. Results are returned on port 0x60. Always test the OBF ("Output Buffer Full") flag before writing commands or parameters to the 8042.

- 0x20 (Read Command Byte) - Returns command byte. (See "Write Command Byte" below).
- 0x60 (Write Command Byte) - Stores parameter as command byte. Command byte defined as follows:

PS/2-compatible mode:

--	XLAT	_EN2	_EN	--	SYS	INT2	INT

Figure 5.13 Command Byte of 8042

- INT (Input Buffer Full Interrupt) - When set, IRQ 1 is generated when data is available in the input buffer.
  0: IBF Interrupt Disabled - You must poll STATUS<IBF> to read input.
  1: IBF Interrupt Enabled - Keyboard driver at software INT 0x09 handles input.
- SYS (System Flag) - Used to manually set/clear SYS flag in Status register.
  0: Power-on value - Tells POST to perform power-on tests/initialization.
  1: BAT code received - Tells POST to perform "warm boot" tests/initialization.
- OVR (Inhibit Override) - Overrides keyboard's "inhibit" switch on older motherboards.
  0: Inhibit switch enabled - Keyboard inhibited if pin P17 is high.
  1: Inhibit switch disabled - Keyboard not inhibited even if P17 = high.

- _EN (Disable keyboard) - Disables/enables keyboard interface.
  0: Enable - Keyboard interface enabled.
  1: Disable - All keyboard communication is disabled.
- PC ("PC Mode") - Enables keyboard interface somehow???
  0: Disable
  1: Enable
- XLAT (Translate Scan Codes) - Enables/disables translation to set 1 scan codes.
  0: Translation disabled - Data appears at input buffer exactly as read from keyboard
  1: Translation enabled - Scan codes translated to set 1 before put in input buffer
- INT2 (Mouse Input Buffer Full Interrupt) - When set, IRQ 12 is generated when mouse data is available.
  0: Auxiliary IBF Interrupt Disabled
  1: Auxiliary IBF Interrupt Enabled
- _EN2 (Disable Mouse) - Disables/enables mouse interface.
  0: Enable - Auxiliary PS/2 device interface enabled
  1: Disable - Auxiliary PS/2 device interface disabled

- 0x90-0x9F (Write to output port) - Writes command's lower nibble to lower nibble of output port (see Output Port definition.)
- 0xA1 (Get version number) - Returns firmware version number.
- 0xA4 (Get password) - Returns 0xFA if password exists; otherwise, 0xF1.
- 0xA5 (Set password) - Set the new password by sending a null-terminated string of scan codes as this command's parameter.
- 0xA6 (Check password) - Compares keyboard input with current password.
- 0xA7 (Disable mouse interface) - PS/2 mode only. Similar to "Disable keyboard interface" (0xAD) command.
- 0xA8 (Enable mouse interface) - PS/2 mode only. Similar to "Enable keyboard interface" (0xAE) command.
- 0xA9 (Mouse interface test) - Returns 0x00 if okay, 0x01 if Clock line stuck low, 0x02 if clock line stuck high, 0x03 if data line stuck low, and 0x04 if data line stuck high.

- 0xAA (Controller self-test) - Returns 0x55 if okay.
- 0xAB (Keyboard interface test) - Returns 0x00 if okay, 0x01 if Clock line stuck low, 0x02 if clock line stuck high, 0x03 if data line stuck low, and 0x04 if data line stuck high.
- 0xAD (Disable keyboard interface) - Sets bit 4 of command byte and disables all communication with keyboard.
- 0xAE (Enable keyboard interface) - Clears bit 4 of command byte and re-enables communication with keyboard.
- 0xAF (Get version)
- 0xC0 (Read input port) - Returns values on input port (see Input Port definition.)
- 0xC1 (Copy input port LSn) - PS/2 mode only. Copy input port's low nibble to Status register (see Input Port definition)
- 0xC2 (Copy input port MSn) - PS/2 mode only. Copy input port's high nibble to Status register (see Input Port definition.)
- 0xD0 (Read output port) - Returns values on output port (see Output Port definition.)
- 0xD1 (Write output port) - Write parameter to output port (see Output Port definition.)
- 0xD2 (Write keyboard buffer) - Parameter written to input buffer as if received from keyboard.
- 0xD3 (Write mouse buffer) - Parameter written to input buffer as if received from mouse.
- 0xD4 (Write mouse Device) - Sends parameter to the auxiliary PS/2 device.
- 0xE0 (Read test port) - Returns values on test port (see Test Port definition.)
- 0xF0-0xFF (Pulse output port) - Pulses command's lower nibble onto lower nibble of output port (see Output Port definition.

5.4.3 PS/2 Mouse Driver

Inputs, Resolution, and Scaling:

The standard PS/2 mouse supports the following inputs: X (right/left) movement, Y (up/down) movement, left button, middle button, and right button. The mouse reads these inputs at a regular frequency and updates various counters and flags to reflect movement and button states.

The standard mouse has two counters that keep track of movement: the X-movement counter and the Y-movement counter. These are 9-bit 2's complement values and each has an associated overflow flag. Their contents, along with the state of the three mouse buttons, are sent to the host in the form of a 3-byte movement data packet (as described in the next section.) The movement counters represent the amount of movement that has occurred since the last movement data packet was sent to the host.

When the mouse reads its inputs, it records the current state of its buttons, then checks for movement. If movement has occurred, it increments (for +X or +Y movement) or decrements (for -X or -Y movement) its X and/or Y movement counters. If either of the counters has overflowed, it sets the appropriate overflow flag.

Movement Data Packet:

The standard PS/2 mouse sends movement (and button) information to the host using the following 3-byte packet:

Y overflow	X overflow	Y sign bit	X sign bit	Always 1	Middle Btn	Right Btn	Left Btn
X Movement							
Y Movement							

Figure 5.14 Mouse movement byte

The movement counters are 9-bit 2's complement integers, where the most significant bit appears as a sign bit in Byte 1 of the movement data packet. These counters are updated when the mouse reads its input and finds movement has occurred. Their value is the amount of movement that has occurred since the last movement data packet was sent to the host (i.e., after a packet is sent to the host, the movement counters are reset.) The range of values that can be expressed by the movement counters is -255 to +255. If this range is exceeded, the appropriate overflow bit is set and the counter is not incremented/decremented until it is reset.

Modes of Operation:

Data reporting is handled according to the mode in which the mouse is operating. There are four standard modes of operation:

- *Reset* - The mouse enters Reset mode at power-up or after receiving the "Reset" (0xFF) command.
- *Stream* - This is the default mode (after Reset finishes executing) and is the mode in which most software uses the mouse. If the host has previously set the mouse to Remote mode, it may re-enter Stream mode by sending the "Set Stream Mode" (0xEA) command to the mouse.
- *Remote* - Remote mode is useful in some situations and may be entered by sending the "Set Remote Mode" (0xF0) command to the mouse.
- *Wrap* - This mode isn't particularly useful except for testing the connection between the mouse and its host. Wrap mode may be entered by sending the "Set Wrap Mode" (0xEE) command to the mouse. To exit Wrap mode, the host must issue the "Reset" (0xFF) command or "Reset Wrap Mode" (0xEC) command. If the "Reset" (0xFF) command is received, the mouse will enter Reset mode. If the "Reset Wrap Mode" (0xEC) command is received, the mouse will enter the mode it was in prior to Wrap Mode.

Modes of operation

Reset Mode:

The mouse enters reset mode at power-on or in response to the "Reset" (0xFF) command. After entering this mode, the mouse performs a diagnostic self-test referred to as BAT (Basic Assurance Test) and sets the following default values:

- Sample Rate - 100 samples/sec
- Resolution - 4 counts/mm
- Scaling - 1:1
- Data Reporting Disabled

It then sends a BAT completion code of either 0xAA (BAT successful) or 0xFC (Error). If the host receives a response other than 0xAA, it may cycle the mouse's power supply, causing the mouse to reset and re-execute its BAT.

Following the BAT completion code (0xAA or 0xFC), the mouse sends its device ID of 0x00. This distinguishes it from a keyboard, or a mouse in an extended mode. I have read documents that say the host is not *supposed* to transmit any data until it receives a device ID. However some BIOS's will send the "Reset" (0xFF) command immediately following the 0xAA received after a power-on reset.

After the mouse has sent its device ID to the host, it will enter Stream Mode. Note that one of the default values set by the mouse is "Data Reporting Disabled". This means the mouse will not send any movement data packets to the host until the "Enable Data Reporting" (0xF4) command is received.

Stream Mode:

In stream mode, the mouse sends movement data when it detects movement or a change in state of one or more mouse buttons. The maximum rate at which this data reporting may occur is known as the *sample rate*. This parameter ranges from 10 samples/sec to 200 samples/sec. Its default value is 100 samples/sec and the host may change that value by using the "Set Sample Rate" (0xF3) command. Stream mode is the default mode of operation.

Remote Mode:

In this mode, the mouse reads its inputs and updates its counters/flags at the current sampling rate, but it only notifies the host of movement (and change in button state) when that information is requested by the host. The host does this by issuing the "Read Data" (0xEB) command. After receiving this command, the mouse will send a movement data packet, and reset its movement counters.

Wrap Mode:

This is an "echoing" mode in which every byte received by the mouse is sent back to the host. Even if the byte represents a valid command, the mouse will not respond to that command--it will only echo that byte back to the host. There are two exceptions to this: the "Reset" (0xFF) command and "Reset Wrap Mode" (0xEC) command. The mouse treats these as valid commands and does not echo them back to the host.

Command Set:

The following are the only commands that may be sent to the mouse... If the mouse is in Stream mode, the host should disable data reporting (command 0xF5) before sending any other commands...

- 0xFF (Reset) - The mouse responds to this command with "acknowledge" (0xFA) then enters Reset Mode.
- 0xFE (Resend) - The host sends this command whenever it receives invalid data from the mouse. The mouse responds by resending the last packet it sent to the host. If the mouse responds to the "Resend" command with another invalid packet, the host may either issue another "Resend" command, issue an "Error" command, cycle the mouse's power supply to reset the mouse, or it may inhibit communication (by bringing the Clock line low). The action taken depends on the host.
- 0xF6 (Set Defaults) - The mouse responds with "acknowledge" (0xFA) then loads the following values: Sampling rate = 100, Resolution = 4 counts/mm, Scaling = 1:1, Disable Data Reporting. The mouse then resets its movement counters and enters stream mode.
- 0xF5 (Disable Data Reporting) - The mouse responds with "acknowledge" (0xFA) then disables data reporting and resets its movement counters. This only effects data reporting in Stream mode and does not disable sampling. Disabled stream mode functions the same as remote mode.
- 0xF4 (Enable Data Reporting) - The mouse responds with "acknowledge" (0xFA) then enables data reporting and resets its movement counters. This command may be issued while the mouse is in Remote Mode (or Stream mode), but it will only affect data reporting in Stream mode.
- 0xF3 (Set Sample Rate) - The mouse responds with "acknowledge" (0xFA) then reads one more byte from the host. The mouse saves this byte as the new sample rate. After receiving the sample rate, the mouse again responds with

"acknowledge" (0xFA) and resets its movement counters. Valid sample rates are 10, 20, 40, 60, 80, 100, and 200 samples/sec.

- 0xF2 (Get Device ID) - The mouse responds with "acknowledge" (0xFA) followed by its device ID (0x00 for the standard PS/2 mouse.) The mouse should also reset its movement counters.
- 0xF0 (Set Remote Mode) - The mouse responds with "acknowledge" (0xFA) then resets its movement counters and enters remote mode.
- 0xEE (Set Wrap Mode) - The mouse responds with "acknowledge" (0xFA) then resets its movement counters and enters wrap mode.
- 0xEC (Reset Wrap Mode) - The mouse responds with "acknowledge" (0xFA) then resets its movement counters and enters the mode it was in prior to wrap mode (Stream Mode or Remote Mode.)
- 0xEB (Read Data) - The mouse responds with acknowledge (0xFA) then sends a movement data packet. This is the only way to read data in Remote Mode. After the data packets have been successfully sent, it resets its movement counters.
- 0xEA (Set Stream Mode) - The mouse responds with "acknowledge" then resets its movement counters and enters stream mode.
- 0xE9 (Status Request) - The mouse responds with "acknowledge" then sends the following 3-byte status packet (then resets its movement counters.):

Always 0	Mode	Enable	Scaling	Always 0	Left Btn	Middle Btn	Right Btn
Resolution							
Sample Rate							

Figure 5.15 Mouse Status byte

Right, Middle, Left Btn = 1 if button pressed; 0 if button is not pressed.
Scaling = 1 if scaling is 2:1; 0 if scaling is 1:1. (See commands 0xE7 and 0xE6)
Enable = 1 if data reporting is enabled; 0 if data reporting is disabled. (See commands 0xF5 and 0xF4)
Mode = 1 if Remote Mode is enabled; 0 if Stream mode is enabled. (See commands 0xF0 and 0xEA)

- 0xE8 (Set Resolution) - The mouse responds with acknowledge (0xFA) then reads one byte from the host and again responds with acknowledge (0xFA) then resets its movement counters. The byte read from the host determines the resolution as follows:

Byte Read from Host	Resolution
0x00	1 count/mm
0x01	2 count/mm
0x02	4 count/mm
0x03	8 count/mm

Table 5.7 – PS/2 Mouse Movement Resolution

- 0xE7 (Set Scaling 2:1) - The mouse responds with acknowledge (0xFA) then enables 2:1 scaling (discussed earlier in this document.)
- 0xE6 (Set Scaling 1:1) - The mouse responds with acknowledge (0xFA) then enables 1:1 scaling (discussed earlier in this document.)

Mouse and Keyboard Driver design and implementation

The keyboard driver initializes the controller so as to use a specific scan set (XT/AT/PS2) and enables the keyboard IRQ.

The kernel gets all hardware interrupts after it sets the IF flag in the EFLAGS register. The PIC is responsible for notifying the kernel about the occurrence of external hardware interrupts. Thus when a key is pressed the keyboard controller notifies the PIC about the event and it is passed to the kernel as INT 0x21. The interrupt handler passes the control to the keyboard driver, which reads data from port 0x60 after checking the OBF bit in the 8042’s status register. It then decodes the code as per the current scan set and forwards the decoded data to the GUI Server for further processing. This transfer of data is completely transparent to the user and thus giving the user an illusion of having directly sent the data to the Graphical User Interface.

The mouse driver has to first initialize the mouse before the mouse can send data to the system. The reason behind this is that the mouse by default enters the Reset mode after the system boots up. This causes it to inhibit its data reporting functionality. Thus the mouse driver does the following to make the mouse report the data.

- Perform Controller Self Test
- Enable PS/2 I/F
- Send Reset 2 Mouse
- Enable Stream Mode
- Enable Data Reporting
- Enable Mouse IRQ

Once the mouse is initialized it can generate interrupts whenever a mouse event occurs. The movement bytes are stored in the output buffer of the keyboard controller's port 0x60 and the OBF and MOBF bits are set. The mouse driver upon invocation by the interrupt handler retrieves the data from the 8042's data port. Once the movement data has been removed the OBF and MOBF bits are cleared by the controller.

The x and y movement data is sent as 9-bit 2's complement form. Thus we use the following algorithm to convert it to usable form.

1. If the status byte contains the sign bit for x movement
   - Then the x data byte is bit-negated and 1 is added to it to the result.
2. If the status byte contains the sign bit for y movement
   - Then the y data byte is bit-negated and 1 is added to it to the result.

Table 5.8 – Algorithm for 2's complement conversion

The mouse driver after retrieving the data from the 8042's output buffer converts it to a usable format by using the first byte. The first byte contains information about which mouse button is held down, and the sign bit for x movement and y movement. The first byte also contains information about the overflow of x and y movement. In case of an overflow the mouse driver reinitializes the mouse so as to smoothen the functioning of the mouse.

