بسمه تعالی

دانشگاه علم و صنعت ایران
دانشکده مهندسی کامپیوتر

جزوه درس برنامه‌سازی سیستم

مدرس: محمد عبداللهی ازگمی

فصل ۱: مبانی برنامه‌سازی سیستم
(System Programming Basics)

۱-۱ برنامه‌سازی سیستم چیست؟

از دید افراد مختلفی که با کامپیوتر کار می‌کنند این پرسش می‌تواند پاسخهای متفاوتی داشته باشد. برخی به هر برنامه به دید یک "سیستم" نگاه می‌کنند و **برنامه‌سازی سیستم** را فرآیند تبدیل یک مسأله به یک برنامه‌ی قابل اجرا می‌دانند. برخی به نوشتن برنامه‌های خاص برای یک سیستم کامپیوتری خاص، برنامه‌سازی سیستم می‌گویند.

گرچه تعریف دوم تا حدی درست است، اما به‌طور دقیق‌تر از دید ما، برنامه‌سازی سیستم دارای تعریف مشخص و معینی است: **برنامه‌سازی سیستم نوع خاصی از برنامه‌سازی است که نیازمند اطلاع از جزئیات فنی سیستم مورد نظر و دسترسی و مدیریت منابع سخت‌افزاری سیستم به‌صورت دلخواه است.** این نوع برنامه‌سازی عموماً برای نوشتن **نرم‌افزارهای سیستمی**[۱] مورد استفاده قرار می‌گیرد.

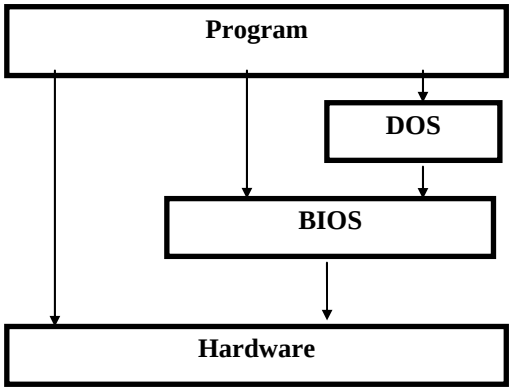
برنامه‌سازی کاربردی[۲] **در مقابل برنامه‌سازی سیستم:** برنامه‌سازی کاربردی شامل مدیریت و ارائه‌ی اطلاعات در قالب یک برنامه‌ی کامپیوتری بوده و نیازمند نگهداری اطلاعات در قالب ساختارهای داده‌ای (نظیر آرایه، رکورد و غیره) و پردازش آنها است. الگوریتم‌های مورد استفاده در برنامه‌سازی کاربردی **مستقل از سیستم**[۳] بوده و برای اغلب کامپیوترها قابل تعریف هستند. اما، روشی که اطلاعات به برنامه منتقل می‌شود و طریقی که اطلاعات نمایش داده می‌شوند یا چاپ می‌شوند، **وابسته به سیستم**[۴] است. در برنامه‌سازی سیستم، کنترل هرنوع سخت‌افزاری که اطلاعات را به کامپیوتر می‌فرستد یا از آن دریافت می‌کند، انجام می‌شود. برای پردازش این اطلاعات نیاز به برنامه‌های کاربردی هستیم. از این‌رو، برای بهره‌برداری از سخت‌افزار کامپیوتر، هر دو نوع برنامه‌سازی لازم هستند.

۱-۲ مدل سه لایه‌ای

مهمترین هدف برنامه‌سازی سیستم، دسترسی و مدیریت سخت‌افزار کامپیوتر است. نوعِ سیستمِ کامپیوتری که مورد نظر ما است، **کامپیوتر شخصی(PC)**[۵] است. این نوع کامپیوتر دارای معماری خاص بوده و اجزاء سخت‌افزاری آن با سایر معماری‌ها متفاوت است. نوعی از برنامه‌سازی سیستم که مورد نظر ما است، برنامه‌سازی همین نوع سیستم کامپیوتری بوده و هدف آن دسترسی و مدیریت سخت‌افزار PC است.

در محیط PC، دسترسی به سخت‌افزار از طریق ROM-BIOS یا DOS نیز امکان پذیر است. ROM-BIOS، مخفف Read Only Memory-Basic Input/output System است و DOS هم مخفف Disk Operating System است. BIOS و DOS واسطهای نرم‌افزاری[۶] هستند که به‌طور خاص برای مدیریت سخت‌افزار ایجاد شده‌اند. در شکل (۱-۱) مدلی سه‌لایه‌ای از ارتباط برنامه‌ها با BIOS و DOS ارائه شده است.

شکل ۱-۱: مدل سه‌لایه‌ای



مهمترین مزیت استفاده از DOS یا BIOS این است که یک برنامه نیازمند دسترسی مستقیم به سخت‌افزار نیست و در عوض یک روال[۷] BIOS یا DOS را فراخوانی می‌کند تا کار موردنظر را برایش انجام دهد. روالهای BIOS پس از انجام یک کار، اطلاعات وضعیت[۸] را به برنامه برمی‌گردانند تا برنامه از نتیجه‌ی عمل آگاه شود. این واسطها باعث می‌شوند که برای نوشتن برنامه‌ها، هزینه و زمان کمتری صرف شود.

---

۱ system softwares
۲ application programming
۳ system independent
٤ system dependent
٥ Personal Computer
٦ software interfaces
۷ routine
۸ status information

مزیت دیگر استفاده از واسطهای فوق، عدم وابستگی برنامه به مشخصات فیزیکی سخت‌افزار است. برای مثال، نمایش اطلاعات بر روی صفحه‌ی نمایش با استفاده از کارتهای ویدئویی[۱] تک رنگ[۲] و رنگی (نظیر EGA، VGA، SuperVGA و غیره) با هم دارای تفاوتهای اساسی است. اگر برنامه بخواهد خودش اطلاعات را با برنامه‌سازی کارت ویدئویی، نمایش دهد، باید روالهای مجزایی برای هر کدام از انواع کارت ویدئویی نوشته شود. اما اگر از روالهای BIOS استفاده نماید، برنامه مستقل از نوع کارت ویدئویی خواهد بود. چون انواع مختلف کارتهای ویدئویی بوسیله‌ی BIOS پشتیبانی می‌شوند.

ROM-BIOS ۳-۱

در شکل (۱-۱) یک مدل سه‌لایه‌ای را مشاهده می‌کنید که واسط BIOS از DOS به سخت‌افزار نزدیک‌تر است. BIOS توابعی را برای دسترسی و مدیریت لوازم[۳] زیر فراهم می‌کند:

۱. کارت ویدئویی،
۲. حافظه (RAM)،
۳. دیسکت،
۴. دیسک سخت[۴]،
۵. درگاههای سریال[۵] (Com1 و Com2)،
۶. درگاههای موازی[۶] (LPT1 و LPT2)،
۷. صفحه کلید،
۸. ساعت بی‌درنگ عمل‌کننده‌با باتری[۷].

گرچه می‌توان بدون استفاده از BIOS نیز با سخت‌افزار ارتباط برقرار نمود، اما عموماً بهتر است از طریق این توابع استاندارد به کامپیوتر دسترسی پیدا کرد، تا برنامه مستقل از سخت‌افزار باشد.

BIOS در یک تراشه‌ی حافظه[۸] از نوع ROM در برد اصلی[۹] سیستم PC وجود دارد . بلافاصله پس از روشن شدن کامپیوتر، قابل دسترسی است. توابع موجود در BIOS، وظایفی نظیر آزمون حافظه و بررسی لوازم جانبی را پس از روشن شدن کامپیوتر برعهده دارند. به همراه هر نوع جدید برد اصلی PC و مدل CPU، نگارشهای[۱۰] جدیدتری از BIOS معرفی می‌شوند. از معروفترین انواع BIOS می‌توان از Award و AMI نام برد.

DOS ۴-۱

DOS که در حقیقت وظیفه‌ی یک سیستم عامل را برعهده دارد، بوسیله‌ی BIOS از روی دیسک در حافظه‌ی کامپیوتر قرار داده شده و اجرا می‌شود. از جمله توابعی که در DOS فراهم شده است، توابعی هستند که امکان کار با دیسک را فراهم می‌نمایند و این امر دلیل نام‌گذاری DOS است. یعنی سیستم عاملی که توانایی کار با دیسک را دارد.

در کنار BIOS، DOS نیز توابعی را برای دسترسی به سخت‌افزار فراهم می‌کند. از آنجایی ک DOS به سخت‌افزار به عنوان **لوازم منطقی**[۱۱] نگاه می‌کند، نه مثل BIOS به عنوان لوازم فیزیکی، لذا توابع DOS سخت‌افزار را به‌طریقی متفاوت مدیریت می‌کنند. برای مثال، BIOS به گرداننده‌های دیسک[۱۲] به‌عنوان گروهی از شیارها[۱۳] و بخشها[۱۴] نگاه می‌کند، اما DOS آنها را به‌عنوان گروه‌هایی از فایلها و فهرست‌ها[۱۵] می‌بیند.

---

۱ video cards
۲ monochrome
۳ devices
۴ harddisk
۵ serial ports
۶ parallel ports
۷ battery-operated real-time clock
۸ memory chip
۹ mother board
۱۰ version
۱۱ logical devices
۱۲ disk drives
۱۳ tracks
۱۴ sectors
۱۵ directory

اگر بخواهید ۱۰۰۰ کاراکتر اول یک فایل را بر روی صفحه نمایش ببینید، از طریق BIOS اگر عمل کنید، باید مکان فایل بر روی درایو (شماره شیار و بخش) را به BIOS بگویید. اما با استفاده از توابع DOS، کافی است به DOS بگوییم که فایل با نام داده‌شده را در گرداننده‌ی A: یا C: باز نموده و نمایش دهد.

گرچه اغلب اوقات DOS از طریق BIOS به سخت‌افزار دسترسی دارد، ولی گاهی اوقات هم به‌صورت مستقیم با سخت‌افزار ارتباط برقرار می‌کند.

۱-۵ انتخاب روش دسترسی به سخت‌افزار

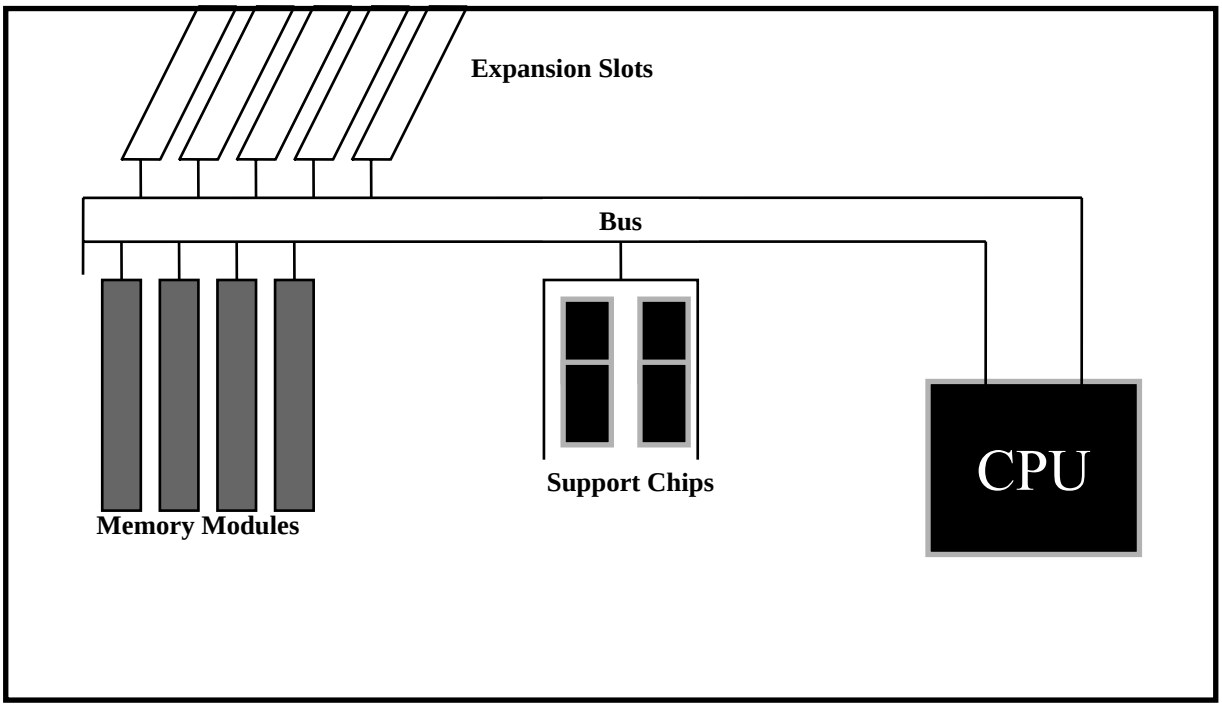
از کدام توابع استفاده کنیم؟ در ادامه‌ی درس در مورد اینکه DOS و BIOS چه توابعی دارند صحبت خواهیم نمود. اما ابتدا باید تصمیم بگیریم که کدام روش دسترسی را انتخاب نماییم: DOS یا BIOS یا دسترسی مستقیم به سخت‌افزار؟ گاهی اوقات مجبور به دسترسی مستقیم هستیم. زیرا ممکن است عمل موردنظر ما را هیچکدام از توابع DOS یا BIOS انجام ندهند. برای مثال اگر بخواهید به کارت ویدئویی بگویید که یک خط یا دایره رسم کند، هیچ تابعی در DOS یا BIOS برای این منظور پیدا نخواهید کرد. بنابر این، مجبورید که یک روال برای دسترسی مستقیم به سخت‌افزار و برنامه‌ریزی آن برای رسم خط یا دایره بنویسید، یا آنکه از یک تابع کتابخانه‌ای نرم‌افزاری[۱] تهیه شده بوسیله‌ی دیگران استفاده کنید.

انتخاب DOS یا BIOS ؟ این انتخاب به نوع کاربرد بستگی دارد. برای مثال اگر می‌خواهید در برنامه با فایل کار کنید، باید از توابع DOS استفاده کنید. ولی اگر می‌خواهید یک دیسکت را قالب‌بندی[۲] کنید، باید از توابع BIOS استفاده کنید. هم توابع DOS و هم توابع BIOS، در برخی اوقات **کند** هستند. زیرا مدل سه‌لایه‌ای شکل (۱-۱) و نیز عمومی بودن توابع، باعث عدم بهینه‌بودن توابع این واسطها می‌شود. حتی به این دلیل هم گاهی اوقات مجبور به برنامه‌سازی مستقیم سخت‌افزار هستیم.

۱-۶ سخت‌افزار PC

بلوک دیاگرام اجزاء سخت‌افزار PC در شکل (۱-۲) نمایش داده شده است:

شکل ۱-۲: بلوک دیاگرام سخت‌افزار PC



اجزاء سخت‌افزار PC شامل موارد زیر است:

- **پردازنده:** CPU در PC، یکی از ریزپردازنده‌های سری 80X86 ساخت شرکت Intel است. اولین PC ساخت شرکت IBM با نام System/23 DataMaster دارای ریزپردازنده‌ی ۸ بیتی ۸۰۸۵ بود. سپس به ترتیب ریزپردازنده‌های ۸۰۸۶، ۸۰۸۸، ۸۰۲۸۶، ۸۰۳۸۶، ۸۰۴۸۶ و Pentium از طرف شرکت Intel معرفی شدند و این پردازنده‌ها در مدلهای مختلف کامپیوترهای شخصی ساخت شرکت IBM یا شرکتهای دیگر مورد استفاده قرار گرفتند.

---

۱ software library function
۲ format

- **گذرگاه**[۱]: یکی از بخشهای اساسی سیستم PC است که وظیفه‌ی اتصال اجزاء مختلف آنرا دارد. گذرگاه در حقیقت یک کابل با ۶۲ خط اتصال است. پردازنده از طریق گذرگاه اطلاعات را از حافظه خوانده و پردازش می‌کند و دوباره در آن می‌نویسد.
- **تراشه‌های پشتیبانی**[۲]: پردازنده توانایی انجام همه‌ی اعمال موردنیاز در یک سیستم کامپیوتری را ندارد. از این‌رو تراشه‌های پشتیبانی مورد نیاز هستند تا اعمالی را که پردازنده قادر به انجام آن نیست، برعهده بگیرند. این تراشه‌ها که به آنها کنترلر[۳] نیز گفته می‌شود، بخشی از سخت‌افزار را کنترل می‌کنند و کارهایی را انجام می‌دهند و پردازنده را برای انجام کارهای مهمتر آزاد می‌گذارند. برخی از مهمترین این تراشه‌ها که در سیستم IBM PC وجود داشت عبارتند از:

- **DMA Controller (8237):** DMA مخفف Direct Memory Access است و این تکنیک به لوازمی نظیر دیسک سخت اجازه می‌دهد که داده‌ها را به‌طور مستقیم بر روی حافظه بنویسند. این تکنیک ضمن آنکه باعث نقل و انتقال سریع داده‌ها بین لوازم و حافظه می‌شود، باعث آزاد ماندن پردازنده برای انجام سایر کارها می‌شود.

- **Interrupt Controller (8259):** وقفه[۴] راهی برای آگاه کردن پردازنده نسبت به آمادگی لوازم (نظیر دیسک سخت، صحفه کلید و غیره) برای نقل و انتقال داده است. با استفاده از وقفه‌ها، نیاز نیست که پردازنده منتظر آماده‌شدن یک وسیله برای تبادل داده باشد(با روش **سرکشی**[۵])، بلکه هر گاه وسیله آماده شد، یک سیگنال کنترلی به پردازنده می‌فرستد و باعث توقف در عملیات آن شده و درخواست سرویسی از پردازنده می‌کند. مثلاً با زدن یک کلید، صفحه کلید یک سیگنال وقفه به پردازنده می‌فرستد و زده شدن کلید را به آن اطلاع می‌دهد. وقوع وقفه از طرف لوازم مختلف باید دارای نظم باشد تا تداخلی بین آنها پیش نیاید و متناسب با الویت لوازم به آنها سرویس داده شود. تراشه‌ی ۸۲۵۹ وظیفه‌ی مدیریت وقفه‌های سخت‌افزاری را در سیستم PC برعهده دارد.

- **Programmable Peripheral Interface (8255):** این تراشه پردازنده را به لوازم جانبی، نظیر صفحه کلید یا مولد صوت[۶] متصل می‌کند و نقش یک میانجی را بازی می‌کند.

- **The Clock (8248):** اگر پردازنده را مغز سیستم کامپیوتری فرض کنیم، ساعت قلب آن خواهد بود. این قلب چند میلیون بار در ثانیه ضربان دارد (مثلاً 233 MHZ) و وسیله‌ای برای همزمان‌سازی پردازنده و لوازم دیگر سیستم کامپیوتری است.

- **The Timer (8253):** این تراشه به‌عنوان یک شمارشگر و نگهدارنده‌ی زمان در سیستم مورد استفاده قرار می‌گیرد. این تراشه دارای خروجی‌هایی است که سیگنالهای الکتریکی منظمی را صادر می‌کند و تناوب آن قابل برنامه‌ریزی است، اینکه سیگنال تایمر در چه فواصل زمانی صادر شود.–

- **حفره‌های گسترش**[۷]: برخی از تراشه‌های دیگر مورد نیاز در سیستم PC در این قسمت قرار داده می‌شوند. دو مورد مهم عبارتند از:

- **CRT Controller (6845):** این تراشه وظیفه‌ی کنترل لوله‌ی اشعه‌ی کاتدی[۸] را بر عهده دارد و در کارت ویدئویی قرار دارد. وظیفه‌ی این کنترلر، مدیریت و کدبندی نحوه‌ی نمایش اطلاعات در صفحه‌ی نمایش است.

- **Disk Controller (765):** این کنترلر هم در قسمت حفره‌های گسترش نصب می‌شود و توسط سیستم عامل نشان‌دهی شده و گرداننده‌ی دیسک را مدیریت می‌کند. حرکت دادن هد دیسک برای خواندن و نوشتن اطلاعات در بخشهای مختلف دیسک، وظیفه‌ی این کنترلر است.

- **حافظه**: در سیستمهای PC دو نوع حافظه وجود دارد. حافظه‌های فقط خواندنی یا ROM[۹]، که داده‌ها و کدهای غیرقابل تغییر در آن نگهداری می‌شود و نمونه‌ی مشخص آن، ROM-BIOS است. نوع دوم حافظه، حافظه‌ی خواندنی–نوشتنی است که به‌صورت تصادفی قابل دستیابی است و به آن RAM[۱۰] گفته می‌شود. این نوع حافظه محل نگهداری داده‌ها و پردازش آنها بوسیله‌ی پردازنده است. اولین نوع PC دارای 16KB حافظه بود و در PCهای امروزی امکان قرار دادن چند صد MB حافظه وجود دارد. تراشه‌های حافظه به صورت ماجولهایی هستند که در مکانهای مشخص در برد اصلی سیستم نصب می‌شوند و قابل کم و زیاد شدن هستند.

سیستم‌های عامل مختلف توانایی استفاده از همه‌یا بخشهایی از حافظه را دارند. سیستم عامل DOS نهایتاً می‌تواند 640KB حافظه را مورد استفاده قرار دهد و اگر مقدار حافظه بیشتر باشد، مابقی بلااستفاده می‌ماند. در PC، برای مدیریت حافظه و تخصیص آن به برنامه‌ها، حافظه‌به قطعه‌های[۱۱] 64KB تقسیم می‌شود. بنابر این کل حافظه‌های موجود در سیستم PC دارای نمای زیر است:

---

۱ Bus
۲ support chips
۳ controler
۴ interrupt
۵ polling
۶ speaker
۷ expansion slots
۸ Cathode Ray Tube (CRT)
۹ Read Only Memory
۱۰ Random Access Memory
۱۱ segment

شکل ۳-۱: نمایی از قسمت‌بندی حافظه‌ی PC

Block	Address	Contents
15	F000:0000 - F000:FFFF	ROM-BIOS
14	E000:0000 - E000:FFFF	Free for ROM Cartridge
13	D000:0000 - D000:FFFF	Free for ROM Cartridge
12	C000:0000 - C000:FFFF	Additional ROM-BIOS
11	B000:0000 - B000:FFFF	Video RAM
10	A000:0000 - A000:FFFF	Additional Video RAM (EGA/ VGA)
9	9000:0000 - 9000:FFFF	RAM from 576K to 640K
8	8000:0000 - 8000:FFFF	RAM from 512K to 576K
7	7000:0000 - 7000:FFFF	RAM from 448K to 512K
6	6000:0000 - 6000:FFFF	RAM from 384K to 448K
5	5000:0000 - 5000:FFFF	RAM from 320K to 384K
4	4000:0000 - 4000:FFFF	RAM from 256K to 320K
3	3000:0000 - 3000:FFFF	RAM from 192K to 256K
2	2000:0000 - 2000:FFFF	RAM from 128K to 192K
1	1000:0000 - 1000:FFFF	RAM from 64K to 128K
0	0000:0000 - 0000:FFFF	RAM from 0K to 64K

۱-۷ ثباتهای پردازنده[۱]

نکته‌ی مهم در مورد انواع پردازنده‌ها، مدل برنامه‌سازی آن است و در این میان، **ثباتها** نقش مهمی در برنامه‌سازی دارند. ثباتها، مکانهای حافظه‌ای در درون پردازنده هستند، و به این دلیل به مراتب سریع‌تر از RAM قابل دستیابی هستند. همچنین، ثباتها مکانهای ویژه‌ای برای انجام اعمال حسابی و منطقی بوسیله‌ی پردازنده هستند. برای برنامه‌سازی سیستم، آشنایی با این ثباتها، بسیار مهم است. زیرا، جریان اطلاعات بین برنامه، DOS و BIOS، با استفاده از ثباتها برقرار می‌شود. برنامه‌ها پارامترها را از طریق ثباتها به توابع واسطهای DOS یا BIOS ارسال نموده و نتایج و اطلاعات وضعیت را به عنوان مقادیر بازگشتی از این توابع دریافت می‌کنند.

از دیدگاه برنامه‌سازی سیستم با وجود معرفی مدلهای جدیدتر پردازنده‌های 80X86، این ثباتها از 8086 به بعد تغییر نکرده‌اند. این امر بدان دلیل است که BIOS و DOS وابسته به این پردازنده ایجاد شده‌اند و به‌دلایل سازگاری، بعداً تغییر اساسی از دید برنامه‌سازی سیستم در آنها داده نشده است. چون ثباتهای این پردازنده، ۱۶ بیتی بود، با استفاده از DOS، تحت پردازنده‌های ۳۲ بیتی 80386 یا 80486 هم فقط نصف ثبات، یا ۱۶-بیت آن قابل دستیابی است.

حال به انواع ثباتهای 8088 نگاه می‌کنیم. ثباتها به چند دسته تقسیم می‌شوند:

1. Common Registers:
   - AX (AH, AL) : Accumulator
   - BX (BH, BL) : Base
   - CX (CH, CL) : Count
   - DX (DH, DL) : Data
   - DI : Destination Index
   - SI : Source Index
   - SP :Stack Pointer
   - BP : Base Pointer
2. Segment Registers:

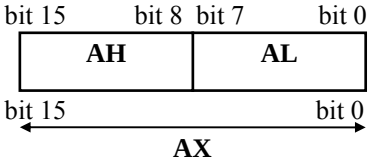
[۱] processor registers

- DS : Data Segment
- CS : Code Segment
- ES : Extra Segment
- SS :Stack Segment

3. Program Counter:
   - IP :Instruction Pointer

4. Flag Register
   - □□□□ODITSZ□A□P□C

ثباتهای مشترک: عموماً برای فراخوانی توابع DOS و BIOS مورد استفاده قرار می‌گیرند. همچنین این ثباتها بوسیله‌ی اعمال ریاضی (مثل جمع، تفریق و غیره) دستکاری می‌شوند. چهار ثبات ۱۶ بیتی AX، BX، CX و DX دارای ویژگی قابل قسمت‌شدن به دو بخش ۸ بیتی High و Low هستند. بنابر این، AX، AL و AH، هر سه قابل دستیابی هستند. مقدار AX با استفاده از مقادیر AL و AH قابل محاسبه است: AX = AH * 256 + AL



ثباتهای ۸ بیتی مثل CL یا CH برای خواندن و نوشتن داده‌های ۸ بیتی در حافظه و ثباتهای ۱۶ بیتی، مثل CX برای خواندن و نوشتن داده‌های ۱۶ بیتی قابل استفاده هستند.

ثبات پرچم[۱]**:** این ثبات برای برقراری ارتباط مابین دستورات پی‌درپی اسمبلی به‌کار می‌رود و برای این منظور وضعیت عملیات ریاضی و منطقی را نگهداری می‌کند. برای مثال با استفاده از Carry Flag یک برنامه می‌تواند تعیین کنید که آیا در هنگام جمع دو ثبات ۱۶ بیتی، حاصل بیش از ۶۵۵۳۵ شده است یا نه.

این ثبات هم ۱۶ بیتی است، ولی فقط ۹ بیت آن مورد استفاده قرار گرفته است که عبارتند از:

- Bit 0: CF (Carry Flag)
- Bit 2: PF (Parity Flag)
- Bit 4: AF (Auxiliary Flag)
- Bit 6: ZF (Zero Flag)
- Bit 7: SF (Sign Flag)
- Bit 8: TF (Trap Flag)
- Bit 9: IF (Interrupt Flag)
- Bit 10: DF (Direction Flag)
- Bit 11: OF (Overflow Flag)

در برنامه‌سازی سیستم با استفاده از زبانهای سطح بالا فقط دو پرچم ZF و CF مورد استفاده قرار می‌گیرند. زیرا اغلب توابع DOS و BIOS از آنها برای مشخص کردن رخداد خطا در هنگام عملیات استفاده می‌کنند.

ثبات نشانی[۲]**:** تعداد مکانهای حافظه که یک پردازنده می‌تواند به آنها دسترسی داشته باشد، به این ثبات بستگی دارد. هر قدر تعداد بیتهای این ثبات بیشتر باشد، حداکثر حافظه‌ی قابل دسترسی، بیشتر می‌شود. اگر این ثبات ۱۶ بیتی باشد، حداکثر ۶۵۵۳۵ مکان حافظه، قابل دستیابی است. به همین دلیل است که پردازنده‌های اولیه فقط قادر به دسترسی به 64K حافظه بودند. برای نشانی‌دهی 1MB حافظه، این ثبات باید دارای حداقل ۲۰ بیت باشد. اما در زمان طراحی ۸۰۸۸، امکان استفاده از ثبات نشانی ۲۰ بیتی وجود نداشت. از این‌رو، راه حل دیگری مورد استفاده قرار گرفت و دو عدد ۱۶ بیتی متفاوت برای تشکیل نشانی ۲۰ بیتی مورد استفاده قرا می‌گیرد. نخستین عدد در یک **ثبات قطعه**[۳] قرار می‌گیرد و دومین عدد در یک ثبات دیگر یا در یک مکان حافظه. به این ترتیب نشانی دارای دو بخش قطعه و مبدأ[۴] خواهد بود. نشانی قطعه، که در ثبات قطعه وجود دارد، شروع یک قطعه حافظه را نشان می دهد. نشانی مبدأ، شماره‌ی مکان حافظه را در درون قطعه مشخص می‌کند. چون نشانی مبدأ یک عدد ۱۶ بیتی است، پس یک قطعه نمی‌تواند دارای بیش از ۶۵۵۳۵ یا 64K مکان حافظه‌باشد.

---

۱ flag register
۲ address register
۳ segment register
٤ offset

چهار نوع ثبات قطعه در ۸۰۸۸ وجود دارد:

1. CS: Code Segment
2. DS: Data Segment
3. SS: Stack Segment
4. ES: Extra Segment

CS، از ثبات IP (Instruction Pointer) به عنوان نشانی مبدأ استفاده می‌کند. به IP، PC (Program Counter) هم گفته می‌شود و در حقیقت نشانی دستور بعدی برنامه را در حافظه نشان می‌دهد و به‌طور خودکار با اجرای یک دستور، یک واحد افزایش می‌یابد. **DS**، نشانی قطعه‌ای از حافظه را که شامل داده‌های مورد دستیابی برنامه است را نشان می‌دهد. **SS**، نشانی شروع پشته[۱] است و **ES**، بوسیله‌ی برخی از دستورات اسمبلی برای نشانی‌دهی بیش از 64K داده یا انتقال داده‌ها مابین قطعه‌های مختلف مورد استفاده قرار می‌گیرد.

۱-۸ درگاه‌ها[۲]

درگاه‌ها، واسطی بین پردازنده و سایر اجزاء سخت‌افزار سیستم هستند. هر درگاه مشابه یک دریچه‌ی ۸ بیتی ورودی یا خروجی است که به بخش خاصی از یک جزء سخت‌افزاری متصل است و با مقادیر بین ۰ تا ۶۵۵۳۵ قابل نشانی‌دهی است. درگاه‌ها در حقیقت، ثباتهایی در اجزاء سخت‌افزاری هستند که برای ارتباط با آن، مقداری در آن نوشته یا از آن خوانده می‌شود. پردازنده از گذرگاه برای دسترسی به درگاه‌ها استفاده می‌کنند و در پردازنده‌های 80x86 از دو دستورالعمل IN و OUT می‌توان برای دسترسی به درگاه‌ها در برنامه‌های اسمبلی استفاده نمود.