Once the mouse driver has a usable movement packet it sends information to the GUI server about the mouse movement. The GUI Server uses this information to provide GUI handling like clicking a window or a button on the screen.

5.4.4 SVGA Video driver

This module acts as an interface between the video controller and the GUI server. It uses the VESA 2.0 standard for Mode Info Retrieval and Setting. The new Linear Frame Buffer (LFB) model is used. After setting of the mode, the drawing is done using double buffering method. It also provides certain primitive graphical library functions. It also deals with the font map and mouse cursor creation.

VBE functions for Mode Setting and Retrieval

VBE Mode Number

VBE mode numbers are 15 bits wide. It has a specific format. The format of VBE mode numbers is as follows:

D0-D8= Mode number

D9-D10 = Reserved (must be 0)

D11 = 0 Use current default refresh rate

= 1 Use user specified CRTC values for refresh rate

D12-13 Reserved for VBE/AF (must be 0)

D14 = 0 Use windowed frame buffer model

= 1 Use linear/flat frame buffer model

D15 = 0 Clear display memory

= 1 Don't clear display memory

The mode number that is used by this video driver is 114h. It is the 800 x 600 64K (5:6:5) graphical mode. The D11 bit is not as the default refresh rate is used. The Linear frame buffer model is used. So the D14 bit is set. The display memory is cleared and so the D15 bit is not set. So, the mode number is 4114h.

VBE Mode Information Retrieval

The VBE function 01h is used for getting information about the mode. The input is given in the following registers:

AX - 4F01h Return VBE mode information

CX - Mode number

ES:DI - Pointer to ModeInfoBlock structure (256-byte buffer)

The output is obtained in AX, which has the VBE Return Status. This function fills the mode information block, ModeInfoBlock, structure with details on the requested mode.

The important portions of this structure are:

1. **ModeAttributes –** This is used for checking whether this mode is supported in hardware. If the D0 bit is set, then the mode is supported in hardware. If the D7 bit is set, then the linear frame buffer mode is supported.
2. **Xresolution** and **Yresolution**
3. **BitsPerPixel**
4. **PhysBasePtr –** This is a 32-bit physical address of the start of frame buffer memory when the controller is in flat frame buffer memory mode. If this mode is not available, then this field will be zero. The driver maps uses direct memory mapping for the LFB address.

VBE Mode Setting

The VBE function 02h is used for setting a particular mode. The input is given in the following registers:

AX	- 4F02h Set VBE Mode
BX	- Desired Mode to set (4114h)
ES:DI	- Pointer to CRTCInfoBlock structure (0:0)

The output is obtained in the AX register as the VBE Return Status. If the VBE function completed successfully, 00h is returned in the AH register. Otherwise the AH register is set to indicate the nature of the failure.

Linear Frame Buffer Model

Once the graphics hardware has been initialized into a mode that supports a hardware linear frame buffer, and then there is no need to use the previous bank-switching model. The pixels can be read and written directly through this address. The steps involved in using this are:

1. Get the mode information for the particular mode and check if there is support for the linear frame buffer model
2. If so, get that address. The linear frame buffer location is a *physical* memory address. It can't be used directly.
3. So, the particular virtual memory region is **directly mapped** to the same physical region.

4. After that, there is no need to access the hardware. The data to be displayed can be directly written to this address.

Double Buffering

Using the linear frame buffer mode, if the data is directly written to the specified address on a bit-by-bit basis, then there is a flickering effect. In order to avoid this, double buffering is used. The data is written first in the secondary buffer and as and when required it is written to the actual frame address. As a result, the flickering effect is removed.

Primitive Graphical Library Functions

The driver also provides certain primitive graphics methods. It also has a default font for the printable characters. The font is maintained as a three-dimensional array and the character width and height is 5x7. The mouse cursor is also embedded. The various functions are:

1. **initGraphics –** This sets the x-resolution and y-resolution, linear frame buffer address to be used.
2. **setPixel** – Sets a pixel when given the x and y position. It writes only to the secondary buffer.
3. **getPixel –** Gets the pixel at a particular location.
4. **getImage –** Gets the image that is present in a specified rectangular region into a buffer.
5. **putImage –** Sets the specified rectangular region with the data from the buffer.
6. **drawLine** – Draws a line from the one point to another.
7. **drawRect** - Draws a rectangular box given the left point, width and height.
8. **drawCircle –** Draws a circle given the center and the radius

5.4.5 Floppy Device Driver

This module deals with interfacing with the floppy disk controller(*NEC µPD765)* to provide an interface to the higher-level modules such as the file system module. The various functions that are to be performed are motor handling, seeking and recalibration, reading and writing of sectors. It also deals with the floppy interrupt handling. The floppy runs as a separate process receiving requests from other processes and performs the requests. It uses DMA for the data transfer. It provides two system calls for reading and writing sectors.

Basics of Floppy drive

Conventional floppy drives contain the following basic components: A *spindle clamping mechanism* to hold the diskette in place as it spins; Either *one or two magnetic read/write heads* mounted on a mechanism that moves the heads radially across the diskette's surface; and A *magnetic sensor* that detects the rotational position of the diskette via an index hole on floppy disks.

When the computer system needs to access data on the diskette, the read/write heads are stepped by signals generated by the computer system's floppy controller. These steps are along invisible concentric cylinders, which are usually referred to as **tracks**. As the computer system's power is first turned on, the *read/write heads* of the drive are automatically set to track 0 (the first track and starting position). In most drives, this starting position is located by means of a sensor in the drive, which has been adjusted to tell the floppy controller when the heads have reached the first track. If this sensor is not in proper adjustment, then this initial starting calibration is also incorrect and the heads are not properly positioned over track 0. In order to move the heads from this first track to other tracks, the head pin simply moves in or out one track for each step pulse received from the computer's floppy controller.

The floppy drive blindly accepts these pulses and *assume*s that it is positioned directly over the proper specified track. It has no accurate feedback mechanism from the diskette concerning whether or not the heads are properly positioned. This differs significantly from many hard drives employing servo systems, which constantly monitor exact head position over each track and make very small and almost instantaneous corrections automatically before performing a read or write operation. This feedback is generated by positioning signals pre-recorded on the hard disk's surface. Since common floppy drives are designed without a positional feedback mechanism, they are referred to as **open-loop** whereas these hard drives are referred to as **closed-loop**. Since common floppy drives don't have a sophisticated closed loop system, they must be carefully aligned in order to ensure the very important ability of reliably exchanging data diskettes with other drives.

It is quite possible for the head pin to become out of alignment in such a way that the read/write head is only over a portion of a track. This reduces the strength of the data signal detected by the head and may also cause unwanted interference between adjacent tracks, or incomplete erasure when data fields are re-recorded during a disk save. Should the relative alignment of the drive that recorded the diskette's data and that of the drive reading the same diskette differ substantially, it will be difficult or impossible to read the recorded data. This undesirable condition is known as **radial misalignment**. In addition to radial misalignment, data errors may also be caused by the heads being rotated slightly on their *axis* (azimuth) or the drive's index sensor being out of position.

There are a lot of delays involved in communicating with the controller. These delays are for a variety of reasons, including the time needed to spin up the drive motor, and the time taken to move the head to a new position and wait for it to settle in place. When the drive motor is started up or seek is requested, there will be a delay until the drive is ready for the next command. An interrupt is issued by the hardware when it is ready for the next command.

In a single tasked environment, the only option is to have the driver constantly wait for an interrupt, and then respond to it. However, in a multi-tasked or multi-threaded environment, it is perfectly acceptable to write a driver which allows other tasks to be executed while waiting for the interrupt.

When performing a read or write operation, data may be transferred a byte at a time by reading from or writing to the appropriate port, or a sector/track at a time through the use of DMA channel *2*.

Floppy Controller Specification

The *floppy disk controller* (FDC) is capable of managing up to 4 separate drives and provides a number of registers and commands that may be used to execute a variety of operations on a specific drive. The base port address used for the controller is dependant on whether the controller is configured as the primary or secondary controller. This base address controls the port addresses used for each of the registers on the controller.

Registers Of FDC	**Primary Address**	**Secondary Address**	**Write (W) Read (R)**
Base address	3f0h	370h	
Status register A (PS/2)	3f0h	370h	R
Status register B (PS/2)	3f1h	371h	R
Digital output register DOR	3f2h	372h	W
Main status register	3f4h	374h	R
Data rate select register (DSR)(PS/2)	3f4h	374h	W
Data register	3f5h	375h	R/W
Digital input register DIR (AT)	3f7h	377h	R
Configuration control register (AT)	3f7h	377h	W

Table 5.9 – Ports of FDC Registers

Floppy Drive Controller Registers

There are some registers, which are common for all systems and some that are AT specific, and some are PS/2 Specific.

Digital Output Register DOR

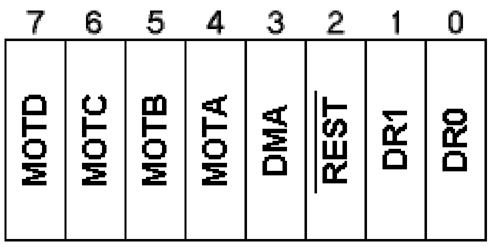


Figure 5.16 – DOR

This register is write only, and controls the drive motors, as well as selecting a drive and the DMA/IRQ mode, and resetting the controller. MOTD, MOTC, MOTB, MOTA control the motor for floppy drive D, C, B, A. If it is set, it corresponds to starting the drive, or else it corresponds to stopping the drive. Setting the DMA bit , enables the usage of the DMA and IRQ channel. If not, the polling method of transfer is used. If the REST bit is set, the controller is enabled, in order to accept and execute commands. If it is equal to 0, the controller ignores all commands and carries out an internal reset of all internal registers (except the DOR). The DR1, DR0 bits are used for Drive select. 00 for drive 0 (A), 01 for drive 1 (B), 10 for drive 2 (C), 11 for drive 3 (D).

Main Status Register

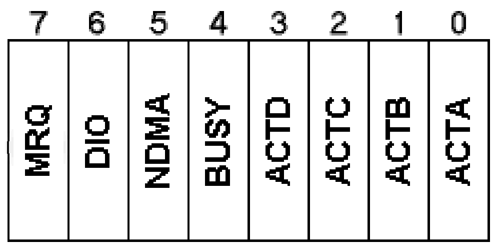


Figure 5.17 – MSR

The MSR is read-only, and contains the controller's status information. This register can be read whatever else the controller is doing. Bit 7 (MRQ) indicates whether the controller is ready to receive or send data or commands via the data register. DIO is used to provide an indication of whether the controller is expecting to receive data from the CPU, or if it wants to output data to the CPU. If the controller is set up to use DMA channel 2 to transfer data to or from main memory, the NDMA bit is not set. If this bit is set, data transfer is carried out exclusively by means of read or write commands to the data register. In this case, the controller issues a hardware interrupt every time that it either expects to receive or wants to supply a data byte. Bit 4 indicates whether the controller is busy or not. If the bit is set, the controller is currently executing a command. Bits 0-3 indicate which (if any) drive is currently in the process of positioning it's read/write heads, or being recalibrated.

Data Register

The data register is an 8-bit register, which provides indirect access to a stack of registers. A command can be one to nine bytes in length, and the first byte tells the controller how many more bytes to expect. The controller sends the command bytes to the correct registers in it's stack, saving the programmer from the need to use a separate index register, as is the case in some other devices (e.g. some VGA registers).

Status Register ST0

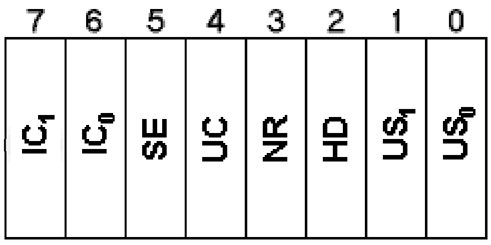


Figure 5.18 – ST0

- IC1 and IC0 contain the interrupt code in binary. A value of 00b indicates normal command termination without errors. 01b indicates abnormal command termination: the controller began executing the command, but could not complete it successfully 10b indicates an invalid command: the command was not started. 11b indicates abnormal termination through polling
- Seek End(SE) is set if the controller successfully executed a Seek or Recalibrate command or read or write operation with an implicit seek
- Equipment Check (EC) is set when a Recalibrate command failed
- Not Ready (NR) indicates that the selected drive is not ready. This bit is not used by all controllers.
- Head Select (HDS) indicates the active drive head: 0 = head 0, 1 = head 1
- Drive Select (DS1, DS0) indicates the currently selected drive in binary: 00b = drive 0, 01b = drive 1, 10b = drive 2 and 11b = drive 3.

Status Register ST1

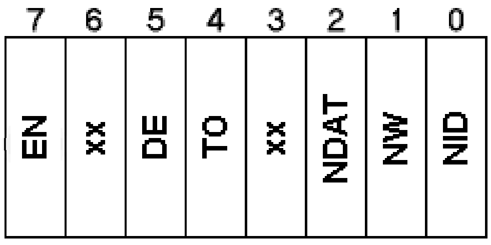


Figure 5.19 – ST1

- Bit *7* is unused and should be *0*
- End of Cylinder (EN). Set if the controller attempted to access a sector beyond the final sector of the track
- Bit 6 is unused and should be *0*
- Data Error (DE). Set if the controller detected a CRC error in either the ID field or data field of a sector
- Overrun (OR). Set if the controller does not receive CPU or DMA service within the required time period
- Bit *3* is unused and should be *0*
- No Data (ND) becomes set if the controller could not locate the requested sector while executing a Read Sector command
- Not Writable (NW) becomes set if a the controller attempted to execute a write operation and the medium in the drive is write protected
- Missing Address Mark. Set if the controller could not locate an ID address mark or failed to locate a deleted address mark

Status Register ST2

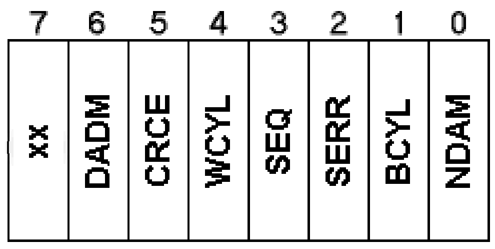


Figure 5.20 – ST2

- Bit *7* is unused and should be *0*
- Control Mark (CM). Set if a Read Sector command encounters a deleted address mark or a Read Deleted Sector command encounters a data address mark
- Data Error in Data Field (DD). Set if the controller detected a CRC error in the data field
- Wrong Cylinder (WC). Set when the track address is different from the address maintained by the controller
- Bit *3* is set to *1* if the controller is a *µPD765* and the seek equal condition is fulfilled, otherwise this bit is always set to *0*

- Bit *2* is set to *1* if the controller is a *µPD765* and a seek operation failed, otherwise this bit is always set to *0*
- Bad Cylinder (BC). Set if the track address differs from the address maintained by the controller and equal to `FFH`
- Missing Data Mark (MD). Set if the controller could not find a valid or deleted address mark

Status Register ST3

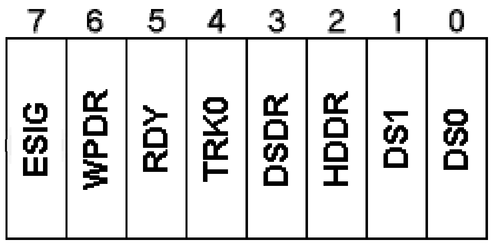


Figure 5.21 – ST3

- Bit *7* is set to *1* if the controller is a *µPD765* and a drive error signal is active, otherwise this bit is always set to *0*
- Write Protected (WP). Set if the medium in the drive is write protected
- Bit *5* is is set to *1* if the controller is a *µPD765* and the active drive is ready, otherwise this bit is always set to *1*
- TRACK 0 (T0). Set if the head is above track *0* and the TRK0 signal is active
- Bit *3* is always set to *1*, but some controllers may use this bit to indicate that the drive is double sided
- Head Address (HD). Indicates the active drive head
- Drive Select (DS1, DS0). Indicates the selected drive: *00b* = drive *0*, *01b* = drive *1*, *10b* = drive *2* and *11b* = drive *3*

FDC Command Set

There are a total of 13 commands available on the µPD765 and compatible FDCs. A further 4 commands are available on the 8207x controllers. The *sector identification* consists of the cylinder, head, sector number and sector size. This tells the controller the position and number of sectors to perform this command on. All commands and status bytes are transferred via the data register, at port 37fh or 377h.

Before the command can be written or the status byte read, it is necessary to read the MRQ bit in the main status register. This determines whether the data register is ready to supply or receive a byte.

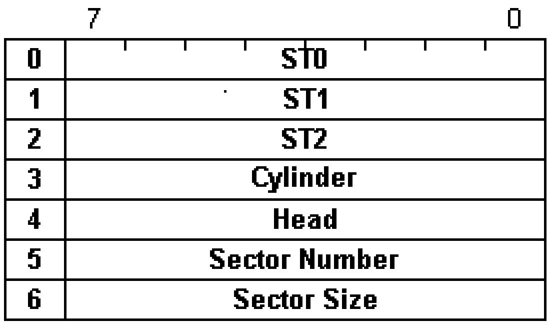
Command Phases

Commands are broken down into three logical phases:

1. **Command Phase** The command, along with all the required parameters are sent to the controller. The first byte contains the operation code for the command and is followed by the command parameters, if any. The number of parameters may vary. However, the controller will know how many parameters to expect based on the command that was issued. For example, the *Recalibrate* command is sent to the controller followed by the drive number that must be recalibrated.

2. **Execution Phase** The controller responds to the command and performs any required action. For example, in response to the *Recalibrate* command, the controller steps the head of the specified drive to track *0*.

3. **Result Phase** The controller updates the status registers depending on the command that was executed. The driver responds by examining these results and takes any necessary actions. The driver responds by informing the host software that executed the command of its failure and returns an appropriate error code or displays an error message. The commands will return the following information during the result phase:



	7 0
0	ST0
1	ST1
2	ST2
3	Cylinder
4	Head
5	Sector Number
6	Sector Size

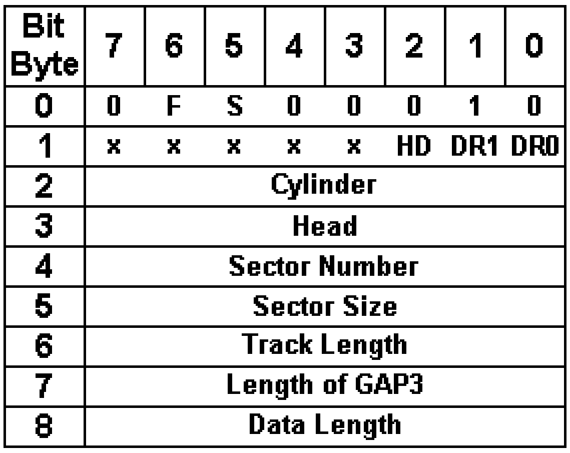
Figure 5.22 - Commands returned during Result Phase

The following fields may be present in the command that is sent to the controller:

- **M** Indicates that the command is a multi-track operation when set
- **F** When set, indicates that the controller must execute the command in double density mode. When cleared, the controller executes the command in single density mode
- **S** When set, the controller will skip deleted address marks
- **HD** Indicates the head number, *0* or *1*
- **DS1, DS0** Specifies drive selection: *00b* = drive *0*, *01b* = drive *1*, *10b* = drive *2* and *11b* = drive *3*
- **SRT** Step rate
- **HUT** Head unload time
- **HLT** Head load time
- **ND** Non-DMA. When set, indicates that the controller is not operating in DMA mode.

Read Sector

This command transfers one or more sectors from the medium to main memory.



Bit / Byte	7	6	5	4	3	2	1	0
0	0	F	S	0	0	0	1	0
1	x	x	x	x	x	HD	DR1	DR0
2	Cylinder							
3	Head							
4	Sector Number							
5	Sector Size							
6	Track Length							
7	Length of GAP3							
8	Data Length							

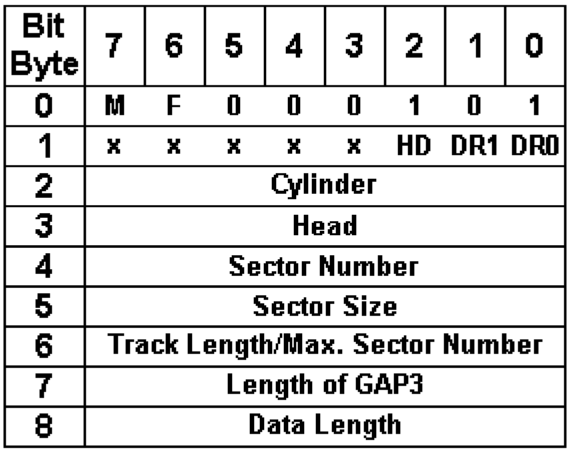
Figure 5.23 - Command Phase of Read Sector

- M, F and S should be set to *1*
- The FDC numbers the sectors in every track starting at *1*.
- The sector size is expressed in multiples of *128* bytes. A value of *0* indicates a sector size of *128* bytes, a value of *1* indicates a sector size of *256* bytes, etc.
- The GAP3 parameter specifies the space between sectors
- The Data Length field should be set to `FFH`

In the result phase, the controller returns the standard information.

Write Sector

This command will transfer data from main memory to the medium in the drive.



Bit Byte	7	6	5	4	3	2	1	0
0	M	F	0	0	0	1	0	1
1	x	x	x	x	x	HD	DR1	DR0
2	Cylinder							
3	Head							
4	Sector Number							
5	Sector Size							
6	Track Length/Max. Sector Number							
7	Length of GAP3							
8	Data Length							

Figure 5.24: Command Phase of Write Sector

- M and F should be set to *1*, S should be set to *0*
- The FDC numbers the sectors in every track starting at *1*
- The sector size is expressed in multiples of *128* bytes. A value of *0* indicates a sector size of *128* bytes, a value of *1* indicates a sector size of *256* bytes, etc.
- The GAP3 parameter specifies the space between sectors
- The Data Length field should be set to `FFH`.

In the result phase, the controller returns the standard information.

Seek

The Seek command moves the read/write head of the selected drive to the specified cylinder (track). The controller will issue step pulses until the current cylinder number matches the cylinder number specified in the command parameters.

Bit Byte	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	1	1
1	x	x	x	x	x	HD	DR1	DR0
2	Cylinder							

Figure 5.25 - Command Phase of Seek

The Seek command does not have a result phase. The Sense Interrupt Status command should be issued to verify the position of the head.

Recalibrate Drive

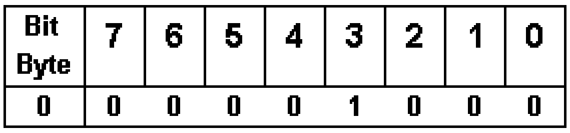
The Recalibrate command will move the read/write head to cylinder (track) *0* by issuing up to *79* step pulses. The controller will examine the TRK0 signal after every pulse and terminate the command if this signal is active and set SE in ST0. The controller will abort the command and set SE and EC in ST0 if the TRK0 signal does not become active after *79* pulses. The Recalibrate command does not have a result phase. The Sense Interrupt Status command should be issued to verify the controller status after the command has been completed.

Bit Byte	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	1
1	x	x	x	x	x	0	DR1	DR0

Figure 5.26 - Command Phase Of Recalibrate Drive

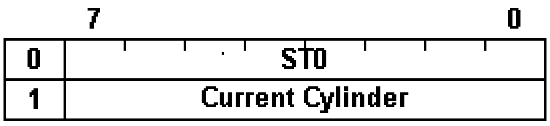
Sense Interrupt

This command is used to check the status of the controller in the result phase of a command if the controller issued an interrupt. Executing this command will reset the interrupt signal. Issuing this command without a pending interrupt will return a value of *80H* in ST0, indicating an invalid command.



Bit Byte	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0

Figure 5.27 - ***Command Phase Of*** **Sense Interrupt**



	7 … 0
0	ST0
1	Current Cylinder

Figure 5.28 - Result for Sense Interrupt

Floppy Driver Interface

System Call Interface

1. read sector(char *buf,int sector) – This reads 512 bytes from the sector specified in the second argument and places in buf.
2. write sector (char *buf,int sector) – This writes 512 bytes from the buffer to the This reads 512 bytes from the sector specified in the second argument.

Error Values

1. ERR_SEEK (-1) – This is returned when the seeking was not performed properly. And this is corrected by doing a recalibrate.
2. ERR_TRANSFER (-2) – This is returned when the reading or writing of sector was not performed correctly.
3. ERR_STATUS (-3) – This is returned when the FDC is not ready to receive commands.
4. ERR_RECALIBRATE (-4) - This is returned when the recalibrate command fails. This can be corrected by performing a reset.

5. ERR_WR_PROTECT (-5) – This is returned when the floppy is write-protected and the write sector system call is executed.
6. ERR_TIMEOUT (-6) – This is returned whenever the floppy interrupt times out.

Floppy Server Architecture

The floppy server runs as a separate process with a higher privilege. It keeps waiting for requests from other processes and services the request.

Flow of Control on a system call

The sequence of steps that happen are:

1. The process executes a system call provided by the floppy and supplies the necessary parameters.
2. The system call handler creates a floppy server request corresponding to the system call and adds that to the sequence of requests to be serviced by floppy server.
3. The handler then removes the requesting process from ready queue and moves it to floppy server queue.
4. The floppy server services the request after servicing the other requests in the queue. It invokes corresponding servicing methods.
5. In this process, the floppy server may request for waiting for the interrupt (0x26) by executing a system call and it is queued in an interrupt queue.
6. After the interrupt is completed, the kernel removes the floppy server from the **interrupt** queue and moves it to the ready queue.
7. The kernel removes the requesting process from floppy server queue to the ready queue after storing the result in the proc structure.
8. The process now begins its normal execution.

Elements in a Floppy Server request

The elements in a request sent to floppy server are

1. **Process Identifier** (pid) of the requesting process: this is used by the floppy server for doing a memory copy on a read or write and on signaling the

completion of the processing so that the kernel can make the requesting process to begin its normal execution.

2. **Operation Code**: this identifies the purpose of this request i.e. a read or write and depending on this the corresponding steps are performed.
3. **Buffer Address**: This is the address of the buffer of the process that should be written to or read from.
4. **Sector Number**: This is the number of the sector that should be written or read.

Interrupt Handling

The kernel provides certain system calls for processes to wait for interrupts till a specific time interval. The process gets queued in the interrupt queue until the interrupt occurs or when the interrupt does not occur till the time is out. The floppy make use of this function and waits for the Interrupt 0x6 mapped by PIC to interrupt 0x26 for a time of 3 seconds. The kernel uses this waiting time for running other processes so that the processor utilization is increased. The floppy driver checks for the return of this system call to determine whether an interrupt has occurred or a time out has occurred and depending on this it triggers certain actions.

Description of Functions

The functions that are implemented in this module are of the following categories:

1. **Motor Handling Functions**: These are the start and stop functions for the floppy motor. The floppy motor must be started explicitly before any other command is sent. These functions check for the current state of the motor and change the motor state as required by sending the value to the DOR port. Start motor function also waits for the interrupt.
2. **Command Send and Result Phase:** The sending of floppy commands is done through the Data Register. There is a function for this which is invoked by all the other modules. It checks for the MRQ bit in the MSR and outputs the command when the FDC is ready. If not, it sets the error value to true. Similarly, there is a funtion for getting the results from the FDC. It also checks for the MRQ and DIO bit in MSR to obtain the result.

3. **Seeking and Recalibrating:** When the data is to be read or writing, the seek function is called for placing the head in the corresponding cylinder and sector. On error, the seek method calls the recalibrate method.

4. **Transfer Method:** This is the important method, which is the starting point when a request to read or write sectors arrives. The steps it does are:

   i) Calculate the cylinder, sector and head value of the offset which is to be read. This is done using the following formula.

   Block = Offset >> 9 (2^9 = 512(sector size))

   Cylinder = block / (no_heads * no_sectors)

   Head = (block % (no_heads * no_sectors)) / no_sectors

   Sector = (block % no_sectors) +1

   ii) Setup the DMA using the driver for the corresponding mode DMA_READ or DMA_WRITE

   iii) Output the seek command and check for the results after interrupt arrives.

   iv) Output the Data Transfer command and wait for the interrupt, then check for the results.

   v) If there is no error in any of these command, copy the data to the user buffer and return.

   vi) If there is error, set the reset_needed value to true and try again.

5. **Reset Method**: This is called whenever the floppy driver has met with an error and so the FDC is reset. This is done by sending zero and the DMA_ENABLE value to the DOR register. After this the entire processing with the FDC is re-begun.

5.5 FILE SYSTEM MODULE

This module deals with all file management and directory management function. It takes the raw floppy interface of reading and writing sectors and implements the FAT12 standard on it. It maintains a descriptor structure for each opened file indexed by the file handle. This file handle is valid only to that process that gives security to this interface. It also provides a set of system calls for the processes. For each file opened, a buffer of 512 bytes is maintained which is synchronized if needed. It also provides a list of system calls for creation, removal, retrieval, rename of files and directories.

5.5.1. FAT12 Specification

FAT12 Regions

FATnn – nn is the number of bits in each entry in the FAT structure on the disk. A FAT file system volume is composed of four basic regions, which are laid out in this order on the volume:

1. **Reserved Region** or the boot sector - This is where the (BIOS Parameter Block) is present.

2. **FAT Region** - There are two copies of FATs for redundancy and these must be kept consistent. Each FAT is 9 sectors long and is a singly linked list of logical cluster numbers. Each entry is 12 bits and 2 entries are packed into 3 bytes.

3. **Root Directory Region** - This is the base directory in the hierarchy of the file system. It follows the 2 FATs and occupies a fixed size of 14 sectors. Each directory entry is 32 bytes.

4. **File and Directory Data Region –** This starts from the 33rd sector to the end. All directories except the root directory are simply files with directory entries and stored in this region. These sectors are linked together by the FAT12 entries to form logical files and directories.

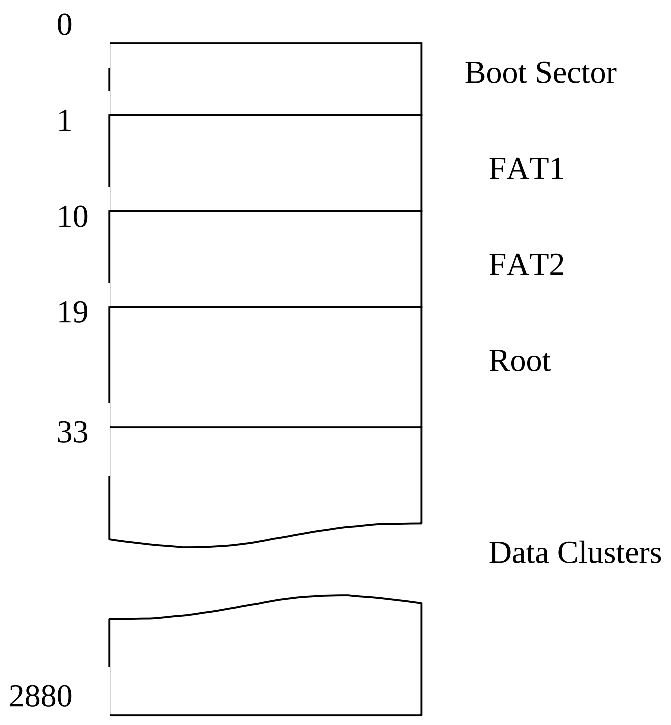


Figure 5.29 - FAT Volume Regions

Various FAT12 structures

Boot Sector and BPB Structure

This structure provides various information about the media type and the file system format, volume label etc.