۱-۹ وقفه‌ها

وقفه مکانیسمی است که پردازنده را مجبور می‌کند تا اجرای برنامه‌ی جاری را متوقف نموده و یک روال خاصی را که وظیفه‌ی مدیریت وقفه[۳] را دارد را اجرا کند. وقفه‌ها هم برای کنترل سخت‌افزارها استفاده می‌شوند و هم برای برقرار ارتباط بین برنامه و توابع DOS و BIOS. به این ترتیب دو نوع وقفه داریم:

- **وقفه‌های نرم‌افزاری[۴]:** این نوع وقفه‌ها برای فراخوانی توابع DOS یا BIOS استفاده می‌شوند. در این حالت، توابع فراخوانی شده، همانند یک زیرروال[۵] برنامه عمل می‌کنند و پس از پایان اجرای تابع، به برنامه‌ی اصلی برگشت انجام می‌شود. توابع DOS و BIOS به ترتیب خاصی در حافظه قرار می‌گیرند و تشکیل یک جدول را می‌دهند که به آن **جدول بردار وقفه**[۶] می‌گویند. این جدول دارای دو ستون ورودی یا شماره تابع و نشانی شروع کد مربوطه در حافظه است. فراخوانی توابع با استفاده از ورودی مربوطه در این جدول انجام می‌شود. برای مثال، اگر تابع DOS 21H را فراخوانی کنیم، پردازنده نشانی شروع کد تابع 21H را از جدول بردار وقفه‌ها بدست آورده و به آن پرش می‌کند.
- **وقفه‌های سخت‌افزاری[۷]:** این وقفه‌ها بوسیله‌ی اجزاء سخت‌افزاری تولید می‌شوند و از طریق کنترلر وقفه‌ها به پردازنده منتقل می‌شوند. برخی از وقفه‌های سخت‌افزاری قابل ناتوان‌سازی[۸] هستند. یعنی باعث می‌شود که یک سخت‌افزار نتواند به پردازنده وقفه بدهد. اما برخی از وقفه‌ها غیرقابل ناتوان‌سازی هستند که به آنها NMI[۹] یا Trap گفته می‌شود.

۱-۷ محاوره‌ی سیستم[۱۰]

برای انجام کارها در سیستم PC، اجزاء مختلف سخت‌افزاری و نرم‌افزاری با هم کار می‌کنند. برای مثال برای خواندن یک کلید فشار داده شده بر روی صفحه کلید، موارد زیر مطرح هستند:

- **سخت‌افزار صفحه کلید:** وقتی یک کلید فشار داده می‌شود، سخت‌افزار صفحه کلید(در صورت توانا بودن وقفه‌های سخت‌افزاری)، وقفه‌ی شماره‌ی 09H را به پردازنده می‌فرستد. پردازنده پس از دریافت این وقفه، براساس الویت وقفه‌ها و اگر درحال اجرای یک وقفه با الویت بالاتر نباشد، روال مدیریت وقفه‌ی شماره‌ی 09H را اجرا خواهد کرد.
- **روال مدیریت صفحه‌کلید BIOS:** روالی که پردازنده برای مدیریت وقفه‌ی 09H فراخوانی می‌کند، مربوط به توابع BIOS است. این روال با توجه به اینکه چه کلیدی فشار داده شده است، کد مربوطه را بدست می‌آورد و معتبر بودن آن را بررسی می‌کند.

---

۱ stack
۲ ports
۳ interrupt handler
۴ software interrupts
۵ subroutine
۶ interrupt vector table
۷ hardare interrupt
۸ disable
۹ Non-Maskable Interrupt
۱۰ system interaction

- **بافر صفحه‌کلید**: پس از آنکه روال مدیریت صفحه‌کلید، کد کلید فشار داده شده را معتبر تشخیص داد، آنرا در یک بافر ۱۶ بایتی در RAM ذخیره می‌کند. این بافر مثل یک صف است و کلیدها به ترتیب فشار داده شدن در آن قرار می‌گیرند و چون ۱۶ بایتی است، در صورتی که بیش از ۱۶ کلید فشار داده شود، پر خواهد شد و در این صورت است که صدای بوق شنیده می‌شود.
- **وقفه‌ی صفحه‌کلید BIOS**: مرحله‌ی بعد، خواندن کاراکتر از بافر فوق و آماده‌سازی آن برای برنامه است. این کار بوسیله‌ی وقفه‌ی شماره‌ی 16H قابل انجام است. پس با فراخوانی این وقفه‌ی نرم‌افزاری، کاراکتر موجود در بافر و وضعیت خوانده می‌شوند. برای این منظور می‌توان از دستور INT اسمبلی استفاده کرد.
- **در سطح DOS:** همچنین در DOS نیز توابعی برای کار کردن با صفحه‌کلید وجود دارد که جزو توابع 21H هستند. این توابع، کارهای بیشتری را در مقایسه به تابع BIOS 16H انجام می‌دهند. برای نمونه کلید زده شده را بر روی صفحه نمایش، نشان می‌دهند.

فصل ۲: برنامه‌سازی سیستم در عمل
(System Programming in Practice)

۲-۱ مقدمه

در این بخش درباره‌ی برنامه‌سازی سیستم با استفاده از زبان اسمبلی، زبان پاسکال (کامپایلرهای توربو یا بورلند پاسکال) و زبان C (با کامپایلرهای میکروسافت و بورلند)، مطالبی ارائه می‌شود. گرچه هدف اصلی زبان C، برنامه‌سازی سیستم بوده و از این نظر از زبان پاسکال مناسب‌تر است، ولی زبان پاسکالی که از طریق کامپایلرهای بورلند در اختیار قرار گرفته، تمامی امکانات برنامه‌سازی سیستم در محیط DOS را در اختیار قرار می‌دهد.

۲-۲ برنامه‌سازی سیستم با زبان اسمبلی

زبان اسمبلی، اصلی‌ترین و بی‌محدودیت‌ترین راه برنامه‌سازی سیستم است. زبان اسمبلی، امکان استفاده از تمامی امکانات سیستم را داده و اجازه‌ی نزدیک‌شدن به سخت‌افزار را تا پائین‌ترین سطح در اختیار قرار می‌دهد. برنامه‌نویسی سطح‌پایین[۱] و با زبان سطح‌پایین اسمبلی، گرچه برای برنامه‌های کاربردی مشکل‌آفرین است، اما برای برنامه‌سازی سیستم مزیت عمده‌ای محسوب می‌شود.

فراخوانی وقفه‌ها در زبان اسمبلی: دستورالعمل فراخوانی وقفه‌ها در زبان اسمبلی 80x86، دستورالعمل **int** است، که یک عملوند دارد و آن شماره‌ی وقفه است. برای مثال دستورالعمل زیر، باعث فراخوانی روال مدیریت وقفه‌ی[۲] شماره 00H که وقفه‌ی "تقسیم بر صفر[۳]" است، می‌شود:

```
int 00H    ; call INT 00H: Division by 0
```

00H، شماره وقفه در جدول بردار وقفه‌ها است و با اجرای دستورالعمل int 00H، نشانی روال مدیریت وقفه‌ی 00H از جدول بردار وقفه‌ها برداشته شده و روال مربوطه که در حافظه‌ی ROM (برای وقفه‌های ROM-BIOS) یا RAM (برای وقفه‌های DOS یا وقفه‌های نوشته شده بوسیله‌ی کاربر) قرار دارد فراخوانی شده و اجرا می‌شود.

```
int  interrupt_no
```

پردازنده‌های 80x86 توانایی مدیریت ۲۵۶ وقفه را دارند. پس interrupt_no مقداری بین 00H و FFH را می‌تواند داشته باشد. دستورالعمل int پس از فراخوانی، مقدار ثبات پرچمها را در پشته push می‌کند و سپس مقادیر پرچمهای IF و TF را 0 می‌کند. مقدار 0 برای IF باعث می‌شود که از وقوع سایر وقفه‌ها جلوگیری شود، مگر آنکه وقفه NMI باشد. همچنین مقدار 1 برای TF باعث می‌شود که پردازنده به حالت تک‌مرحله‌ای[۴] برود که، حالت اشکالزدایی[۵] است و صفر کردن آن این حالت را غیر فعال می‌کند. دلیل صفر کردن IF، ناتوان‌سازی[۶] وقفه‌های دیگر است، چون روالهای وقفه نمی‌توانند به صورت متداخل اجرا شوند. سپس یک far call به روال مدیریت وقفه، که نشانی آن در **جدول بردار وقفه‌ها**[۷] و متناظر با شماره‌ی وجود دارد، انجام می‌شود. برای معرفی یک روال مدیریت وقفه به سیستم یا تغییر روال مدیریت یک وقفه‌ی موجود باید نشانی روال جدید در این جدول قرار داده شود. چون روالها far هستند، پس نشانی قطعه و مبدأ، که ۴ بایت است در این جدول قرار گیرد. به عبارت دقیق‌تر، نشانی مبدأ روال باید در نشانی 4*interrupt_no و نشانی قطعه‌ی آن در نشانی 4*int_no+2 حافظه ذخیره شود. در صورت تغییر روال وقفه‌های حساس، روال قبلی باید در روال جدید فراخوانی شود.

روالهای مدیریت وقفه: یک روال مدیریت وقفه، از هر جهت شبیه یک روال معمولی (البته far proc) در زبان اسمبلی است. با این تفاوت که برای برگشت از این گونه روالها باید از دستورالعمل **iret** (interrupt return) به جای ret استفاده شود. iret کار اضافه‌ای که نسبت به ret انجام می‌دهد، pop کردن خودکار مقدار ثبات پرچمها از روی پشته است.

سایر دستورات مربوط به وقفه‌ها: دو دستور دیگر نیز در ارتباط با وقفه‌ها وجود دارد، که یکی **cli** است و مقدار IF=0 و دیگری **sti** است که IF=1 می‌کند و به ترتیب باعث ناتوان‌سازی و تواناسازی وقفه‌ها می‌شوند.

---

۱ low-level
۲ interrupt handler or Interrupt Service Routine (ISR)
۳ dicision by zero
٤ single-step mode
٥ debugging mode
٦ disable
۷ interrupt vector table

دستورات دسترسی به درگاه‌ها: درگاه‌ها، واسطی بین پردازنده و سایر اجزاء سخت‌افزار سیستم هستند. هر درگاه مشابه یک دریچه‌ی ۸ بیتی ورودی یا خروجی است که به بخش خاصی از یک جزء سخت‌افزاری متصل است و با مقادیر بین 00H تا FFFFH قابل نشانی‌دهی است. درگاه‌ها در حقیقت، ثباتهایی در اجزاء سخت‌افزاری هستند که برای ارتباط با آن، مقداری در آن نوشته یا از آن خوانده می‌شود. برای مثال برای چاپ یک کاراکتر بوسیله‌ی چاپگر، باید کد اَسکی آن کاراکتر در درگاه موازی[۱] که چاپگر به آن متصل است، نوشته شود.

در پردازنده‌های 80x86 از دو دستورالعمل **in** و **out** می‌توان برای دسترسی به درگاه‌ها در زبان اسمبلی استفاده کرد:

Instuction	Destination Operand	Source Operand	Instuction	Destination Operand	Source Operand
in	al	port_number	**out**	port_number	al
in	ax	port_number	**out**	port_number	ax
in	al	dx	**out**	dx	ax
in	ax	dx	**out**	dx	al

این دستورالعملها خیلی شبیه به دستورالعمل mov هستند و می‌توانند یک byte یا یک word را منتقل کنند. هر دو دارای دو شکل هستند، که در آن مقصد یا شماره‌ی درگاه می‌تواند به‌صورت بلافصل (عدد) یا ثبات DX باشد که محتوی آن شماره‌ی درگاه است. برای مثال دستورالعمل (۱) معادل دستورالعملهای (۲) است:

```
(1)     in   ax, 07ch

(2)     mov  dx, 07ch
        in   ax, dx
```

اما برای شماره درگاههای بزرگتر از FFH، استفاده از شکل اول غیرمجاز است. مثلاً دستورالعمل زیر درست نیست:

```
out  04a5h, al      ; illegal port address for this format
```

و باید مقدار 04a5h در ثبات DX قرار داده شود:

```
mov  dx, 04a5h
out  dx, al
```

۲-۳ برنامه‌سازی سیستم با زبان پاسکال

فراخوانی وقفه‌ها از توربو پاسکال: توربو پاسکال دو روال Intr و MsDos را فراهم نموده است که در یونیت DOS تعریف شده‌اند. همچنین در این یونیت، انواع و ثابت‌های مورد نیاز برای این منظور معرفی شده‌اند.

```
Intr ( InterruptNumber: Byte, Regs: Registers);
```

پارامتر InterruptNumber، شماره‌ی وقفه‌ی موردنظر را مشخص می‌کند. از آنجایی که این پارامتر می‌تواند مقداری بین 0 تا 255 را قبول کند، بنابر این همه‌ی وقفه‌ها، و از جمله وقفه‌های سخت‌افزاری را می‌توان فراخوانی نمود. روال MsDos شکل خاصی از روال Intr است:

```
MsDos (Regs: Registers);
```

این روال شماره‌ی وقفه را به‌عنوان پارامتر اول نیاز ندارد. این روال امکان دستیابی به وقفه‌ی 21H را که شامل توابع واسط برنامه‌ی کاربردی DOS[۲] است را فراهم می‌کند. این API شامل حدود ۲۰۰ تابع است که بوسیله‌ی فرمانها و برنامه‌های DOS فراهم می‌شوند.

نوشتن روال مدیریت وقفه‌ی جدید: در برنامه‌ی زیر، نمونه‌ای از یک روال مدیریت وقفه ارائه شده است. روالTimerHandler ، وقفه‌ی 1CH را که به "Clock Tick" معروف است و در هر ثانیه 18.2 بار یک بار فعال می‌شود، عوض نموده است. در این روال یک شمارشگر سراسری یک واحد اضافه شده و سپس نمایش داده می‌شود. در برنامه‌ی اصلی، ابتدا نشانی روال قبلی مدیریت وقفه‌ی 1CH با فراخوانی GetIntVec در متغیر Int1CSave، ذخیره شده است و سپس روال جدید برای وقفه‌ی 1CH با فراخوانی SetIntVec، نصب شده است. در حالیکه برنامه‌ی اصلی، منتظر زده شدن یک کلید است، روال وقفه‌ی 1CH، شمارشگر را افزایش داده و نمایش می‌دهد. پس از زدن کلید، روال قبلی 1CH برگردانده می‌شود.

```
uses Dos, Crt;
var
  Int1CSave : Pointer;
  Count : Integer;  { Global counter }
  Ch : Char;
procedure TimerHandler; interrupt ;
begin
```

---

۱ parallel port
۲ DOS Application Program Interface (API)

```
  { New Timer ISR }
  Inc(Count);
  GotoXY(10, 10);
  WriteLn(Count);
end;

begin  { main program }
  Count := 0;
  GetIntVec($1C, Int1CSave);
  SetIntVec($1C, Addr(TimerHandler));
  writeln('Press any key  to exit');
  Ch := ReadKey;
  SetIntVec($01C, Int1CSave);
end.
```

دستیابی به ثباتهای پردازنده: هر دو تابع فوق به متغیری از نوع Registers نیاز دارند که در یونیت DOS تعریف شده‌اند. این نوع متغیرها مقادر ثباتهای پردازنده را قبل از فراخوانی وقفه می‌پذیرند و به وقفه ارسال می‌کنند. پس از برگشت از توابع Intr یا MsDos، این متغیرها شامل مقادیر ثباتهای پردازنده در پایان فراخوانی وقفه‌ها هستند.

برای ساده‌سازی ارجاع به ثباتها، نوع Registers به‌صورت یک رکورد متغیر[۱] تعریف شده است:

```
type Registers = record
                   case  integer  of
                     0 :  (AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags : word);
                     1 :  (AL, AH, BL, BH, CL, CH, DL, DH : byte);
                   end;
```

ثباتهای ۱۶ بیتی پردازنده، یعنی AX تا BP بوسیله‌ی متغیرهایی از نوع word و با نام مشابه نمایش داده شده‌اند. ثباتهای ۸ بیتی AL تا DH نیز با متغیرهای از نوع byte ارائه شده‌اند. در ذخیره‌سازی مقادیر ثباتها در حافظه، ثباتهای ۸ بیتی و ۱۶ بیتی از فضای حافظه‌ی یکسانی استفاده می‌کنند. یعنی AL و AH دو بایت از یک کلمه‌ی ۱۶ بیتی AX را تشکیل می‌دهند و دارای حافظه‌ی مستقلی نیستند.

اگر Regs یک متغیر از نوع Registers باشد، برای دسترسی به ثباتهای پردازنده به‌سهولت می‌توان از فیلدهای این متغیر از نوع رکورد استفاده نمود:

- Regs.AX,
- Regs.BX,
- Regs.AH,
- Regs.DL, etc.

برای ارسال مقدار D3H به ثبات DL برای فراخوانی یک وقفه، به‌صورت زیر عمل کنید:

```
Regs.DL := $D3;
```

قبل از فراخوانی یک وقفه با Intr یا MsDos، مقادیر موردنظر را در ثباتهایی که بوسیله‌ی تابع، مورد استفاده قرار خواهند گرفت، قرار دهید. وقفه فقط از اطلاعات قرار داده شده در ثباتهای موردنظرش استفاده می‌کند و از مابقی صرف‌نظر می‌کند.

خواندن پرچم‌های ثبات پرچم: در بسیاری از حالت‌ها، ثبات پرچم می‌تواند اطلاعاتی را به برنامه برگرداند. توابع DOS به‌طور گسترده از Carry Flag استفاده می‌کنند، که پس از پایان فراخوانی تابع یا اجرای ناموفق آن، مقداردهی می‌شود. برای سهولت بررسی پرچم‌ها، در یونیت DOS ثابت‌های مختلفی تعریف شده‌است و متناظر با مقادیر بیتهای پرچمهای پردازنده است. برای تعیین اینکه کدامیک از بیت‌ها مقدار گرفته است، می‌توان از عملگر AND استفاده نمود. عبارت زیر در صورتی‌که مقدار بیت Carry Flag یک شده باشد، به متغیر بولیError مقدار True می‌دهد:

```
Error := ((Regs.Flags and FCarry) <> 0);
```

Constant	Bit Position	Bit Value
FCarry	0	1
FParity	2	4
FAuxiliary	4	16
FZero	6	64

[۱] variant record

FSign	7	128
FOverflow	11	2048

بافرها و توربو پاسکال: بسیاری از توابع وقفه به پارامترهایی از نوع اشاره‌گر به بافر برای فراخوانی نیاز دارند. این توابع یا اطلاعات را از بافر برمی‌دارند یا در آن قرار می‌دهند. این اشاره‌گرها همیشه FAR بوده و شامل نشانی قطعه و نشانی مبدأ هستند. اشاره‌گرهای FAR برخلاف اشاره‌گرهای NEAR، به داده‌های موجود در هر جای حافظه و خارج از قطعه‌ی جاری برنامه می‌توانند اشاره کنند.

ارسال اشاره‌گرها به توابع وقفه: تابع DOS 09H، مثالی از یک تابع است که پارامتری از نوع اشاره‌گر دارد. این تابع یک رشته را بر روی صفحه‌ی نمایش و از نقطه‌ی شروع محل فعلی مکان‌نما، نشان می‌دهد. همانند همه‌ی توابع DOS، این تابع انتظار دارد که شماره‌ی تابع در ثبات AH باشد و نشانی شروع بافر حاوی رشته‌ای که قرار است نمایش داده شود در زوج ثبات DS:DX قرار داشته باشد. DS مقدار نشانی قطعه و DX نشانی مبدأ بافر است.

با وجودیکه ایجاد رشته در پاسکال آسان است، ولی باید بدانید که چگونه نشانی بافر رشته را بدست آورده و در ثباتهای DS و DX قرار دهید. برای این منظور در توربو پاسکال دو تابع Seg() و Ofs() وجود دارد، که به‌ترتیب نشانی‌های قطعه و مبدأ یک شئ را در حافظه بدست می‌دهند.

مثال زیر نشان می‌دهد که چگونه می توانید این توابع را با تابع DOS 09H مورد استفاده قرار دهید. این برنامه از تابع DOS 09H برای نمایش رشته‌ی موجود در متغیر پیغام بر روی صفحه‌ی نمایش استفاده می‌کند. برخلاف سایر توابع DOS، تابع 09H نیاز به مشخص شدن پایان رشته با کاراکتر '$' دارد، نه بایت null .

```
program 9HDemoP;
uses    DOS;
var     Regs    : Registers;
        Message: string[20];
begin
  Message := 'DOSPrint$';
  Regs.AH := $09;
  Regs.DS := Seg( Message[1] );
  Regs.DX := Ofs( Message[1] );
  MsDos (Regs );
end.
```

دریافت اشاره‌گرها از توابع وقفه: برنامه‌ی زیر تابع DOS 1BH را فراخوانی می‌کند که یک اشاره‌گر را در زوج ثبات DS:BX برمی‌گرداند. این اشاره‌گر به یک بایت که شامل کد رسانه‌ی[1] گرداننده‌ی جاری است، اشاره می‌کند. DOS از کدهای رسانه F0H تا FFH برای توصیف انواع مختلف گردانندهها استفاده می‌کند. مقدار F8H همه‌ی انواع دیسک سخت را مشخص می‌کند.

برای تعیین شناسه‌ی رسانه از اشاره‌گر برگشتی، نوع MediaPtr به عنوان اشاره‌گری به یک بایت، در ابتدای برنامه تعریف شده است. از آنجایی که اشاره‌گرها در توربو پاسکال همیشه FAR هستند، مطمئن خواهید بود که یک اشاره‌گر FAR ایجاد نموده‌اید. در برنامه MP به‌عنوان متغیری از این نوع تعریف شده است. پس از فراخوانی تابع DOS 1BH، برنامه اشاره‌گر برگشتی را پس از بدست‌آوردن از زوج ثبات DS:BX در MP قرار می‌دهد. برنامه برای این منظور از تابع Ptr توربو پاسکال استفاده می‌کند. این تابع نشانی‌های قطعه و مبدأ را دریافت نموده و یک اشاره‌گر از آنها تشکیل می‌دهد. این اشاره‌گر برای دسترسی به اطلاعات مورد اشاره، قابل استفاده است. دستور WriteLn در پایان برنامه این مسأله را نشان می‌دهد.

```
program MediaIdP;
uses    Dos;
type    MediaPtr = ^byte;
var     Regs    :   Registers;
        MP      :    MediaPtr;
begin
   Regs.AH := $1B;
   MsDos ( Regs );
   MP := Ptr (Regs.DS, Regs.BX );
   WriteLn ( 'Media ID = ', MP^ );
end.
```

دسترسی به حافظه با Mem، MemW و MemL : در توربو پاسکال از سه تابع ازقبل‌تعریف شده‌ی Mem، MemW و MemL برای دسترسی به حافظه به‌ترتیب با انواع byte، word و longint(dwords) می‌توان استفاده کرد. از یک نحو خاص برای کار با این آرایه‌ها استفاده می‌شود که در داخل کروشه‌ها نشانی قطعه از نشانی مبدأ با ':' جدا می‌شود. برای مثال شناسه‌ی رسانه را می‌توان به صورت زیر هم در برنامه‌ی فوق بدست آورد:

[1] media code

```
mem[ Regs.DS :Regs.BX ];
```

دسترسی به درگاه‌ها در تورپو پاسکال: درگاههای PC با استفاده از یک آرایه‌ی ازقبل‌تعریف شده در تورپو پاسکال شناسایی می‌شوند. اما، تورپو پاسکال دو آرایه را برای دسترسی به درگاه‌ها پشتیبانی می‌کند: Port (برای درگاه‌های ۸ بیتی) و PortW (برای درگاه‌های ۱۶ بیتی). PortW اجازه‌ی ارسال مقادیر ۱۶ بیتی را به درگاه‌ها اجازه می‌دهد، در حالیکه Port فقط مقادیر ۸ بیتی را می‌پذیرد. انتخاب هرکدام از این آرایه‌ها به نوع برد متصل یا تراشه مورد دستیابی بستگی دارد. اگر برد یا تراشه ۱۶ بیتی است می‌توانید از PortW استفاده کنید، وگرنه باید از Port برای دسترسی به آن استفاده کنید.

می‌توانید برای خواندن اطلاعات از درگاه‌ها یا نوشتن اطلاعات در آنها از نحو معمولی آرایه استفاده کنید. برای مثال هر دو جمله‌ی زیر محتوی درگاه 3C4H را می‌خوانند، که بخشی از کنترلر گرافیکی در یک کارت EGA/VGA است:

```
XByte := Port [ $3C4 ];
XWord := PortW [ $3C4 ];
```

جمله‌های زیر اجازه‌می‌دهند که یک بایت یا کلمه به درگاه ارسال شود:

```
Port [ $3C4 ] := XByte;
PortW [ $3C4 ] := XWord;
```

۴-۲ برنامه‌سازی سیستم با زبان C

برخلاف پاسکال، بازار کامپایلرهای C بین میکروسافت و بورلند تقسیم شده است. هر دو شرکت دارای چند محصول در این زمینه هستند: Microsoft QuickC و Microsoft C 6.0 و Turbo C++ و Borland C++. برنامه‌های C ارائه شده در این کتاب با همه‌ی کامپایلرهای فوق کامپایل می‌شوند و فقط تعدای پیغام هشدار ممکن است بر روی صفحه ظاهر شود. به دلیل تفاوت در کتابخانه‌های کامپایلرهای فوق، در برخی اوقات در برنامه‌ها از Directiveهای #ifdef برای تعریف ماکروهایی استفاده شده است. چون هدف برنامه‌سازی سیستم است، برنامه‌ها از قابلیتهای شئ‌گرایی زبان C++ استفاده نمی‌کنند و برنامه‌ها، در حقیقت C هستند.

فراخوانی وقفه‌ها از C: هم کامپایلرهای میکروسافت و هم بورلند، توابع int86()، int86x()، intdos() و intdosx() را برای فراخوانی وقفه‌های نرم‌افزاری فراهم می‌کنند. درحالیکه توابع int86() و int86x() می‌توانند همه‌ی ۲۵۶ وقفه‌ی پردازنده‌ی Intel را فراخوانی کنند، توابع intdos() و intdosx() مخصوص وقفه‌ی 21H (0x21) هستند، که شامل توابع واسط برنامه‌ی کاربردی DOS است و بیش از ۲۰۰ تابع را می‌پوشاند.

اعلان این توابع در هر دو کامپایلر در فایل DOS.H قرار دارد و به صورت زیر است:

```
int  intdos( union REGS *inregs, union REGS *outregs);
int  intdosx( union REGS *inregs, union REGS *outregs, struct SREGS *segregs);
int int86( int intno, union REGS *inregs, union REGS *outregs);
int  int86x( int intno, union REGS *inregs, union REGS *outregs, struct SREGS *segregs);
```

دسترسی به ثباتهای پردازنده: هر چهار تابع فوق به اشاره‌گرهایی از نوع REGS به عنوان پارامتر نیاز دارند. توابع intdosx() و int86x()، به متغیری از نوع SREGS هم نیاز دارند. این ساختارها برای نمایش ثباتهای پردازنده مورد استفاده قرار می‌گیرند. توابع فوق قبل از فراخوانی وقفه، مقادیر ثباتهای پردازنده را از inregs بارمی‌کند و پس از خاتمه‌ی فراخوانی وقفه، مقادیر ثباتهای پردازنده را در outregs قرار می‌دهند.

برای سهولت نشانی‌دهی به ثباتهای ۸ بیتی و ۱۶ بیتی، REGS یک union را که شامل دو ساختار WORDREGS و BYTEREGS است را نشان می‌دهد:

```
union   REGS   {
                struct  WORDREGS x;
                struct  BYTEREGS h;
                };
struct WORDREGS  {
                    unsigned int ax, bx, cx, dx, si, di, cflag;
                };
struct BYTEREGS   {
                    unsigned char   al, ah, bl, bh, cl, ch, dl, dh;
                 };
```

ثباتهای ۱۶ بیتی پردازنده(AX تا ES) با متغیرهای unsigned int با نامی مشابه در ساختار WORDREGS نشان داده شده‌اند. ثباتهای ۸ بیتی پردازنده (AL تا DH) با متغیرهای unsigned char در ساختار BYTEREGS ارائه شده‌اند. ساختار REGS یک رکورد متغیر است که حافظه‌های ثباتهای ۱۶ و ۸ بیتی را به‌طور اشتراکی نشان می‌دهد.

اگر pregs متغیری از نوع REGS باشد، برای دسترسی به ثباتهای پردازنده می‌توان به شکل زیر عمل کرد:

- pregs.x.ax,
- pregs.x.bx,
- pregs.h.ah,
- pregs.h.dl, etc.

اگر بخواهید مقدار D3H (0xD3) را قبل از فراخوانی وقفه در ثبات DL قرار دهید، به‌صورت زیر عمل کنید:

```
pregs.h.dl = 0xD3;
```

قبل از فراخوانی یک وقفه با توابع فوق، مقادیر موردنظر را در ثباتهایی که بوسیله‌ی تابع مورد استفاده قرار خواهند گرفت، قرار دهید. وقفه فقط از اطلاعات قرار داده شده در ثباتهای موردنظرش استفاده می‌کند و از مابقی صرف‌نظر می‌کند.

در تعریف WORDREGS دیده می‌شود که ثباتهای قطعه در اعلان ساختار وجود ندارند. این امر به آن دلیل است که اغلب توابع به این ثباتها کاری ندارند. اگر تابعی به این ثباتها نیاز داشته باشد باید از توابع intdosx() و int86x() استفاده کنیم که یکی از پارامترهای آن از نوع SREGS است و در آن مقادیر ثباتهای قطعه بارگذاری می‌شوند. تعریف SREGS به‌شکل زیر است:

```
struct  SREGS   {
                unsigned int    es;
                unsigned int    cs;
                unsigned int    ss;
                unsigned int    ds;
};
```

همچنین در کامپایلرهای بورلند امکان دسترسی به ثباتها با استفاده از شبه‌متغیرهای[1] ثباتها وجود دارد. شبه‌متغیرهایی با نامهای AX_، BX_ و ... و AH_، BH_ و ... و ES_ و ... و نیز FLAGS_ و متناظر با ثباتهای پردازنده وجود دارد که امکان دسترسی مستقیم خواندنی و نوشتنی به ثباتها را می‌دهند.