Name	Offset (byte)	Size (bytes)	Description

BS_jmpBoot	0	3	Jump instruction to boot code. This field has two allowed forms:
			jmpBoot[0] = 0xEB, jmpBoot[1] = 0x??, jmpBoot[2] = 0x90 and
			jmpBoot[0] = 0xE9, jmpBoot[1] = 0x??, jmpBoot[2] = 0x??
			0x?? indicates that any 8-bit value is allowed in that byte. This forms a three-byte Intel x86 unconditional branch (jump) instruction that jumps to the start of the operating system bootstrap code.

BS_OEMName	3	8	The system that formatted the volume.
BPB_BytsPerSec	11	2	Count of bytes per sector. For FAT12, it is 512.
BPB_SecPerClus	13	1	Number of sectors per allocation unit. This value must be a power of 2 that is greater than 0. For FAT12, the typical value is 1.

BPB_RsvdSecCnt	14	2	Number of reserved sectors in the Reserved region of the volume starting at the first sector of the volume. This field must not be 0. For FAT12 volumes, this value should never be anything other than 1.
BPB_NumFATs	16	1	The count of FAT data structures on the volume. This field should always contain the value 2 for any FAT volume of any type.
BPB_RootEntCnt	17	2	For FAT12 volumes, this field contains the count of 32-byte directory entries in the root directory. This value is 224.
BPB_TotSec16	19	2	This field is the old 16-bit total count of sectors on the volume. This count includes the count of all sectors in all four regions of the volume. For FAT12 volumes, this value is 2880.
BPB_Media	21	1	0xF8 is the standard value for “fixed” (non-removable) media. For removable media, 0xF0 is frequently used. The legal values for this field are 0xF0, 0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, and 0xFF. The only other important point is that whatever value is put in here must also be put in the low byte of the FAT[0] entry. For FAT12 volumes, this value is 0xF0
BPB_FATSz16	22	2	This field is the FAT12 16-bit count of sectors occupied by ONE FAT. For FAT12 volumes, this value is 9.
BPB_SecPerTrk	24	2	Sectors per track for interrupt 0x13. For FAT12 volumes, this value is 18.
BPB_NumHeads	26	2	Number of heads for interrupt 0x13. For example, on a 1.44 MB 3.5-inch floppy drive this value is 2.
BPB_HiddSec	28	4	Count of hidden sectors preceding the partition that contains this FAT volume.
BPB_TotSec32	32	4	This field is the new 32-bit total count of sectors on the volume. For FAT12 volumes, the value is 0.
BS_DrvNum	36	1	Int 0x13 drive number (e.g. 0x80).
BS_Reserved1	37	1	Reserved (used by Windows NT). Code that formats FAT volumes should always set this byte to 0.
BS_BootSig	38	1	Extended boot signature (0x29). This is a signature byte that indicates that the following three fields in the boot sector are present.
BS_VolID	39	4	Volume serial number. This field, together with BS_VolLab, supports volume tracking on removable media. These values allow FAT file system drivers to detect that the wrong disk is inserted in a removable drive. This ID is usually generated by simply combining the current date and time into a 32-bit value.

BS_VolLab	43	11	Volume label. This field matches the 11-byte volume label recorded in the root directory. **NOTE:** FAT file system drivers should make sure that they update this field when the volume label file in the root directory has its name changed or created. The setting for this field when there is no volume label is the string "`NO NAME    `".
BS_FilSysType	54	8	The string "`FAT12   `".

Table 5.10 -Boot Sector and BPB Structure

FAT Data Structure

The next data structure that is important is the FAT itself. This data structure does is define a singly linked list of the "extents" (clusters) of a file. The FAT maps the data region of the volume by cluster number. The first data cluster is cluster 2. There are 2 FAT structures continuously. This structure is composed of 12-bit FAT12 entries and 2 entries are packed in 3 bytes.

FAT12 Entry Values

FAT12 entry value indicates the next cluster in the cluster chain. The various FAT12 entry values are :

• 0	Unused cluster
• 0xFF0-0xFF6	Reserved cluster
• 0xFF7	Bad cluster
• 0xFF8-0xFFF	End Of Clusterchain mark (EOC) <EOF>
• Other Next cluster in file	

The list of free clusters in the FAT is nothing more than the list of all clusters that contain the value 0 in their FAT cluster entry. Any cluster that contains the "BAD CLUSTER" value in its FAT entry is a cluster that should not be placed on the free list because it is prone to disk errors. Reserved clusters can be used by the operating system.

FAT12 Entry Packing

The three bytes from 3x to 3x + 2 contains the fat12 entries of 2x and 2x + 1.

Byte 3x	- 8 least significant bits (LSB) of entry 2x
Byte 3x+1	- 4 LSB are the 4 most significant bits (MSB) of entry 2x 4 MSB are the 4 LSB of entry 2x+1
Byte 3x+2	- 8 MSB of entry 2x+1

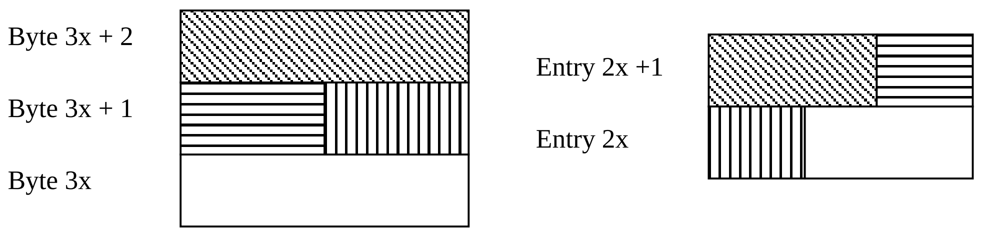


Figure 5.30 - FAT12 Entry Packing

In order to get a fat12 entry, the following steps must be done.

1. Determine byte offset in the FAT using the formula
   FatIndex = (LogicalCluster * 3)/2
2. Determine if an Even or Odd logical cluster.
3. If even , 4 LSB of FAT[FatIndex + 1] forms the 4 MSB of entry and FAT[FatIndex] forms the 8 LSB of entry.
4. If odd, FAT[FatIndex + 1] forms the 8 MSB of entry and 4 LSB of FAT[FatIndex] forms the 4 LSB of entry.

In order to set a fat12 entry, the following steps must be done.

1. Determine byte offset in the FAT using the formula
   FatIndex = (LogicalCluster * 3)/2
2. Determine if an Even or Odd logical cluster.
3. If even, 4 LSB of FAT [FatIndex + 1] is set to the 4 MSB of entry and FAT [FatIndex] is set to the 8 LSB of entry.
4. If odd, FAT [FatIndex + 1] is set to the 8 MSB of entry and 4 LSB of FAT [FatIndex] is set to the 4 LSB of entry.

Cluster Chaining

The way the data of a file is associated with the file is as follows. In the directory entry, the cluster number of the first cluster of the file is recorded. The first cluster (extent) of the file is the data associated with this first cluster number, and the location of that data on the volume is computed from the cluster number using the formula:

Cluster = Sector – ReservedSectors – RootSectors – (NoOfFats * SectorsPerFat) + 2

The next cluster is found by getting the FAT12 entry from the FAT structure. If the next cluster has the EOC mark, the chaining sequence is complete. The physical location of the cluster is found using the formula

Sector = Cluster + ReservedSectors + RootSectors + (NoOfFats * SectorsPerFat) - 2

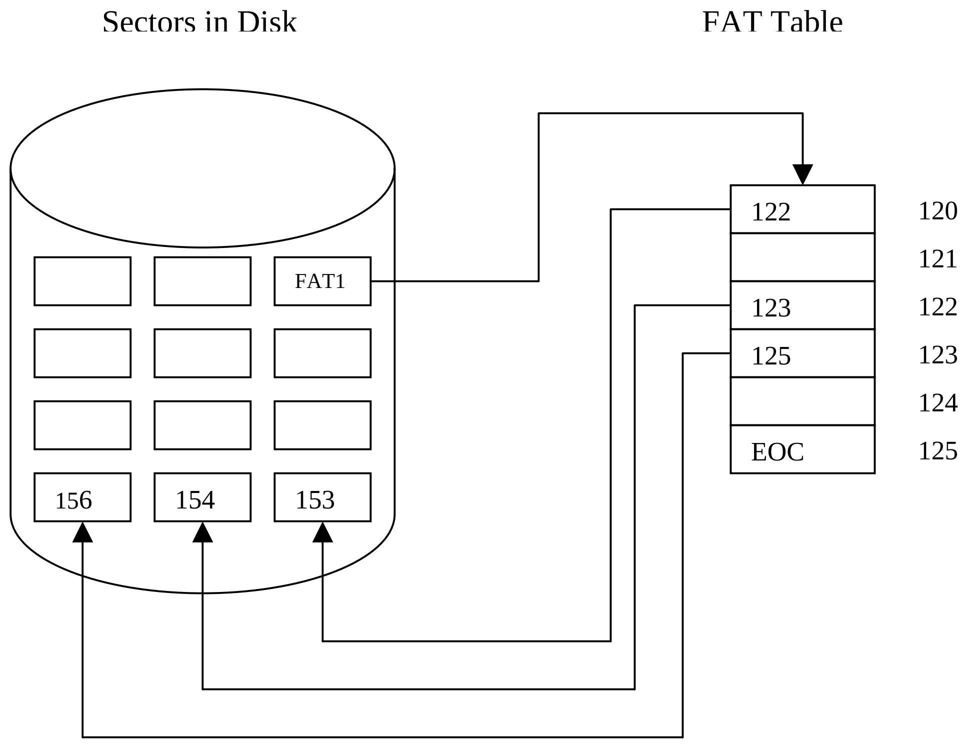


Figure 5.31 – Cluster Chaining

Example of Chaining

Suppose the starting sector is 153. The next sector can be calculated from the FAT. The cluster is 153 - 1 - 18 - 14 - 2 = 122. The fat value on 122nd cluster is 123. So, the next sector is 123 + 1 + 18 + 14 + 2 = 154. Similarly, this process is continued until EOC mark is found in 125th cluster.

FAT Directory Structure

A FAT directory is nothing but a “file” composed of a linear list of 32-byte structures. The only special directory, which must always be present, is the root directory. This is present following the two FAT tables.

Name	Offset (byte)	Size (bytes)	Description

DIR_Name	0	11	Short name.
DIR_Attr	11	1	The upper two bits of the attribute byte are reserved and should always be set to 0 when a file is created.
DIR_NTRes	12	1	Reserved for use by Windows NT. Set value to 0 when a file is created and never modify or look at it after that.
DIR_CrtTimeTenth	13	1	Millisecond stamp at file creation time. This field actually contains a count of tenths of a second. The granularity of the seconds part of DIR_CrtTime is 2 seconds so this field is a count of tenths of a second and its valid value range is 0-199 inclusive.
DIR_CrtTime	14	2	Time file was created.
DIR_CrtDate	16	2	Date file was created.
DIR_LstAccDate	18	2	Last access date. Note that there is no last access time, only a date. This is the date of last read or write. In the case of a write, this should be set to the same date as DIR_WrtDate.
DIR_FstClusHI	20	2	High word of this entry’s first cluster number (always 0 for a FAT12 or FAT16 volume).
DIR_WrtTime	22	2	Time of last write. Note that file creation is considered a write.
DIR_WrtDate	24	2	Date of last write. Note that file creation is considered a write.
DIR_FstClusLO	26	2	Low word of this entry’s first cluster number.
DIR_FileSize	28	4	32-bit DWORD holding this file’s size in bytes.

Table 5.11 - FAT 32 Byte Directory Entry Structure

DIR_Name[0]

This bye carries a specific

If DIR_Name[0] == 0xE5, then the directory entry is free (there is no file or directory name in this entry).

If DIR_Name[0] == 0x00, then the directory entry is free (same as for 0xE5), and there are no allocated directory entries after this one (all of the DIR_Name[0] bytes in all of the entries after this one are also set to 0).

If DIR_Name[0] == 0x05, then the actual file name character for this byte is 0xE5. 0xE5 is actually a valid KANJI lead byte value for the character set used in Japan. The special 0x05 value is used so that this special file name case for Japan can be handled properly and not cause FAT file system code to think that the entry is free.

The following characters are not legal in any bytes of DIR_Name: Values less than 0x20 except for the special case of 0x05 in DIR_Name[0] described above and 0x22, 0x2A, 0x2B, 0x2C, 0x2E, 0x2F, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F, 0x5B, 0x5C, 0x5D, and 0x7C.

DIR_Attr specifies attributes of the file:

ATTR_READ_ONLY	Indicates that writes to the file should fail.
ATTR_HIDDEN	Indicates that normal directory listings should not show this file.
ATTR_SYSTEM	Indicates that this is an operating system file.
ATTR_VOLUME_ID	There should only be one “file” on the volume that has this attribute set, and that file must be in the root directory. This name of this file is actually the label for the volume. DIR_FstClusHI and DIR_FstClusLO must always be 0 for the volume label (no data clusters are allocated to the volume label file).
ATTR_DIRECTORY	Indicates that this file is actually a container for other files.
ATTR_ARCHIVE	This attribute supports backup utilities. The FAT file system driver sets this bit when a file is created, renamed, or written to.
ATTR_LONG_NAME	ATTR_READ_ONLY | ATTR_HIDDEN | ATTR_SYSTEM | ATTR_VOLUME_ID. This attribute bit combination indicates that the “file” is actually part of the long name entry for some other file.

Date and Time Formats

Date Format. A FAT directory entry date stamp is a 16-bit field that is basically a date relative to the MS-DOS epoch of 01/01/1980. Here is the format (bit 0 is the LSB of the 16-bit word, bit 15 is the MSB of the 16-bit word):

Bits 0–4: Day of month, valid value range 1-31 inclusive.

Bits 5–8: Month of year, 1 = January, valid value range 1–12 inclusive.

Bits 9–15: Count of years from 1980, valid value range 0–127 inclusive (1980–2107).

Time Format. A FAT directory entry time stamp is a 16-bit field that has a granularity of 2 seconds. Here is the format (bit 0 is the LSB of the 16-bit word, bit 15 is the MSB of the 16-bit word).

Bits 0–4: 2-second count, valid value range 0–29 inclusive (0 – 58 seconds).

Bits 5–10: Minutes, valid value range 0–59 inclusive.

Bits 11–15: Hours, valid value range 0–23 inclusive.

The valid time range is from Midnight 00:00:00 to 23:59:58.

Long File Name Standard

As the original FAT12 is confined to 8.3 names i.e. names of length 8 bytes and extension of 3 bytes, this standard was proposed. It has backward compatibility with the original FAT12. The 8.3 names and long names share the same namespace. Both a long and a short name will always be created for a file.

FAT Long Directory Entry Structure

A long directory entry is just a regular directory entry in which the attribute field has a value of ATTR_LONG_NAME.

Name	**Offset (byte)**	**Size (bytes)**	**Description**
LDIR_Ord	0	1	The order of this entry in the sequence of long dir entries associated with the short dir entry at the end of the long dir set. If masked with 0x40 (LAST_LONG_ENTRY), this indicates the entry is the last long dir entry in a set of long dir entries. All valid sets of long dir entries must begin with an entry having this mask.
LDIR_Name1	1	10	Characters 1-5 of the long-name sub-component in this dir entry.
LDIR_Attr	11	1	Attributes - must be ATTR_LONG_NAME
LDIR_Type	12	1	If zero, indicates a directory entry that is a sub-component of a long name. NOTE: Other values reserved for future extensions. Non-zero implies other dirent types.
LDIR_Chksum	13	1	Checksum of name in the short dir entry at the end of the long dir set.

LDIR_Name2	14	12	Characters 6-11 of the long-name sub-component in this dir entry.
LDIR_FstClusLO	26	2	Must be ZERO. This is an artifact of the FAT "first cluster" and must be zero for compatibility with existing disk utilities. It's meaningless in the context of a long dir entry.
LDIR_Name3	28	4	Characters 12-13 of the long-name sub-component in this dir entry.

Table 5.12 - FAT Long Directory Entry Structure

Organization and Association of Short & Long Directory Entries

A set of long entries is always associated with a short entry that they always immediately precede. Long entries are paired with short entries for one reason: only short directory entries are visible to previous versions of FAT12 drivers. A long entry never legally exists all by itself. If long entries are found without being paired with a valid short entry, they are termed *orphans*. The following figure depicts a set of n long directory entries associated with it's single short entry.

Long entries always immediately precede and are physically contiguous with, the short entry they are associated with. The file system makes a few other checks to ensure that a set of long entries is actually associated with a short entry.

Entry	**Ordinal**
Nth Long entry	LAST_LONG_ENTRY (0x40) | N
… Additional Long Entries	…
1st Long entry	1
Short Entry Associated With Preceding Long Entries	(not applicable)

Table 5.13 - Sequence Of Long Directory Entries

First, every member of a set of long entries is uniquely numbered and the last member of the set is or'd with a flag indicating that it is, in fact, the last member of the set. The LDIR_Ord field is used to make this determination. The first member of a set has an LDIR_Ord value of one. The nth long member of the set has a value of (n OR LAST_LONG_ENTRY). Note that the LDIR_Ord field cannot have values of 0xE5 or 0x00. Values for LDIR_Ord must run from 1 to (n OR LAST_LONG_ENTRY). If they do not, the long entries are "damaged" and are treated as orphans by the file system.

Second, an 8-bit checksum is computed on the name contained in the short directory entry at the time the short and long directory entries are created. All 11 characters of the name in the short entry are used in the checksum calculation. The check sum is placed in every long entry. If any of the check sums in the set of long entries do not agree with the computed checksum of the name contained in the short entry, then the long entries are treated as orphans. The algorithm, implemented in C, for computing the checksum is:

```
unsigned char ChkSum (unsigned char *pFcbName)
{
        short FcbNameLen;
        unsigned char Sum;
        Sum = 0;
        for (FcbNameLen=11; FcbNameLen!=0; FcbNameLen--) {

                Sum = ((Sum & 1) ? 0x80 : 0) + (Sum >> 1) + *pFcbName++;
        }
        return (Sum);
}
```

As a consequence of this pairing, the short directory entry serves as the structure that contains fields like: last access date, creation time, creation date, first cluster, and size. It also holds a name that is visible on previous versions of FAT12. Names are also NUL terminated and padded with 0xFFFF characters in order to detect corruption of long name fields by errant disk utilities. A name that fits exactly in a n long directory entries (i.e. is an integer multiple of 13) is not NUL terminated and not padded with 0xFFFFs.

The Basis-Name Generation Algorithm

The *basis-name* generation algorithm is outlined below. This is a ***sample*** algorithm and serves to illustrate how short names can be auto-generated from long names. An implementation ***should*** follow this basic sequence of steps.

1. The UNICODE name passed to the file system is converted to upper case.
2. Strip all leading and embedded spaces from the long name.
3. Strip all leading periods from the long name.
4. While not at end of the long name and char is not a period and total chars copied < 8 Copy characters into primary portion of the basis name.

5. Insert a dot at the end of the primary components of the *basis-name* iff the basis name has an extension after the last period in the name.
6. Scan for the last embedded period in the long name.

```
If   the last embedded period was found
     While not at end of the long name and total chars copied < 3
               Copy characters into extension portion of the basis name.
```

The Numeric-Tail Generation Algorithm

If the long name fits within the 8.3 naming conventions and the basis-name does not collide with any existing short name, the short name is only the basis-name without the numeric tail. Else, Insert a numeric-tail "~n" to the end of the primary name such that the value of the "~n" is chosen so that the name thus formed does not collide with any existing short name and that the primary name does not exceed eight characters in length.

5.5.2. File System Interface

System Call Interface

The various system calls provided for file system are :

1. open(char *fname,int mode) – This takes the filename and the mode to be opened (READ-ONLY or READ-WRITE) as the parameter and returns a file handle or an error.
2. close(int fd_in) – This takes the file handle as input and closes the file and performs the synchronization operation.
3. read(int fd_in,char *buf, int length) – This reads ‘length’ bytes from the file with handle fd_in and places in buf.
4. write(int fd_in,char *buf,int length) – This writes ‘length’ bytes from the buffer to the file opened with fd_in as handle.
5. creat(char *fname) – This creates a new file in the root or the sub directory specified.
6. opendir(char *name) – This is used to open directories but there is only READ mode.
7. createdir(char *name) – This creates a new directory in a sub directory or the root directory.
8.closedir(int dd_in) – This closes the directory opened with handle dd_in.

9.readdir(int dd_in,DIRENT *dir) – This reads the next directory entry from the directory with handle dd_in.
10. remove(char *name) – This removes the file, if it exists.
11.rename(char *old_name,char *new_name) – This renames the file with the new name.
Another system call, which is used by the kernel itself for creating processes is
12. Load (char *name,int addr[]) – This loads the specified binary file into the memory location addr.

Modes of Opening

1. O_RDONLY – The file is opened only for reading. The system call write is not supported for this and no synchronization is done on close.
2. O_RDWR - Both read and write calls are supoorted. Synchronisation of the block is done on close.

Dirent structure

This structure is returned when the readdir system call is executed on the opened directory. By using this, all the present files and directories can be read. It parses through the directory and returns the long name of the file or directory. It also returns various other information about the file. They are :

1.Long Name of the file or directory (upto 255 characters in length)
2. Attribute – Directory or file, Hidden, System
3. Creation Date and Time
4. Accessed Date
5. Modified Date and Time
6. First cluster
7. Size of the file or directory

Error values

The various error values returned by the File System are

1. E_DISK (-1) – This is returned when there is any problem in reading or writing the floppy.
2. E_FS_NEXISTS (-2) – This is returned when a file or directory to be opened or removed or renamed does not exist.

3. E_FS_EXISTS (-3) – This is returned when a file or directory to be created or renamed as exists
4. E_FS_FNAME (-4) - This is returned when the file name does not conform to the standards.
5. E_FS_SPACE (-5) - This is returned when there is no space available for creation of a file or directory.
6. E_FS_BUSY (-6) - This is returned when there are maximum number of files or directories that are opened.

5.5.3 File System Server Architecture

The file system server runs as a separate process with a higher privilege. It keeps waiting for requests from other processes and services the request.

Flow of Control on a system call

The sequence of steps that happen are:

1. The process executes a system call provided by the File System and supplies the necessary parameters.
2. The system call handler creates a FS request corresponding to the system call and adds that to the sequence of requests to be serviced by FS.
3. The handler then removes the requesting process from ready queue and moves it to FS queue.
4. The FS services the request after servicing the other requests in the queue. It invokes corresponding servicing methods.
5. In this process, the FS may request for reading or writing of a sector to the floppy server by executing a system call.
6. After the request has been completed, the FS sends the result and process completion interrupt to kernel.
7. The kernel removes the requesting process from FS queue to the ready queue after storing the result in the proc structure.
8. The process now begins its normal execution.

Elements in a FS request

The elements in a request sent to FS are

1.**Process Identifier** (pid) of the requesting process: this is used by the FS for doing a memory copy on a read or write and on signaling the completion of the processing so that the kernel can make the requesting process to begin its normal execution.

2. **Type of request**: this identifies the purpose of this request and the data present in the next part of the request.

3. **Body of request**: Depending on the type of request field, it contains the necessary parameters of the request. This is present as a union of structures.

Descriptor and Buffer Management

The maintenance of a session across multiple reads and writes is done using file handlers and mapping a descriptor to it. A buffer of size 512 bytes is allocated for each opened file and directory. The descriptor array is indexed using the file handle and it contains the needed information about that opened file or directory and also holds the buffer pointer.

Elements of Descriptor

1. **Available** bit: This is used as an indication of whether the descriptor is in use or not.
2. **Name** of the opened file or directory
3. **Mode** in which the file is opened (Read-Only or Read-Write)
4. **Sector offset** - The current sector of the opened file or directory which is being written or read.
5. **Offset** into the current sector – This is the current byte offset being written or read.
6. **Buffer** pointer - A pointer to the allocated file buffer.
7. **Length –** The size in bytes of the opened file or directory. This is used as an indication of reaching EOF. It is also used for holding the modified size of file.
8. **Total Offset –** This is the byte count starting from the beginning of the file. This is also used in finding the EOF of the opened file or directory.

9. **Directory Entry** – This is the initial directory entry of the opened file or directory. This is later modified and written in the corresponding location indicated by the next two fields.

10. **Sector of directory entry** – The sector in which the directory entry for this opened file or directory is present. This is used for writing the last accessed date, the modified date and time, and the changed file size on close of this file or directory

11. **Offset** into the sector of **directory entry**

The descriptor is maintained independently for each process. As a result, each process has its own handler space and it can't access the files opened by the other processes. This is guaranteed as the kernel itself gives the process id of the requesting process and so it is trustworthy. The handle returned is maintained unique within a process by using the available bit in the descriptor structure. The descriptor is also used for maintaining the state of the session. It also puts a threshold on the number of opened files or directories so that a process can't flood the FS with multiple requests. When the maximum number is reached, an error with value E_FS_BUSY is returned.

Buffer Management

The buffer holds the data temporarily and so it reduces the accessing of the disk. It extremely optimizes the disk access as most requests are of very small sizes. The way in which the buffer is used is as follows :

1. On opening a file or directory, the buffer is filled with the first sector, if present and the sector and offset fields are updated.

2. When a read request is performed and the requested data limit is within the buffer itself, then the data is returned and the offset is updated accordingly.

3. If the data limit is beyond the buffer, the data present in the buffer is first copied and then the next sector of the file is read into the buffer and the remaining of the data is copied and returned to the process.

4. A write request also works in a similar way and a write back is done whenever the file buffer is updated with the next sector or when the file is closed.

Request Handler Methods

The File System server waits for requests and dispatches it according to the type to the respective request handling methods. The various request handling methods are described here.

Creating a file

The various steps for creating a file are :

1. Extract the parent directory and the file name. Check whether the parent directory exists, if not return error.
2. Check whether a file with the new file name is present. If so, return error.
3. Check whether the new file, follows the naming conventions of FAT12. If not, return an error.
4. Get a location in the directory where free entry can be created.
5. Using the long name convention, form a short name if needed.
6. Construct the short directory entry with size and first cluster fields as 0. Compute the checksum of the entire short name.
7. Form the long name directory entry sequences and add the checksum field.
8. Write all the long and short directory entries in the correct sequence to the obtained free entry location in the parent entry.

Creating a directory

Creating a directory also involves a few additional steps to be taken care of. They are:

1. When a directory is created, DIR_Attr field of the directory entry is set to the value ATTR_DIRECTORY, and the DIR_FileSize is set to 0.
2. One cluster is allocated to the directory and the DIR_FstClusLO and DIR_FstClusHI is set to the corresponding cluster number and place an EOC mark in that clusters entry in the FAT.
3. Initialize all bytes of that cluster to 0.
4. Create two special entries in the first two 32-byte directory entries of the directory. (The first two 32 byte entries in the data region of the cluster). The first directory entry has DIR_Name set to: “.” The second has DIR_Name set

to”..”. These are called the *dot* and *dotdot* entries. The *dot* entry is a directory that points to itself. The *dotdot* entry points to the starting cluster of the parent of this directory (which is 0 if this directories parent is the root directory). The DIR_FileSize field on both entries is set to 0, and all of the date and time fields in both of these entries are set to the same values as they were in the directory entry for the directory that was just created. The DIR_FstClusLO and DIR_FstClusHI for the *dot* entry (the first entry) are set to the same values in those fields for the directories’ directory entry. The DIR_FstClusLO and DIR_FstClusHI for the *dotdot* entry (the second entry) are set to the first cluster number of the directory in which this directory was created.

Opening a file or a directory

The various steps performed for handling this request are:

1. Obtain a free descriptor for the corresponding process.
2. Check if the file or directory exists in the corresponding parent directory.
3. Update the sector, offset of the directory entry and the directory entry fields in the descriptor.
4. Update the file length, total offset, current sector and offset accordingly.
5. Read the first sector and place it in the buffer for that descriptor.
6. Return the index of the allocated descriptor as handle.

Reading or Writing to a file

The sequence of steps for handling read and write are:

1. Check whether a file is opened with the handle and the mode is set appropriately. If not, return an error.
2. Check if the data in buffer is enough for satisfying this request. If so, copy the requested bytes or write the requested bytes, update the offset in the descriptor and return.
3. If not, first copy the available data or write the available data. Then, write back the data in the buffer to the corresponding sector. Then, obtain the next sector in the cluster chain, and complete the request.

4. For read call, if the next sector is EOF, then return the already read number of bytes.
5. For write call, if the next sector is EOF, then allocate a new sector and write an EOC mark in it's fat entry, update the previous cluster's fat entry to point to this cluster. If there is no previous entry, make this allocated cluster as the first cluster in the root directory entry.
6. Repeat the steps from 2 until the requested number of bytes is read or written to.

Closing a file

Depending on the mode in which the file was opened certain steps must be performed.

1. Check whether a file is opened with the handle and the mode is set appropriately. If not, return an error.
2. If the file was opened for Read-Write, then write back the data in buffer to disk. Change the modified date and time to the current date and time.
3. Change the last accessed date in the directory entry.
4. Write the directory entry to the parent directory's sector at the offset stored in the descriptor.

Removing a file

1. Check whether a file is present in the corresponding directory. If not, return error.
2. Set the long name entries in the parent directory to deleted entries by setting the first byte to 0xE5 by going backward till the ordinal field has the 0x40 (LAST_LONG_ENTRY).
3. Set the short directory entry's first byte to 0xE5.
4. Get the first cluster in the short entry and by traversing through the FAT table update all cluster entries in the FAT to unused.

Renaming a file

1. Check whether a old file is present in the corresponding directory. If not, return error.
2. Check whether a file with the new name is present in the corresponding directory. If so, return error.
3. Set the long name entries and short directory entries in the parent directory to deleted entries by setting the first byte to 0xE5.
4. Form short and long entries for the new name, the other fields of date and time, file size are set as in the old entry.
5. The first cluster values are also set as in the old file. This is done so that the contents of the old file are retained.

Loading a program

1. Check whether the binary file to be loaded is present. If not, return error.
2. Open the file by using the name in the request.
3. While there are more bytes to read, read it as chunks of 512 bytes and store it in the address given in the request.
4. Close the opened file.

Formatting or FAT Volume Initialization

1. Initialize the values for jump boot and System Name. Set the values for bytes per sector to 512, sectors per cluster to 1, head count to 2, total sectors to 2880, sectors per track to 18.
2. Initialize the values for fat count to 2, max root entries to 224, sectors per fat to 9, media descriptor to 0xF0, extended boot signature to 0x29.
3. Generate volume serial number from the date and set the volume label.
4. Set the first fat entry to 0xFF0 (media type) and the upper byte to EOC (0xFFF). Set the rest of the Fat entries in both FATs to 0.
5. Create a volume label directory entry and set it as the first entry in Root sector. Clear all the other entries in Root sector.

5.5.4 Implementation of File System

The FAT12 specification was implemented along with the long name standard. The file system server was implemented. The system call interface as designed was implemented.

Stages in Implementation

The implementation of the file system was done in several stages:

1. **FS Server:** Methods for starting file server, adding requests, system call handling functions that forms the requests, the run server loop that dispatches requests.
2. **Primitive functions**: Various functions for reading and writing FAT entries, boot sector, long directory entries, short directory entries. All these functions convert the data types of int, long, date and time into the little endian format and generate buffers for writing to sectors.
3. **Cluster Chaining functions:** To implement the cluster chaining, some base functions were first coded. They are:

   i) **Get Next Sector:** This function reads the fat entry for the given sector and checks if it is EOC and if not, calculates the sector number from the cluster and returns. This function is used for reading, writing a file.

   ii) **Remove Chain:** This function sets the fat entries of a chain to unused. It reads the fat entry of the first sector and passes through the chain setting all entries to unused.
4. **Long Name Standard:** The functions that were implemented are:

   i) **Get Long Name**: This function returns the long name when given the index into the short directory entry. It checks for all the rules of a long name namely the checksum, ordinal value. This is invoked for searching through a directory.

   ii) **Remove Long Name**: This function removes the long name entry sequences given the short directory entry index. This is invoked when removing a file or a directory.

iii) **Create Long Name**: This function generates the set of long name entries when given the short and the long name.

iv) **Generate Short Name:** This is used for the base name and the numerical tail generation for a long name given. This function is used when a file or directory is created.

5. **Identifying free entries and sectors:** The get free sector function is used for finding free sectors by checking for unused fat entries in FAT and the get free entry gives the index into a directory sector to find the free entry.
6. **Descriptor Handling:** This set of functions is used for creating, removing and accessing descriptors. This is called by the open, read, write and close calls for files and directories.
7. **Request Handler Methods:** These functions perform the specific operations for a request. These are called by the file server loop. After performing the necessary functions, it returns the result through the file server to the kernel.
8. **Synchronization Methods:** The file buffer needs to be synchronized as it employs a write back cache scheme. The synchronization is done when a file is closed or a process exits or when the system shuts down or restarts. The file system has functions for syncing on a file basis, or process basis or for all opened files.

5.6. DSERVER (GUI) MODULE DESIGN

This module deals with the creation and management of windows and components. It provides an easy-to-use model using which various applications can be designed. Its architecture is based on the client-server model wherein the client sends requests for the GUI-based functions and the GUI server performs the operations. The model uses the inter-process communication model provided by the kernel. The GUI uses the functions of Video driver for displaying the GUI elements. The client is prevented from accessing the video driver directly and the GUI server acts as a bridge between the client and the video driver.

5.6.1 Conceptual Design

The GUI server is organized into the following sub-modules in order to provide the functions required by a client. The sub-modules are:

GUI Server and the Request Handling

The GUI server runs as a separate process with a high priority level. It receives messages from the various client processes and also the device drivers like mouse when a click is occurred and the keyboard when a key is typed. The GUI server handles the message depending upon the type present in the message and performs the corresponding functions. The sequence of steps that happen are:

1. The client sends requests to the server through messages.
2. The client is removed from the ready queue and queued in a special queue called as GUI queue by the kernel.
3. The server waits for messages from the client or the drivers using the recv system call of the process management module and handles it.
4. After performing the operations, it indicates to the kernel that the processing is over through an system call.
5. The kernel removes the respective process from the GUI queue into the ready queue.

Desktop and Taskbar Management

The desktop provides shortcuts for commonly used applications like DynaPlorer, DynaPad and the DynaCalc. The taskbar holds the currently visible windows and it differentiates between the active and the inactive window. By clicking on this task link, the window toggles from maximized/restored state to minimized state. It also displays the time and date. On clicking this clock, a process that displays a window, which is used for setting the time, is invoked. This module also handles the start menu, which is displayed on a mouse click. The start menu holds links to other programs, which are dynamically loaded using the create system call.

Window and Component Management

The GUI provides an interface for the client to create, destroy, hide, show, disable and enable the windows and the components. Window is a container which holds the various components. The window is composed of the tittle bar and the buttons for minimizing, maximizing or restore and the close button. The next part in the window is the menu that is handled by a separate module. Next is the region where the client can create any components of choice.

The various components that can be created are:

Label: It is a component that is used for displaying text messages.

Button: It is a component that is used for invoking certain actions. The GUI server handles the clicking of the button and sends a message to the client process that can perform the actions accordingly.

Text: It is a component that is used for getting a line of text as input. The GUI server handles the clicking of this text by making this the focus component after which any key typed is displayed. It also provides functions for cutting, copying and pasting and navigation on specific keys. It provides certain messages for getting, setting the text.

TextArea: It is also an area for receiving input across multiple rows. It also provides the same function of editing and navigating of the displayed text.

Folderview: It is a component that provides the display of folder and file icons and handles the click event on these icons and sends a message to the client process that performs the corresponding operations. DynaPlorer uses this for providing a file and directory browser.

Dialog: This is similar to a window but it is embedded within a window. The dialog can also contain the other components. It is used for displaying information or getting information needed by the parent window. During the time in which the dialog is active, the parent window is inactive.

All these components can over-ride the default operation by using certain style. For example, the window can be made non-resizable or non-closable, text and textarea can be disabled so that the keys pressed are not displayed.

Menu Management

Each window can optionally have a menu. This menu has a menu bar on top below the title bar and a drop-down menu interface. The client application this hierarchy of menu and sends a request message to the GUI. The GUI server handles the click event on the menu bar and displays or hides the drop-down menu alternatively. When a menu item is clicked, the GUI server sends a message to the client which contains the id of the clicked menu item. The client can do the respective actions depending on the id of the menu item that was clicked.

Alias Id Mapping

The client uses a unique id for windows and components that it creates. The alias id's namespace is only within a process space. So, the (pid, alias id) pair is unique. The GUI maps between this alias id and the actual id that the GUI uses. This helps the application to handle events related to the component easily. The client sends the alias id of a component during its creation to the GUI server which maps this id into the id in its space. In all further communications, the client uses only this alias id and the GUI server uses this for getting actual id. The GUI maintains a list of windows and components that are indexed with this actual id.

Z-Order Processing

The processing of a mouse click and the painting of windows are done with the help of a z-order queue structure. As windows are shown, they are added in the top of the z-order queue. The click event is processed from the top of the z-order queue to the bottom. And if within the specific window, actions are taken depending on whether it was clicked on the title bar or any components. When painting the windows, the windows are painted from the bottom of the queue to the top, so that the top-most or the recent window is shown.

5.6.2 Client Interface Design

The client process can create, destroy, show, hide, disable, enable windows and components, dialogs and menus. The client does so by using certain functions. The client can have alias id for each of these and using the alias id they can handle events. The GUI process has a specific format of message that is sent when a event occurs.

Functions Provided

The functions defined are:

1. **Create –** This type of function is defined for window, all components and dialogs. It has the parametes of starting co-ordinates, width, height, alias id, style. For windows, the title can also be passed. For components, the container id is passed.
2. **Destroy –** This function is used for destroying a window. It takes the alias id as the parameter.
3. **Show –** This function is used for displaying the window. It takes the alias id of the window as the parameter.
4. **Hide –** This prevents the window from being painted. It also takes the alias id as argument.
5. **Resize –** This is used for resizing the window. It takes the alias id, the new starting position and the new width and height of the window.
6. **Enable –** This is used for enabling the window or components. Enabling makes the window or components to respond to mouse and keyboard events as defined.
7. **Disable –** This makes the window and components not to respond to mouse and keyboard events.
8. **SetText –** This function is used for changing the current text in text and textarea components.
9. **GetText –** This function is used for getting the current text in text and textarea components.

10. **isChanged –** This funtion is used for checking whether the state of the text or textarea is changed.

Styles Defined

There are specific styles that are defined for the windows and components. These styles can be specified when they are created. They are as follows:
For windows, the styles defined are:

1. **W_MINIMIZE** (0x2) – This style sets the state of the window as minimized.
2. **W_MAXIMIZE** (0x4) – This makes the state of the window as maximized.
3. **W_DISABLED** (0x8) – This disables the window.
4. **W_NORESIZE** (0x10) – This style disables the maximize/restore button on the title bar so that the window stays in its initial state.
5. **W_NOCLOSE** (0x20) - This style disables the close button on the title bar so that the window is not closed on its click and only a message is sent to the client. The client can choose whether the window must be destroyed or not.

Styles that are common to all components are:

1. **HIDDEN** (0x1) – This prevents the component from being displayed.
2. **DISABLED** (0x2) – This prevents the component from responding to mouse clicks and keyboard events.

Handling Events

The client after performing the creating and displaying functions can wait for event messages from the GUI. The type of messages the GUI can send are:

1. **WM_DESTROY** (0x1) – This message is called when the window is closed. The body of the message has the alias id of the window.

2. **WM_CLICKED** (0x2) – This message is called when a click occurs on a button or a menu. The alias id is passed in the body of the message. The sub type indicates this which can be:

i) BUT_TYPE

ii) MENU_TYPE

3. **WM_KEYPRESS** (0x3) – This message is called when a key is pressed. The sub type can be:

i) TEXT_TYPE

ii) TEXTAREA_TYPE

5.6.3 GUI Message Syntax

The client interface makes use of the message-passing model of our kernel to accomplish the defined functions. There are messages of specific formats that are sent.

The type of the message can be any of the following depending on the function to be performed:

1. **CREATE –** The subtype can have the values of WINDOW, BUTTON, TEXT, TEXTAREA, MENU, LABEL,FOLDERVIEW and DIALOG. The body of the message has the following structures. For windows, it can have the structure contains the x, y , width, height, style, alias_id, title. For components, the structure has container id and the type in addition to the above. For menu, a pointer to a menu bar structure is sent. It in turn contains the

   i) Number of top level menus

   ii) Array of Menus

   The menu in turn contains

   i) The name of the menu

   ii) Number of Sub-menu items

   iii) Array of Sub-menu items

   The menu item is composed of

   i) Menu id – This is sent by the GUI server when this sub menu item is clicked.

   ii) Name of the menu item.

2. **SHOW –** The subtype can have the values of WINDOW, DIALOG. The body contains the alias id.

3. **HIDE** - The subtype can have the values of WINDOW, DIALOG. The body contains the alias id.
4. **RESIZE** - The subtype can have the values of WINDOW. The body contains the alias id of window, new x, y, width and height.
5. **ENABLE** - The subtype can have the values of WINDOW, BUTTON, TEXT and TEXTAREA. The body contains the alias id.
6. **DISABLE** - The subtype can have the values of WINDOW, BUTTON, TEXT, and TEXTAREA. The body contains the alias id.
7. **DESTROY** - The subtype can have the values of WINDOW, BUTTON, TEXT, TEXTAREA, MENU, LABEL, FOLDERVIEW and DIALOG. The body contains the alias id.
8. **FINISHED** – This message is sent by the client after processing a WM_CLICK event of the button. After the click of the button until this message is invoked, the window is disabled. Only after the client sends this message the window can respond to other events.
9. **SET** - The subtype can have the values of TEXT, TEXTAREA. The body contains the alias id and the pointer to the string.
10. **GET** - The subtype can have the values of TEXT, TEXTAREA. The body contains the alias id and the pointer to the string.
11. **CUT** - The subtype can have the values of TEXT, TEXTAREA. The body contains the alias id.
12. **COPY** - The subtype can have the values of TEXT, TEXTAREA. The body contains the alias id.
13. **PASTE** - The subtype can have the values of TEXT, TEXTAREA. The body contains the alias id.
14. **IS_CHG** - The subtype can have the values of TEXT, TEXTAREA. The body contains the alias id. It returns whether the text or textarea was changed.

Apart from these, there is a category of message received by the GUI. These are from the kernel when a mouse is clicked or a key is pressed.

15. **ACTION** - The type of the message contains this value and the subtype can have a value of MOUSE or KBD. For the subtype of MOUSE, the body of the message consists of

    i) The x and y point co-ordinates

    ii) The status – whether it is a click or a move.

    For the subtype of KBD, the body of the message contains

    i) The character's ascii value

    ii) The state – whether CTRL, SHIFT, ALT key is pressed and whether CAPS lock is on.

5.6.4 Implementation of the GUI Server

The various sub modules were implemented and tested with the sample applications. The sub modules and their functions are explained below.

GUI Server startup and the run

This has functions for starting this special process with higher priority. It initializes the stack, code and data segments of this process. The GUI server first executes the **recv** system call and branches out to the message handler function. After processing of the request, it invokes the special **GUI processing over** system call.

Alias Id Mapping and the list maintenance

This module maintains two basic data structures. One for the mapping between the alias id the client sends and the other for the maintenance of lists of window and component pointers. The alias id mapping is done with the structure which has

i) Type
ii) Alias id
iii) Id

The list manipulations use the structure, which contains

i) Type
ii) Id
iii) Pointer depending on the type.

There are functions that add, remove, remove all by process id, find by alias id and find by id present. The add method is invoked when a window or component is created. The remove is invoked on destroy. The find methods are invoked on all messages for getting the actual id.

Z-Order Processing

This is a queue with additions on one end. It has a structure, which contains

i) Process id
ii) Window id
iii) Task id.

There are methods for addition and removal based on the id or process id. There is one other important moveTop. This is used for moving a window from somewhere in the order to the top. This is invoked when a click on the window occurs.

Window Management

The window was implemented as a C ++ class.

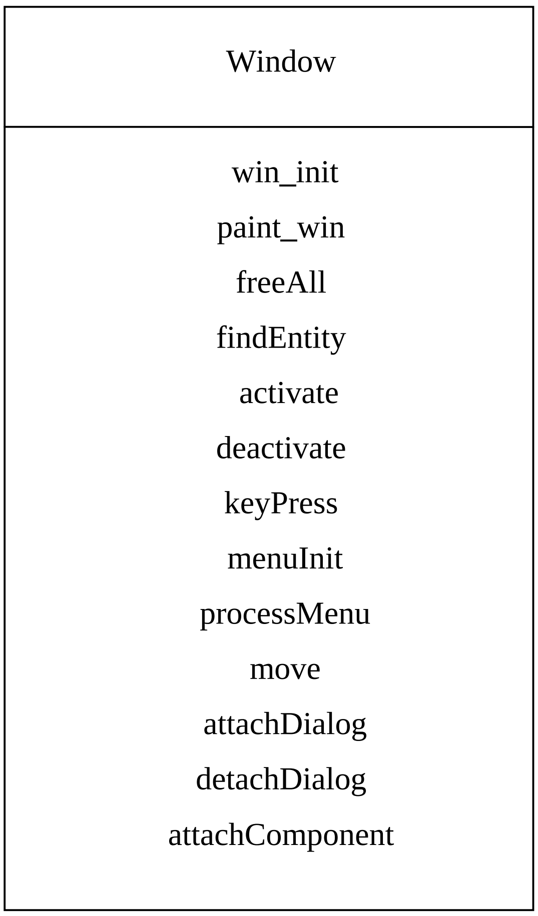


Figure 5.32: Class Diagram of the Window

The various data that are to be maintained about a window are:

1. id, style, title
2. x, y, width, height
3. comp_focus – This is used for identifying the component which has the focus so that all key press go to this.
4. num_comps – Number of Components attached to it.

5. Array of Component pointers
6. hasMenu – it indicates whether this window has a menu.
7. menubar *MenuBar – The pointer to the menubar if window is present.
8. COLOR *buf – This is used for moving the window at which point the back image is buffered here.
9. window *diag – This stores the pointer to the dialog, if one is present.

The various functions that were implemented were:

1. **win_init** – This is called when the object is created. It initializes all the data.
2. **paint_win** – This is called for drawing the window. It in turns the components, menu and dialog if present.
3. **freeAll** – This is called when this window is destroyed. It frees the components and menu if present.
4. **findEntity** – This is called when the window is clicked. It checks for the component which was clicked.
5. **activate** – This enables the window by changing the style.
6. **deactivate** - This disables the window by changing the style.
7. **keyPress** – This is called when a key was pressed and this is the window on focus.
8. **menuInit** – This is called when a menu is created for this window.
9. **processMenu** – This is called when the click was within the menu area.
10. **move** – This is called when the window is moved. This restores the image present behind the window and backs up the image on the new location and draws the window.
11. **attachDialog** - This is called when a dialog is created for this window.
12. **detachDialog** – This is called when a dialog is removed for this window.
13. **attachComponent** – This is called when a component is added in this window.

Component Management

The various data that it stores are:

1. Type of Component
2. id, container id,alias id

3. x, y, width, height – This is relative to the container
4. Pointer to container, container type
5. Style

All components inherit from the base class component.

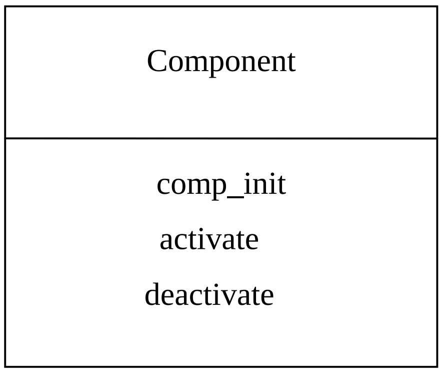


Figure 5.33: Class Diagram of the Component

Button Class

This inherits from the component class. It implements all the necessary behavior of the button.

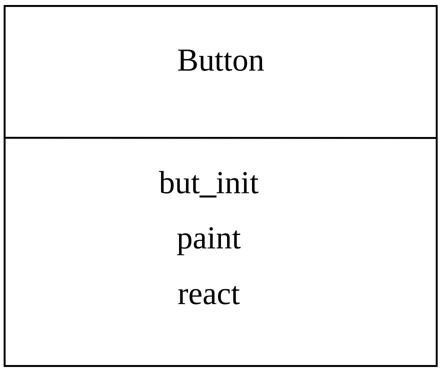


Figure 5.34: Class Diagram of the Button

The data elements in this class are:

1. label – this holds the text to be displayed on the button.
2. state – This holds the state of the button. This is used when the button is painted. It is set to pushed_back state when it is clicked.

The various functions that are present in this class are:

1. **but_init** – This is called when an object is created for this class. It initializes all the values.
2. **paint –** It paints the button depending upon the state.
3. **react** – This method is called when a click occurs on the button. It disables the parent window and changes its state.

Text Class

This is also a subclass of text. The additional data elements present are:

1. txt – This stores the text that is displayed on the textbox.
2. total_chars, allow_chars – The count of characters that are present and the allowable number of characters depending on the width of the textbox.
3. start – This is the character number starting from which the text is displayed. This is changed when a navigation key (arrow keys or home or end key) is pressed.
4. curs – This is the current location of the cursor. This is used for setting the globally shared cursor. Only from this position, that characters are inserted.
5. sel start, sel end – This indicates the starting and ending character that is selected. This is used for displaying the highlighted portion of the text and to execute a copy or cut or delete.
6. is sel – This indicates that whether any portion is selected or not.
7. changed – This indicates whether the text has been changed after it was set.

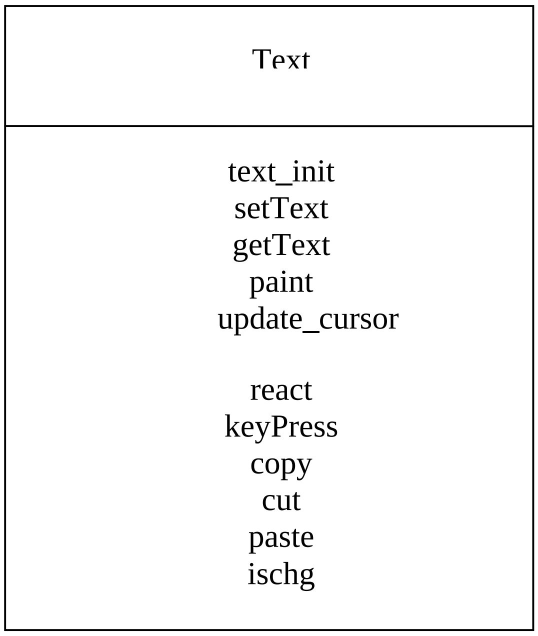


Figure 5.35 – Class Diagram of Text

The member functions in this class are:

1. **text_init –** This is used for setting the intial values.
2. **setText** – This function sets the current text value to the text sent by the client.
3. **getText –** This function returns the current text to the client.
4. **paint** – This function displays the textbox. It checks for the selection data to dispay highlighted text portions.
5. **update_cursor** – This method is called on navigation to update the cursor. It changes the start location if the cursor goes beyond the allowable character.
6. **react** – This method is called when a mouse click occurs within the region of the text. It reacts by updating the cursor and displaying it. Also sets this text as the focus component.
7. **keyPress –** This is called when the key is pressed with this as the component in focus. It either displays the character typed or updates cursor on a navigation keys or selects the text on selection keys or take any corresponding editing actions.
8. **copy** – This is invoked on request from the client or when the special key is pressed. It stores the selected portion into a globally shared clipboard.

9. **cut** – This is called on message from client or on a special key. This stores the selected portion into the clipboard. It also removes the selected portion of text.
10. **paste** – This inserts the text from the global clipboard to the current cursor location. It pushes the rest of the text backwards.
11. **isChanged** – This function returns whether the text has been changed after the text has been set.

TextArea

This class also inherits from the component class. It is almost similar to the text but has another dimension for all parameters. The text is a two dimensional array. The starting position also indicates the row and column. The cursor is also a two dimensional quantity indicating the row and column. The methods are also similar but it also has the functionality to manipulate many rows.

Label

This class inherits from the component class. It is used for displaying any information. It has only one member: the label – the text that is to be displayed.

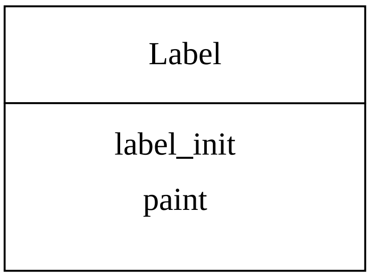


Figure 5.36 – Class Diagram of Label

The member functions are:

1. **label_init** – Initializes a newly created object of this class.
2. **paint –** Paints the label to the allowable width.

FolderView

The folderview component is used for providing file and directory browsing. Thus component inherits from the component class.

This component has the following members

1. cache – This is an array of DIRENT structure that is used to cache up a folders contents. The use of this cache is to reduce the overhead of reading the file from the file system for every repaint.
2. inited – This field is used to check whether the current cache is valid or not.
3. dirname – This member contains the name of the present working directory.
4. no_entries – This field contains the number of valid entries into the components DIRENT cache.

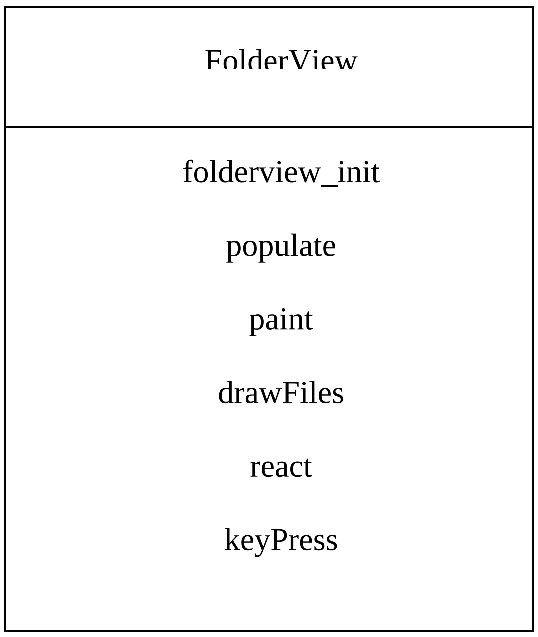


Figure 5.37 – Class Diagram of Folderview

The folderview component has the following member functions:

1. folderview_init – This is used to initializ the folderview component and inform it about its coordinates and size.

2. populate – This method is provided to allow the client application to populate the cache of the folderview.

3. paint – This function is exposed to draw the component. The GUI Server when painting this component invokes this method.

4. drawFiles – This is an internal function that is used to paint the internal contents of a folder basically from the cache. If the inited is set to false it calls informs the client application to populate the cache with valid entries.

5. react – This is basically an event handler function that is used to handle events from the mouse.

6. keyPress – This is another event handler that handles events from the keyboard. The folderview component allows only *arrow keys, page up key, page down key* for navigation and the *enter key* for opening files and folders.

Desktop and Taskbar Management

The various methods involved in the desktop and taskbar painting are:

1. **drawDesktop** – It paints all the icons onto the respective locations
2. **drawTaskbar** – It paints the start menu, the various window tasks and the timer.
3. **drawTime** – This is called from the timer interrupt whenever the time needs to be updated.

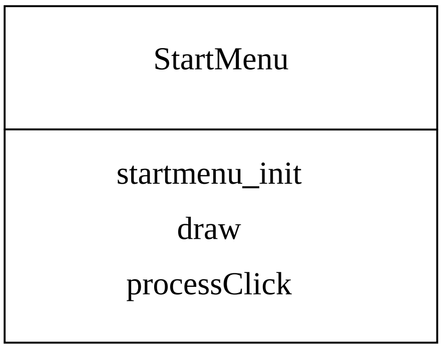


Figure 5.38 – Class Diagram of StartMenu

The various methods involved in the desktop and taskbar event management are:

1. **ProcessStartMenu** – It processes the clicking of items in the start menu. It uses the current state of the start menu to either display it or hide it or process an event.

2. **ProcessTask** – It processes the clicking on the task links so that the window is toggled between maximized/restored to minimized.

The start menu is implemented as a class. The various data present in it are:

1. startopened – indicates whether the menu is clicked open or not.
2. x , y , width , height
3. COLOR *bk_buf – Used for backing up the background data
4. no_items - The number of menu items

The various methods present are:

1. **startmenu_init** – Initializes the values
2. **draw** – paints the menu depending on the state
3. **processClick** – called when a mouse click occurs.

Menu Management

The menu is implemented as classes, which are added to windows. There are two base classes: menu and the menu bar. There is one another structure for the menu item.

The menu item is composed of the two elements:

1. label – The string displayed in the drop-down menu item
2. Code – This is the id of the item. The client sends it when the menu is created. It is sent to the client to indicate that this menu item has been clicked so that suitable actions can be taken at the client side.

The next is a class that corresponds to each drop-down **menu**. The various data elements are:

1. Container id and pointer
2. x, y, width, height
3. Number of items present
4. Array of the menu items present
5. COLOR *buf – a backup buffer for caching the background data when the menu is dropped down. When the menu is restored, the background information is painted.

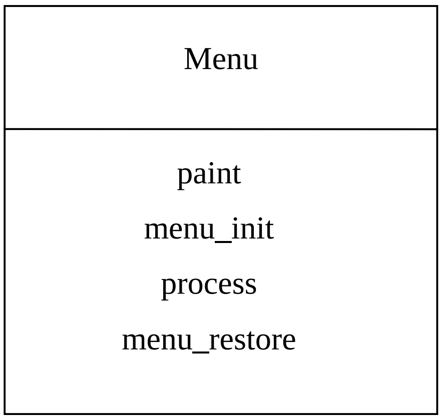


Figure 5.39 – Class Diagram of Menu

The member functions in the menu class are:

1. **menu_init** – This initializes the data related to the menu.
2. **paint** – It paints the menu after grabbing the data behind it.
3. **process** – It is called when a click occurs on this drop down menu. It identifies the corresponding menu item and sends a message indicating it has been clicked.
4. **menu_restore** – This is done when the menu is being removed when the menubar is clicked again. This restores the background data that was cached.

The next is a class that corresponds to the **menu bar**. The various data elements are:

1. Container id and pointer
2. x, y, width, height
3. Number of top-level menus
4. Current menu if, any that is opened.
5. A structure containing the menu pointer and label of each top-level menu.

The various member functions that are present in the class are:

1. **menubar_init** – This initializes the data related to the menubar, the menu, and the menu items.
2. **paint** – It paints the menu bar below the title bar of the window.
3. **process** – It is called when a click occurs on this window. It checks whether it is within a specific menu. If so, it displays or removes the drop-down menu depending on the current state of the menu.

4. **freeAll** – It is called when the menu is destroyed. It removes all the associated pointers.

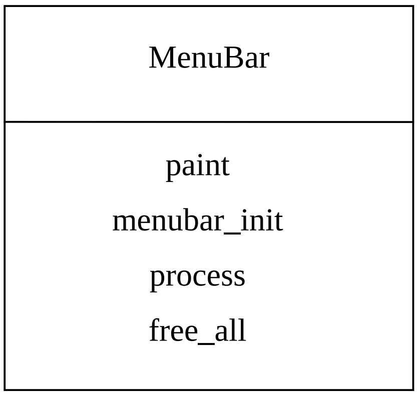


Figure 5.40 – Class Diagram of Menubar

Global Objects

There are two objects of the following two classes that are shared by all text and textarea components. Only one cursor object is present at any time. The clipboard is also global allowing for the data movement from one text/textarea to other.

Cursor

The cursor is implemented with a class. If a cursor is present, then it is painted once in a specific number of timer events to get the blinking effect.

The data elements present in the cursor class are:

1. The location of the cursor – It is set by the component that requires a cursor.
2. State – It indicates whether a cursor is present or not.
3. Number of times – It indicates the number of timer interrupts after which the cursor is repainted.

The member functions of this class are:

1. **Cursor_init** – This is called once when the global cluster is created.
2. **setCursor**(POINT pt) – This is called by the various components which sets the cursor to the specific locations.
3. POINT **getCursor**() – This returns the current cursor location.
4. **show –** This sets the state of the cursor to be visible.

5. **hide** – This sets the state of the cursor to be invisible.
6. **paint** – This paints the cursor and is called by the timer interrupt handling routine. It is alternatively painted in black and white.
7. **erase** – This is called when the cursor is being removed.

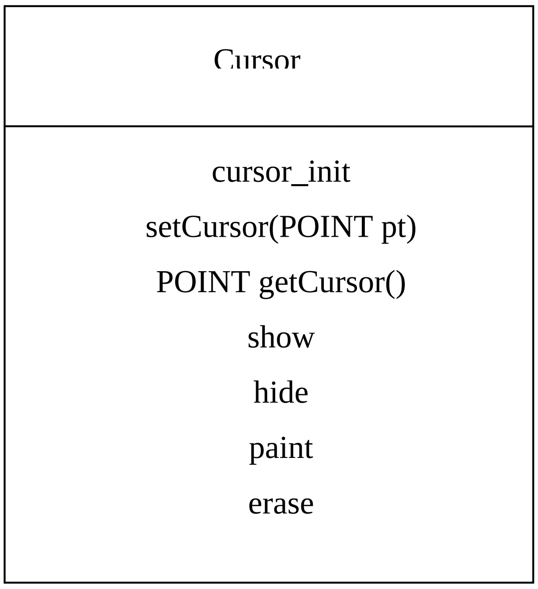


Figure 5.41 – Class Diagram of Cursor

Clipboard

There is a globally shared structure, which contains the copied text and an indication whether there is any text in clipboard. When a text/ textarea copy is performed, then the data is placed in this buffer. When a paste is done, the data in this clipboard is copied from here to the corresponding component.

Request Handler Methods

The GUI server gets request methods from clients to perform certain operations and messages from the kernel on certain events like mouse click or key press. The processing done on each of these requests is as follows:

1. **Create** – for the window and all components. The steps done are:
   - i) Extract the information from buffer. For components, check if container is present.
   - ii) Allocate memory space for the component or window by using the memory allocator provided by the kernel.

   iii) Add the alias id mapping and the pointers to the corresponding structure based on the process id.
   iv) Initialize the allocated space with the passed values through the respective init methods.

2. **Show** – for windows and dialogs
   i) For window, check if the alias id is present. If so, get the actual id and add it to the z-order on top. Change the state.
   ii) For dialog, check if the alias id is present. If so, add the dialog pointer to the parent window
3. **Hide** – for windows and dialogs
   i) For window, check if the alias id is present. If so, get the actual id and remove it from the z-order. Change the state.
   i) For dialog, check if the alias id is present. If so, remove the dialog pointer from the parent window
4. **Resize –** for windows. Check if the alias id is present. If so, change the x, y, width and height accordingly.
5. **Enable** – for window and all components. Check if the alias id is present. If so, change the style accordingly.
6. **Disable** – for window and all components. Check if the alias id is present. If so, change the style accordingly.
7. **Destroy** – for window and all components. The steps performed for this
   i) Check if alias id is present.
   ii) Call the freeAll method in window for freeing all components
   iii) Deallocate memory space for the component or window by using the memory allocator provided by the kernel.
   iv) Remove the alias id mapping and the pointers to the corresponding structure based on the process id.
8. **Action –** for mouse and keyboard events
   i) **Mouse** – Check if click is on taskbar items. If so call the process function of taskbar. Process the z-order from top to check if click is within a window. If so , check for

clicks on specific components by calling the **findEntity** method. If there is a click on the component, ask the component to do the necessary steps by calling its **react** method. Send the message to kernel if click is on button or menu. Set the corresponding window as focus. Move the window to the top of z-order. If the event is **move**, then call the window's move event.

ii) **Keyboard –** Check if there is any window on focus. If so, call the respective **keyPress** method. This method checks if there is any component on focus. If so, it calls the **keyPress** method of the component.

9. **Finished –** the client sends it after the processing of a button click event is over. The steps done are:

- i) Check if the alias id is present.
- ii) If so, enable the parent window so that it can receive the mouse and keyboard methods.

Painting Method

This method is called whenever any change is made to the z-order

1. Paint the window from the top of the z-order by calling the paint method of the window. It calls the paint method of the components, dialog and menubar.
2. Draw the taskbar items, timer and the start icon.
3. Draw the desktop and the icons.

6. SAMPLE APPLICATIONS

The Dynacube operating system's main modules were tested using three sample applications. They are the:

1. DynaPlorer
2. DynaPad
3. DynaCalc

6.1 DYNAPLORER

This Gui-based file browser helps in creation, removal, and copying of the folders and files. It uses the folderview component of GUI. It has a menu driven interface for file and directory based operations. This acts as a test bed for the following modules:

File System – As it uses the file and directory create, open, read, write and close calls.

Floppy Device Driver - As the file system uses the floppy's read and write sector.

GUI Server – It uses the window, component and menu, and dialog functions of GUI.

Kernel Module – It runs as a separate process and thus uses the process and memory modules of kernel.

Device Driver Module – It uses the mouse and keyboard handling functionality provided by the GUI server.

6.2 DYNAPAD

This is a GUI-based file editor, which helps in creating and editing files. It uses the textarea component of the GUI library. It has a menu driven interface for *file* and *edit* operations. It maintains a session interface and indicates an error when a file that is changed is not saved. It also acts as a test bed for the all modules that were tested in the DynaPlorer application.

6.3 DYNACALC

This is a GUI-based calculator. It is used for performing the basic calculations. It uses the button component of GUI for the getting input from the user, and the text component, which is disabled to direct keyboard input. This is done to prevent the user from entering irrelevant information into the text component. This application acts as a test bed for the following modules:

GUI Server – It uses the window, component and menu, and dialog functions of GUI.

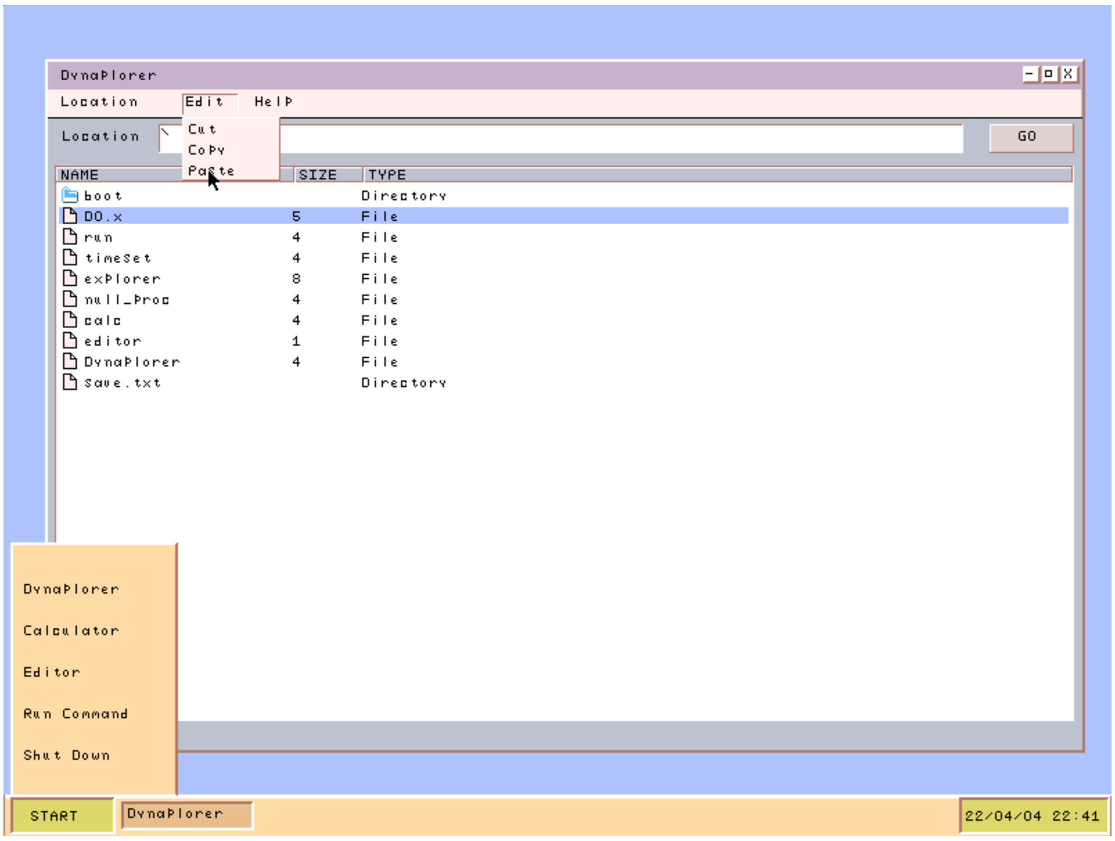
Kernel Module – It runs as a separate process and thus uses the process and memory modules of kernel.

Device Driver Module – It uses the mouse and keyboard handling functionality provided by the GUI server.

7. SCREENSHOTS

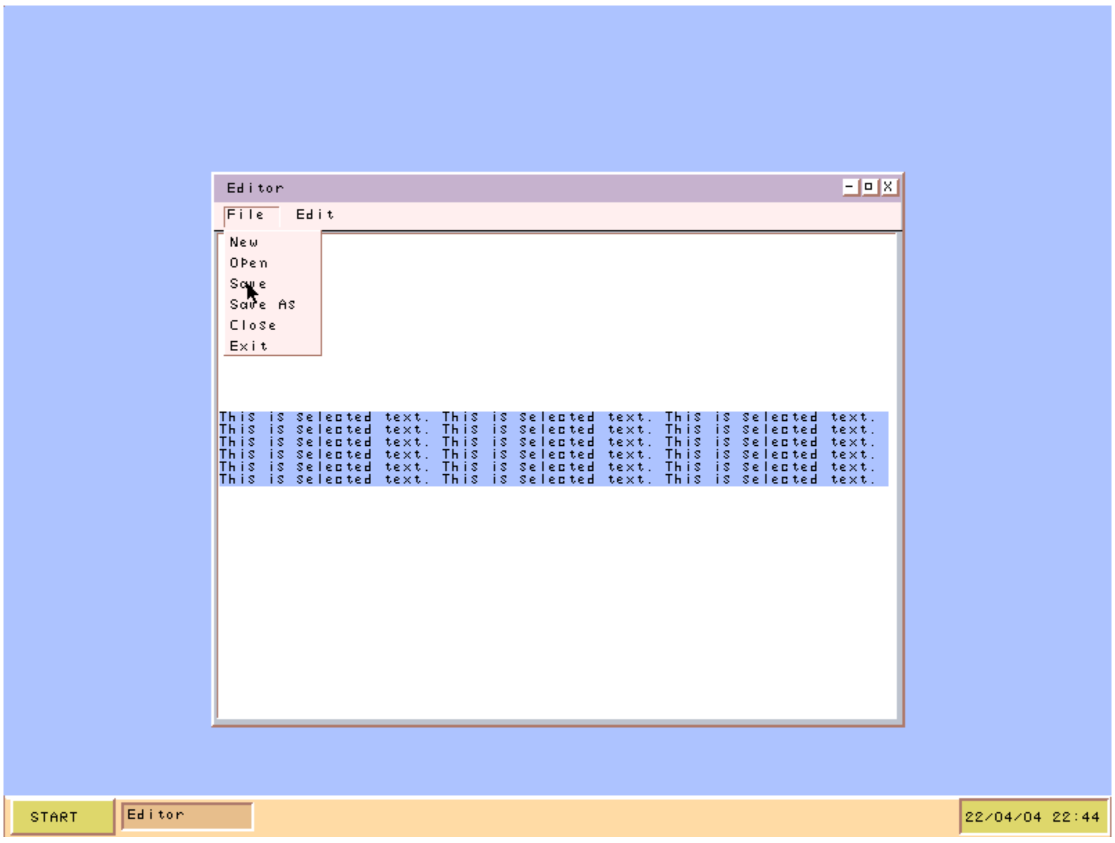
The following screenshots show the user-friendly Graphical User Interface (GUI) provided by the Dynacube operating system.

7.1 DynaPlorer Screenshot



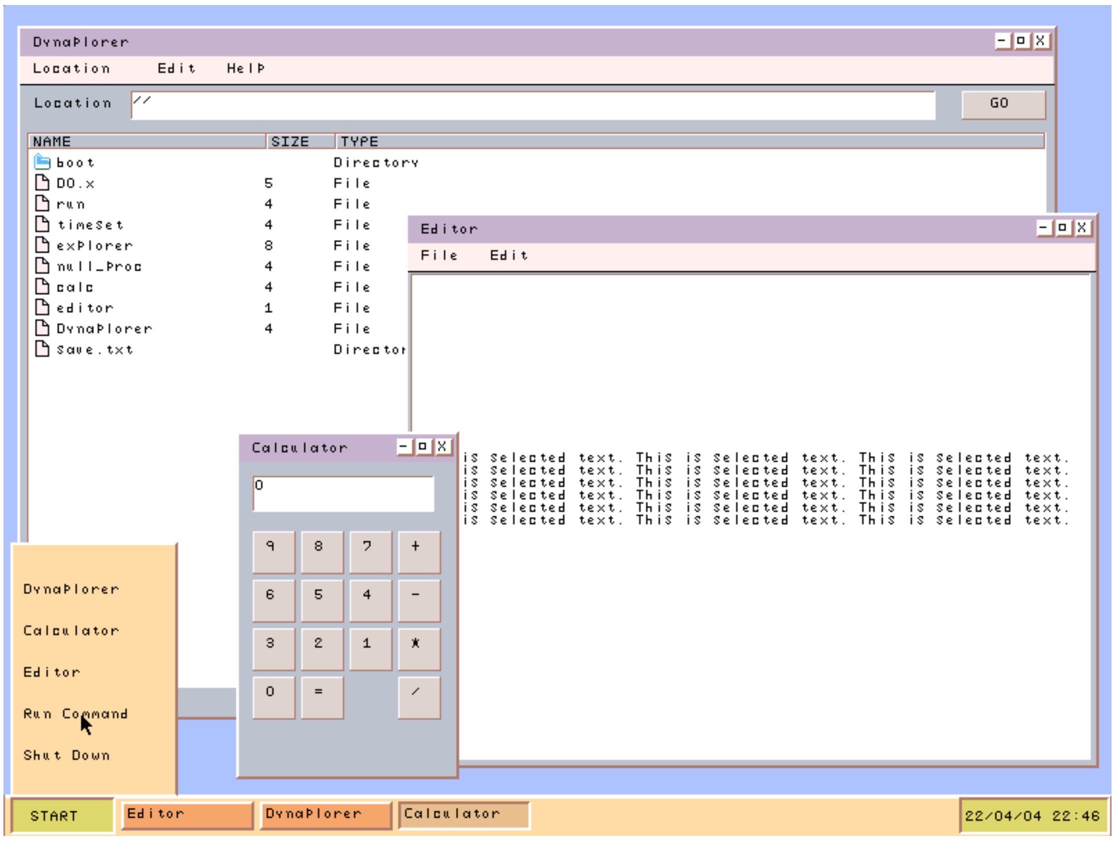
NAME	SIZE	TYPE
boot		Directory
DO.x	5	File
run	4	File
timeSet	4	File
explorer	8	File
null_Proc	4	File
calc	4	File
editor	1	File
DynaPlorer	4	File
Save.txt		Directory

7.2 DynaPad Screenshot



```
This is selected text. This is selected text. This is selected text.
This is selected text. This is selected text. This is selected text.
This is selected text. This is selected text. This is selected text.
This is selected text. This is selected text. This is selected text.
This is selected text. This is selected text. This is selected text.
This is selected text. This is selected text. This is selected text.
```

7.3 DynaCalc with other applications – Test for overlapped windows



Location //

```
NAME              SIZE   TYPE
boot                     Directory
DO.x               5     File
run                4     File
timeSet            4     File
exPlorer           8     File
null_Proc          4     File
calc               4     File
editor             1     File
DynaPlorer         4     File
Save.txt                 Director
```

```
0
```

```
is selected text. This is selected text. This is selected text.
is selected text. This is selected text. This is selected text.
is selected text. This is selected text. This is selected text.
is selected text. This is selected text. This is selected text.
is selected text. This is selected text. This is selected text.
is selected text. This is selected text. This is selected text.
```

22/04/04 22:46

8. CONCLUSION

The project has successfully achieved its stated goals. We have implemented and thoroughly tested all the modules – Dynacube kernel, Dynacube Disk Server, Dynacube File Server, and the Dynacube GUI Server module. This project has improved our understanding of operating systems and helped us analyze it from a different perspective. We have unraveled the mysteries shrouding computer hardware, system level programming, and system software. We have mastered the control flow analysis of Intel processors. We have honed our design skills, and have grasped the concept of virtualization of hardware resources.

We have also found out that using advanced technologies in the development of large projects can help in considerably reducing the development time. We used *System Simulators* to test our operating system during the initial phases of our project, which drastically reduced the development time. Otherwise we would have wasted far more time in booting up our system rather than developing our Dynacube operating system.

Having completed our project, we are looking forward to make the following enhancements to our project in near future:

- Extend Dynacube to 64bit architectures.
- To provide SMP support.
- To port GCC and G++ to our Dynacube operating system.
- To include File System support for EXT2, EXT3 and NTFS formats.
- To create separate ABIs for porting applications written for Linux and Windows.
- To provide Unicode support.
- To provide more sophisticated user interfaces.
- To design a configurable Window manager.

9. BIBLIOGRAPHY

The following literatures were consulted during the course of development of Dynacube Operating System.

Design Literature

- Andrew S. Tanenbaum and Albert S. Woodhull, Operating Systems: Design and Implementation
- Douglas Comer and Timothy V. Fossum, Operating System Design: The XINU Approach
- Microsoft Official FAT12 documentation

Hardware Literature

- IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture
- IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference
- IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming guide
- VESA BIOS EXTENSION (VBE) Core Functions Standard Version: 3.0
- Adam Chapweske, PS/2 Keyboard Interface.
- Adam Chapweske, PS/2 Mouse Protocol.
- NEC µPD765 and Intel 82072-7 Floppy Disk Controller specification.
- Xavier's Programmable Interrupt Controller documents
- Programmable System Timer documents.
- CMOS tutorials
- DMA tutorials

Implementation Literature

- Andrew S. Tanenbaum and Albert S. Woodhull, Operating Systems: Design and Implementation
- Official GCC man pages
- NASM Documentation
- Randall Hyde, Art of Assembly
- Brennan's Guide to Inline Assembly by Brennan Underwood
- Tim Robinson, Virtual 8086 mode

Web-links

- **www.osdev.org**
- **www.osdever.net**
- **www.osjournal.com**