نوشتن روال مدیریت وقفه‌ی جدید: برنامه‌ی زیر، نمونه‌ای از یک روال مدیریت وقفه 1CH به زبان C و با کامپایلر بورلند است:

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>

#define INTR 0X1C    /* The clock tick interrupt */

#ifdef __cplusplus
   #define __CPPARGS ...
#else
   #define __CPPARGS
#endif

void interrupt ( *oldhandler)(__CPPARGS);
int count=0; /* Global counter */
void interrupt handler(__CPPARGS)
{
  /* increase the global counter */
  count++;
  gotoxy(10, 10);
   printf("%d", count);
   /* call the old routine */
  oldhandler();
}

int main(void)
```

[1] pseudovariables

```
{
  /* save the old interrupt vector */
  oldhandler = getvect(INTR);
   /* install the new interrupt handler */
  setvect(INTR, handler);
  puts("Press any key  to exit");
  getch();
  /* reset the old interrupt handler */
  setvect(INTR, oldhandler);
  return 0;
}
```

خواندن پرچمها در ثبات پرچم: در بسیاری از حالت‌ها، ثبات پرچم می‌تواند اطلاعاتی را به برنامه برگرداند. توابع DOS به‌طور گسترده از Carry Flag استفاده می‌کنند، که پس از پایان فراخوانی تابع یا اجرای ناموفق آن، مقداردهی می‌شود. برای سهولت بررسی پرچمها، ساختار WORDREGS دارای یک فیلد به‌نام cflag است که مقدار Carry Flag پس از فراخوانی وقفه در آن بار می‌شود و یکی از مقادیر 0 یا 1 را خواهد داشت. در برنامه‌ی زیر این پرچم پس از فراخوانی تابع 13H وقفه‌ی 21H بررسی می‌شود:

```
#include <dos.h>
void test ( void )
{
    union REGS pregs;
    pregs.h.ah = 0x13;
    pregs.h.dl = 0;
    intdos( &pregs, &pregs );
    if  ( &pregs.x.cflag )
       ;  /* Carry flag set */
   else
       ;  /* Carry flag unset */
}
```

اگر نیاز به خواندن همه‌ی پرچمها باشد، در کامپایلرهای میکروسافت راه‌حلی برای آن وجود ندارد. ولی در کامپایلرهای بورلند فیلد دیگری به‌نام flags در ساختار WORDREGS وجود دارد که همه‌ی پرچمهای پردازنده را برمی‌گرداند. بنابر این در کامپایلرهای بورلند داریم:

```
struct WORDREGS  {    /* Borland only! */
                        unsigned int ax, bx, cx, dx, si, di, cflag, flags;
                    };
```

با توجه به جدول زیر می‌توانیم تعیین کنیم که کدام پرچم مقدار گرفته است:

Constant	Bit Position	Bit Value
Carry	0	1
Parity	2	4
Auxiliary	4	16
Zero	6	64
Sign	7	128
Overflow	11	2048

بافرها و زبان C: بسیاری از توابع وقفه به پارامترهایی از نوع اشاره‌گر به بافر برای فراخوانی نیاز دارند. این توابع یا اطلاعات را از بافر برمی‌دارند یا در آن قرار می‌دهند. این اشاره‌گرها همیشه FAR بوده و شامل نشانی قطعه و نشانی مبدأ هستند. اشاره‌گرهای FAR برخلاف اشاره‌گرهای NEAR، به داده‌های موجود در هر جای حافظه و خارج از قطعه‌ی جاری برنامه می‌توانند اشاره کنند.

ارسال اشاره‌گرها به توابع وقفه: تابع DOS 09H، مثالی از یک تابع است که پارامتری از نوع اشاره‌گر دارد. این تابع یک رشته را بر روی صفحه‌ی نمایش و از نقطه‌ی شروع محل فعلی مکان‌نما، نشان می‌دهد. همانند همه‌ی توابع DOS، این تابع انتظار دارد که شماره‌ی تابع در ثبات AH باشد و نشانی شروع بافر حاوی رشته‌ای که قرار است نمایش داده شود در زوج ثبات DS:DX قرار داشته باشد. DS مقدار نشانی قطعه و DX نشانی مبدأ بافر است.

با وجودیکه ایجاد رشته در زبان C آسان است، ولی باید بدانید که چگونه نشانی بافر رشته را بدست آورده و در ثباتهای DS و DX قرار دهید. برای این منظور در کامپایلرهای بورلند و میکروسافت دو ماکرو با نامهای FP_SEG() و FP_OFF() وجود دارد، که به‌ترتیب نشانی‌های قطعه و مبدأ یک شئ را در حافظه بدست می‌دهند. اما روش تعریف آنها در این دو نوع کامپایلر با هم متفاوت است:

Borland:

```
#define FP_SEG( fp ) ( (unsigned )( void _seg * )( void far * ) ( fp))
#define FP_OFF( fp ) ( (unsigned )( fp))
```

Microsoft:

```
#define FP_SEG( fp ) (*((unsigned  _far *) & (fp)+1))
#define FP_OFF( fp ) (*((unsigned  _far *) & (fp)))
```

درحالیکه ماکروی تعریف شده بوسیله‌ی بورلند یک متغیر را می‌پذیرد، ماکروی میکروسافت، اشاره‌گری به یک متغیر و از نوع FAR را قبول می‌کند. در مثال زیر مشاهده می‌کنید که چگونه می‌توانید این ماکروها را در دو کامپایلر با تابع DOS 09H مورد استفاده قرار دهید. این برنامه از تابع DOS 09H برای نمایش رشته‌ی موجود در متغیر پیغام بر روی صفحه‌ی نمایش استفاده می‌کند. برخلاف سایر توابع DOS، تابع 09H نیاز به مشخص شدن پایان رشته با کاراکتر '$' دارد، نه بایت null.

```
/* Borland Version */
#include <dos.h>
void main ( void )
{
   union REGS pregs;
   struct SREGS sregs;
   char Message[20] = "PC Interns$";
   pregs.h.ah = 0x09;
   pregs.h.ds = FP_SEG ( Message );
   pregs.x.dx = FP_OFF ( Message );
   intdosx ( &pregs, &pregs, &sregs );
}

/* Microsoft Version */
#include <dos.h>
void main ( void )
{
   union REGS pregs;
   struct SREGS sregs;
   char Message[20] = "PC Interns$";
   void far *mesptr = Message; /* FAR pointer to string */
   pregs.h.ah = 0x09;
   pregs.h.ds = FP_SEG ( mesptr );
   pregs.x.dx = FP_OFF ( mesptr );
   intdosx ( &pregs, &pregs, &sregs );
}
```

دریافت اشاره‌گرها از توابع وقفه: برنامه‌ی زیر تابع DOS 1BH را فراخوانی می‌کند که یک اشاره‌گر را در زوج ثبات DS:BX برمی‌گرداند. این اشاره‌گر به یک بایت که شامل کد رسانه‌ی گرداننده‌ی جاری است، اشاره می‌کند. DOS از کدهای رسانه F0H تا FFH برای توصیف انواع مختلف گردانندهها استفاده می‌کند. مقدار F8H همه‌ی انواع دیسک سخت را مشخص می‌کند.

برای تعیین شناسه‌ی رسانه از اشاره‌گر برگشتی، نیاز به یک اشاره‌گر FAR است. برای این منظور در DOS.H یک ماکرو به نام MK_FP تعریف شده است و با استفاده از نشانی‌های قطعه و مبدأ، یک اشاره‌گر FAR ایجاد می‌کند. تعریف این ماکرو در برنامه‌ی زیر آمده است. در این برنامه mp به‌عنوان متغیری far و از نوع unsigned char تعریف شده است. پس از فراخوانی تابع DOS 1BH، برنامه اشاره‌گر برگشتی را پس از بدست‌آوردن از زوج ثبات DS:BX در mp قرار می‌دهد. برنامه برای این منظور از ماکروی MK_FP استفاده می‌کند. این اشاره‌گر برای دسترسی به اطلاعات مورد اشاره، قابل استفاده است. دستور printf در پایان برنامه این مسأله را نشان می‌دهد.

```
#include <dos.h>
#include <stdio.h>
#ifndef  MK_FP
   #define MK_FP ( seg, ofs)  ( (void far * ) ((unsigned long) (seg) << 16)(ofs)))
#endif
```

```
void main (void)
{   union REGS pregs;
    struct SREGS sregs;
    unsigned char far *mp;
    pregs.h.ah = 0x1B;
    intdosx( &pregs, &pregs, &sregs );
    mp = MK_FP ( sregs.ds, pregs.x.bx );
     printf (“Media ID = %d\n”, *mp);
}
```

دسترسی به حافظه با poke، pokeb، peek و peekb: در کامپایلرهای بورلند C، از توابع poke و pokeb می‌توان برای نوشتن integer یا byte در نشانی segment:offset حافظه و از توابع peek و peekb برای خواندن یک integer یا byte از نشانی segment:offset حافظه استفاده نمود:

```
void poke(unsigned segment, unsigned offset, int value);
void pokeb(unsigned segment, unsigned offset, char value);
int peek(unsigned segment, unsigned offset);
char peekb(unsigned segment, unsigned offset);
```

دسترسی به درگاه‌ها در C: هم در کامپایلرهای بورلند و هم میکروسافت، توابع متعددی برای کار با درگاه‌ها معرفی شده‌اند که در بورلند در DOS.H و در میکروسافت در CONIO.H تعریف شده‌اند. تعریف این توابع در دو نوع کامپایلر با هم متفاوت است:

Microsoft: Include File <CONIO.H>

```
int             inp ( unsigned port );
unsigned        inpw ( unsigned port );
int             outp ( unsigned port, int databyte );
unsigned        outpw ( unsigned port, int dataword );
```

Borland: Include File <DOS.H>

```
int             inport ( int __portid );
unsigned  char  inportb ( int __portid );
void            outport ( int __portid, int __value );
void            outportb ( int __portid, unsigned char __value );

#define  inp ( portid )       inportb ( portid )
#define  outp ( portid, v )   outportb ( portid, v )
```

هر دو کامپایلر توابعی برای دسترسی به درگاه‌های ۸ و ۱۶ بیتی دارند. در کامپایلر بورلند برای سازگاری با کامپایلر میکروسافت دو ماکروی inp() و outp() برای درگاه‌های ۸ بیتی تعریف شده است.

در مثال زیر محتوی درگاه 3C4H، که بخشی از کنترلر گرافیکی در یک کارت EGA/VGA است، خوانده می‌شود:

```
XByte = inp (0x3C4 );
XWord = inpw ( 0x3C4 );
```

جمله‌های زیر اجازه‌می‌دهند که یک بایت یا کلمه به درگاه ارسال شود:

```
outp ( 0x3C4, XByte);
outpw ( 0x3C4, XWord);
```

فصل ۳: سیستم مبنایی ورودی/خروجی
(The BIOS)

۳-۱ مقدمه

بسیاری از کاربران اصطلاح *سیستم عامل* در PC را با DOS برابر می‌دانند. اما، DOS تنها سیستم عامل در PC نیست. قبل از آنکه گرداننده دیسک سخت در سیستم شناخته شده باشد، PC برای BIOS جستجو می‌کند، تا روالهای مبنایی ورودی و خروجی مورد نیاز برای ارتباط سخت‌افزار و نرم‌افزار در اختیار بگیرد. BIOS بر روی یک ROM در حافظه‌ی در برد اصلی PC وجود دارد و بلافاصله پس از روشن شدن PC، در دسترس است. BIOS شامل همه‌ی روالهای اساسی موردنیاز PC برای ارتباط با سخت‌افزار و لوازم جانبی است. این روالها شامل دستورات مدیریت خروجی صفحه‌نمایش، خروجی چاپی، تاریخ، زمان و غیره است.

چرا BIOS مهم است؟ از آنجایی که توابع BIOS استاندارد هستند، برنامه‌نویس مجبور به نوشتن برنامه برای یک PC با پیکره‌بندی سخت‌افزاری خاص نیست. به این معنی که برنامه را می‌توان بر روی یک PC نوشت و آنرا بر روی همه‌ی PCهای سازگار[۱] با آن بدون خطا اجرا نمود، حتی اگر سخت‌افزار یا روالهای خاصی از BIOS آنها با هم به‌طور کامل سازگار نباشد.

BIOS بخش مهمی از PC است. مهم نیست که PC ساخت کدام شرکت است یا ظرفیت دیسک سخت آن چند مگابایت است. مهم این است که در هر حال، توابع پشتیبانی دیسک سخت در BIOS، یکسان هستند.

BIOS، وابسته به سخت‌افزار است و عامل اصلی مرسوم شدن PC است. BIOS، سازندگان مختلف PC را قادر می‌سازد که یک PC متفاوت با IBM PC بسازند، اما دارای BIOS یکسانی با آن باشد و بتواند برنامه‌هایی را که بر روی آن اجرا می‌شوند، اجرا کند.

چندین شرکت تراشه‌های BIOS را می‌سازند که می‌توان از AMI، Phoenix، Award و Quadtel نام برد. گرچه این BIOSها دارای تفاوتهایی با هم هستندن، ولی وظایف اساسی آنها یکسان است.

۳-۲ استاندارد BIOS

IBM، انواع مختلف توابع BIOS و پارامترهای مورد نیاز در PC را تعریف نموده است. ۲۵۶ وقفه در BIOS وجود دارد که به توابع تقسیم شده‌اند. این توابع ارتباط با سخت‌افزار را میسر می‌سازند. در جدول زیر این توابع نمایش داده‌شده‌اند. BIOS به برخی از وقفه‌ها به‌عنوان متغیر نگاه می‌کند، نظیر توابع کارت ویدئویی و گرداننده‌ی دیسک سخت. در این باره در ادامه بیشتر توضیح داده خواهد شد.

Number	Meaning
10H	Video card access
11H	Configuration test
12H	RAM test
13H	BIOS disk functions
14H	Serial interface functions
15H	Cassette and extended AT functioms
16H	Keyboard functions
17H	Parallel interface functions
1AH	Date/tme/realtime clock functions

[۱] compatilbe

۳-۳ تعیین نگارش BIOS

نگارش و تاریخ BIOS در نشانی ROM-BIOS F000:FFF0 وجود دارد و قابل دسترسی است. برای این منظور می‌توان با استفاده از برنامه‌ی DEBUG استفاده نمود:

```
C> debug
```

فرمان زیر را وارد کنید:

```
- d  F000:FFF0  L  10
F000:FFF0  CD 19 E0 00 F0 30 33 2F-32 37 2F 39 38 00 FC 5C   .....03/27/98 ..\
```

تاریخ BIOS در سمت راست خط فوق نمایش داده شده‌است.

```
- q
C>
```

تعیین نوع PC: توابع معینی از BIOS برای شناسایی مدل PC به‌کار می‌رود. برنامه‌نویس می‌تواند مدل PC را از بایتی که در آخرین مکان ROM-BIOS در نشانی F000:FFFE قرار دارد، شناسایی کند. این بایت ممکن است شامل یکی از کدهای زیر باشد:

Model identification byte codes	
Code	Meaning
FCH	AT
FEH FBH	XT
FFH	PC

این مقادیر دقیق نیستند و تنها در کامپیوترهای ساخت شرکت IBM به‌طور دقیق پشتیبانی می‌شوند و به دلیل تفاوتهای جزئی در BIOSهای ساخت شرکتهای مختلف این روش برای شناسایی مدل PC، روش کاملی نیست.

۴-۳ متغیرهای BIOS

BIOS در ۲۵۶ بایت حافظه (RAM)، که *محدوده‌ی متغیر BIOS*[۱] یا *قطعه‌ی متغیر BIOS*[۲] نامیده می‌شود، متغیرهای داخلی خود را ذخیره می‌کند. این روش به‌صورت استاندارد در همه‌ی انواع BIOSها وجود دارد، زیرا بوسیله‌ی بسیاری از توابع DOS استفاده می‌شود. در فهرست زیر متغیرهای مختلف، اهداف و نشانی مبدأ آنها مشخص شده است. نشانی کامل از تلفیق نشانی مبدأ با نشانی قطعه‌ی 0040H بدست می‌آید. برای مثال نشانی یک متغیر با مبدأ 10H، 0040H:0010H است.

Offset	Function	Size	Access
00H	Serial interface port addresses	4 words	INT 14H

در هنگام روشن شدن PC، تست خودکار زمان روشن‌شدن یا POST[۳]، پیکره‌بندی سیستم را تعیین می‌کند. یکی از مواردی که جزو پیکره‌بندی سیستم است، نشانی واسطهای سریال یا RS-232 است، که با COM1 تا COM4 مشخص می‌شوند. به تعدادی که واسط سریال نصب شده است، نشانی درگاه آنها تعیین شده و در چهار کلمه پشت‌سر هم که از نشانی مبدأ 00H شروع می‌شوند، قرار می‌دهد.

08H	Parallel interface port addresses	4 words	INT 17H

در مورد واسطهای موازی (LPT1-LPT4) هم نشانی درگاه‌آنها تعیین شده و در چهار کلمه ذخیره می‌شوند.

10H	Configuration	1 word	INT 11H

۱ BIOS variable segment
۲ BIOS variable range
۳ Power On Self Test

سایر اطلاعات مربوط به پیکره‌بندی سیستم، از قبیل تعداد گرداننده‌های دیسک سخت، حالت ویدئویی، تعداد واسطهای سریال و موازی و غیره با این یک کلمه تعیین می‌شوند.

12H	POST status #1	1 byte	POST
13H	RAM size	1 word	INT 12H
15H	POST status #2	1 word	POST
17H	Keyboard status byte	1 byte	INT 16H

وضعیت کلیدهای کنترلی، نظیر Shift، Ctrl، Alt و غیره را تعیین می‌کند.

18H	Extended keyboard status byte	1 byte	INT 16H
19H	ASCII code entry	1 byte	INT 16H
1AH	Next character in keyboard buffer	1 word	INT 16H
1CH	Last character in keyboard buffer	1 word	INT 16H
1EH	Keyboard buffer	16 words	INT 16H

به بافر چرخه‌ای صفحه‌کلید، که حداکثر ۱۶ کاراکتر در هر زمان می‌تواند در آن باشد، از این طریق می‌توان دسترسی پیدا نمود.

3EH	Disk drive recalibration	1 byte	INT 13H
3FH	Disk drive motor status	1 byte	INT 13H
40H	Disk drive motor timer	1 byte	INT 13H
40H	Disk error status	1 byte	INT 13H
42H	Disk controller status	7 byte	INT 13H
49H	Current video mode	1 byte	INT 10H
4AH	Number of screen columns	1 word	INT 10H
4CH	Screen page size	1 word	INT 10H
4EH	Offset address of current screen page	1 word	INT 10H
50H	Cursor position in eight screen pages	8 words	INT 10H
60H	Starting line of screen cursor	1 byte	INT 10H
61H	Ending line of screen cursor	1 byte	INT 10H
62H	Current screen page namber	1 byte	INT 10H
63H	Port address of video controller	1 word	INT 10H
65H	Mode selector register contents	1 byte	INT 10H
66H	Palette register contents	1 byte	INT 10H
67H	Miscellaneous selector register contents	5 bytes	POST
6CH	Timer	1 dword	INT 1AH
70H	24-hours flag	1 byte	INT 1AH
71H	CTRL-Break flag	1 byte	INT 16H

فصل ۴: کارتهای ویدئویی
(Video Cards)

۴-۱ مقدمه

کارتهای ویدئویی مختلفی، از قبیل MDA، CGA، EGA، VGA، و Hercules تاکنون معرفی شده‌اند و استاندارد واحدی در این زمینه وجود ندارد. حتی برخی از انواع کارتهای جدید، نظیر Super VGA و TIGA، خود نیز دارای گونه‌های متفاوتی هستند. تفاوت گونه‌های مختلف کارتهای گرافیکی در وضوح[۱] تصویر، کارآیی یا سرعت نمایش اطلاعات، میزان حافظه‌ی ویدئویی[۲] و روش کدبندی اطلاعات جهت نمایش است.

برخی از مهمترین گونه‌های کارتهای گرافیکی، با توجه به سابقه‌ی آنها به صورت زیر است:

- **Monochrome** Display Adapter (MDA): به همراه CGA از قدیمی‌ترین انواع کارتهای گرافیکی است و همراه با معرفی IBM PC، در سال ۱۹۸۱، عرضه شد. این کارت فقط یک حالت عملیاتی[۳] را پشتیبانی می‌کند که نمایش متن تک رنگ، در قالب ۸۰ ستون و ۲۵ ردیف است. این کارت دارای حافظه‌ی ویدئویی بسیار کمی است و تنها یک صفحه نمایشی را می‌تواند در RAM نگهداری کند. همچنین این نوع کارت نمی‌تواند گرافیک را نمایش دهد.

- Color/Graphics Adapter (CGA): این نوع کارت نیز در سال ۱۹۸۱ معرفی شد و توانایی نمایش گرافیک و متنهای رنگی را داشت. کسانی که از این نوع کارت استفاده می‌کردند می‌توانستند از تلویزیون به جای صفحه‌نمایش[۴] استفاده کنند. این نوع کارت توانایی تولید خروجی RGB را داشت که در این روش، از سیگنالهای الکتریکی متفاوتی برای رنگهای قرمز، سبز و آبی استفاده می‌شود. این نوع کارت نیز در حالت متن، ۸۰ ستون در ۲۵ ردیف را نمایش می‌دهد و در حالت گرافیک دارای وضوح 320x200 نقطه[۵] با چهار رنگ است.

- Hercules Graphics Card (HGC): در همان زمانها، کارت دیگری که از هرجهت مشابه MDA بود ولی توانایی نمایش گرافیک را نیز داشت، معرفی شد. این نوع کارت توانایی نمایش گرافیک با وضوح 720x348 نقطه را در دو صفحه دارد.

- Enhanced Graphics Adapter (EGA): این کارت در سال ۱۹۸۵ معرفی شد و ضمن فراهم‌سازی وضوح بهتر، دارای قیمت کمتری بود. این نوع کارت وضوح 640x350 نقطه را با ۱۶ رنگ در یک زمان و ۶۴ سری‌رنگ[۶] فراهم می‌کند. این نوع کارت دارای حداکثر 256K حافظه‌ی ویدئویی بود. EGA بوسیله‌ی ROM-BIOS استاندار پشتیبانی نمی‌شود، بلکه EGA ROM-BIOS، به‌جای BIOS استاندارد سیستم قرار گرفته و امکان دسترسی به قابلیت‌های این نوع کارت را فراهم می‌سازد. غیر از IBM، عده‌ی دیگری از سازندگان، شروع به تولید گونه‌های دیگری از EGA نمودند که حالتهای عملیاتی دیگری را فراهم می‌کرد، ولی با EGA ساخت IBM، ناسازگار بودند. این مسأله، کار برنامه‌نویسان را دچار مشکل نمود.

- Video Graphics Array (VGA): این نوع کارت در سال ۱۹۸۷ و همزمان با کامپیوترهای IBM PS/2، معرفی شد و چون از فن‌آوریهای جدیدتر استفاده می‌کرد، رنگهای بیشتر، وضوح بالاتر و نمایش بهتر متن را فراهم نمود. بر خلاف EGA، کارتهای VGA سیگنالهای رنگ را به‌صورت آنالوگ به صفحه‌نمایش ارسال می‌کنند، نه دیجیتال. به این ترتیب، کارتهای VGA، بیش از ۲۶۰۰۰۰ رنگ متفاوت را در حالتهای عملیاتی ۲، ۴، ۱۶ و ۲۵۶ نمایش می‌دهند. این نوع کارت وضوح 640x480 را با ۲، ۴، یا ۱۶ رنگ، با توجه به حالت عملیاتی انتخاب‌شده، فراهم می‌کند. در یک حالت عملیاتی دیگر، با وضوح 320x200، ۲۵۶ رنگ قابل نمایش است. این نوع کارت دارای حافظه‌ی ویدئویی 256K است که تا 512K نیز قابل افزایش است. این نوع کارت نیز دارای ROM-BIOS خاص خود است و تولید کنندگان مختلف، از استاندارد یکسانی تبعیت نکرده‌اند.

- Super VGA: این نوع کارت دارای سخت‌افزاری مشابه با VGA است، اما نقطه‌ها را سریع‌تر و با تعداد رنگ بیشتر و وضوح بالاتر نمایش می‌دهد. ضمن‌آنکه، تمامی حالتهای عملیاتی VGA را نیز پشتیبانی می‌کند. در حالیکه، VGA در حالت چهارم توانایی نمایش 320x200 نقطه را با ۲۵۶ رنگ داشت، کارتهای Super VGA، سه حالت عملیاتی دیگر را نیز دارند که وضوح 640x200، 640x350، و 640x480 را با ۲۵۶ رنگ تأمین می‌کنند. حالتهای عملیاتی دیگری نیز وجود دارند که وضوح 800x600 و 1024x768 نقطه را نیز، در صورتی که از یک صفحه‌نمایش سازگار با VGA یا Multiscan استفاده شود، و نیز حافظه‌ی ویدئویی به‌اندازه‌ی کافی وجود داشته باشد، در دسترس قرار می‌دهند. در مورد این نوع کارت نیز تا سال ۱۹۹۰،

---

۱ resolution
۲ video RAM
۳ operationg mode
٤ monitor
٥ pixel
٦ color palette

استانداردی وجود نداشت و در این سال یک کنسرسیوم با نام اختصاری VESA[1]، سعی در تعریف حالتهای عملیاتی و BIOS استاندارد برای این نوع کارت نمود.

- Texas Instrument Graphics Architecture (TIGA): نوعی کارت است که برنامه‌پذیر[2] است و دارای گونه‌های متفاوتی، نظیر TI34010 و TI 34020 است. این نوع کارتها، تا ۵ بار سریع‌تر از کارتهای VGA هستند.

۲-۴ BIOS ویدئویی

در ROM-BIOS، توابع مختلفی در قالب وقفه‌ی شماره‌ی 10H پیش‌بینی شده است که ابتداً فقط با کارتهای MDA و CGA کار می‌کردند. اما، کارتهای EGA و VGA، دارای BIOS خاص خود هستند که در یک تراشه بر روی این کارتها وجود دارند و با روشن شدن سیستم، توابع بسطیافته فعال می‌شوند. این مجموعه از توابع بسطیافته‌ی BIOS در ارتباط با وقفه‌ی 10H عمل نموده و توابع EGA و VGA را به BIOS موجود اضافه می‌کنند. توابعی که برای EGA هستند، از توابع VGA کمتر هستند.

علاوه بر توابع BIOS، از توابع DOS نیز می‌توان برای کار با صفحه‌ی نمایش استفاده نمود. اما، این توابع از توابع BIOS، کندتر هستند. توابع DOS، دارای درجه‌ی بالاتری از سازگاری با کامپیوترهای مختلف هستند. همچنین روش دسترسی مستقیم به سخت‌افزار کارت ویدئویی نیز برای کار با صفحه‌ی نمایش، قابل استفاده است و از دو روش دیگر سریع‌تر است. دسترسی مستقیم در مورد سخت‌افزار کارت ویدئویی، نوشتن مستقیم اطلاعات در حافظه‌ی ویدئویی از نشانی B800H:0000 است و دلیل سریع‌تر بودن این روش، عدم نیاز به انجام کارهای مربوط به فراخوانی وقفه‌های BIOS یا DOS، از قبیل ذخیره سازی نشانی برگشت، ثباتها و پارامترها بر روی پشته و غیره است. عیب مهم روش دسترسی مستقیم، وابستگی برنامه‌ها به سخت‌افزار است. برای مثال، روالی که یک کاراکتر را بر روی صفحه‌ی نمایش نشان می‌دهد، برای CGA و MDA متفاوت خواهد بود و روال مربوط به هر کدام در دیگری عمل نخواهد کرد.

در مورد توابع ROM BIOS، یک دلیل دیگر کاهش سرعت، دسترسی به ROM BIOS برای اجرای کدهای توابع است. زیرا ROM به‌صورت ۸ بیتی قابل دسترسی است و این باعث کند شدن سیستم، به‌ویژه در پردازنده‌های ۳۲ بیتی به بالا می‌شود. یک راه رفع این مشکل، بارکردن ROM BIOS در RAM و در محدوده‌ی حافظه‌ی ویدئویی و حد ۱ مگابایت است که به آن، shadow ROM گفته می‌شود. این کار، دسترسی ۱۶ یا ۳۲ بیتی به توابع ROM BIOS را فراهم می‌کند.

توابع BIOS ویدئویی دارای شماره‌های 00H تا 1CH هستند که سرویسهای مختلفی را دراختیار قرار می‌دهند. برای فراخوانی هر کدام از این توابع، باید ثبات AH با شماره‌ی موردنظر مقداردهی شود. اگر یک زیرتابع مورد نظر باشد، شماره‌ی آن در ثبات AL قرار می‌گیرد.

Video BIOS functions and support from EGA, VGA and standard BIOS

No.	Meaning
00H	Determine video mode
01H	Define cursor size
02H	Set cursor position
03H	Read cursor position
04H	Read light pen
05H	Define current screen page
06H	Scroll screen up
07H	Scroll screen down
08H	Read character and attribute
09H	Write character and attribute
0AH	Write character to cursor position
0BH	Set color palette for graphics mode
0CH	Set screen pixel in graphics mode
0DH	Read screen pixel in graphics mod
...	...

[1] Video Electronic Standard Association (VESA)
[2] programmable

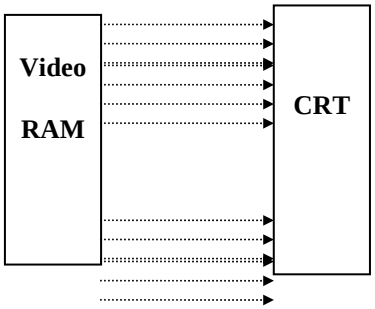
تعیین نوع کارت ویدئویی: در VGA BIOS با استفاده از تابع 1AH و زیرتابع 00H، با توجه به مقادیر برگشتی زیر می‌توان نوع کارت را تشخیص داد:

Code	Meaning
00H	No video card
01H	MDA card / monochrome monitor
02H	CGA card / color monitor
03H	Reserved
04H	EGA card / high resolution monitor
05H	EGA card / monochrome monitor
06H	Reserved
07H	VGA card / analog monochrome monitor
08H	VGA card / analog color monitor

در EGA BIOS با استفاده از تابع 12H و زیرتابع 10H، می‌توان نوع کارت را تشخیص داد.

۳-۴ حافظه‌ی ویدئویی[۱]

حافظه‌ی ویدئویی جزو فضای نشانی[۲] PC است و می‌تواند بر روی کارت ویدئویی باشد یا آنکه از قسمتی از RAM استفاده شود. محتویات این حافظه متناظر با صفحات متن و گرافیک است که بر روی صفحه‌ی نمایش نشان داده می‌شوند. یعنی در حالت متن، هر کاراکتر و رنگ آن، با دو بایت از این حافظه و درحالت گرافیک، هر نقطه[۳] بر روی صفحه نمایش، یک یا بیشتر بیت این حافظه متناظر است(که در ادامه دقیق‌تر توضیح داده می‌شود).



در کارتهای ویدئویی تک‌رنگ مثل MDA، حافظه‌ی ویدئویی در محدوده‌ی نشانی B000:0000-B000:7FFF قرار دارد و در کارتهای ویدئویی رنگی، مثل EGA، از نشانی B800:0000 شروع می‌شود. چون این حافظه در فضای نشانیهای سیستم قرار دارد، با روشهای معمولی دسترسی به حافظه قابل دستیابی است. یعنی برای نوشتن یک کاراکتر بر روی صفحه، می‌توان یک بایت کد کاراکتر و بایت رنگ (اصلی[۴] و زمینه[۵]) آنرا در این حافظه نوشت. حاصل کار، نمایش کاراکتر بر روی صفحه نمایش است.

ساختار حافظه‌ی ویدئویی در حالت متن: در حالت متن برای نمایش هر کاراکتر به دو بایت حافظه‌ی ویدئویی نیاز است. در بایت نخست، کد آسکی[۶] کاراکتر و در بایت دوم مشخصه‌ی[۷] آن قرار می‌گیرد. در بایت مشخصه، متناسب با نوع کارت و براساس کدبندی آن، رنگ اصلی، رنگ زمینه، شدت نور، و غیره مشخص می‌شوند. مثلاً در MDA، ۴ بیت نخست این بایت رنگ اصلی و ۴ بیت دوم رنگ زمینه را مشخص می‌کنند.

اولین بایت حافظه‌ی ویدئویی متناظر با اولین کاراکتر نمایش داده شده بر روی صفحه نمایش است و بایت دوم حافظه‌ی ویدئویی، مشخصه‌ی آن است. نشانی مبدأ اولین کاراکتر، 0000H و نشانی مبدأ مشخصه‌ی آن، 0001H است. کاراکتر دوم و مشخصه‌ی آن در نشانیهای 0002H و 0003H قرار دارند و الی آخر. به این ترتیب برای نمایش یک صفحه اطلاعات بر روی صفحه نمایش، حافظه‌ی ویدئویی باید 80x25x2 = 4000 byte ظرفیت داشته باشد، که به آن

---

۱ video RAM
۲ address space
۳ pixel
۴ foreground
۵ background
۶ ASCII code
۷ attribute

یک صفحه[۱] گفته می‌شود. اگر ظرفیت حافظه‌ی ویدئویی بیش از 4KB باشد(مثلاً 8KB)، می‌توان بیش از یک صفحه را در حافظه نگهداشت، که در کارتهای EGA و VGA این امکان وجود دارد، ولی در کارتهای MDA این‌گونه نیست. با استفاده از توابع وقفه 10H، می‌توان این صفحه‌ها را تعویض نمود و روشی مفید برای بهبود نمایش اطلاعات بر روی صفحه نمایش است.

دسترسی مستقیم به حافظه‌ی ویدئویی: از فرمول زیر می‌توان برای نوشتن مستقیم یک کاراکتر در حافظه‌ی ویدئویی یا خواندن آن، استفاده نمود:

character_offset_position (row, col) = row * 160 + col * 2

و نشانی بایت مشخصه‌ی آن به صورت زیر خواهد بود:

attribute_offset_position (row, col) = row * 160 + col * 2 + 1

برای نوشتن مستقیم متنها در زبان پاسکال می‌توان از دستور Mem استفاده نمود، که نشانی قطعه، نشانی قطعه حافظه‌ی ویدئویی است که در کارتهای رنگی، B800H و در MDA، B000H است و نشانی مبدأ هم از فرمول فوق بدست می‌آید. به این ترتیب می‌توان روالی به‌صورت زیر برای نوشتن مستقیم یک کاراکتر در نقطه‌ی (row, col)، نوشت:

```
procedure DirectWriteCh (row, col, ch, att : byte);
begin
   Mem[$B800:(row * 160 + col * 2)] := ch;
   Mem[$B800:(row * 160 + col * 2 + 1)] := att;
end;
```

همچنین برای خواندن یک کاراکتر و مشخصه‌ی آن از نقطه‌ی (row, col)، می‌توان به صورت زیر عمل کرد:

```
procedure DirectReadCh (row, col : byte; var ch, att: byte);
begin
  ch := Mem[$B800:(row * 160 + col * 2)];
  att := Mem[$B800:(row * 160 + col * 2 + 1)];
end;
```

در زبان C از دو روش می‌توان برای این منظور استفاده نمود. روش نخست استفاده از دستورات pokeb و peekb و روش دوم، ایجاد یک far pointer به حافظه‌ی ویدئویی است.

الف) استفاده از دستورات pokeb و peekb:

```
void DirectWriteCh (unsigned char row, unsigned char col, unsigned char ch, unsigned char att)
{
   pokeb(0xB800, row * 160 + col * 2, ch);
   pokeb(0xB800, row * 160 + col * 2+1, att);
}

void DirectReadCh (unsigned char row, unsigned char col, unsigned char *ch, unsigned char *att)
{
   *ch = peekb(0xB800, row * 160 + col * 2);
   *att = peekb(0xB800, row * 160 + col * 2+1);
}
```

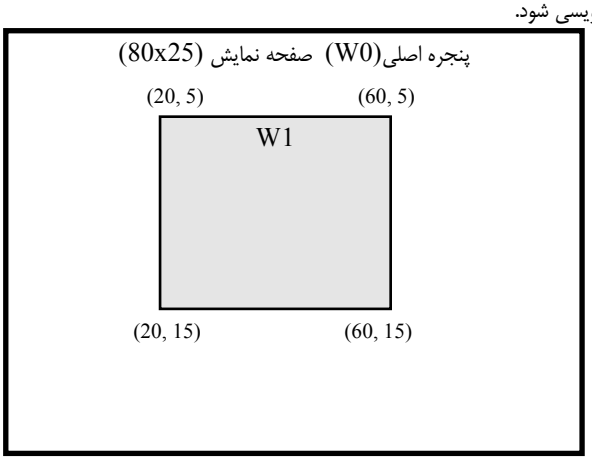
ب)استفاده از دستورات far pointer:

```
struct VelB {
        unsigned char ch, att;
};
typedef struct VelB far *VP;

void DirectWriteCh (unsigned char row, unsigned char col, unsigned char ch, unsigned char att)
{
  VP lptr;
```

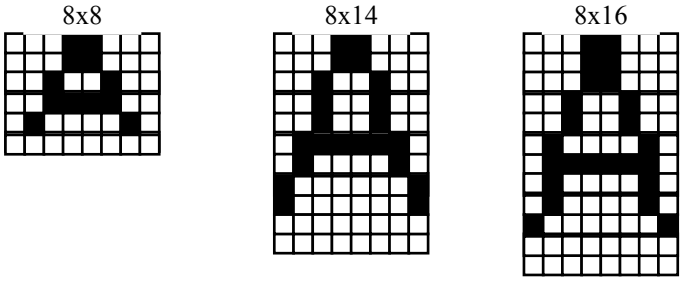
[۱] page

```
    lptr = MK_FP (0xB800, 0) + row * 80 + col;
    lptr->ch = ch;
    lptr->att = att;
}

void DirectReadCh (unsigned char row, unsigned char col, unsigned char *ch, unsigned char *att)
{
    VP lptr;
    lptr = MK_FP (0xB800, 0) + row * 80 + col;
    *ch = lptr->ch;
    *att = lptr->att;
}
```

مدیریت پنجره‌ها[۱]: از روشهای فوق برای دسترسی مستقیم به حافظه می‌توان استفاده نموده و برنامه‌هایی نوشت که در آنها از پنجره‌ها استفاده می‌شود. آشکارسازی و پنهان کردن پنجره‌ها، مستلزم خواندن و نوشتن سریع اطلاعات صفحه‌ی نمایش و ذخیره و بازیابی اطلاعات نوشته شده بر روی صفحه در بافرهای موقتی است. برای مثال اگر بخواهیم در شکل زیر پنجره‌ی W1 را بر روی صفحه نشان دهیم، لازم به نگهداری اطلاعات پنجره‌ی اصلی W0 هستیم. بنابر این، باید یک بافر با ابعاد 2*(1+5-15)*(1+20-60) برای نگهداری کاراکترها و مشخصه‌های آنها که در این محدوده صفحه نمایش قرار دارند، اختصاص دهیم. سپس با روشهای دسترسی مستقیم فوق و با استفاده از توابع DirectReadCh، اطلاعات پنجره را در این بافر ذخیره کنیم. این کار قبل از رسم پنجره W1 انجام می‌شود. پس از نمایش W1 و عدم نیاز به آن، برای پاک کردن آن از روی صفحه نمایش و برگشتن به صفحه‌ی قبلی، کافیست اطلاعات موجود در بافر، با استفاده از تابع DirectPutCh، در حافظه‌ی ویدئویی بازنویسی شود.



انتخاب و تعریف فونتها[۲]: کارتهای EGA و VGA اجازه‌ی انتخاب و تعریف فونتهای جدید را می‌دهند. این کارتها از مولدهای قوی کاراکتر استفاده می‌کنند که شکل هر کاراکتر را از روی bitmap آن که در حافظه‌ی ویدئویی قرار دارد، بر روی صفحه‌ی نمایش ترسیم می‌کنند. کارتهای EGA، با توجه به حالت انتخاب‌شده، اجازه‌ی نمایش کاراکترهای با ابعاد 8x8 و 8x14 نقطه را می‌دهند. همچنین، کارتهای VGA، اجازه‌ی نمایش کاراکترهای با ابعاد 8x8، 8x14 و 8x16 نقطه را می‌دهند.



به ازای هر ردیف در ماتریسهای فوق، نیاز به یک بایت است. به این ترتیب برای هر کاراکتر، در سه حالت فوق، به‌ترتیب به ۸، ۱۴ و ۱۶ بایت حافظه نیاز خواهد بود و برای تعریف فونت برای ۲۵۶ کاراکتر، به 256x8، 256x14 و 256x16 بایت حافظه نیاز است. پس برای معرفی فونتهای جدید(مثلاً فونتهای فارسی) می‌توان یک بافر(character table) با اندازه‌های فوق تعریف نموده و ماتریسهای فونتهای طراحی شده را ایجاد نمود. سپس برای تعریف فونتها، از زیرتابع00H تابع 11H وقفه‌ی EGA/VGA BIOS 10H استفاده کرد:

```
AH = 11H
AL = 00H
BH = Lines per character (also bytes per character)
```

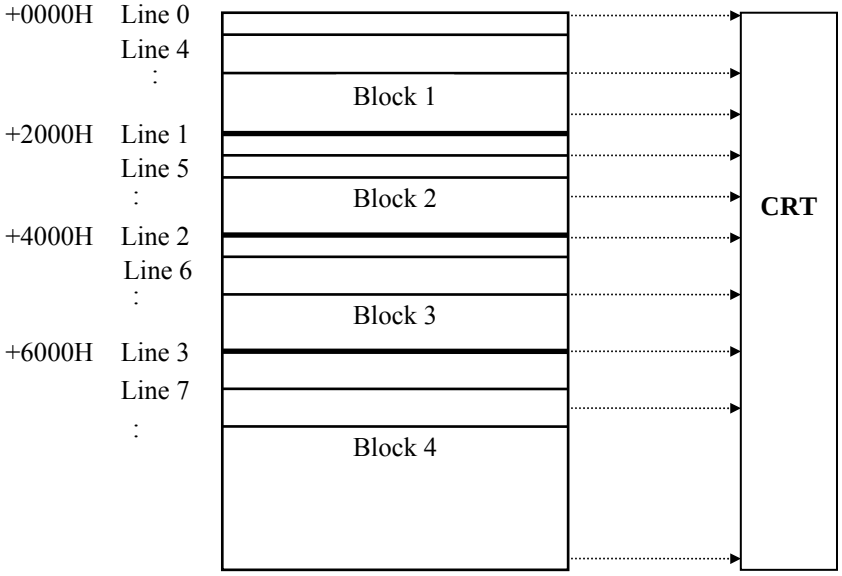
۱ window management
۲ fonts

BL = Character table (0 or 1)
CX = Number of character in table
DX = ASCII code of first character table
ES = Segment address of character table in RAM
BP = Offset address of character table in RAM

۴-۴ حالت گرافیکی در کارت Hercules

کارت HGC در حالت گرافیکی وضوح 348x720 را فراهم می‌کند. هر نقطه بر روی صفحه نمایش، متناظر با یک بیت در حافظه‌ی ویدئویی است. اگر مقدار بیت ۱ باشد، یک نقطه بر روی صفحه نمایش دیده می‌شود. وگرنه، نقطه وجود ندارد. محورهای مختصات صفحه‌ی نمایش، از صفر شروع می‌شوند و در هر خط از ۰ تا ۷۱۹ نقطه و شماره خطوط هم بین ۰ و ۳۴۷ است.

ساختار حافظه‌ی ویدئویی در حالت گرافیکی:



حافظه ویدئویی 32K است که به ۴ بلوک 8K تقسیم شده است. خطوط مختلف پشت سر هم نیستند و خطوط با شماره‌های ۰، ۴، ۸، ... در بلوک اول، خطوط با شماره‌های ۱، ۵، ۹، ... در بلوک دوم و الی‌آخر. هر خط شامل ۹۰ بایت است که تعداد نقاط هر خط، 720 = 90x8 است. در هر بلوک هم ۳۴۸ خط وجود دارد و مابقی حافظه‌ی ویدئویی بلااستفاده است.

برای محاسبه‌ی نشانی یک نقطه‌ی صفحه‌ی نمایش (X, Y) در حافظه‌ی ویدئویی باید نشانی بایت و شماره‌ی بیت آن بایت را بدست آوریم:

ByteAddress(X, Y) = 2000H * (Y mod 4) + 90 * int(Y/4) + int(X/8)

BitNumber(X) = 7 - X mod 8 { بیت هشتم نقطه‌ی صفرم را و بیت اول نقطه‌ی هفتم را مشخص می‌کند }

به این ترتیب باید بیت بدست آمده در بایت موجود در نشانی داده شده، برای روشن شدن نقطه، به 1 و برای خاموش شدن، به 0 مقداردهی شود. برای مثال در زبان پاسکال برای روشن کردن بیت (X, Y):

```
procedure PutPixel (X, Y : Integer);
var   PixelByte, PixelBit : Byte;
begin
  PixelByte := Mem[$B000:ByteAddress(X, Y)];
  PixelBit := 1;
  PixelBit := PixelBit Shl BitNumber(X);    { Shl = Shift Left }
  PixelByte := PixelByte Or PixelBit;
  Mem[$B000:ByteAddress(X, Y)] := PixelByte;
end;
```

۴-۵ حالت گرافیکی در کارت CGA

این نوع کارت سه حالت گرافیکی را پشتیبانی می‌کند، که دو مورد بیشتر مورد استفاده قرار گرفتند:

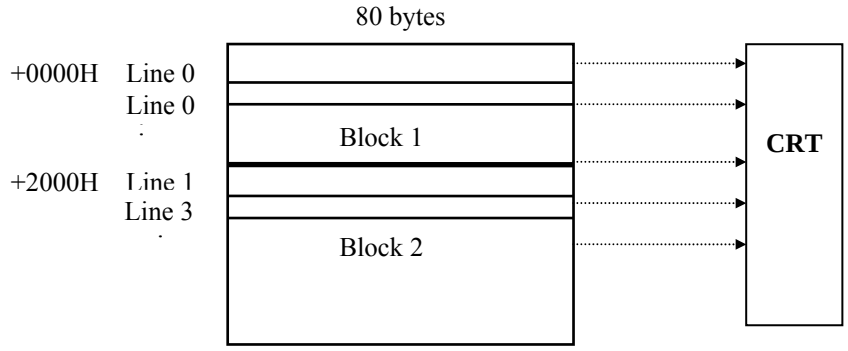
- Color-Suppressed: وضوح 160x100 با ۱۶ رنگ، که CRT controler این حالت را پشتیبانی می‌کرد، اما سایر بخشهای سخت‌افزار، این حالت را پشتیبانی نمی‌کردند.
- وضوح 320x200 با ۴ رنگ،
- وضوح 640x200 با ۲ رنگ.

وضوح 320x200: کارت CGA می‌تواند از 16K حافظه‌ی ویدئویی استفاده کند و این وضوح را با ۴ رنگ فراهم نماید. رنگهایی که با برنامه‌ریزی ثبات انتخاب رنگ یا تابع 0BH وقفه‌ی 10H، قابل انتخاب هستند، عبارتند از:

– رنگ زمینه: از ۱۶ سری رنگ[۱] قابل انتخاب است.
– رنگ اصلی: سه رنگ دیگر، که از دو سری رنگ زیر قابل انتخاب هستند:

Palette 1: { Cyan, Violet, White} Palette 2: { Green, Red, Yellow }

ساختار حافظه‌ی ویدئویی در حالت گرافیکی: برای هر نقطه، به دو بیت نیاز داریم:



حافظه‌ی ویدئویی 16K است که به دو بلوک 8K تقسیم شده است. خطوط زوج در بلوک اول و خطوط فرد در بلوک دوم قرار دارند. برای محاسبه‌ی نشانی یک نقطه‌ی صفحه‌ی نمایش (X, Y) در حافظه‌ی ویدئویی باید نشانی بایت و شماره‌ی اولین بیت آن بایت را بدست آوریم:

ByteAddress(X, Y) = 2000H * (Y mod 2) + 80 * int(Y/2) + int(X/4)
BitNumber(X) = 6 - 2 *(X mod 4) {بیتهای ۷ و ۸ نقطه‌ی ۰ را مشخص می‌کنند}

به این ترتیب اگر برای مثال فرمول دوم مقدار ۴ را برگرداند، اطلاعات یک نقطه در بیتهای ۴ و ۵ بایت داده شده، کدبندی شده است.

وضوح 640x200: در این حالت، فقط یک بیت برای هر نقطه کافی است و دو رنگ بیشتر نداریم. در نتیجه، عرض صفحه دو برابر می‌شود:

ByteAddress(X, Y) = 2000H * (Y mod 2) + 80 * int(Y/2) + int(X/8)
BitNumber(X) = 7 - X mod 8

ثباتهای CGA: این نوع کارت دارای ۳ ثبات است:

- Mode Selection Register: با این ثبات ۸ بیتی می‌توان حالتهای سه‌گانه‌ی فوق را انتخاب نمود و ثباتی قابل نوشتن و غیرقابل خواندن است. نشانی درگاه این ثبات، 3D8H است. بیت 0 این ثبات، تعداد کاراکترهای قابل نمایش را مشخص می‌کند که اگر مساوی 0 باشد، 40x25 و اگر مساوی 1 باشد، 80x25 کاراکتر در صفحه نمایش داده می‌شود. بیت 1، حالت ویدئویی را مشخص می‌کند، که مقدار 0 برای حالت متن و مقدار 1 برای حالت گرافیک است. برای تغییر حالت از متن به گرافیک و برعکس باید مقدار این بیت تنظیم شود.
- Status Register: وضعیت مانیتور و اطلاعات مربوط به سیگنالینگ را برمی‌گرداند و یک ثبات فقط خواندنی است که نشانی درگاه آن، 3DAH است.
- Color Selection Register: با این ثبات می‌توان سری رنگ زمینه و اصلی را انتخاب نمود. این ثبات فقط نوشتنی است و نشانی درگاه آن، 3D9H است.

[۱] color palette

۴-۶ کارتهای EGA و VGA

همانطوری که قبلاً گفته شد، سازندگان مختلف از یک استاندارد واحد پیروی نکرده‌اند. از اینرو، برنامه‌سازی مستقیم آنها، بسیار پیچیده است و گاهی بهتر است که از توابع EGA/VGA BIOS استفاده شود.

نوع مانیتور: کارآیی کارتهای EGA یا VGA، به نوع مانیتور وابستگی زیادی دارد و درصورتی‌که مانیتور از نوع مناسبی نباشد، قابلیت‌های این نوع کارت در اختیار قرار نمی‌گیرد. این نوع کارتها به انواع مانیتورهای زیر قابل اتصال هستند:

- EGA: یک کارت EGA قابل اتصال به مانیتورهای CGA، EGA، multisync یا Monochrome است و براساس نوع مانیتور، کارت EGA مشابه یک کارت CGA یا MDA قوی عمل می‌کند.
- VGA: کارتهای VGA قابل اتصال به مانیتورهای Analog Monochrome و نیز مانیتورهای VGA هستند. در حالت تک‌رنگ، وضوح بالای کارتهای VGA در دسترس خواهد بود.

حرکت آرام تصویر[۱]: کارتهای CGA و HGC، توانایی حرکت دادن آرام تصویر را نداشتند. اما، در سخت‌افزار این کارتهای EGA و VGA، قابلیت‌هایی برای ایجاد تصاویر متحرک[۲] پیش‌بینی شده است، که از آن جمله، حرکت آرام تصویر در حالت گرافیک است. با این قابلیت، می‌توان تصویر را در جهتهای افقی و عمودی حرکت داد و نیازی به بازنویسی حافظه‌ی ویدئویی نیست. برای این منظور دو ثبات با نام pel panning registers در این نوع کارتها وجود دارد، که یکی برای حرکت افقی و دیگری برای حرکت عمودی تصویر است و با افزایش مقدار این ثباتها، تصویر یک خط به سمت چپ یا بالا حرکت می‌کند. نشانی درگاه این ثباتها استاندارد نبوده و برای نمونه به ترتیب 3C0H و 3D4H می‌تواند باشد.

حالت ۲۵۶ رنگ کارت VGA: این حالت، یکی از ساده‌ترین حالتهای کارت VGA است. یکی از مزایای کارت VGA نسبت به EGA، توانایی نمایش ۲۵۶ رنگ متفاوت به‌صورت همزمان بر روی صفحه است. ۲۵۶ رنگ مختلف را می‌توان از ۲۶۲۰۰۰ سری‌رنگ متفاوت انتخاب نمود. اما در حالت ۲۵۶ رنگ، وضوح، 320x200 نقطه خواهد بود که در این حالت وضوح کمتری را نسبت به حالت ۱۶ رنگ خواهیم داشت، که 640x480 نقطه است.

تنظیم حالت ۲۵۶ رنگ: نخستین کار برای تنظیم این حالت، فعال‌سازی[۳] آن است. برای این منظور تابع 00H وقفه‌ی 10H باید استفاده شود. شماره‌ی این حالت، 13H است که در ثبات AL باید قرار گیرد. در نتیجه، با فراخوانی این تابع حالت ۲۵۶ رنگ با وضوح 320x200 فعال خواهد شد:

```
mov  ah, 00h
mov  al, 13h
int  10h
```

ساختار حافظه‌ی ویدئویی در این حالت: هر نقطه به یک بایت نیاز دارد که کد نقطه است. حافظه‌ی ویدئویی 64KB است و نشانی قطعه‌ی آن B800H است:

```
VideoSegAddr = B800H
```

در هر خط، ۳۲۰ نقطه داریم. پس برای بدست‌آوردن نشانی هر نقطه P(X, Y) در حافظه‌ی ویدئویی:

```
PixelOffset (X, Y) = Y * 320 + X
```

و روال PutPixel به صورت زیر خواهد بود:

```
procedure PutPixel(X, Y : Integer; Color :  Byte);
begin
   Mem[VideoSegAddr:PixelOffset(X, Y)] := Color;
end;
```

و تابع GetPixel هم به صورت زیر خواهد بود، که رنگ یک نقطه را برمی‌گرداند:

```
function GetPixel(X, Y : Integer): Byte;
begin
  GetPixel := Mem[VideoSegAddr:PixelOffset(X, Y)];
end;
```

---

۱ smooth scrolling
۲ animation
۳ enabling

فصل ۵: برنامه‌سازی صفحه کلید
(Keyboard Programming)

۵-۱ مقدمه

صفحه کلید یکی از لوازم ورودی PC است و باید بتوان آنرا مدیریت نمود. سخت‌افزار صفحه کلید پالس‌های الکتریکی حاصل از فشار دادن کلید را به اعدادی تبدیل نموده و به PC ارسال می‌کند. این اعداد "Scan code" نامیده می‌شوند و تلفیق کلیدهای معمولی با کلیدهای کنترلی، scan code متفاوتی را ایجاد می‌کند.

ارتباط صفحه کلید با PC از طریق یک کابل انجام می‌شود که نوع ارتباط، سریال و همگام[۱] است و از این نظر با واسط سریال که ناهمگام[۲] است، دارای تفاوت است.

make and break codes: به ازای فشاردادن و رهاکردن یک کلید، کدهای متفاوتی تولید می‌شود و به PC ارسال می‌شود تا تعین شود که یک کلید زده شده یا رها شده است و از طریق فشاردادن دو کلید در یک زمان قابل تشخیص می‌شود و بدون این روش نمی‌توان از کلیدهای ترکیبی مثل Shift+A یا Ctrl+X استفاده نمود. موقعی که یک کلید زده می‌شود، بیت هفتم scan code صفر است و این یک make code است و موقع رهاکردن این کلید، بیت هفتم یک خواهد بود و به عنوان یک break code ارسال می‌شود.

scan code یک بایت است و با استفاده از آن نمی‌توان ترکیب همه‌ی کلیدهای کنترلی و حروف بزرگ و کوچک و غیره را ایجاد نمود. راه حل این است که به‌ازای هر کلید معمولی یا کنترلی یک scan code ارسال شود. یعنی، مثلاً برای Shift+A، ابتدا یک make code کلید Shift (36H) و سپس make code کلید A (1EH) ارسال می‌شود. سیستم چون دوتا make code دریافت شده و break code دریافت نشده، پس دو کد را در هم ترکیب نموده و حرف بزرگ[۳] A را ایجاد می‌کند و کد اَسکی متناظر را به برنامه‌ها ارسال می‌کند. این گونه تفسیرها، توسط روال مدیریت وقفه‌ی صفحه کلید ROM-BIOS[۴] (INT 09H) انجام می‌شود و با توجه به make & break codes، ASCII code ایجاد می‌شود.

تبدیل scan code به ASCII code: چون scan code به ASCII code می‌شود، پس با توجه به یکسان و استاندارد بودن ASCII، در صورت تغییر سخت‌افزار صفحه کلید و ارسال scan codeهای متفاوت، با تغییر روال مدیریت وقفه‌ی صفحه‌کلید در ROM-BIOS، برنامه‌ها مشکلی نخواهند داشت. برای مثال در کامپیوترهای Laptop سخت‌افزار دیگری برای صفحه کلید وجود دارد که متفاوت با صفحه کلیدهای معمولی ۱۰۱ یا ۱۰۲ کلیدی است.

۵-۲ دسترسی به صفحه کلید از طریق BIOS

وقفه‌ی 16H، سه تابع برای خواندن صفحه کلید و وضعیت آن فراهم نموده است. اما این توابع ناقص هستند و مثلاً تابعی برای برداشتن یک یا چند کلید از بافر صفحه‌کلید و دور ریختن آنها، وجود ندارد، که جزو توابع DOS است و با برنامه‌نویسی مستقیم امکان‌پذیر است.

Function 00H: Read keyboard (Remove character from buffer)

در صورتیکه کاربر قبلاً کلیدی را زده باشد این تابع کاراکتر را از بافر بر می‌دارد و به برنامه برمی‌گرداند. در غیر این‌صورت، منتظر می‌شود. مقادیر برگشتی این تابع در ثباتهای AL و AH قرار می‌گیرند. اگر کلید زده شده، جزء ۱۲۸ کاراکتر اول جدول ASCII و غیر کنترلی باشد، یک مقدار غیر 00H در AL وجود خواهد داشت. در غیر این‌صورت، AH=00H است و کد متناظر از جدول Extended ASCII در AH وجود خواهد داشت. این تابع، کاراکتر را از بافر برمی‌گرداند و اشاره‌گر بافر چرخه‌ای صفحه کلید را به جلو می‌برد. این تابع معادل ReadKey در زبان پاسکال یا ()getch در زبان C است.

Function 01H: Read keyboard (Don't remove character from buffer)

برخلاف تابع 00H، این تابع کاراکتر را از بافر برنمی‌دارد و دوباره می‌توان آنرا خواند. به‌این ترتیب، می‌توان فهمید که یک کلید زده شده است یا نه؟ اگر زده نشده، منتظر نشد یا تا زمان زدن یک کلید، عملی انجام شود. کاری که در زبان پاسکال با تابع KeyPressed و در زبان C با تابع ()kbdhit انجام می‌شود. در صورت زده شدن یک کلید، ZF=0 است و ثباتها همانند تابع قبلی مقدار می‌گیرند، و در غیر این‌صورت، ZF=1 خواهد بود و با چک کردن ZF می‌توان فهمید که کلیدی زده شده است یا نه.

---

۱ syncronous
۲ asynchronous
۳ upper case
٤ ROM-BIOS keyboard handler

Function 02H: Read control keys

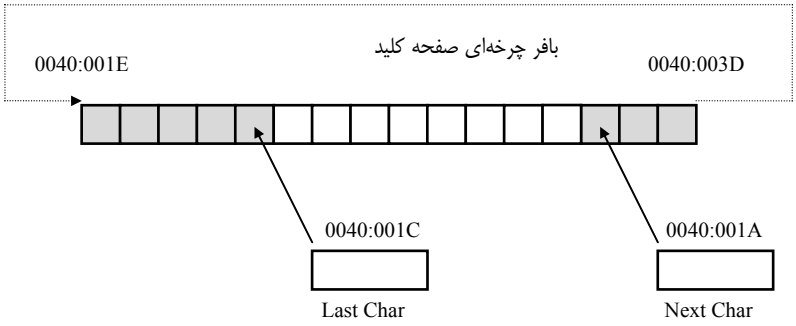
با این تابع، بایت کنترلی صفحه‌کلید را می‌توان خواند تا وضعیت کلیدهای کنترلی را بررسی کرد. بیتهای مختلف این بایت متناظر با کلیدهای زیر است:

بیت ۰ : Right Shift	بیت ۴ : Scroll Lock
بیت ۱ :Left Shift	بیت ۵ : Num Lock
بیت ۲ : Ctrl	بیت ۶ : Caps Lock
بیت ۳ : Alt	بیت ۷ :Insert

۵-۳ متغیرهای صفحه کلید در قطعه‌ی متغیرهای BIOS

در فصل ۳ گفتیم که تعدادی از متغیرهای BIOS مربوط به صفحه کلید هستند. این متغیرها بوسیله‌ی INT 09H و INT 16H دستکاری می‌شوند. این متغیرها، اندازه و نشانی مبدأ آنها به شرح زیر است(نشانی قطعه 0040H):

Offset	Meaning	Size
0017H	Keyboard status byte	1 byte
0018H	Extended keyboard status byte	1 byte
0019H	ASCII code entry	1 byte
001AH	Next character in keyboard buffer	1 word
001CH	Last character in keyboard buffer	1 word
001EH	Keyboard buffer	16 words



متغیرهای نشانیهای 1AH، 1CH و 1EH، بافر چرخه‌ای صفحه‌کلید را تعیین می‌کنند. اولین متغیر، اشاره‌گری به کاراکتر بعدی در بافر چرخه‌ای، دومین متغیر، اشاره‌گر به آخرین کاراکتر و متغیر سوم که از 1EH شروع شده و تا 3DH ادامه می‌یابد و دارای طول ۱۶ کلمه یا ۳۲ بایت است، محدوده‌ی بافر چرخه‌ای است. دلیل انتخاب بافر چرخه‌ای، عدم نیاز به شیفت‌دادن بافر در صورت برداشتن یک کاراکتر از ابتدای بافر است و کافی است که فقط اشاره‌گر Next یکی اضافه شود، و در صورت رسیدن با آخر بافر، دوباره به ابتدای بافر اشاره کند. این روش زمان پردازش بافر بوسیله‌ی INT 09H را کاهش می‌دهد.

۵-۴ سایر نکات

روال جدید مدیریت وقفه‌ی صفحه‌کلید: اگر بخواهیم تفسیر جدیدی از ترکیب کلیدها داشته باشیم، و برای نمونه ترکیب کلیدهای Alt+Ctrl+Delete باعث reset شدن سیستم نشود، یا با زدن ترکیب کلیدهای مثلاً Alt+F10 یک برنامه‌ی مقیم در حافظه (TSR)[۱] فعال شود، باید روال جدیدی برای مدیریت وقفه‌ی صفحه کلید(INT 09H) بنویسیم و با روال موجود جایگزین کنیم.

برنامه‌ریزی مستقیم صفحه کلید: باید تراشه‌ی 8255 را برنامه‌ریزی کرد. 8255 یک تراشه همه‌منظوره برای ارتباطات ورودی-خروجی است، که دستگاههایی مثل صفحه کلید به آن متصل می‌شوند. این تراشه سه درگاه به نامهای PA، PB و PC دارد که نشانیهای آنها به ترتیب 60H، 61H و 62H هستند. ارتباط بین صفحه کلید و پردازنده از طریق مکانیزم وقفه و درگاههای PA و PB برقرار می‌شود. در درون صفحه کلید یک پردازنده کوچک وجود دارد که همیشه وضعیت کلیدها را بررسی می‌کند و منتظر تغییرات در وضعیت آنها است و متناظر با فشار دادن یا رهاکردن کلیدها، make & break code را به سیستم ارسال می‌کند. برای خواندن یک کلید، scan code آن از درگاه PA خوانده می‌شود و سپس با یک کردن بیت ۷، Ack از طریق PB ارسال می‌شود.

حرکت دادن مکان‌نما: برای این منظور باید تراشه‌ی 6845 که CRT Controller است، برنامه‌ریزی شود، که در اینجا به آن نمی‌پردازیم. اما قطعه کد زیر به زبان C، کار حرکت دادن مکان نما را بر روی صفحه نمایش انجام می‌دهد:

```
void GoToXY(int X, int Y)
{
```

[۱] Terminate Stay Resident

```
    int V = X * 80 + Y;
    outportb(0x3D4,  14);
    outportb(0x3D5,  (V >> 8) & 0xFF);
    outportb(0x3D4,  15);
    outportb(0x3D5, V & 0xFF);
}
```

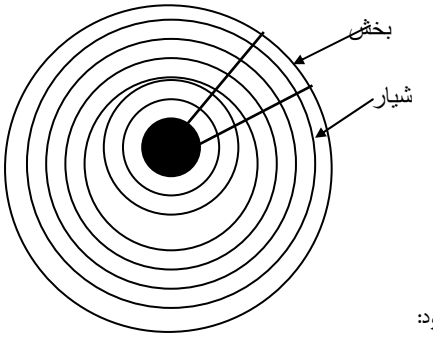
فصل ۶: گردانندهای دیسکت و دیسک سخت
(Diskette and Hard Disk Drives)

۶-۱ مقدمه

BIOS و DOS، هر دو توابعی را برای کارکردن با دیسکت و دیسک سخت در اختیار قرار می‌دهند. توابع BIOS سطح پایین هستند و به دیسک به عنوان یک وسیله‌ی فیزیکی نگاه می‌کنند. اما توابع DOS سطح بالا بوده و به دیسک به‌صورت منطقی نگاه می‌کنند. برای نوشتن برنامه‌های کمکی مثل Norton Utilities یا PC Tools، نیاز به دسترسی به دیسک درسطح BIOS است. دسترسی مستقیم به دیسکت و به‌ویژه دیسک سخت نیز مورد نیاز است. مثلاً در صورتی که بخواهیم از یک کنترلر خاص برای دسترسی به دیسک استفاده کنیم که بوسیله‌ی BIOS پشتیبانی نمی‌شود، یا آنکه بخواهیم یک روش دسترسی ویژه پیاده‌سازی کنیم. دسترسی مستقیم به دیسک سخت، نیازمند برنامه‌ریزی کنترلرهایی نظیر IDE یا SCSI است.

۶-۲ ساختار دیسکت و دیسک سخت

ساختار دیسکت: یک دیسکت یا دیسک نرم[۱] دارای تعدادی شیار[۲] است که از 0 تا N شماره‌گذاری می‌شوند. هر شیار به تعدادی بخش[۳] تقسیم می‌شود و میزان معینی داده را می‌تواند در خود ذخیره کند.



تعداد بخشها در هر شیار به نوع دیسکت و نوع قالب‌بندی یا فرمت آن بستگی دارد. از نظر اندازه، دو نوع دیسکت‌وجود دارد که نوع قدیمی "5.25 و نوعی که الان استفاده می‌شود، "3.5 است. در فرمت DOS، هر بخش دیسکت 512 بایت است و در هر بار دسترسی به دیسکت، حداقل یک بخش خوانده یا نوشته می‌شود و خواندن و نوشتن کمتر از آن امکان‌پذیر نیست.

محاسبه ظرفیت دیسکت: با فرمول زیر می‌توان ظرفیت یک دیسکت را محاسبه نمود:

Floppy Disk Capacity = Tracks * Sector_Per_Track * 512

---

۱ floppy disk
۲ track
۳ sector

این فرمول برای محاسبه‌ی فرمول یک طرف دیسکت است و اگر گردانندهی دیسکت دارای دو هِد[۱] باشد، دوطرف دیسکت قابل خواندن و نوشتن است و در این صورت دیسکت *چگالی دوبرابر*[۲] خواهد بود و ظرفیت دیسکت دو برابر می‌شود. پس ساختار دیسکت شامل تعدادی طرفها، تعداد شیارها، تعداد بخشهای هر شیار و ظرفیت یک بخش است.

دیسکت‌های "3.5، یا چگالی دوبرابر هستند و 720KB ظرفیت دارند و یا چگالی بالا[۳] بوده و 1.44MB ظرفیت دارند. در دیسکتهای چگالی بالا، تعداد بخشها در هر شیار دو برابر چگالی دوبرابر بوده و مساوی ۱۸ است. تعداد شیارها در هر دو نوع، ۸۰ در هر طرف است. به این ترتیب:

Double-dinsity: Capacity = 2 Side * 80 Tracks * 9 Sector_Per_Track * 512 Byte_Per_Sector = 720KB

High-dinsity: Capacity = 2 Side * 80 Tracks * 18 Sector_Per_Track * 512 Byte_Per_Sector = 1.44MB

ساختار دیسک سخت: دیسکت فقط یک صفحه یا دوصفحه پشت و رو (Side 0, Side 1) است. دیسک سخت هم تعدادی صفحه است که روی هم قرار گرفته‌اند و به هر کدام یک سیلندر[۴] گفته می‌شود. برای آنکه خواندن داده‌ها از صفحه‌های مختلف سریع‌تر باشد، تعداد هدها زیادتر است. حرکت‌دادن یک هد روی شیارهای مختلف کند است، پس اگر بخواهیم داده‌ای را که حجم زیادی دارد در چند شیار بنویسیم، یک راه، نوشتن آن در شیارهای پشت سر هم است. اما این کار باعث می‌شود که بازویی[۵] که هدها به آن متصل هستند، نیاز به حرکت داشته باشد. ولی برای آنکه این کار سریع‌تر باشد، داده‌هایی را که در یک شیار جا نمی‌شوند، در شیارهای هم‌شماره در صفحات مختلف می‌نویسند. این کار نیاز به حرکت بازو را برطرف می‌کند و باعث تسریع در نوشتن و خواندن داده‌ها می‌شود. ظرفیت دیسک سخت:

Hard Disk Capacity = Cylenders * Tracks * Sector_Per_Track * 512

۶-۳ دسترسی به دیسکت از طریق BIOS

وقفه‌ی 13H توابع دسترسی به دیسکت را فراهم می‌کند. این توابع عبارتند از:

Function	Task
00H	Reset
01H	Read status
02H	Read
03H	Write
04H	Verify
05H	Format
08H	Request format
09H	Define drive type
16H	Detect diskette change
17H	Determine diskette format
18H	Determine diskette format

وضعیت گرداننده: تابع 01H، پس از فراخوانی، بایت وضعیت گرداننده را در ثبات AH برمی‌گرداند. همچنین پس از فراخوانی سایر توابع، اگر مقداری غیرصفر در AH باشد یا CF (Carry Flag)، مقدار 1 داشته باشد، خطا رخ داده است. مقادیر مختلف بایت وضعیت گرداننده، نمایش‌دهنده انواع خطاها است:

00H: No error
01H: Illegal function number
02H: Address marking not found
03H: Attempt to write to write-prootected diskette
...

نوع گرداننده و فرمت دیسکت: تابع 08H، اطلاعات زیر را در ثباتهای مختلف برمی‌گرداند:

---

۱ head
۲ double-density
۳ high-density
۴ cylender
۵ arm

RegisterInformation

BL	Drive type :	01H = 5.25”, 360KB 02H = 5.25”, 1.2MB 03H = 3.5”, 720KB 04H = 3.5”, 1.44MB
DH	Max. number of sides (always 1 for 2 sides)	
CH	Max. number of tracks	
CL	Max. number of sectors	
ES:DI	Pointer to DDPT (Disk Drive Parameter Table)	

* DDPT، جدولی است که پارامترهای موردنیاز BIOS برای دسترسی به کنترلر دیسک در آن قرار دارد.

تشخیص تعویض دیسکت: تابع 16H تعویض دیسکت را پس از دستیابی قبلی کنترل می‌کند و تشخیص می‌دهد. پس از خواندن اطلاعات از دیسکت و قبل از نوشتن بر روی آن، باید کنترل شود که دیسکت تعویض نشده باشد. در غیر این صورت ممکن است توابع سطح بالاتر، مثل توابع DOS، اطلاعات دیسکت را خراب کنند. مثلاً با توجه به FAT دیسکت قبلی، اطلاعاتی را بر روی دیسکت تعویض شده بنویسند. این تابع مقادیر زیر را در ثبات AH برمی‌گرداند:

00H: Drive not present
01H: Disk drive does not recognize diskette change
02H: Disk drive recognize diskette change
03H: Hard drive!

خواندن بخشهای دیسکت: تابع 02H این کار را می‌کند و با یک بار فراخوانی آن می‌توان چند بخش را خواند. برای این منظور باید پارامترهای زیر فراهم شود:

AL: No of sectores to be read

DL: Drive specification value (0: A, 1:B, etc.)

DH: Side (0 or 1)

CL: Sector number (1...N)

CH: Track number (1..N-1)

ES:BX: Address of buffer for the data to be read

پس از خواندن، وضعیت خطا در AH برگردانده می‌شود و تعداد بخشهای خوانده شده در AL قرار می‌گیرد. اگر CF مقدار 1 داشته باشد، خطا رخ داده است.

نوشتن بخشهای دیسکت: تابع 03H برای این کار قابل استفاده است و با یک بار فراخوانی، می‌توان چند بخش را بر روی دیسکت نوشت. برای این منظور باید پارامترهای زیر فراهم شود:

AL: No of sectores to be written

DL: Drive specification value (0: A, 1:B, etc.)

DH: Side (0 or 1)

CL: Sector number (1...N)

CH: Track number (1..N-1)

ES:BX: Address of buffer for the data to be written

پس از نوشتن، وضعیت خطا در AH برگردانده می‌شود و تعداد بخشهای نوشته شده در AL قرار می‌گیرد. اگر CF مقدار 1 داشته باشد، خطا رخ داده است.

وارسی[۱] بخشهای دیسکت: تابع 04H انتقال درست داده‌ها به دیسکت را پس از نوشتن بررسی می‌کند. برای این منظور داده‌های موجود در بافر و نوشته شده بر روی دیسکت (بایت به بایت) با هم مقایسه نمی‌شوند، بلکه، کد CRC[۲] آنها با هم مقایسه می‌شود. CRC، یک تابع محاسبه‌ی مجموع مقابله‌ای[۳] است که پارامتر ورودی آن داده‌ها است و مقدار خروجی آن، یک کد است. برای مثال می‌توان همه‌ی داده‌های موجود در بافر را بایت به بایت باهم XOR نمود تا یک

۱ verifying
۲ Cyclic Redundancy Check
۳ checksum

بایت حاصل شود و همین کار را بر روی داده‌های نوشته شده بر روی دیسک انجام داد و سپس این دو بایت را با هم مقایسه نمود. محاسبه‌ی مجموع مقابله‌ای و مقایسه‌ی تنها دو بایت، از مقایسه‌ی همه‌ی داده‌ها به‌مراتب سریع‌تر است. CRC، نوعی مجموع مقابله‌ای است که حاصل آن بسیار مطمئن بوده و با احتمال بسیار پایین برای داده‌های متفاوت حاصل آن یکسان است و به کمک سخت‌افزار نیز قابل محاسبه است.

قالب‌بندی(فرمت‌کردن) شیارها: با فراخوانی تابع 05H انجام می‌شود و پارامترهای لازم به شرح زیر است:

AL: Number of sectors in the track

BL: Number of the drive

CH: Number of the track

ES:BX: Pointer to format table, containing the following information:

Offset	Information
0:	Track to be formatted
1:	Side (0 or 1)
2:	Number of sectors
3:	Number of bytes in sector:

0:	128 bytes
1:	256 bytes
2:	512 bytes
3:	1024 bytes

۶-۴ دسترسی به دیسک سخت از طریق BIOS

از همان توابع وقفه‌ی 13H برای دسترسی به دیسک سخت استفاده می‌شود. البته تعدادی از توابع این وقفه فقط با دیسک سخت قابل استفاده هستند. این توابع عبارتند از:

Function	Task
00H	Reset
01H	Read status
02H	Read
03H	Write
04H	Verify
05H	Format
08H	Check format
09H	Adapt to foreign drives (not defined in BIOS)
0AH	Extended Read
0BH:	Extended Write
0CH:	Move read/write head
0DH:	Reset
10H:	Drive ready
11H:	Recalibrate drive
14H	Controller diagnostic
15H	Determine drive type

۶-۵ تقسیمات[۱] دیسک سخت

برای آماده‌کردن دیسک سخت، سه کار باید انجام شود:

- **قالب‌بندی سطح‌پایین[۲]:** به منظور سازماندهی گرداننده به سیلندرها، شیارها و بخشها، با نوشتن علامتهای نشانی[۳] مناسب در سطح گرداننده انجام می‌شود. این علامتهای نشانی بعداً بوسیله‌ی کنترلر برای دسترسی به دیسک سخت و تشخیص بخشها مورد استفاده قرار می‌گیرد. قالب‌بندی

۱ patritions
۲ low-level format
۳ address marker

سطح پایین همچنین برای خارج کردن بخشهایی از دیسک که به‌طور فیزیکی خراب شده‌اند از فهرست بخشهای قابل استفاده‌ی دیسک مورد استفاده قرار می‌گیرد. این نوع قالب‌بندی برای کنترلرهای SCSI مورد نیاز نبوده و فقط برای کنترلرهای IDE استفاده می‌شود. برنامه‌ی قالب‌بندی عموماً در برنامه‌ی تنظیم[۱] BIOS یا در برخی از برنامه‌های کمکی وجود دارد.

- **تقسیم‌بندی:** یک دیسک سخت به قسمتهایی، تقسیم می‌شود. این کار به‌کمک برنامه‌ی DOS FDISK انجام می‌شود و به دو دلیل مورد نیاز است. نخست، برای تقسیم یک دیسک بزرگ به گردانندههای منطقی، تا بتوان از دیسکهای بزرگتر از 2Giga Byte تحت سیستم عامل DOS استفاده نمود. زیرا DOS نمی‌تواند به گردانندههای بیش از 2GB دسترسی داشته باشد(به دلیل محدودیت FAT). دوم، نگهداری سیستمهای عامل مختلف بر روی یک دیسک سخت. مثلاً، نگهداری همزمان سه سیستم عامل DOS، NT و Linux.. به این ترتیب می‌توان هر کدام از قسمت‌ها را فعال نمود تا آن سیستم عامل اجرا شود.
- **قالب‌بندی دیسک:** این کار همان قالب‌بندی معمولی است که با استفاده از برنامه DOSFormat انجام می‌شود.

وقتی با برنامه‌ی DOS FDISK یک دیسک را تقسیم‌بندی می‌کنیم، دیسک به دو بخش فیزیکی تقسیم می‌شود:

1. PRI DOS: Primary DOS partition (drive C:)
2. EXT DOS: Extended DOS partition

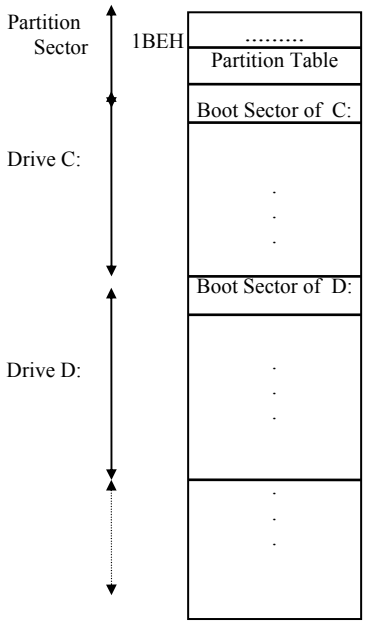
قسمت دوم (extended)، قابل تقسیم‌بندی به تعدادی گردانندهی منطقی است که می‌تواند از D: تا Z: باشد.

بخش نخست هر دیسک سخت، بخش تقسیم‌بندی[۲] است و محل نگهداری اطلاعات نشانی شروع، ختم، نوع و غیره در مورد تقسیمات دیسک است، که به آن جدول تقسیمات[۳] گفته می‌شود. این بخش در هنگام روشن شدن کامپیوتر در حافظه بارمی‌شود و در نشانی 0000:7C00 قرار می‌گیرد(در صورتیکه دیسکتی در گردانندهی A: قرار نداشته باشد).

اگر BIOS پس از بارکردن بخش تقسیم‌بندی در دو بایت آخر آن مقادیر 55H و AAH را ببیند، به این معنی خواهد بود که آن بخش قابل‌اجرا یا بوت‌کننده است و از بایت اول، آنرا اجرا می‌کند. در غیر این‌صورت پیغام قرار دادن دیسکت بوت‌کننده به کاربر داده می‌شود.

در مورد دیسکت‌ها، بخش 0 حاوی برنامه‌ی بارکننده یا بوت‌کننده‌ی[۴] سیتسم عامل است. درصورتی‌که، یک دیسکت در گردانندهی A: قرار داشته باشد، بخش 0 آن خوانده شده و در نشانی فوق در حافظه قرار می‌گیرد و آنگاه در صورتی‌که اجرایی باشد، اجرا شده و در غیر این‌صورت، پیغام تعویض دیسکت داده می‌شود.

برنامه‌ای که در بایت اول بخش تقسیم‌بندی وجود دارد، قسمت فعال که سیستم عامل باید از آن بارشود را پیدا نموده آنرا اجرا می‌کند. در هر زمان، یکی از قسمت‌های مختلف دیسک‌سخت می‌تواند فعال باشد. برنامه‌ی بوت‌کننده، از بخش بوت (نخستین بخش) آن قسمت، اجرا می‌شود. در صورتی‌که، بخش 0 قسمت فعال، اجرایی یا بوت‌شدنی[۵] نباشد، پیغام قرار دادن دیسکت در گرداننده داده می‌شود.



۱ setup
۲ partition sector
۳ partition table
٤ boot strap
٥ bootable

ساختار هر ردیف جدول تقسیمات به صورت زیر است:

Address	Contents	Type
+00H	Partition Status: 00H = Inactive 01H = Active (Boot-Partition)	1 Byte
+01H	Read/write head, with which the partition begins	1Byte
+02H	Sector and cylender, with which the partition begins	1 Word
+04H	Partition type: 00H = Entry not allocated 01H = DOS with 12-bit-FAT (primary partition) 02H = XENIX 03H = XENIX 04H = DOS with 16-bit-FAT (primary partition) 05H = Extended DOS partition 06H = DOS-4.0 partition with more than 32MB DBH = Concurrent DOS **H = Other codes possible in combination with other operations systems or special driver software.	1 Byte
+05H	Read/write head, with which the partition ends	1Byte
+06H	Sector and cylender, with which the partition ends	1 Word
+08H	Removal of first sector of the partition (boot-sector) of partition sector in sectors.	1 DWord
+0CH	Number of sectors in this partition	1 DWord

** مقادیر دیگری که ممکن است بوسیله‌ی سایر سیستمهای عامل، نظیر Linux یا NetWare برای فیلد نوع قسمت قرار دهند، بوسیله‌ی برنامه‌ی DOS FDisk تحت عنوان Non-DOS نمایش داده می‌شود.

بر روی یک PC می‌توان بیش از یک دیسک سخت نیز قرار داد. در این صورت باید در برنامه‌ی FDisk، شماره‌ی دیسک جاری یا "Current Fixed-Disk" را انتخاب کرد. در این صورت، گردانندهٔ C:، نخستین قسمت اولین دیسک و D: نخستین قسمت دیسک دوم خواهد بود. اگر هرکدام قسمتهای دیگری نیز داشته باشند، سایر قسمتهای دیسک دوم یا گردانندهای منطقی آن، از E: شروع می‌شوند و مثلاً تا G: ادامه می‌یابند. آنگاه، قسمت‌های دیگر یا گردانندهای منطقی دیسک دوم از H: شروع خواهد شد.

فصل ۷: درگاه موازی
(The Parallel Port)

۷-۱ مقدمه

وسایلی نظیر چاپگر، از طریق درگاه موازی به PC وصل می‌شوند. سه روش برای دسترسی به درگاه موازی وجود دارد:

۱) برنامه‌ریزی مستقیم سخت‌افزار درگاه موازی،

۲) از طریق فراخوانی توابع ROM-BIOS،

۳) از طریق فراخوانی توابع DOS.

اگر بخواهیم چاپگر را بهتر مدیریت کنیم، نمی‌توانی به توابعDOS بسنده کنیم. چون این توابع در هنگاه چاپ با برخورد به اولین خطا، مثلاً آماده نبودن کاغذ، خطای بحرانی[۱] داده و اغلب اوقات چاپ قطع می‌شود و این مسأله مشکلات زیادی را برای برنامه‌های کاربردی ایجاد می‌کنند. از اینرو، در این فصل با روشهای دسترسی به درگاه موازی از طریق BIOS و برنامه‌ریزی مستقیم سخت‌افزار درگاه موازی آشنا می‌شوید.

۷-۲ دسترسی به درگاه موازی از طریق BIOS

وقفه‌ی 17H، برای ارتباط با درگاه موازی اختصاص داده شده است و عموماً به این وقفه، وقفه‌ی چاپ[۲] گفته می‌شود. ولی هر وسیله‌ی دیگری را نیز می‌توان به این درگاه متصل نمود و مختص چاپگر نیست.

تحت سیستم عامل DOS، به حداکثر سه تا از چهار درگاه موازی ممکن در PC می‌توان دسترسی داشت که تحت نامهای منطقی :LPT1 یا PRN، :LPT2 و :LPT3 قابل دسترسی هستند. توابع DOS. به‌طور پیش‌فرض با :LPT1 کار می‌کنند. اما در وقفه‌ی 17H، سه تابع برای کار با درگاه موازی وجود دارد که با هر سه درگاه کار می‌کنند. برای تعیین شماره‌ی درگاه باید متناظر با :LPT1، مقدار 0، متناظر با :LPT2، مقدار 1 و متناظر با LPT3، مقدار ۲ را در ثبات DX قرار داد:

Function	Task
00H	Display (write or send) character
01H	Initialize printer
02H	Request printer status (read or receive character)

وضعیت چاپگر: هرکدام از توابع فوق، پس از اجرا وضعیت چاپگر را در ثبات AH برمی‌گرداند. بیتهای مختلف بایت وضعیت چاپگر، نشاندهنده‌ی وضعیت‌های زیر است:

Bit 0: Time out error
Bit 1: Unused
Bit 2: Unused
Bit 3: Transfer error
Bit 4: Printer online
Bit 5: Printer out of paper
Bit 6: Receive mode selected
Bit 7: Printer busy

سرعت درگاه موازی ۱۰۰۰۰۰ کاراکتر در ثانیه است. اگر در یک دوره‌ی زمانی کمتر از ۱/۱۰۰۰۰۰ ثانیه داده به درگاه ارسال شود، بیت ۷ مساوی ۱ می‌شود و چاپگر مشغول دیده می‌شود و به این دلیل خطای "Time out error" اتفاق افتاده و بیت ۰ مساوی ۱ می‌شود.

---

۱ critical error
۲ print interrupt

تابع چاپ یا ارسال داده‌ها: تابع 00H برای این منظور مورد استفاده قرار می‌گیرد. شماره‌ی تابع(00H) باید در ثبات AH و کد اَسکی کاراکتر در ثبات AL قرار گیرد. پس از فراخوانی، ثبات AH حاوی بایت وضعیت چاپگر خواهد بود.

مقداردهی اولیه‌ی درگاه: تابع 00H باید همواره قبل از شروع ارسال داده‌ها فراخوانی شود و درگاه یا چاپگر را مقداردهی اولیه نماید. پس از فراخوانی، ثبات AH حاوی بایت وضعیت چاپگر خواهد بود.

تابع درخواست بررسی وضعیت چاپگر: تابع 03H برای این منظور استفاده می‌شود و بایت وضعیت را در AH برمی‌گرداند. این بایت را در برنامه‌های C یا پاسکال برای یافتن نوع خطا قابل استفاده است.

فراخوانی توابع BIOS در زبان C: کامپایلرهای Borland C++ و QuickC، به ترتیب توابع biosprint و bios_print را برای چاپ کاراکتر با استفاده از توابع BIOS فراهم کرده‌اند.

نوشتن روال جدید برای مدیریت وقفه‌ی چاپ: برای آنکه خطاهای چاپ در همه‌ی برنامه‌هایی اجرایی تحت DOS کنترل شود، به دلیل آنکه توابع DOS از توابع BIOS استفاده می‌کنند، می‌توان روال مدیریت وقفه‌ی 17H را تغییر داد و قبل از چاپ هر کاراکتر، وضعیت چاپگر را کنترل نمود و در صورت آماده نبودن چاپگر، پیغام مناسب بر روی صفحه چاپ کرد.

ارسال دادها از طریق درگاه موازی: گرچه این درگاه برای ارسال داده‌ها از PC به لوازم خروجی طراحی شده است، ولی از تابع 02H می‌توان برای دریافت دادههانیز استفاده نمود. ولی سرعت بسیار پائین بوده و این درگاه درحقیقیت مسر یکطرفه‌ای برای ارسال داده‌ها است، نه دریافت آن.

۷-۳ دسترسی مستقیم به درگاه موازی

نشانیهای درگاههای موازی نصب شده در PC، در هنگام روشن شدن کامپیوتر توسط BIOS پیدا شده و در قطعه‌ی متغیرهای BIOS قرار داده می‌شود. از طریق این نشانیها، با دستورات دسترسی به درگاهها، می‌توان به درگاههای موازی دسترسی داشت:

Port	Interface
3BCH-3BFH	Parallel interface on MDA card
378H-37FH	Parallel interface #1
278H-37FH	Parallel interface #2

* کارتهای ویدیویی قدیمی MDA، دارای یک درگاه موازی نیز بودند، که نشانی این گونه درگاهها، 3BCH-3BFH بود.

* نشانیهای فوق به‌ترتیب چک می‌شوند. اگر در 3BCH-3BFH یک درگاه یافته شود، به عنوان درگاه نخست به :LPT1 منسوب می‌شود و در فضاهای بعدی هم اگر پیدا شده به :LPT2 و :LPT3.

* همانطوری که گفته شد، نشانیهای بدست آمده در قطعه‌ی متغیرهای BIOS ذخیره می‌شوند:

Address	Contents
0040:0008H	Base address of LPT1
0040:000AH	Base address of LPT2
0040:000CH	Base address of LPT3
0040:000EH	Base address of LPT4

برای دسترسی به هر درگاه باید ابتدا نشانی آن از مطابق جدول فوق نشانیهای مشخص شده بدست آید. مثلاً برای بدست آوردن نشانی :LPT1 در زبان پاسکال به صورت زیر عمل می‌شود:

```
LPT1Addr := Mem[$0040:$0008];
Port[Lpt1Addr] := XByte;          // ارسال یک بایت
```

تغییر نشانی درگاهها: می‌توان با تعویض مقادیر ردیفهای جدول فوق، نشانی درگاهها را عوض کرد. برای مثال قطعه کد زیر در پاسکال، نشانی :LPT1 و :LPT2 را تعویض می‌کند:

```
DummyWord := Mem[$0040:$0008];
Mem[$0040:$0008] := Mem[$0040:$000A];
Mem[$0040:$000A] := DummyWord;
```

ثباتهای درگاه موازی: هر درگاه سه ثبات دارد:

- **ثبات ۱ (Data Lines):** حاوی بیتهای ۰ تا ۷ داده‌ی ارسالی است و داده برای ارسال در آن قرار می‌گیرد و در نشانی 378H قابل دسترسی است.

- **ثبات ۲ (Printer Status):** حاوی بایت وضعیت چاپگر است و نشانی آن، 379H است.
- **ثبات ۳ (Printer Control):** سیگنالهای کنترلی چاپگر (نظیر ACK، INT و ...) را نشان می‌دهد و نشانی آن 37AH است.

۸-۵ دسترسی به درگاه سریال از طریق BIOS

علاوه بر روش برنامه‌نویسی مستقیم، توابع وقفه‌ی BIOS 14H برای برنامه‌نویسی درگاه سریال قابل استفاده است. توابع این وقفه‌عبارتند از:

Function 0: Passing protocol

با این تابع، پارامترهای ارتباطی پروتکل تعیین می‌شوند و بایت حاوی پارامترهای ارتباطی را در ثبات AL قرارداده و این تابع را فراخوانی می‌کنیم:

Bits 0-1: Data length:
10 = 7 bits
11 = 8 bits
Bit 2: Number of stop bits:
0 = 1 stop bit
1 = 2 stop bits
Bits 3-4: Parity check:
00 = None
01 = Odd
10 = Even
Bits 5-7: Baud rate:
000 = 110 baud
001 = 150 baud
010 = 300 baud
011 = 600 baud
100 = 1200 baud
101 = 2400 baud
110 = 4800 baud
111 = 9600 baud

Function 1: Transmit character

با این تابع می‌توان یک کاراکتر را ارسال نمود. کد کاراکتر باید در AL و مقدار ۱ در AH قرار داده شود. پس از فراخوانی تابع اگر مقدار 0 در AH باشد، داده به‌درستی ارسال شده است و اگر خطایی رخ داده باشد، مقدار 1 در AH باقی می‌ماند.

Function 2: Receive character

مقدار کد کاراکتر در ثبات AL دریافت می‌شود. اگر AH=0 باشد، خطایی رخ نداده است، و در غیر این‌صورت، AH حاوی کد نوع خطا است.

Function 3: Line/Modem Status

اطلاعات ثبات Line Control یا Line Status را در AH و Modem Status را در AL بر می‌گرداند. Line Status، شامل اطلاعات پارامترهای ارتباطی (نظیر طول کلمه،بیت توقف و غیره) بوده و Modem Status شامل اطلاعات زیر است:

Bit 0: (Delta) Modem ready to send
Bit 1: (Delta) Modem is on
Bit 2: (Delta) Telephone is ringing
Bit 3: (Delta) Connection to receiver modem
Bit 4: Modem ready to send
Bit 5: Modem is on
Bit 6: Telephone is ringing
Bit 7: Connection to receiver modem

بیتهای ۴ تا ۷، تکرار بیتهای ۰ تا ۳ هستند و برای مشخص کردن تغییر از آخرین بار قرائت وضعیت به‌کار می‌روند. اگر تغییری در وضعیت رخ داده باشد، بیتهای متناظر ۱ می‌شوند. برای مثال اگر بیت شماره ۲ مقدار ۱ داشته باشد، با این معنی است که مقدار بیت ۶ از آخرین قرائت قبلی، تغییر کرده است.

فصل ۹: برنامه‌نویسی ماوس
(Mouse Programming)

۹-۱ مقدمه

توابع معرفی شده بوسیله‌ی میکروسافت برای ماوس، به عنوان استانداردی برای دسترسی به ماوس پذیرفته شده است. این توابع که تشکیل دهنده‌ی واسط نرم‌افزاری ماوس یا درایور ماوس[۱] هستند، در قالب وقفه‌ی 33H در اختیار قرار می‌گیرند که بوسیله‌ی نصب برنامه‌ی MOUSE.DRV در فایل CONFIG.SYS یا اجرای برنامه‌ی مقیم در حافظه‌ی[۲] MOUSE.COM تحت سیستم عامل DOS، فراهم می‌شوند.

۹-۲ نمایش مکان‌نمای ماوس

در برنامه‌هایی که ماوس را پشتیبانی می‌کنند، در صورت نصب‌بودن واسط نرم‌افزاری ماوس، مکان‌نمای ماوس[۳] را نمایش می‌دهند. مکان‌نمای ماوس، در دو حالت متن و گرافیک قابل نمایش است، که در حالت گرافیک، وابسته به کارت ویدئویی و وضوح تصویر است. شکل مکان نما در حالت متن، با دو مقدار 16بیتی مشخص می‌شود که Screen Mask و Cursor Mask نامیده می‌شوند، که در حافظه‌ی ویدئویی قرار دارند. برای نمایش مکان نما در هر نقطه از صفحه نمایش، شکل مکان نما با AND کردن مقدار دو بایت S.M. با مقدار دو بایت مشخص‌کننده‌ی کاراکتر و مشخصه در آن نقطه، و سپس XOR کردن مقدار حاصله با C.M. و نوشتن نتیجه‌ی نهایی در همان نقطه بدست می‌آید. این روش، انتخابهای مختلفی را برای نمایش مکان‌نمای ماوس در اختیار قرار می‌دهد که به‌طرح زیر است:

- یک کاراکتر خاص با رنگ خاص، مثلاً همیشه با کاراکتر ↑، رنگ سفید در زمینه‌ی آبی،
- یک کاراکتر خاص با رنگ کاراکتر موجود در نقطه‌ی جاری،
- تغییر رنگ کاراکتر موجود در نقطه‌ی جاری به صورت سخت‌افزاری و با استفاده از حالت معکوس[۴] کارت ویدئویی، به این صورت که به کارت ویدئویی فرمان داد تا رنگ بیتهای صفر رنگ را ۱ و بیتهای ۱ را صفر کند. یعنی، بایت رنگ را با FFH، XOR کند.
- تغییر رنگ کاراکتر موجود در نقطه‌ی جاری به صورت نرم‌افزاری. برای این منظور باید مقدار بایت رنگ در نقطه ی جاری خوانده شده و پس از XOR کردن در همان نقطه نوشته شود. این کار بوسیله‌ی درایور ماوس انجام می‌شود.

قطعه کدهای زیر برای موارد ۲ و ۴ قابل استفاده هستند:

```
Procedure ShowCursor2(row, col : Byte);
var
   ChAtt, CursorShape : Word;
begin
   ChAtt := Mem[$B800:(row * 160 + col * 2)];
   CursorShape := CharAtt AND $FF00 ; {FF00H is the S.M. }
   CursorShape := CursorShape XOR $0018H;  { 0018H is the C.M. and 18H is ASCII code of '↑' }
   Mem[$B800:(row * 160 + col * 2)] := CursorShape; { mouse cursor = '↑': yellow over black }
end;


Procedure ShowCursor4(row, col : Byte);
var
   ChAtt, CursorShape : Word;
begin
   ChAtt := Mem[$B800:(row * 160 + col * 2)]; { 'A': yellow over black }
   CursorShape := CharAtt AND $FFFF ; {FFFFH is the S.M. }
```

---

۱ mouse driver
۲ TSR
۳ mouse cursor
٤ inverse mode

```
    CursorShape := CursorShape XOR $FF00H;  { FF00H: to inverse the attribute }
    Mem[$B800:(row * 160 + col * 2)] := CursorShape; { mouse cursor = 'A': white over blue }
end;
```

۹-۳ واحد اندازه‌گیری دقت ماوس

از واحد میکی[۱] برای اندازه‌گیری دقت ماوس استفاده می‌شود و نمایش دهنده‌ی تعداد نقطه‌ها در هر اینچ است. ماوسهای قدیمی دارای دقت ۲۰۰ نقطه در اینچ یا میکی بودند، اما ماوسهای جدید تا دقت ۴۰۰ میکی را پشتیبانی می‌کنند. البته این مسأله به نوع درایور ماوس وابسته است.

۹-۴ توابع وقفه‌ی ماوس

وقفه‌ی 33H دارای توابع اصلی زیر است:

Function	**Task**
00H:	Reset mouse drivers
01H:	Display mouse cursor
02H:	Remove(hide) mouse cursor
03H:	Move mouse cursor
07H & 08H:	Set range of movement (محدوده‌ی حرکت ماوس در صفحه)
10H:	Exclusion area (محدوده‌ی عدم نمایش مکان نمای ماوس)
1DH:	Set display page
0FH:	Set cursor speed
0AH:	Set cursor shape: passing S.M. & C.M. and selecting the method for updating the mouse cursor: - Software Specific or Hardware Specific - Polling or Interrupt
03H:	Get cursor position/button status
Function 0CH:	Set event handler

اینکه با توجه به حرکت دادن ماوس و زدن دکمه‌ها، چه کاری انجام شود را می‌توان در یک ISR[۲] قرار داد و به درایور ارسال نمود.

Function 18H:	Install alternate event handler

ترکیب رخدادهای ماوس(نظیر حرکت کردن یا زدن دکمه‌ها) با برخی از کلیدهای کنترلی ماوس را می‌توان با این تابع و با برنامه‌نویسی در ISR ارسالی به درایور ماوس، پردازش نمود. مثلاً با گرفتن کلید شیفت و حرکت دادن مکان‌نمای ماوس در یک ادیتور، متن انتخاب می‌شود و غیره.

برنامه‌نویسی مستقیم ماوس: باید از طریق درگاه سریال این کار را کرد. چون ماوس به یکی از درگاههای سریال، نظیر COM1 یا COM2 متصل می‌شود. کسانی که درایور ماوس را می‌نویسند، نیازمند این کار هستند.

---

۱ Mickey، پس از کارتون Mickey Mouse والت دیسنی!
۲ Interrupt Service Routine

فصل ۱۱: دسترسی و برنامه‌نویسی ساعت
(Accessing and Programming the Real-time Clock)

۱۱-۱ اندازه‌گیری زمان با وقفه‌ی 08H

تراشه‌ی تایمر(Intel 8254)، در هر ثانیه 1,193,180 بار از مولد ارتعاش[۱]، سیگنال دریافت می‌کند. پس از ۶۵۵۳۶ سیگنال، یا تقریباً 18.2 بار در ثانیه، این تراشه وقفه‌ی سخت‌افزاری 08H را فعال می‌کند. کنترلر وقفه[۲]، این وقفه را به CPU منتقل می‌کند و روال سرویس این وقفه، اجرا می‌شود. این روال وقفه، یک شمارشگر برای زمان (time counter) را که جزو متغیرهای BIOS است را افزایش می‌دهد. این شمارشگر، یک متغیر ۳۲ بیتی است و در ابتدای روشن شدن سیستم دارای مقدار اولیه‌ی صفر است، مگر آنکه به‌طور دستی یا بوسیله‌ی ساعت بی‌درنگ(RTC) مقداردهی شود.

از این شمارشگر زمان، می‌توان ثانیه‌ها، و از ثانیه‌ها، دقیقه و ساعت را محاسبه کرد:

seconds = time_counter /18.2

کار دیگر وقفه‌ی 08H، خاموش کردن موتور گرداننده‌ی دیسک است، که این کار پس از یک مدت زمان غیرفعال بودن گرداننده انجام می‌شود و در ارتباط با وقفه‌ی 13H، این کار انجام می‌شود. پس از انجام این کار، وقفه‌ی 08H، وقفه‌ی 1CH را فراخوانی می‌کند. در حالت عادی، روال سرویس وقفه‌ی 1CH، فقط شامل یک دستورالعمل iret است، ولی می‌توان این وقفه را تعویض نمود تا کارهای دیگری انجام دهد.

۱۱-۲ تنظیم زمان با وقفه‌ی 1AH

این وقفه، توابع خواندن و تنظیم تاریخ و ساعت را در اختیار قرار می‌دهد:

Function	Task
00H	Get clock (32-bit time_counter)
01H	Set clock
02H	Get current time (HH:MM:SS.PP)
03H	Set current time (HH:MM:SS.PP)
04H	Get current date (CCYY/MM/DD)
05H	Set current date (CCYY/MM/DD)
06H	Set alarm time*
07H	Reset alarm time

* alarm time، زمانی است در روز جاری که در آن زمان وقفه‌ی 4AH فعال می‌شود و می‌توان با تعویض روال سرویس این وقفه، پیغام خاصی را بر روی صفحه نمایش داد، و یا با زدن زنگ، مطلبی را به کاربر یادآوری نمود.

۱۱-۳ ساعت بی‌درنگ

ساعت بی‌درنگ(RTC) در کامپیوترهای AT به بعد وجود دارد و با باتری کار می‌کند و با خاموش شدن کامپیوتر، reset نمی‌شود. این ساعت یک حافظه از نوع RAM دارد که 64 بایت ظرفیت دارد و از طریق درگاه 70H تا 7FH قابل دسترسی است. این حافظه، در حقیقت ثباتهای RTC است، که عبارتند از:

Register	Meaning
00H	Current second
01H	Alarm second
02H	Current minute

---

۱ quartz crystal
۲ interrupt controller

03H	Alarm minute
04H	Current hour
05H	Alarm hour
06H	Day of week (1=Sunday, ...)
07H	Number of day
08H	Month
09H	Year
0AH	Clock status register A
0BH	Clock status register B
0CH	Clock status register C
0DH	Clock status register D
32H	Century (19 or 20)

در ابتدای روشن شدن سیستم، مقدار شمارشگر زمان، با استفاده از مقادیر زمان RTC، مقدار دهی اولیه می‌شود.

روش دسترسی به این ثباتها، همانند دسترسی به حافظه است:

برای خواندن ثباتها:

```
mov  al, mem_loc  ; خواندن یک ثبات با شماره مشخص
out  70h, al      ; درگاه نوشتن
in  al, 71h       ; درگاه خواندن
```

برای نوشتن در ثباتها:

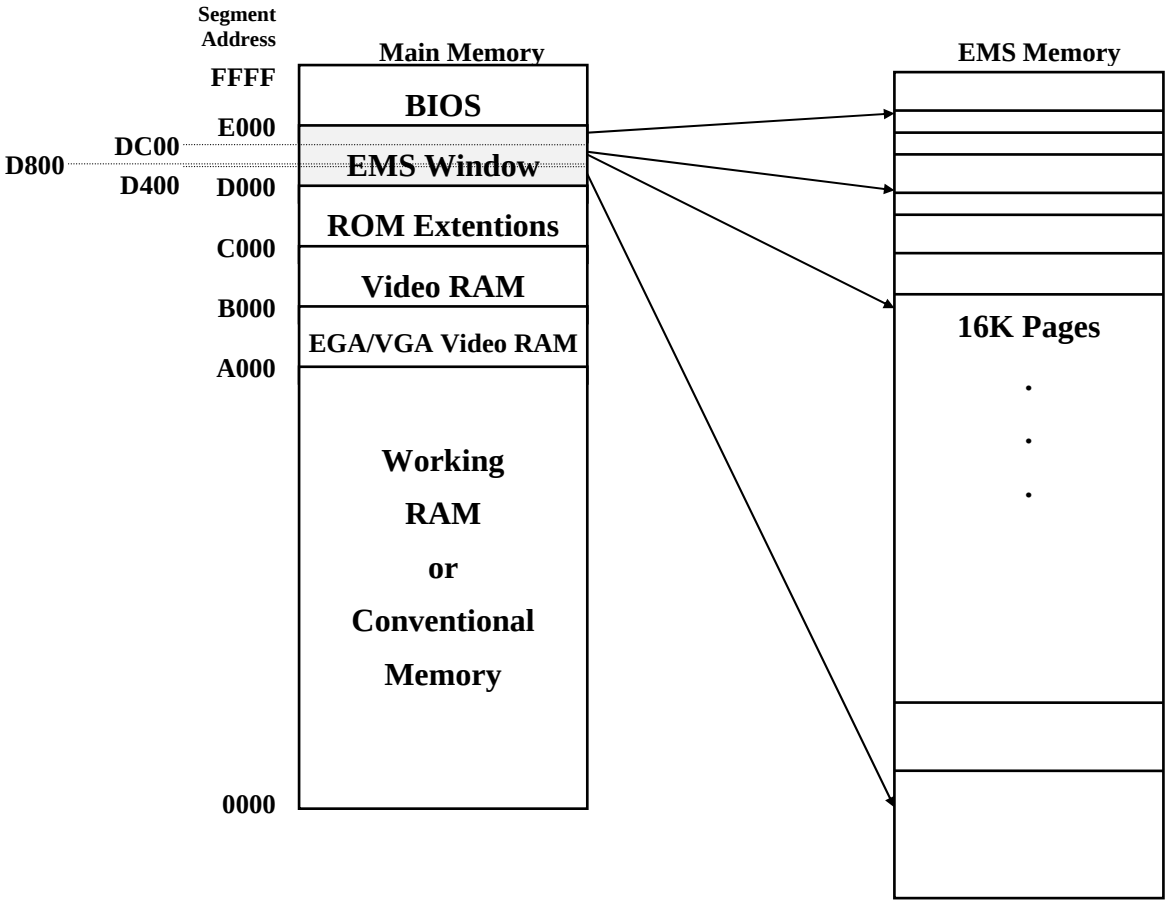
```
mov  al, mem_loc
out  70h, al
mov  al, new_content
out  71h, al
```

فصل ۱۲: گسترش حافظه
(Memory Expansion)

۱۲-۱ مقدمه

در سال ۱۹۸۰ که نخستین PC عرضه شد، 640KB حافظه، خیلی زیاد بود و PCهای اولیه 64KB یا 128KB یا حداکثر 256KB حافظه داشتند. DOS هم برای چنین نیازهایی طراحی شده بود و حداکثر 640KB حافظه را پشتیبانی می‌کند که در حال حاضر حداقل حافظه‌ی موردنیاز بوسیله‌ی اکثر کاربران است. در شکل (۱۲-۱) پیکره‌بندی حافظه‌ی PC نشان داده شده است.

شکل ۱۲-۱: پیکره‌بندی حافظه PC و بسط حافظه مطابق استاندارد LIM



مطابق این شکل فقط 640KB از حافظه به‌عنوان حافظه‌ی اصلی می‌تواند مورد استفاده‌ی برنامه‌ها قرار گیرد. اما برخلاف PC که دارای پردازنده‌ی 8086 بود، کامپیوترهای شخصی بعدی، دارای پردازنده‌های 80286 یا بالاتر، امکان دسترسی به بیش از 1MB حافظه را فراهم می‌کردند. اما تحت سیستم عامل DOS، مابقی حافظه قابل نشانی‌دهی نیست. دلیل این امر این است که DOS در حالت واقعی[۱] این پردازنده‌ها کار می‌کند و پردازنده در این حالت، که سازگار با

---

[۱] real mode

روش نشانی‌دهی 8086 است نمی‌تواند بیش از 1MB نشانی‌دهی کند. قابلیت نشانی‌دهی بیش از 1MB در حالت محافظت‌شده[۱] فراهم می‌شود و سیستمهای عاملی نظیر Windows یا Unix که توانایی دسترسی به بیش از 1MB حافظه را دارند در این حالت پردازنده عمل می‌کنند.

تحت سیستم عامل DOS، راه حلهای مختلفی برای حل مشکل کمبود حافظه ارائه گردید که دو راه حل مهم عبارتند از:

۱) حافظه‌ی گسترش یافته[۲]
۲) حافظه‌ی بسط‌یافته[۳]

این روشها میزان حافظه‌ی قابل استفاده بوسیله‌ی برنامه‌ها را تا چند مگابایت افزایش می‌دهند و دسترسی به‌این نوع حافظه‌ها، مستلزم پیش‌بینی ترتیبات لازم در برنامه‌ها است. تفاوت این دو روش به‌طور خلاصه عبارت است از:

- **حافظه‌ی گسترش یافته:** حافظه‌ی اضافی در کامپیوترهای PC و XT است و به دلیل محدودیت ۸۰۸۶ و ۸۰۸۸ برای کارکردن با بیش از 640KB حافظه، با این روش میزان حافظه با تکنیکی که مفصلاً مورد بحث واقع می‌شود، به بیش از 1MB گسترش داده می‌شود.
- **حافظه‌ی بسط‌یافته:** حافظه‌ی اضافی در کامپیوترهای دارای پردازنده‌های 80286 به بالا است و میزان حافظه با تکنیکی که مستلزم عمل نمودن در حالت محافظت‌شده‌ی پردازنده است، به بیش از 1MB بسط می‌یابد.

۱۲-۲ حافظه‌ی گسترش یافته

حافظه‌ی متعارف[۴] در کامپیوترهای PC/XT به 640KB محدود بود. شرکتهایی برای غلبه بر مشکل محدودیت حافظه، اقدام به ارائه‌ی راه‌حلهایی برای افزودن حافظه‌ی قابل استفاده تحت سیستم عامل DOS نمودند تا برنامه‌ها در کامپیوترهای PC و XT بتوانند از آن استفاده نمایند. سه شرکت معروف در این زمینه عبارت بودند از Lotus، Intel و Microsoft، که استانداردی را با نام LIM، که برگرفته از حروف نخست نامهای آنها است، معرفی نمودند. این استاندارد اجازه می‌دهد که تا 8MB حافظه با بهره‌گیری از یک سخت‌افزار کارت گسترش[۵] به حافظه‌ی PC اضافه شود. ماجولها با بانکهای حافظه‌ی اضافی، که هر کدام 16KB ظرفیت داشتند، بر روی این کارت قرار داشتند و در هر زمان تنها چهار بانک یا 64KB از این حافظه‌ی اضافی در فضای نشانی‌دهی PC زیر 1MB و در محدوده‌ی D000H تا E000H و تحت سیستم عامل DOS قابل دستیابی است و به آن یک قاب صفحه[۶] گفته می‌شود. این نوع حافظه یا حافظه‌ی بسط‌یافته که در کامپیوترهای AT به بالا و بالاتر از فضای نشانی‌دهی 1MB وجود دارد، متفاوت است. به سیستمی که دارای حافظه‌ی گسترش یافته باشد، EMS[۷] گفته می‌شود.

استاندارد LIM از یک تکنیک مطمئن به نام **تعویض بانک**[۸] برای دسترسی به حافظه‌ی اضافی استفاده می‌کند. در این روش از یک پنجره‌ی حافظه در محدوده‌ی D00H تا E000H، به‌نام EMS Window برای دسترسی به بانکهای 16MB حافظه‌ی گسترش‌یافته استفاده می‌شود. نرم‌افزار به‌طور تنگاتنگ با سخت‌افزار کار می‌کند و پنجره را در کل محدوده‌ی حافظه حرکت می‌دهد تا در هر زمان به ۴ صفحه یا بانک آن دسترسی داشته باشد، که ظرفیت هر کدام 16MB است و در مجموع ۴ صفحه در نشانیهای D000H، D400H، D800H و DC00H که در شکل (۱۲-۱) مشخص شده‌اند، قابل دسترسی در برنامه‌ها بوده و مابقی حافظه‌ی گسترش یافته غیرقابل دسترسی در آن زمان است.

باز کردن یک پنجره حافظه: تحت DOS حداقل 64KB در فضای نشانی‌دهی 1MB، علی‌رغم وجود حافظه‌ی متعارف، BIOS، Video RAM، یا سایر موارد، بدون استفاده می‌ماند. طراحان EMS از این محدوده برای دسترسی به حافظه‌ی گسترش‌یافته استفاده کرده‌اند. عموماً این پنجره در نشانی قطعه‌ی D000H قرار می‌گیرد، اما سخت‌افزار EMS اجازه می‌دهد که در جای دیگری هم باشد. چون این پنجره در فضای نشانی‌دهی 1MB قرار دارد پس بوسیله‌ی دستورالعملهای عادی زبان اسمبلی، همانند حافظه‌ی ویدئویی قابل دسترسی(خواندنی / نوشتنی) است.

تقسیم‌بندی قاب صفحه: قاب صفحه به ۴ صفحه‌ی 16KB تقسیم شده است و به برنامه‌نویس اجازه می‌دهد که در هر زمان به چهار صفحه‌ی متفاوت در حافظه‌ی گسترش‌یافته دسترسی داشته باشد. ثباتهای EMS به برنامه‌نویس اجازه می‌دهند که صفحات مورد نظرش را انتخاب نماید که براساس آن خطوط نشانی EMS تنظیم می‌شوند تا صفحات موردنظر قابل دسترسی باشند و بر این اساس آن صفحات در فضای نشانی‌دهی 8088 نگاشت[۹] می‌شوند، و این همان تکنینک تعویض بانک است.

---

۱ protected mode
۲ expanded memory
۳ extended memory
۴ conventional memory
۵ expansion card
۶ page frame
۷ Expanded Memory System
۸ bank switching
۹ map

EMS، علاوه بر سخت‌افزار دارای یک واسط نرم‌افزاری نیز هست، که وظایف برنامه‌ریزی ثباتهای EMS و سایر وظایف مدیریت حافظه را انجام می‌دهد، که به آن واسط مدیریت حافظه‌ی گسترش‌یافته[۱](EMM) گویند و واسط استانداردی برای دسترسی به کارتهای EMS ساخت شرکتهای مختلف فراهم می‌کند.

برخی از نرم‌افزارهای EMM، نظیر QEMM386 بر روی کامپیوترهای 386 به بعد، دسترسی به حافظه‌ی بسط‌یافته را با همین روش و همانند حافظه‌ی گسترش‌یافته و با مکانیسم EMS فراهم نموده‌اند.

برنامه‌های گرداننده‌ی EMM[۲]: این برنامه‌ها به‌صورت مقیم درحافظه[۳] (TSR) هستند و یکی از وقفه‌ها، مثل 67H را تعویض نموده و از این طریق امکان دسترسی به حافظه‌ی گسترش‌یافته را فراهم می‌کنند. توابع مختلفی برای این منظور پیش‌بینی شده است، که شماره‌ی تابع مثل توابع DOS در ثبات AH قرار داده می‌شود. پس از فراخوانی، AH حاوی وضعیت خطا خواهد بود و اگر مقدار آن مساوی 00H باشد، به معنی عدم وقوع خطا است و مقادیر مساوی یا بزرگتر 80H نشاندهنده‌ی خطا خواهند بود. برخی از توابع گرداننده‌ی EMS در اینجا تشریح می‌شوند:

40H: Get EMM status

این تابع برای بررسی وضعیت نصب بودن EMM فراخوانی می‌شود و اگر AH=00H باشد، به معنی آن است که EMM نصب شده است.

41H: Get segment address of page frame

در هنگام روشن شدن PC، با توجه به اینکه کدام محدوده از فضای بین 640KB و 1MB بلااستفاده است، EMS قطعه‌ای را به عنوان EMS Window یا قاب صفحه درنظر می‌گیرد، که عموماً D000H است. با این تابع می‌توان نشانی شروع آنرا بدست آورد.

42H: Get number of pages

برنامه می‌تواند تعدادی از صفحات EMS را به‌طور منطقی تخصیص دهد، که تعداد صفحات تخصیص داده شده را می‌توان با این تابع بدست آورد، تا امکان تخصیص اضافی بررسی شود.

43H: Allocate EMS pages

این تابع یک صفحه‌ی EMS را به‌صورت منطقی تخصیص داده و یک شماره به عنوان دسته[۴] برای ارجاعات بعدی به آن برمی‌گرداند.

44H: Set mapping

این تابع یکی از صفحات فیزیکی EMS را به یکی از صفحات منطقی در قاب صفحه نگاشت می‌کند. در این صورت، خواندن یا نوشتن در صفحه‌ی منطقی در محدوده‌ی قاب صفحه، به منزله‌ی انجام این کار برروی صفحه‌ی متناظر EMS است.

45H: Release EMS pages

یکی از صفحات از قبل تخصیص داده‌شده‌ی EMS را با دادن شماره‌ی دسته‌ی آن آزاد می‌کند.

۱۲-۳ حافظه‌ی بسط‌یافته

قبلاً اشاره شد که پردازنده‌های بعد از ۸۰۸۸، نظیر ۲۸۶، ۳۸۶، و غیره، توانایی کار در دو حالت واقعی و محافظت‌شده را دارند. در حالت محافظت شده، این پردازنده‌ها توانایی دسترسی به بیش از 1MB حافظه را دارند. در حالت واقعی، پردازنده توانایی دسترسی به بیش از 1MB حافظه را ندارد و با EMS،حافظه‌ی اضافی در همان فضای نشانی 1MB استفاده می‌شود. DOS چون در حالت واقعی کار می‌کند، تنها می‌تواند به 1MB حافظه دسترسی داشته باشد. اما اگر بخواهیم از کل حافظه‌ی موجود بر روی سیستم استفاده کنیم، باید پردازنده را به حالت محافظت‌شده ببریم. این کار با 1 کردن یک بیت در ثبات پرچم این پردازنده‌ها امکانپذیر است.

اگر در یک برنامه‌ی اسمبلی تحت DOS این کار را بکنیم، برنامه به حالت محافظت‌شده می‌رود، اما برنامه بلافاصله قفل می‌کند. زیرا مدیریت حافظه در حالت محافظت‌شده با حالت واقعی کاملاً متفاوت است. در حالت محافظت‌شده پردازنده از طریق مفهوم توصیف‌گرهای قطعه[۵] عمل می‌کند، نه مفهوم نشانی قطعه[۶]، که در آن هر نشانی از دو بخش مبدأ و قطعه تشکیل یافته است. در حالت محافظت‌شده،توصیف‌گر قطعه یک جدول است که لیست توصیف‌گرهای محلی و سراسری در آن قرار دارند. DOS به این جدول دسترسی ندارد، چون در حالت واقعی کار می‌کند.

---

۱ Extended Memory Management
۲ EMM drivers
۳ Terminate Stay Resident
٤ handle
٥ segment descriptor
٦ segment address

قبل از انتقال به حالت محافظت‌شده، باید جدول توصیف‌گر قطعه ایجاد و مقدار دهی شود. این کار نیازمند برنامه‌نویسی اسمبلی و آشنایی با عملکرد پردازنده در حالت محافظت‌شده است. واسطهای برنامه‌نویسی متعددی نیز وجود دارند که انجام این کار را تسهیل می‌کنند. نظیر واسط DPMI، که در فصل آخر درس به آن می‌پردازیم.

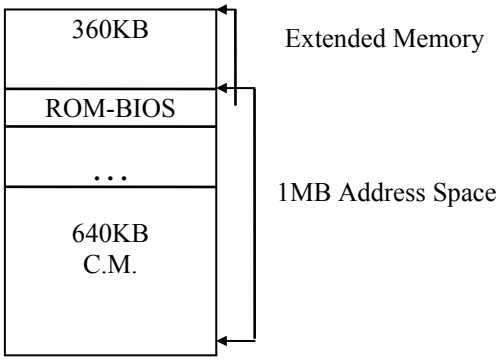
برخی توابع BIOS و گردانندههای XMS، دسترسی به حافظه‌ی بسط‌یافته را فراهم می‌کنند، و برای این منظور پردازنده را به‌صورت موقتی به حالت محافظت‌شده برده و سپس به حالت واقعی بر می‌گردانند. توابع XMS، بهتر از توابع BIOS حافظه‌ی بسط‌یافته را مدیریت می‌کنند و توانایی استفاده اشتراکی از حافظه‌ی بسط‌یافته را فراهم می‌کنند، در حالیکه توابع BIOS استفاده مجزا را فراهم می‌کنند.

در ادامه در مورد سه موضوع صحبت می‌کنیم:

۱) توابع BIOS برای حافظه‌ی بسط‌یافته،
۲) مفهوم ناحیه‌ی بالای حافظه[1] (HMA)،
۳) توابع گرداننده‌ی XMS برای حافظه‌ی بسط‌یافته.

۴-۱۲ توابع BIOS برای دسترسی به حافظه‌ی بسط‌یافته

PCهایی که بیش از 640KB RAM دارند، دارای XM هستند. یعنی حتی اگر یک PC تنها 1MB RAM داشته باشد، میزان XM آن، 1MB - 640KB = 360KB است. این 360KB بالاتر از فضای نشانی 1MB قرار دارد:



برخی از توابع وقفه‌ی 15H، برای دسترسی به XM قابل استفاده هستند. وقفه‌ی 15H، در اصل برای واسط نوار کاست طراحی شده بود، اما به دلیل مرسوم شدن دیسکها، این توابع استفاده نشدند. از اینرو، برخی از توابع این وقفه برای پشتیبانی XM و Joystick در نظر گرفته شدند.

تابع 88H وقفه 15H، مقدار XM را در AX برمی‌گرداند. حال می‌دانیم چقدر XM داریم. چگونه می‌توانیم از آن استفاده کنیم؟ تابع 87H، بلوکهای حافظه را در کل حافظه، یعنی بین 640KB و بالاتر از 1MB مبادله می‌کند. یعنی می‌توان با استفاده از آن، یک بلوک حافظه را از زیر فضای 1MB به بالای فضای 1MB و برعکس منتقل نمود. تابع 87H، تابعی پیچیده است و به این دلیل زیاد استفاده نشده است.

پارامترهای ورودی به تابع 87H:

AH = 87H
CX = تعداد کلمات انتقالی
ES:SI = Address of GDT

در CX باید تعداد کلمات (دو بایتی) قرار داده شود و حداکثر مقدار 8000H است و لذا می‌توان 64K را هر بار منتقل نمود.

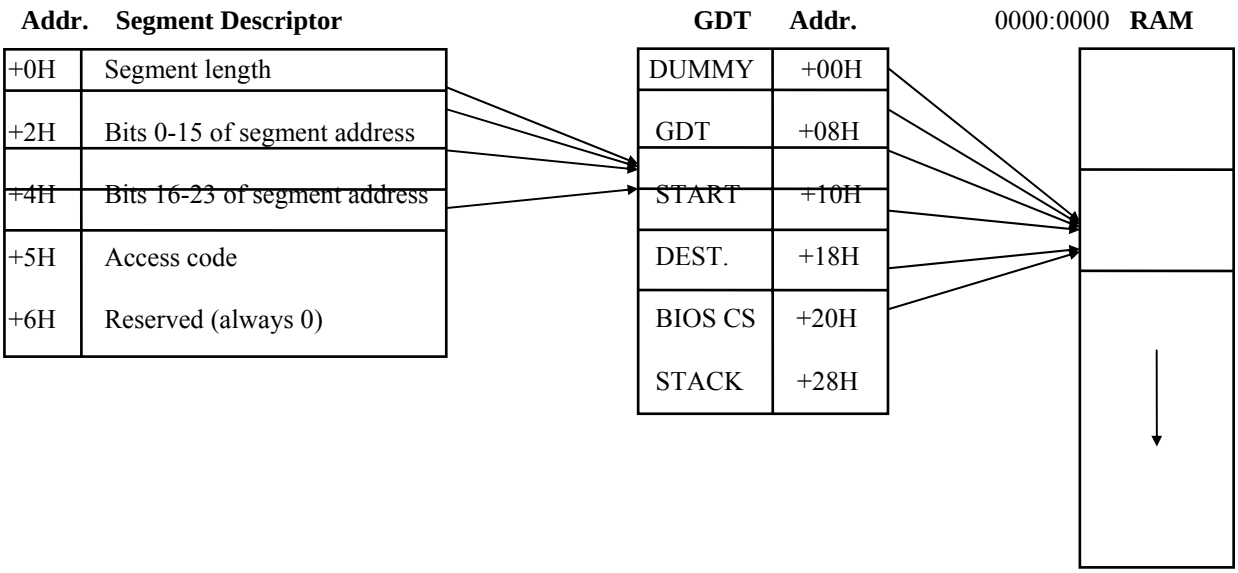
جدول توصیف‌گر سراسری[2](GDT): در زوج ثبات ES:SI، باید نشانی GDT قرار داده شود، که باید در برنامه‌ی کاربر قرار داشته باشد. GDT قطعات مجزای حافظه را توصیف می‌کند. قطعات در حالت محافظت شده می‌توانند از هر نشانی شروع شوند و نیازی به مضربی از 16 بودن ندارند و می‌توانند از یک بایت تا 64KB باشند. یک نکته‌ی دیگر این است که برای هر قطعه باید مشخص نمود که کد(CS) است یا داده (DS). فقط CSها قابل اجرا هستند و خود پردازنده DSها را اجرا نمی‌کند. برای CS می‌توان الویت تعیین نمود، و اینکه اصلاً اجازه‌ی دسترسی داده شده است یا نه؟

هر توصیف‌گر قطعه شامل ۸ بایت است. در طی فراخوانی تابع 87H، فرض می‌شود که شش توصیف‌گر قطعه در GDT آماده شده‌اند. در شکل (۱۲-۲) این مسأله توضیح داده شده است.

---

[1] High Memory Area
[2] Global Descriptor Table

شکل ۱۲-۲: ساختار توصیف‌گر قطعه از نظر وقفه‌ی 87H



تابع 87H نیاز دارد که فقط فیلدهای Start و Destination را مقداردهی کنیم و سایر فیلدها را خود تابع مقداردهی می‌کند و اطلاعاتی را در آن قرار می‌دهد که برای برگشت به حالت واقعی و برنامه‌ی فراخواننده مورد نیاز هستند. Start، قطعه‌ای را توصیف می‌کند که داده‌ها در آن قرار دارند و باید منتقل شوند و Destination قطعه‌ای را مشخص می‌کند که داده‌ها باید در آن کپی شوند. طول هر دو قطعه می‌تواند 0FFFFH یا 64KB باشد و حتی می‌تواند تعدادی کلمه (دوبایتی) نیز باشد. اگر تنها یک بایت باید کپی شود، لازم است که در دو ضرب شود و اگر تعداد کلمات کمتر از دو یا بیشتر از 0FFFFH باشد، خطا می‌گیرد.

نشانی‌های فیزیکی حافظه در 8088، ۲۰ بیتی هستند، اما در 80286، ۲۴ بیتی هستند و به این دلیل است که اولی تنها 2^{20} = 1MB و دومی 2^{24} = 16MB را نشانی‌دهی کند. در حالت محافظت‌شده، نشانی‌های حالت واقعی (مبدأ، قطعه) باید به نشانیهای فیزیکی ۲۴بیتی تبدیل شوند. ۱۶ بیت مبدأ وارد فیلد دوم (+2H) شده و ۴ بیت قطعه وارد فیلد سوم (+4H) می‌شوند و نشانی ۲۴بیتی بدست می‌آید.

فیلد چهارم کد دسترسی قطعه است و همیشه مقدار 92H در آن قرار داده می‌شود که بیانگر (۱) داده‌ای بودن قطعه، (۲) الویت بالا[۱]، (۳) در حافظه قرار داشتن و (۴) قابل نوشتن بودن آن است.

فیلد پنجم برای سازگاری با 80386 است و مقدار 0 در آن قرار داده می‌شود.

پس از اجرای تابع، کد خطا در AH برگردانده می‌شود:

00H: No error
01H: RAM parity error
02H: GDT defective at function call
03H: Protected mode could not be properly initialized

...

معایب دسترسی به XM از طریق BIOS: در حالتی که پردازنده در حالت محافظت‌شده است، همه‌ی وقفه‌ها باید ناتوان[۲] یا منع شوند. اما، ممکن است در شرایطی که در حالت محافظت‌شده هستیم، تایمر یا صفحه کلید وقفه بدهند، و از اینرو درست مدیریت نخواهند شد، زیرا روالهای مدیریت وقفه‌ها برای حالت واقعی هستند. به خصوص در مورد تایمر، این مشکل آشکارا ملاحظه می‌شود و اگر برنامه‌ای مرتباً از طریق تابع 87H به XM دستیابی داشته باشد، سرعت ساعت سیستم کم می‌شود. چون، شمارشگر ساعت بوسیله‌ی روال سرویس وقفه‌ی 08H، بهنگام نمی‌شود.

مشکل دیگر، **تداخل[۳] در XM** است. زیرا در تئوری XM می‌تواند بین تمامی برنامه‌ها مشترک باشد. اما برخی برنامه‌های cache، نظیر SmartDrv، یا برنامه‌های کمکی ممکن است بخواهند از XM استفاده کنند. این مسأله باعث می‌شود که برنامه‌ای اصلی و برنامه‌هایی که به‌صورت مقیم در حافظه عمل می‌کنند، حافظه‌ی همدیگر را بازنویسی نموده و در کار یکدیگر تداخل نمایند. این مشکل نتیجه‌ی نبود کنترل یا مدیریت است، و از اینرو نیازمند یک برنامه‌ی مدیریت XM در سیستم هستیم. این در حالی است که 88H به همه‌ی برنامه‌ها، کل XM را به‌عنوان حافظه‌ی آزاد برمی‌گرداند و میزان اشغال شده توسط سایر برنامه‌ها را در نظر نمی‌گیرد. راه‌حل این مشکل، استفاده از گرداننده‌های XM، نظیر XMS است، که مدیریت حافظه‌ی بسط‌یافته را فراهم می‌کند.

۱ high priority
۲ disable
۳ conflict

۱۲-۵ ناحیه‌ی بالای حافظه (HMA)

HMA، یک قطعه از XM است که بدون انتقال به حالت محافظت‌شده و در حالت واقعی و تحت DOS قابل دسترسی است. این کشف مهمی بود که همزمان با ارائه‌ی DOS 5.0 قابل حصول شد. وجود HMA،از نحوه‌ی تشکیل نشانی فیزیکی از نشانی‌های مبدأ و قطعه در حالت واقعی نشأت می‌گیرد:

Physical Address (20 bits) = Segment Address * 16 + Offset Address

example: 0040H:0020H $\Rightarrow$ 0040H * 10H + 0020H = 00420H

and 0000H:0420H $\Rightarrow$ 0000H * 10H + 0420H = 00420H

پس روش تبدیل نشانی‌ها باعث می‌شود که نشانی‌های متفاوت با هم همپوشانی[1] داشته باشند. یعنی بیش از یک نشانی با روش Seg:Ofs به یک نشانی فیزیکی اشاره می‌کند.

نشانی شروع آخرین قطعه در فضای نشانی‌های 1MB، که مربوط به ROM-BIOS است، عبارت است:

F0000H * 10H + 0000H = F0000H

و نشانی خاتمه آن:

F0000H * 10H + FFFFH = FFFFFH

که این معادل است با:

FFFFH * 10H + 000FH = FFFFFH

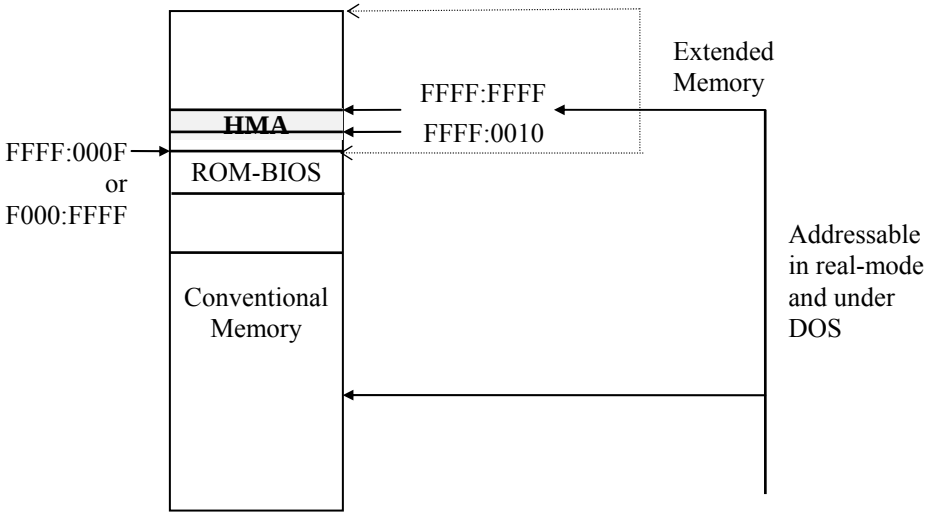
حال اگر در این فرمول به جای 0000FH، مقداری بزرگتر از آن قرار دهیم، به شرط عدم سرریزی نشانی[2]، یک نشانی فیزیکی بدست خواهد آمد که به فضای بالای 1MB اشاره می‌کند. برای مثال:

FFFFH * 10H + 0010H = 100000H

یک نشانی ۲۱بیتی است که به اولین بایت خارج از فضای 1MB را نشان می‌دهد، و این همان نشانی شروع HMA است که تا نشانی مبدأ FFFFH ادامه می‌یابد، که نشانی فیزیکی آن عبارت است از:

FFFFH * 10H + FFFFH = 1FFFEFH

شکل ۱۲-۳: نحوه‌ی شکل‌گیری HMA



به این ترتیب اگر نشانی قطعه FFFFH و نشانی مبدأ 0010H یا بیشتر باشد، یک نشانی فیزیکی ۲۱بیتی بدست می‌آید و دسترسی به XM ممکن می‌شود. مشروط به اینکه نشانی ۲۱بیتی قابل تولید باشد. در PC/XT و دارای پردازنده‌های ۸۰۸۶ یا ۸۰۸۸ این امر میسر نیست، چون نشانیهای فیزیکی ۲۰بیتی هستند. اما، در AT با پردازنده‌ی ۸۰۲۸۶، نشانیهای فیزیکی ۲۴ بیتی هستند. در ۸۰۸۶ یا ۸۰۸۸ و نیز در حالت واقعی ۸۰۲۸۶، اگر مقادیر مبدأ و قطعه به‌گونه‌ای باشد که نشانی فیزیکی حاصله بیش از ۲۰ بیت باشد، اصطلاحاً یک سرریزی نشانی اتفاق می‌افتد و بیت ۲۱ام درنظر گرفته نشده و مثلاً FFFFH:0010H، معادل 0000H:0000H درنظر گرفته شده و به ابتدای RAM اشاره خواهد نمود.

---

[1] overlap
[2] address overflow

در 8088/8086 خطوط نشانی دارای اسامی A0-A19 و در 80286، دارای اسامی A0-A23 هستند. برای جلوگیری از سرریزی نشانی، باید خط ۲۱ام نشانی یا A20 فعال شود. این خط در ابتدای روشن شدن سیستم، غیرفعال است، و با فعال شدن آن، مشکل فوق پیش نمی‌آید. به این ترتیب برای دسترسی به HMA، مراحل زیر باید طی شود:

مراحل دسترسی به HMA:

- بررسی شود که امکان دسترسی به HMA وجود دارد یا نه؟ اگر پردازنده، 80286 به بالا باشد، امکان‌پذیر است. چون خط نشانی A20 وجود دارد.
- تابع 88H وقفه‌ی 15H فراخوانی شود تا مقدار XM مشخص شود. اگر بیش از 64K باشد، آنگاه امکان دسترسی به HMA وجود دارد.
- بیت 1 درگاه خروجی کنترلر صفحه‌کلید باید 1 شود، تا خط A20 نشانی فعال شود(!). در DOS 5.0 به بالا، دستور DOS=HIGH, UMB در CONFIG.SYS همین کار را می‌کند.
- باید در برنامه‌ها بررسی شود که A20 فعال است یا نه؟ اگر فعال است، یعنی HMA وجود دارد و از آن استفاده شود. برای این منظور می‌توان مقدار خانه‌های حافظه از FFFF:0000H را با 0000:0000H مقایسه کرد. مثلاً ۲۵۶ بایت را مقایسه می‌کنیم. اگر یکسان بودند، به معنی عدم دسترسی به HMA است. البته این یک حقه[1] است و با احتمال خیلی کمی ممکن است درست کار نکند.

اندازه‌ی HMA، اندکی کمتر از 64K است(65520 bytes) است، ولی این میزان برای برنامه‌های کوچک TSR کافی است.

۱۲-۶ استاندارد XMS برای دسترسی به حافظه‌ی بسط‌یافته

این واسط امکان دسترسی اشتراکی چند برنامه به XM را فراهم می‌کند. این استاندارد انواع دسترسی زیر را فراهم می‌کند:

- دسترسی به HMA (1024K-1088K)
- چهار EMB (Extended Memory Block) که پس از 1088K شروع می‌شوند.
- UMB (Upper Memory Block)، مشابه EMS در فضای بین 640K تا 1024K امکان دسترسی به XM فراهم می‌شود.

HIMEM.SYS، که جزو گردانندههای DOS است، معروفترین گردانندهی XMS است و در CONFIG.SYS نصب می‌شود. به همراه گردانندهی XMS، برنامه‌ی EMM386.EXE نیز نصب می‌شود که استاندارد LIM را پشتیبانی نموده و XM را در قالب UMB قابل دسترسی می‌کند.

قبل از دسترسی به XM از طریق XMS باید بررسی شود که گردانندهی XMS نصب است یا نه؟ برای این منظور وقفه‌ی 2FH باید فراخوانی شود:

```
AX = 4300H
INT 2FH
```

آنگاه اگر AL = 80H باشد، گردانندهی نصب شده است و گرنه، نصب نشده است.

توابع XMS:

Function	Task
00H	Determine XMS driver version
01H	Allocate High Memory Area (HMA)
02H	Free High Memory Area (HMA)
03H	Globally enable address line A20
04H	Globally disable address line A20
05H	Locally enable address line A20
06H	Locally disable address line A20
07H	Query status of address line A20
08H	Query free extended memory
09H	Allocate Extended Memory Block (EMB)
0AH	Free allocated Extended Memory Block (EMB)
0BH	Move Extended Memory Block (EMB)
0CH	Lock Extended Memory Block (EMB)
0DH	Unlock Extended Memory Block (EMB)
0EH	Get EMB handle information
0FH	Resize Extended Memory Block (EMB)
10H	Allocate Upper Memory Block (UMB)
11H	Free allocated Upper Memory Block (UMB)

[1] trick

توابع به چند دسته تقسیم می‌شوند:

۱) توابع اولیه: برای بررسی وضعیت گرداننده، خط A20 و میزان حافظه‌ی آزاد(شامل توابع 00H و 03H-08H)
۲) توابع HMA: شامل توابع 01H برای تخصیص HMA و 02H برای آزادسازی آن است.
۳) توابع EMB: شامل توابع 09H-0FH است و برای تخصیص و آزادسازی و مدیریت EMB هستند.
۴) توابع UMB: شامل تابع 10H برای تخصیص UMB و 11H برای آزادسازی آن است.

این توابع با Far Call و نه فراخوانی وقفه، قابل فراخوانی هستند. برای این منظور باید نشانی XMS Driver در دست باشد. با فراخوانی تابع AX 4310H = وقفه‌ی 2FH، نشانی قطعه در ثباتهای ES:BX برگردانده می‌شود. توابع ۱۸گانه‌ی فوق آنگاه با قرار دادن شماره‌ی آنها که در جدول فوق ذکر شده، در ثبات AH قابل فراخوانی هستند.

فصل ۱۶: ساختار داخلی داس
(Internal Structure of DOS)

۱۶-۱ اجزاء داس

داس دارای سه مؤلفه‌ی اصلی است:

- **DOS-BIOS:** در یک فایل سیستم ذخیره شده و دارای یکی از اسامی IBMBIO.COM (در PC-DOS) یا IO.SYS (در MS-DOS) است. مشخصه‌ی[۱] این فایل Hidden و Sys است و با فرمان Dir در فهرست فایلها ظاهر نمی‌شود. این فایل شامل گرداننده‌های وسیله[۲] زیر است:

  CON: Keyboard and display
  PRN: Printer
  AUX: Serial interface
  CLOCK: Clock
  Disk drives and/or hard disks which have the drive specifiers A:, B:, and C:

  اگر داس بخواهد با یکی از موارد فوق ارتباط برقرار کند، به یک گرداننده‌ی وسیله دسترسی می‌یابد، که در این ماجول از داس قرار دارد. خود این ماجول هم از روالهای ROM-BIOS استفاده می‌کند و بخشی از سیستم عامل است که بیش از همه وابسته به سخت‌افزار PC است.

- **هسته‌ی داس[۳]:** هسته‌ی داس در فایل IBMDOS.COM یا MSDOS.SYS قرار دارد و عموماً دارای مشخصه‌ی Hidden و Sys است. این ماجول، شامل روالهای دسترسی به فایل (مدیریت فایل)، ورودی-خروجی کاراکتر، و موارد دیگر می‌شود. این ماجول شامل توابع API داس است که با وقفه‌ی 21H فراخوانی می‌شوند. این توابع مستقل از سخت‌افزار بوده و از گردانندههای وسیله‌ی موجود در DOS-BIOS برای دسترسی به صفحه‌کلید، صفحه نمایش، و دیسک استفاده می‌کنند. وقفه‌ی 21H در هنگام بوت شدن داس، تعویض شده و توابع آن قابل فراخوانی می‌شوند. روش فراخوانی این توابع مشابه توابع BIOS بوده و با وقفه‌ی نرم‌افزاری فراخوانی می‌شوند و انتقال پارامترها با استفاده از ثباتها صورت می‌گیرد.

- **پردازشگر فرمان[۴]:** برخلاف دو ماجول قبلی، این بخش در فایل COMMAND.COM وجود دارد که Hidden و Sys نبوده و علامت داس[۵]، یعنی A> یا C> را بر روی صفحه نمایش نشان داده و فرامین کاربر را دریافت نموده و اجرا می‌کند. برخی گمان می‌کنند که COMMAND.COM، سیستم عامل است، در حالیکه اینگونه نیست و این فایل فقط یک برنامه‌ی اجرایی بوده و تحت کنترل داس اجرا می‌شود. این برنامه شامل سه قسمت زیر است:

الف) بخش مقداردهی اولیه[۶]: هنگام بوت شدن داس به متغیرهای داس، مقدار اولیه می‌دهد.

ب) بخش مقیم در حافظه[۷]: شامل روالهای مدیریت خطاهای بحرانی[۸](زدن کلیدهای Ctrl+C یا Ctrl+Break)، نمایش علامت داس و دریافت فرامین و پردازش آنها، فراخوانی برنامه‌هایی که اسامی آنها توسط کاربر وارد شده و بارکردن آنها در حافظه و اجرای آنها، و بارکردن مجدد بخش گذرا، در صورتی که حافظه‌ی مورد استفاده‌ی بخش گذرا توسط سایر برنامه‌ها بازنویسی شده باشد. برای تشخیص این امر، از مجموع مقابله‌ای[۹] استفاده می‌شود.

ج) بخش گذرا[۱۰]: وظیفه‌ی اجرای فرمانهای داخلی[۱۱] داس (نظیر RENAME، COPY، DIR و ...)، فرمانهای خارجی[۱۲] (نظیر FORMAT، XCOPY و ... و به‌طور کلی هر برنامه‌ی COM. یا EXE.)، و فایلهای دسته‌ای[۱۳] (یک فایل متنی با پسوند BAT. که شامل یک سری از فرمانهای داس است، که یک نمونه آن، AUTOEXEC.BAT است). دلیل اینکه به این بخش گذرا گفته می‌شود این

---

۱ attribute
۲ device drivers
۳ DOS kernel
۴ command processor
۵ DOS prompt
۶ initialization part
۷ resident part
۸ critical error handler
۹ checksum
۱۰ transient part
۱۱ internal commands
۱۲ external commands
۱۳ Batch files

است که حافظه‌ای که برنامه‌ی این بخش در آن قرار دارد، در صورت نیاز سایر برنامه‌ها توسط داس به آنها تخصیص داده شده و با اصطلاح رزرو نمی‌شود. از اینرو پس از خاتمه‌ی اجرای برنامه‌ها و در زمان برگشت به پردازشگر فرمان، با مجموع مقابله‌ای بررسی می‌شود که حافظه‌ی این بخش بازنویسی شده است یا نه و در صورت نیاز دوباره از دیسک بار می‌شود. همانطوری که گفته شد، این بررسی و بارکردن مجدد، توسط بخش مقیم درحافظه، انجام می‌شود.

۲-۱۶ راه‌اندازی[۱] داس

فرآیند راه‌اندازی شامل فراخوانی سه ماجول DOS-BIOS، DOS kernel و Command Processor است. وقتی PC روشن می‌شود، برنامه‌ی راه‌اندازی که در ROM-BIOS وجود دارد، اجرا می‌شود. این برنامه کارهای POST[۲] را انجام می‌دهد و سپس شروع به بارکردن سیستم عامل می‌کند. ابتدا بررسی می‌شود که گرداننده‌ی دیسکت وجود دارد یا نه و آیا در آن دیسکتی هست یا نه؟ اگر باشد، در Boot Sector آن به دنبال سیستم عامل می‌گردد. اگر دیسکت در گرداننده نباشد، به دنبال دیسک سخت می‌گردد و قسمت فعال[۳] را یافته و Boot Sector آنرا بار نموده و اجرا می‌کند. اگر Boot Sector دیسکت یا قسمت فعال دیسک سخت، اجرایی نباشد، پیغام تعویض دیسکت یا قرار دادن دیسکت داده می‌شود.

برنامه‌ی موجود در Boot Sector چک می‌کند که فایلهای IO.SYS و MSDOS.SYS وجود دارند یا نه، اگر باشند در آخرین مکان حافظه بارگذاری شده و اجرا می‌شوند. MSDOS.SYS، وقفه‌ی 21H را تعویض نموده و گردانندههای وسیله‌ی موجود در DOS-BIOS را که با IO.SYS بار شده‌اند را مقدار دهی اولیه می‌کند. سپس در Boot Disk، به دنبال فایل CONFIG.SYS می‌گردد و در صورت وجود، فرمانهای موجود در آن را اجرا می‌کند، تا گردانندههای معرفی شده در آن را به داس اضافه نماید و نیز بافرها و لوازم ورودی-خروجی را مقدار دهی اولیه می‌کند. در خاتمه COMMAND.COM را بار نموده و کنترل را به آن منتقل می‌کند.

---

۱ booting
۲ Power On Self Test
۳ active partition

فصل ۱۷: برنامه‌های COM و EXE
(COM and EXE Programs)

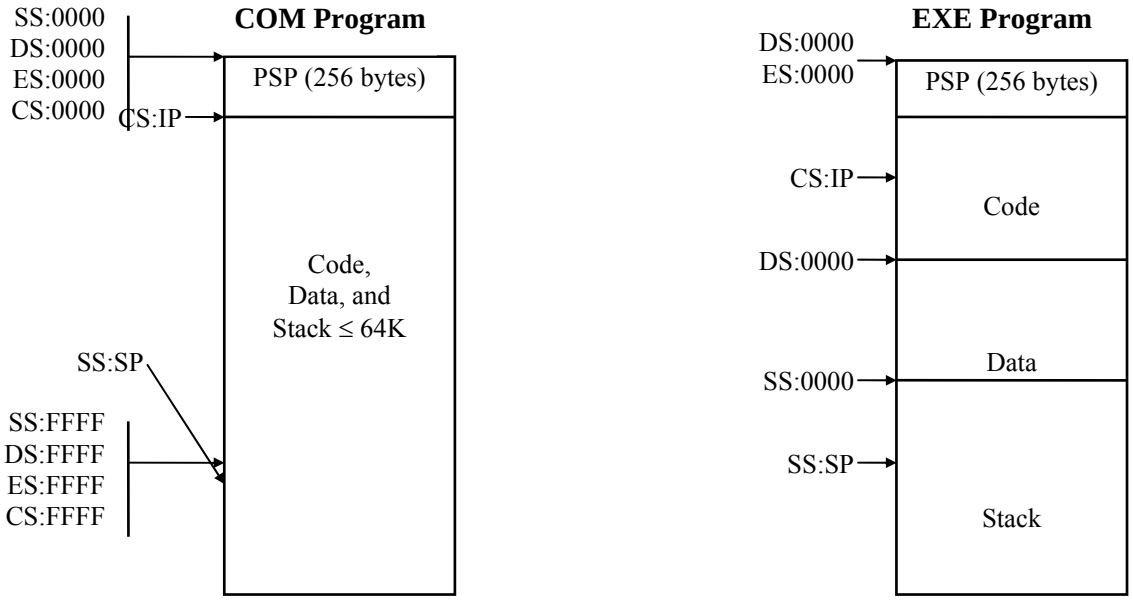
۱-۱۷ مقدمه

دو نوع فایل اجرایی در داس وجود دارد: COM و EXE. برنامه‌های COM دارای اندازه‌ای کوچک بوده و CS، DS و SS آنها در مجموع حداکثر 64K است. اغلب کامپایلرهای پاسکال و C، فقط فایل EXE تولید می‌کنند، ولی برخی کامپایلرهای C برای برنامه‌های دارای مدل TINY می‌توانند COM هم تولید کنند. برنامه‌ی EXE2BIN داس و TLINK بورلند نیز توانایی تبدیل برنامه‌ی EXE به COM یا تولید برنامه‌ی COM را دارند.

۲-۱۷ تفاوت برنامه‌های COM و EXE

برنامه‌های COM از زمان سیستم عامل CP/M مطرح شدند، که حافظه PC بسیار کم و حدود 64K بود. در DOS برنامه‌های COM نمی‌توانند بیشتر از 64K باشند. اما برنامه‌های EXE می‌توانند هر اندازه‌ای که در حافظه‌ی متعارف (640K) جا شود، داشته باشند. برنامه‌های COM فقط یک قطعه هستند، اما برنامه‌های EXE دارای قطعات مجزای CS، DS و SS هستند.

PSP (Program Segment Prefix) : داس برای اجرای برنامه، بدون درنظر گرفتن نوع آن، یک ساختار در حافظه ایجاد می‌کند که PSP نامیده می‌شود و آنرا قبل از شروع برنامه ایجاد می‌کند و کد برنامه بلافاصله پس از آن در حافظه بار می‌شود. این ساختار که PSP نامیده می‌شود، 256 بایت است و مطابق شکل زیر، قبل از برنامه در حافظه قرار می‌گیرد. در ادامه‌ی این فصل در مورد PSP بیشتر صحبت می‌شود.



۳-۱۷ برنامه‌های COM

این برنامه‌ها به‌عنوان یک تصویر حافظه[۱] در دیسک ذخیره می‌شوند. از اینرو، نیاز به پیش‌پردازش ندارند، تا نشانی‌های نسبی[۲] تولید شده بوسیله‌ی کامپایلر یا اسمبلر براساس مکان بارشده در حافظه، به نشانی‌های واقعی اصلاح شوند. از اینرو، سریع‌تر در حافظه بارمی‌شوند. اما امروزه به‌دلیل افزایش حافظه و سرعت کامپیوترها، این مزایای برنامه‌های COM چندان اهمیت ندارد و فقط برای کاربردهای خاص، مثلاً نوشتن گرداننده‌های وسیله، از این نوع برنامه‌ها استفاده می‌شود تا حجم کمی از حافظه را اشغال نموده و سریع باشند.

برنامه‌ای به‌نام EXE2BIN جزو فرامین خارجی داس است که از آن می‌توان برای تبدیل برنامه‌های EXE به برنامه‌ی COM استفاده نمود. این برنامه چک می‌کند که این امر امکان‌پذیر است یا نه؟ یعنی کد برنامه‌ی EXE آنقدر کوچک باشد که در 64K قرار گیرد. برنامه‌های اسمبل شده با MASM و متصل شده با LINK، EXE خواهند بود، که برای تبدیل آنها به COM می‌توان از EXE2BIN استفاده نمود. اما اگر از اسمبلر TASM استفاده شود، با TLINK می‌توان مستقیماً برنامه‌ی COM تولید کرد.

برای اجرای برنامه‌ی COM، محتوای فایل COM بلافاصله پس از PSP و نشانی مبدأ 100H، در حافظه بار می‌شود. پس اولین دستورالعمل پس از 100H اجرا می‌شود. از اینرو، برنامه‌ی COM حتماًباید با یک دستورالعمل اجرایی شروع شوند، حتی اگر این دستورالعمل، یک jmp باشد.

۴-۱۷ برنامه‌های EXE

یک مزیت عمده برنامه‌های EXE نسبت به COM، محدود نبودن اندازه‌ی این برنامه‌ها است. اما، فایلهای EXE پیچیده هستند، زیرا علاوه بر کدهای اجرایی برنامه، اطلاعات دیگری هم باید در فایلهای EXE نگهداری شود. این برنامه‌ها دارای قطعات مجزای کد، داده و پشته هستند و پس از بارشدن از دیسک باید توسط تابع EXEC پردازش شوند تا برای اجرا آماده شوند. برنامه‌های COM همانند یک تصویر حافظه هستند و همیشه باید در یک مکان ثابت در حافظه قرار گیرند و اجرا شوند. اما برنامه‌های EXE در هر مکان حافظه که مضربی از 16 باشد، می‌توانند قرار گرفته و اجرا شوند. به این دلیل در یک برنامه‌ی EXE باید از دستورات FAR استفاده شود. چون ممکن است برنامه‌ی اصلی و روالها در قطعات متفاوت قرار گیرند. از آنجایی‌که، برنامه هر بار در مکان متفاوتی بار می‌شود، نشانی قطعات در اجراها مختلف، متفاوت خواهد بود. بنابر این، باید در ابتدای فایل EXE اطلاعاتی وجود داشته باشد که از طریق آن بتوان نشانی قطعات در فایل EXE را تصحیح نمود. به این دلیل در ابتدای فایل EXE، **سرآیندی**[۳] وجود دارد که اطلاعات مهمی توسط کامپایلر یا اسمبلر در آن ذخیره می‌شود و مورد استفاده‌ی بارکننده‌ی داس(که همان تابع EXEC با شماره‌ی 4BH) قرار می‌گیرد. قالب سرآیند فایل EXE مطابق جدول صفحه‌ی بعد است.

Address	Contents	Type
00H	EXE program identifier (5A4DH or “MZ”)	1 word
02H	File length MOD 512	1 word
04H	File length DIV 512	1 word
06H	Number of segment addresses for passing	1 word
08H	Header size in paragraph	1 word
0AH	Minimum number of paragraphs needed	1 word
0CH	Maximum number of paragraphs needed	1 word
0EH	Stack segment displacement	1 word
10H	SP register contents when program starts	1 word
12H	Checksum based on EXE file header	1 word
14H	IP register contents when program starts	1 word
16H	Start of code segment in EXE file	1 word
18H	Relocation table address in EXE file	1 word
1AH	Overlay number	1 word
1CH	Buffer memory	??
??H	Address of passing segment addresses (relocation table)	??
??H	Program code, data and stack segment	??

بارکننده‌ی داس، یا همان تابع EXEC که در فصلهای بعدی در باره‌ی آن صحبت می‌شود، میزان حافظه‌ی مورد نیاز برنامه را از سرآیند فایل EXE بدست می‌آورد. این میزان براساس پاراگراف(۱۶ بایت) است. ”relocation table“ جدول است که نشانیهای قطعه‌ای که در فایل EXE باید تصحیح شوند، را مشخص می‌کند.

---

۱ memory image
۲ relative addresses
۳ header

۵-۱۷ PSP (Program Segment Prefix)

PSP از یادگارهای سیستم عامل CP/M در داس است و یک ساختار ۲۵۶ بایتی است که قبل از کد برنامه در حافظه قرار می‌گیرد و حاوی اطلاعات مدیریت برنامه توسط داس است. همچنین شامل اطلاعاتی، نظیر پارامترهای برنامه است که مورد استفاده‌ی برنامه‌نویسان قرار می‌گیرد. این پارامترها در برنامه‌هایی که با زبانهای سطح بالا نوشته می‌شوند، توسط کامپایلر خوانده شده و در متغیرهای سراسری قرار داده می‌شوند، اما در برنامه‌های اسمبلی باید توسط برنامه‌نویس از PSP خوانده شوند. جدول زیر ساختار PSP را نشان می‌دهد، که اغلب فیلدهای آن بلااستفاده هستند و مهمترین فیلدها، دو فیلد آخر هستند که اطلاعات مربوط به خط فرمان برنامه یا همان پارامترها هستند.

Address	Contents	Type
00H	Interrupt 20H call	2 bytes
02H	Segment address of memory allocated for program	1 word
04H	Reserved	1 byte
05H	Interrupt 21H call	5 bytes
0AH	Copy of interrupt vector 23H	2 words
0EH	Copy of interrupt vector 23H	2 words
12H	Copy of interrupt vector 23H	2 words
16H	Reserved	22 bytes
2CH	Segment address of environment block	1 word
2EH	Reserved	46 bytes
5CH	FCB 1	16 bytes
6CH	FCB 2	16 bytes
80H	Number of characters in command line	1 byte
81H	Command line (CR-LF)	127 bytes

فصل ۱۸: ورودی-خروجی کاراکتر از طریق داس
(Character Input and Output from DOS)

۱-۱۸ مقدمه

توابع ورودی خروجی داس، امکان دسترسی به صفحه کلید، صفحه نمایش، چاپگر و واسط سریال را میسر می‌سازند. این توابع به دو دسته تقسیم می‌شوند:

۱) توابع مشابه سیستم عامل CP/M

۲) توابع مشابه سیستم عامل Unix

توابع یونیکسی، از یک دسته[۱] که یک شماره‌ی عددی است، به عنوان مشخصه‌ی وسیله استفاده می‌کنند.

در ادامه در مورد هر دو گروه توابع صحبت می‌کنیم.

۲-۱۸ توابع دسته[۲]

این توابع هم با فایل و هم با وسایل کاراکتری[۳] قابل استفاده هستند. نوع یا نام دسته، وسیله را تعیین می‌کند. اگر نام دسته، یکی از لوازم استاندارد و شناخته شده‌ی زیر باشد، ورودی-خروجی روی آن وسیله انجام می‌شود، و در غیر این‌صورت، باید دسته، مربوط به یک فایل باشد که باید قبلاً منسوب به یک فایل فیزیکی شده و باز شده باشد.

CON: Keyboard and display

PRN: Printer

AUX: Serial interface

NUL: Imaginary device (nothing happens on access)

وقتی یک برنامه شروع می‌شود، ۵ دسته برای موارد فوق با شماره‌های ۰ تا ۴ از قبل وجود دارند و شناخته شده هستند.

Handle	**Purpose**
0	Standard input (CON/ keyboard)
1	Standard output (CON/ display)
2	Standard output for error messages (CON)
3	Standard serial interface (AUX)
4	Standard printer (PRN)

---

۱ handle
۲ handle functions
۳ character devices

توابع دسترسی به وسیله‌ها:

1. Function 40H of INT 21H: (Send/write data) ارسال داده به وسیله

AH = 40H
BH = handle
CX = number of characters
DS:DX = offset and segment of data

2. Function 3FH of INT 21H: (Receive/read data) دریافت داده از وسیله

AH = 3FH
BH = handle
CX = number of characters
DS:DX = offset and segment of data

3. Function 3DH of INT 21H: (Open) بازکردن وسیله

AH = 3DH
AL = 0: read 1: write 2: read/write
DS:DX = address of device name (nul-terminated) string

4. Function 3EH of INT 21H: (Close) بستن وسیله

AH = 3EH
BX = handle

۱۸-۳ توابع CP/M

توابع سنتی داس بوده و مشابه سیستم عامل CP/M هستند و نیازی به دسته ندارند و برای صفحه‌نمایش، صفحه کلید، چاپگر، و واسط سریال توابع مجزایی وجود دارند:

- **صفحه کلید:** توابع 01H، 06H، 07H، و 08H برای خواندن کاراکتر از صفحه کلید به‌صورتهای مختلف با انتظار، بدون انتظار و غیره، قابل استفاده هستند. تابع 0BH، تعداد کاراکترها در بافر صفحه کلید را برمی‌گرداند. تابع 0CH، بافر صفحه کلید را خالی می‌کند و با تابع 0AH می‌توان یک رشته را از ورودی خواند.
- **صفحه نمایش:** تابع 02H، برای نمایش کاراکتر و تابع 09H برای نمایش رشته در خروجی استاندارد قابل استفاده هستند.
- **چاپگر:** تابع 05H برای چاپ یک کاراکتر در چاپگر، یا همان واسط موازی قابل استفاده است.
- **واسط سریال:** تابع 03H برای دریافت داده و تابع 04H برای ارسال داده به واسط سریال قابل استفاده هستند.

فصل ۱۹: مدیریت فایل در داس
(File Management in DOS)

۱۹-۱ مقدمه

داس دارای توابعی برای مدیریت فایل است. از طرفی، زبانهای برنامه‌سازی سطح بالا، اغلب توابع خاصی را برای مدیریت فایل فراهم می‌کنند و از اینرو، برنامه‌نویسان از توابع داس بی‌نیاز می‌شوند. اما، کسانی که با زبان اسمبلی برنامه می‌نویسند، کماکان به توابع داس نیاز خواهند داشت.

برای مدیریت فایل باید توابعی برای ایجاد، حذف، بازکردن، بستن، خواندن و نوشتن در فایل فراهم شود، که سیستم عامل داس، جزو توابع وقفه‌ی 21H، این توابع را فراهم نموده است. همانند ورودی-خروجی کاراکتر، برای مدیریت فایلها هم دو دسته توابع سازگار با CP/M (توابع FCB) و سازگار با Unix (توابع دسته[۱]) در داس وجود دارد، که در ادامه به آنها می‌پردازیم.

۱۹-۲ توابع FCB

این توابع سازگار با سیستم عامل CP/M هستند و مبتنی بر ساختار FCB[۲] هستند. اطلاعات مورد نیاز در مورد فایل در زمان پردازش آن در ساختار FCB قرار دارند که داس از آنها استفاده می‌کند و برنامه‌نویس باید ساختمان داده‌ی FCB را در برنامه داشته باشد. FCB یک ساختار سلسله‌مراتبی برای فایلها نیست و لذا با این روش فقط می‌توان به فایلهای فهرست جاری دسترسی پیدا نمود. اما، وجود FCB در برنامه، باعث دسترسی آسان و سریع به اطلاعات فایل، نظیر تاریخ، زمان، و اندازه و غیره می‌شود. ساختمان داده FCB، ۳۷ بایت بوده و حاوی اقلام اطلاعاتی زیر است:

Address	Contents	Size
+00H	Device name(0: A, 1:B ...)	1 byte
+01H	File name	8 bytes
+09H	File mode (extention)	3 bytes
+0CH	Current block number	1 word
+0EH	Data record size	1 word
+10H	File size	2 words
+14H	Modification date 1 word	
+16H	Modification time	1 word
+18H	Reserved	1 word
+20H	Current data record number	1 byte
+21H	Data record no. for random access	2 words

نشانی FCB در زوج ثبات ES:DI به تابع 29H که در ادامه ذکر می‌شود، ارسال می‌شود، تا FCB را به یک فایل که نام آن در DS:SI قرار داده می‌شود، مرتبط می‌کند.

اما توابع FCB عبارتند از:

Function	Purpose
0FH:	Open file
10H:	Close file
13H:	Delete file
14H:	Sequential read
15H:	Sequential write
16H:	Create file
17H:	Rename file
1AH:	Set DTA* address
21H:	Random read (of record)
22H:	Random write (of record)
23H:	Determine file size
24H:	Set record number for random access

---

۱ handle function
۲ File Control Block

27H:	Random read
28H:	Random write
29H:	Enter file name into FCB

* برای فایلهای دارای رکورد بزرگتر از 128 بایت، باید یک بافر برای رکورد به عنوان DTA[1] مشخص شود، تا به جای DTA پیش‌فرض، استفاده شود.

۱۹-۳ توابع دسته

در داس نگارش 2.0 به بعد، این مفهوم معرفی شد و همان مفهومی است که در سیستم عامل Unix هم وجود دارد. به ازای هر فایل در موقع بازکردن آن یک دسته(یک شماره‌ی عددی) برگردانده می‌شود که برای ارجاعات بعدی مورد استفاده قرار می‌گیرد و مشکل توابع FCB را ندارد. یعنی می‌توان به فایلهای موجود در فهرستها دسترسی پیدا نمود و ساختار سلسله‌مراتبی فایلها نیز پشتیبانی می‌شود.

این روش نیازمند نگهداری ساختمان داده‌ی FCB در برنامه نیست. سیستم عامل، اطلاعاتی را به‌ازای هر فایل پس از بازکردن آن در بافرهای داخلی خود نگهداری می‌کند. به این دلیل در بازکردن فایلها محدودیت وجود دارد. تعداد فایلهای باز را می‌توان در CONFIG.SYS با متغیر FILES مشخص نمود، که حداکثر می‌تواند ۲۵۵ باشد:

FILES=n

توابع دسته‌ای که در بخش قبل معرفی شد، هم با لوازم ورودی-خروجی استاندارد و هم با فایلها کار می‌کنند:

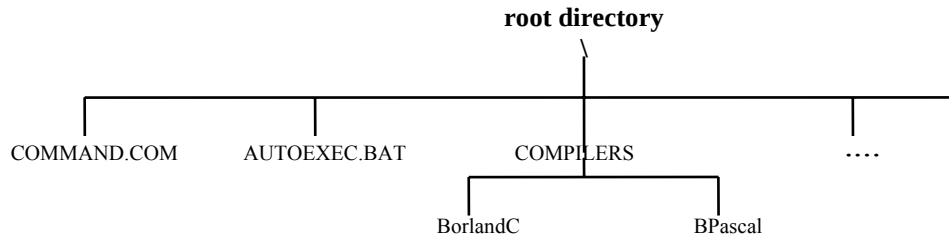
Function	**Purpose**
3CH:	Create file
3DH:	Open file
3EH:	Close file
42H:	Move file pointer/determine file size
43H:	Read/write file attribute
56H:	Rename file
57H/00H:	Read write modification and date/time of file
57H/01H:	Read write modification and date/time of file
5AH:	Create temporary file
5BH:	Create new file
5CH/00H:	Release protected file range
5CH:	Extended open function

[1] Disk Transfer Area

فصل ۲۰: دسترسی به فهرست داس
(Accessing DOS Directory)

۲۰-۱ مقدمه

تعدادی از توابع داس برای کارکردن با فهرست[۱] و جستجوی فایلها[۲] هستند. ساختار سلسله‌مراتبی فایلها و فهرست، در DOS 2.0 معرفی شد و به این وسیله می‌توان دیسک را به فهرستهایی به‌صورت منطقی تقسیم نمود و ساختار درخت فهرست[۳] را ایجاد کرد:



در درخت فهرست، تعداد زیرفهرستها[۴] به‌صورت پویا تغییر می‌کند. لذا توابع زیر مورد نیاز است:

۱) اضافه کردن ورودی[۵]، معادل فرمان MD در محیط فرمان داس: تابع 39H وقفه‌ی 21H
۲) حذف کردن ورودی، معادل فرمان RD در محیط فرمان داس: تابع 3AH وقفه‌ی 21H
۳) تغییر فهرست جاری، معادل فرمان CD در محیط فرمان داس: تابع 3BH وقفه‌ی 21H
۴) تغییر دیسک درایو پیش‌فرض، معادل A: یا C: در محیط فرمان داس: تابع 0EH وقفه‌ی 21H

در موقع فرمت کردن هر دیسکت، بخشهای ۱۹ تا ۳۲ به فهرست ریشه تخصیص داده می‌شود و در هر بخش، اطلاعات ۱۵ ورودی فهرست نگهداری می‌شود. به این ترتیب بر روی فهرست ریشه می‌توان حداکثر ۲۱۰ فایل نگهداری نمود. در مورد دیسک سخت هم به صورتی مشابه است و در هر حال در مورد تعداد فایلها در فهرست ریشه محدودیت وجود دارد (از برنامه‌ی Norton DiskEdit می‌توان برای دیدن اطلاعات فهرستها استفاده نمود). در هر ورودی فهرست، اطلاعاتی نظیر موارد زیر در مورد فایلها وجود دارد:

Name(8 char) Ext(3 char) Attribute(1 byte) Date Time ... First Cluster of File in FAT

به ازای هر زیرفهرست، یک فایل بر روی دیسک نگهداری می‌شود، و همان ساختار فوق برای فایلها در آن نیز رعایت می‌شود. ولی در مورد زیرفهرستها، محدودیتی در مورد تعداد فایلها در آن نداریم.

تابع 47H برای یافتن دایرکتوری جاری است و مسیر کامل[۶] فهرست جاری را برمی‌گرداند. درایو موردنظر باید در ثبات DL مشخص شود:

```
DL=0  :   Current directory
DL=1  :   A:
DL=2  :   B:
DL=3  :   C:
...
```

۲۰-۲ جستجوی فایلها

دو دسته تابع برای جستجوی فایلها وجود دارد:

---

۱ directory
۲ file search
۳ directory tree
٤ subdirectory
٥ entry
٦ full path

۱) توابع FCB : تابع 11H برای یافتن نخستین فایل(FindFirst) و تابع 12H برای یافتن بعدی(FindNext)
۲) توابع دسته: تابع 4EH برای یافتن نخستین فایل(FindFirst) و تابع 4FH برای یافتن بعدی(FindNext)

جستجو برای فایلها براساس مشخصه[۱] آن و نیز یک الگو[۲] (*.TXT) انجام می‌شود. به ازی هر فایل در ورودی فهرست، یک بایت به‌عنوان مشخصه وجود دارد که در جستجو مورد مقایسه قرار می‌گیرد. انواع مشخصه‌ی فایلها به‌صورت زیر است:

Bit	Attribute	Meaning if set
0	Read-Only	File is Read-Only
1	Hidden	File is hidden (invisible to DIR command)
2	System	File is part of operating system
3	Volume label	Entry is volume label and not a file
4	Subdirectory	Entry is subdirectory and not a file
5	Archive	Entry is archive
6	Reserved	Reserved for later implementation
7	Reserved	Reserved for later implementation

در جستجوی فایلها باید مشخصه‌ی فایلهای مورد نظر داده شود. دستور DIR داس نیز در حقیقت فراخوانی تابع FindFirst و سپس فراخوانی تکراری FindNext تا رسیدن به خطا (CF=1)، است. در دستور DIR بدون سوئیچ‌های اضافی(مثل H/ و...) فایلها و فهرستهای Non-Hidden نشان داده می‌شوند.

* روشی که در Win95 برای پیاده‌سازی نام طولانی فایلها[۳] استفاده شده، استفاده از همین بایت مشخصه فایلها در فهرستها است. به این ترتیب که در نخستین ورودی، بخشی از نام با مشخصه‌ی عادی و ادامه‌ی نام در ورودیهای دیگر با مشخصه‌ی رزرو است. برای مثال اگر نام فایل PCSystemProgramming.TXT باشد، تقریباً به‌صورت زیر خواهد بود:

1st entry with normal attribute: PCSyst~1.TXT
2nd entry with reserved attribute: emProgra.---
3rd entry with reserved attribute: mming.---

۱ attribute
۲ wildcard
۳ long file name

فصل ۲۲: مدیریت حافظه
(RAM Management)

۲۲-۱ مدیریت حافظه‌ی داس

یکی از وظایف سیستم عامل، مدیریت حافظه است. در هر زمان ممکن است چندین برنامه در حافظه قرار داشته باشند، همانند: گرداننده‌های وسیله[۱]، برنامه‌های TSR و برنامه‌های کاربردی، که با هم کار می‌کنند. ولی نباید در حافظه‌ی مورد استفاده‌ی همدیگر تداخل داشته باشند. کل بحث مدیریت حافظه در داس، مبتنی است بر استفاده از یک تابع داس برای تخصیص یک بلوک حافظه با اندازه‌ی از قبل مشخص شده. یک بلوک از حافظه به یک برنامه تخصیص داده می‌شود و تا زمانی که توسط تابع دیگری آزاد نشود، قابل تخصیص توسط همان برنامه یا برنامه‌های دیگر نخواهد بود.

توابع مدیریت حافظه عبارتند از:

Function	Purpose
48H	Allocate memory
49H	Free memory
4AH	Change size of a memory block
58H	Read/set memory management model

خود داس هم بوسیله‌ی تابع Exec، برای بارکردن برنامه‌ها در حافظه، با فراخوانی تابع 48H برای برنامه حافظه تخصیص می‌دهد و پس از پایان اجرای برنامه با فراخوانی تابع 49H آنرا آزاد می‌کند. مقدار حافظه‌ی تخصیص داده شده به نوع برنامه بستگی دارد. برای برنامه‌های COM. کل حافظه رزرو می‌شود و چنانچه برنامه به کل حافظه نیاز نداشته باشد، باید ابتدا مابقی حافظه را آزاد نماید، وگرنه حافظه‌ای آزاد نخواهد ماند. اما، برای برنامه‌های EXE. میزان حافظه‌ای که توسط تابع Exec تخصیص داده می‌شود، همان مقداری است که در سرآیند فایل EXE. مشخص شده است. اگر Exec نتواند به اندازه‌ی مشخص شده در سرآیند فایل EXE. حافظه تخصیص دهد، پیغام "Insufficent memory" یا "Program too big to fit in memory" داده و برنامه را اجرا نمی‌کند. علاوه بر حافظه‌ای که Exec برای برنامه‌ها تخصیص می‌دهد، خود برنامه‌ها هم می‌توانند به‌طور پویا حافظه تخصیص دهند.

تخصیص حافظه با تابع 48H: اگر از یک زبان سطح بالا برای برنامه‌نویسی استفاده می‌کنیم، مشکلی برای تخصیص حافظه نداریم. زیرا، زبانهای سطح بالا از یک ساختار heap استفاده می‌کنند و در ابتدا به میزان مورد نیاز، که در سرآیند فایل EXE. مشخص می‌شود، حافظه تخصیص داده و در برنامه با روش مدیریت حافظه‌ی مستقل از توابع داس، آنرا به برنامه تخصیص می‌دهند، و دستورات مربوط به تخصیص حافظه‌ی پویا، از این حافظه‌ی heap استفاده می‌کنند. اما، در برنامه‌های زبان اسمبلی، باید برنامه‌نویس تخصیص حافظه را با استفاده از تابع 48H انجام دهد. برای فراخوانی این تابع باید اندازه‌ی بلوک حافظه در ثبات BX مشخص شود، که براساس پاراگراف(۱۶ بایت) است و می‌تواند حداقل 1H (۱۶ بایت) و حداکثر 1111H (یک قطعه‌ی 64KB) باشد. در بازگشت، ثبات AX حاوی نشانی قطعه بلوک حافظه‌ی تخصیص داده شده است. مثلاً اگر یک پاراگراف حافظه درخواست شود، بلوک تخصیص داده شده، در محدوده‌ی AX:0000 تا AX:000F خواهد بود.

تعیین میزان حافظه‌ی قابل دسترسی: برای این منظور می‌توان از همان تابع 48H استفاده نمود. مقدار 0FFFFH (16*64KB) در BX قرار داده شده و سپس 48H فراخوانی شود. در پاسخ چون داس نمی‌تواند این مقدار حافظه تخصیص دهد، در BX تعداد پاراگرافهای آزاد را برمی‌گرداند.

تابع 4AH: این تابع اندازه‌ی یک بلوک ازقبل تخصیص داده شده را تغییر می‌دهد. برای کم‌کردن اندازه‌ی بلوک، مشکلی پیش نمی‌آید. اما برای افزایش آن، به دلیل آنکه ممکن است ادامه‌ی حافظه، توسط سایر برنامه‌ها تخصیص داده شده باشد، ممکن است امکان افزایش وجود نداشته باشد. در این صورت CF=1 شده که نشانگر وقوع خطا است و در ثبات BX، اندازه‌ی قابل افزایش برگردانده می‌شود.

تابع 58H: استراتژی‌های مختلف تخصیص حافظه[2] را می‌توان با این تابع تنظیم نمود. انواع استراتژی‌ها عبارتند از:

First Fit: Use first memory block large enough
Best Fit: Use smallest memory block large enough
Last Fit: Use high part of last usable memory block

---

۱ device drivers
۲ allocation strategy

۲-۲۲ انواع حافظه‌ی قابل تخصیص

تا نگارش DOS 5.0، کل حافظه‌ای که تابع 48H به برنامه‌ها تخصیص می‌داد، محدود به 640K حافظه‌ی متعارف بود. اما DOS 5.0 به بعد، امکان تخصیص حافظه‌ی بین 640KB و 1MB را تحت عنوان UMB (Upper Memory Block) فراهم نمود. این همان محدوده‌ای است که مورد استفاده‌ی EMS قرار می‌گیرد، یعنی EMS page frame، که برای این منظور باید برنامه‌ای مثل EMM386.EXE نصب شده باشد، تا امکان استفاده از XM را در این محدوده فراهم نماید. برای این منظور باید در CONFIG.SYS دستور DOS=HIGH, UMB قرار گرفته باشد. در صورت بودن این فرمانها در CONFIG.SYS، داس چک می‌کند که گرداننده‌ی XMS، یعنی HIMEM.SYS، نصب شده است یا نه؟ در صروت نصب بودن برنامه با استفاده از فرمان DEVICEHIGH یا LOADHIGH می‌توان برنامه‌ها را در UMB بارنمود. البته این به آزاد بودن UMBs بستگی خواهد داشت و چنانچه به مقدار مورد نیاز در UMBs حافظه آزاد نباشد، برنامه در حافظه‌ی متعارف بار خواهد شد.

۳-۲۲ استفاده از UMB

برای استفاده از UMBs در مدیریت حافظه‌ی معمولی و در صورت فعال شدن UMB در CONFIG.SYS، با فراخوانی تابع 48H ممکن است یک بلوک حافظه که جزو UMBs است برگردانده شود، یا آنکه جزو حافظه‌ی متعارف باشد. برای تشخیص اینکه بلوک برگردانده شده، UMB است یا متعارف، کافی است نشانی قطعه‌ی برگردانده شده در AX چک شود. اگر بالاتر از A000H باشد، UMB است. زیرتوابع تابع 58H، کارهای مورد نیاز در این زمینه را انجام می‌دهند. اما کار اصلی 58H، تنظیم یا بررسی مدل تخصیص حافظه است:

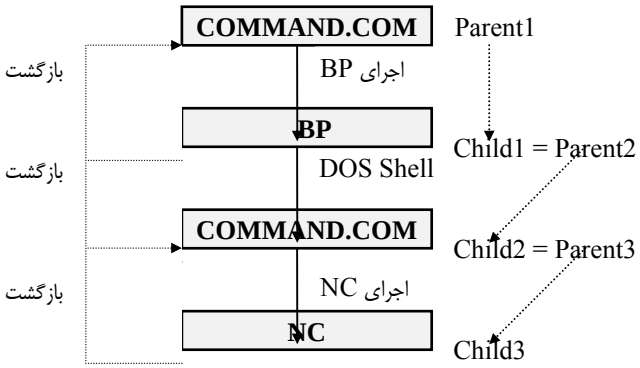
Subfunction	Description
00H	Read memory model
01H	Set memory model
02H	Query UMB status
03H	Set UMB status

فصل ۲۳: تابع Exec
(The Exec Function)

۲۳-۱ مقدمه

تابع شماره‌ی 4BH داس، دارای نام Exec است. این تابع در حقیقت بارکننده[۱] است و به یک برنامه‌ی اصلی[۲] اجازه می‌دهد که یک برنامه فرعی[۳] را از دیسک در حافظه بار نموده و اجرا کند. در صورتی که برنامه‌ی فرعی یک برنامه‌ی مقیم در حافظه (TSR) نباشد، پس از پایان اجرا، حافظه‌ی اشغال شده بوسیله‌ی آن، که بوسیله‌ی Exec تخصیص داده شده، آزاد خواهد شد.

برنامه‌ی فرعی هم می‌تواند برنامه‌ی دیگری را فراخوانی نموده و اجرا کند و در واقع خودش یک برنامه‌ی اصلی باشد. این روش تا جایی که حافظه آزاد باشد، می‌تواند ادامه یابد. یک مثال اصلی از اجرای Exec، برنامه‌ی COMMAN.COM است. وقتی در خط فرمان، تایپ می‌کنیمBP، BP درحقیقت یک برنامه‌ی فرعی نسبت به COMMAND.COM خواهد بود. در داخل BP می‌توانیم DOS Shell کنیم. در این صورت، یک نسخه‌ی دیگر از COMMAND.COM اجرا می‌شود و این نسخه یک برنامه‌ی فرعی نسبت به BP می‌شود. در این شرایط در خط فرمان می‌توانیم برنامه‌های دیگر را فراخوانی کنیم، که آنها نسبت به COMMAND.COM، فرعی خواهند بود...



۲۳-۲ ارتباط برنامه‌های اصلی و فرعی

برنامه‌ی اصلی می‌تواند پارامترهایی(نظیر سوئیچ‌های خط فرمان) را به برنامه‌ی فرعی ارسال کند، که در PSP برنامه‌ی فرعی قرار گرفته و مورد استفاده‌ی آن قرار می‌گیرند. روش اجرا هم مشابه است. یعنی هم برنامه‌های فرعی و هم اصلی، در حافظه پس از PSPشان قرار می‌گیرند. برنامه‌ی فرعی پس از فراخوانی به همه‌ی لوازم سیستم دسترسی خواهد داشت. همچنین با استفاده از توابع دسته می‌تواند به همه‌ی لوازم و فایلهای بازشده بوسیله‌ی برنامه‌های اصلی قبلی، دسترسی داشته باشد. مثلاً فایلی را که قبلاً بازشده، بخواند یا در آن بنویسد. برای این منظور کافیست دسته‌ی فایل را به‌عنوان پارامتر دریافت کند، و نیازی به دانستن نام فایل و بازکردن آن نخواهد داشت. برای ارسال دسته‌ی فایل از برنامه‌ی اصلی به فرعی، از یکی از روشهایی که در ادامه تشریح خواهند شد، می‌توان استفاده نمود. باید توجه کرد که دسترسی به فایل با این روش، اشاره‌گر فایل[۴] را تحت تأثیر قرار خواهد داد و تغییرات برای برنامه‌ی اصلی قابل مشاهده خواهد بود.

پس از برگشت به برنامه اصلی، از نقطه‌ی فراخوانی و مشابه فراخوانی روالها، ادامه‌ی برنامه اصلی اجرا خواهد شد. برای برگرداندن وضعیت اجرا، برنامه‌ی فرعی یک کد عددی را به برنامه اصلی برمی‌گرداند، که برای این منظور باید تابع 4CH وقفه‌ی 21H را فراخوانی نمود، که برنامه‌ی اصلی را خاتمه داده و یک کد برمی‌گرداند و باید قراردادی در زمینه‌ی تفسیر این کد بین برنامه‌ی اصلی و فرعی وجود داشته باشد. برنامه‌ی اصلی هم برای چک کردن کد، تابع 4DH را باید فراخوانی نماید، که مقدار AH نشاندهنده‌ی وضعیت ختم برنامه‌ی فرعی است. عموماً AH=0 برای ختم عادی[۵] است و مقدار AH=1 نشان‌دهنده‌ی حالتی است که برنامه‌ی فرعی با زدن کلیدهای Ctrl+C یا Ctrl+Break قطع شده باشد. همچنین AH=2 برای حالتی‌است که خطا در دسترسی به دیسک اتفاق افتاده باشد و AH=3 نشان‌دهنده‌ی این است که برنامه با فراخوانی یکی از توابع 31H یا 27H، در حافظه مقیم شده و یک برنامه‌ی TSR بوده است.

---

۱ loader
۲ parent program
۳ child program
٤ file pointer
٥ normal termination

اگر برنامه‌ی اصلی از نوع EXE. باشد، در صورتیکه حافظه به اندازه‌ی کافی آزاد باشد، می‌تواند برنامه‌های فرعی را اجرا کند. اما برنامه‌های COM. چون کل حافظه را اشغال می‌کنند، نمی‌توانند برنامه‌ی دیگری را اجرا کنند، مگر آنکه قبلاً با فراخوانی 49H حافظه‌های غیرضرور را آزاد نمایند.

۲۳-۳ روشهای ارسال پارامتر

سه روش برای ارسال پارامتر از برنامه‌ی اصلی به فرعی وجود دارد:

- خط فرمان[۱]،
- بلوک محیط[۲]،
- PSP،

قبل از فراخوانی تابع Exec یا 4BH، باید پارامترهای لازم به آن ارسال شود:

AH = 4BH

AL = 0 or 3 (0: File is executable, 3: File is overlay)

DS:DX = Address of .EXE or .COM program name(null-terminalted string)

ES:BX = Address of parameter block

فایل BAT را نمی‌توان با این روش فراخوانی نمود و باید COMMAND.COM را با پارامتر ”/C“ فراخوانی نمود:

Exec(“COMMAND.COM”, “/C PROG.BAT”)

بلوک پارامتر[۳]: پارامترها را می‌توان از طریق بلوک پارامتر به برنامه‌ی فرعی ارسال نمود. نشانی بلوک پارامتر در ES:BX به تابع Exec ارسال می‌شود. ساختار بلوک پارامتر به‌صورت زیر است:

Field	Addr.	Contents
1.	0-1	Segment address of Environment Block
2.	2-3	Offset address of command parameter
3.	4-5	Segment address of command parameter
4.	6-7	Offset address of first FCB
5.	8-9	Segment address of first FCB
6.	10-11	Offset address of second FCB
7.	12-14	Segment address of second FCB

* فیلد ۱ نیازی به نشانی مبدأ ندارد، چون همیشه از نشانی مضرب ۱۶ شروع می‌شود و لذا مبدأ آن صفر است.

بلوک محیط: پردازشگر فرمان و سایر برنامه‌ها، اطلاعاتی را از بلوک محیط بدست می‌آورند، که یک سری رشته است و دارای نحو زیر است و با دستور DOSSET می‌توان آنرا تنظیم نمود و حداکثر می‌تواند 32K باشد:

Name = parameter

* فیلدهای ۲ و ۳ نشانی پارامتر فرمان را مشخص می‌کنند که در مبدأ PSP 80H برنامه قرار می‌گیرد.

* فیلدهای بعدی نیز نشانی‌های مبدأ و قطعه‌ی FCB های برنامه هستند.

۲۳-۴ وظایف EXEC

کار مهم Exec، تصحیح نشانی‌های جابجاپذیر[۴] برنامه‌های EXE است، که با توجه به اطلاعاتی انجام می‌شود که کامپایلر یا اسمبلر در سرآیند فایل EXE در قسمت جدول جابجایی[۵] قرار داده است. کار دیگر، مقداردهی CS، IP، SS و SP است. آنگاه، برنامه را اجرا می‌کند، که این کار با مقداردهی IP و پرش به نشانی ابتدای کد اجرایی برنامه انجام می‌شود. پس از اجرای برنامه، براساس نوع خاتمه‌ی برنامه به CF مقدار 0 یا 1 می‌دهد. در موقع فراخوانی Exec، اگر AL=3 باشد، به معنی **همپوشا[۶]** بودن فایل EXE یا COM است و بنابر این آنرا در حافظه بارمی‌کند، ولی آنرا اجرا نمی‌کند، بلکه پس از بارکردن، به برنامه‌ی

---

۱ command line
۲ environment block
۳ paramter block
٤ relocatable
٥ relocation table
٦ overlay

فراخواننده برگشت می‌شود. برنامه‌های همپوشا، در هرجای حافظه که Exec بخواهد بارنمی‌شوند، بلکه در جایی بار می‌شوند که فراخواننده‌ی Exec، نشانی آنرا بدهد. در این حالت بلوک پارامتر شامل اطلاعات زیر است:

1. 0-1 Segment address where overlay is located
2. 2-3 Relocation factor

اگر برنامه‌ای آنقدر بزرگ باشد که در حافظه متعارف(640KB) قابل بارشدن نباشد، قسمتهای همپوشای برنامه در حافظه‌ی یکسانی بار می‌شوند. نظیر فرمهای ورودی یا گزارشهای مختلفی که در یک برنامه تجاری وجود دارند و نیاز نیست که به‌طور همزمان در حافظه قرار داشته باشند. بنابراین، برنامه قسمتی از حافظه را از heap تخصیص داده و قسمتهای همپوشای برنامه را در آن بارمی‌کند. روالهای موجود در قسمتهای همپوشا درصورتی قابل فراخوانی می‌شوند که در حافظه بارشده باشند و قبل از آن، نشانی‌های معتبری نیستند. برای فراخوانی توابع، باید از FAR CALL استفاده نمود.

* reloaction factor: نشانی قطعه‌ی برنامه‌ی فراخوانی‌شده(یکی از قسمتهای همپوشای برنامه) را تنظیم می‌کند و فقط برای برنامه‌های EXE معنی دارد و برای برنامه‌های COM صفر است.

برنامه‌ای که همپوشا باشد و به این ترتیب بارشود، قبل از آن PSP قرار نمی‌گیرد. این کار درمورد برنامه‌های COM مشکل‌آفرین است. زیرا درحالت عادی برنامهCOM از بایت 100H اجرا می‌شود، ولی در این حالت چون PSP بارنمی‌شود، نشانیهای jump برنامه‌ی COM اشتباه خواهند بود. از آنجایی که همه‌ی دستورات jump دسترسی به داده‌ها در درون برنامه‌ی COM نسبت به نشانی 100H هستند و نه صفر، نمی‌توان دستور FAR CALL را با قطعه‌ای که برنامه در آن بارشده به‌عنوان نشانی قطعه و مقدار صفر به‌عنوان نشانی مبدأ، فراخوانی نمود. نشانی قطعه برای FAR CALL باید نشانی قطعه‌ای که برنامه در آن بارشده منهای 10H (طول سرآیند فایل COM) و 100H به‌عنوان مبدأ باشد. اگر برنامه‌ی COM به‌عنوان همپوشای یک برنامه‌ی دیگر باشد، باید دارای ورودیهای نشانی[۱] غیر از 100H در داخل خود باشد و لذا فقط کافی است نشانی‌های مبدأ دستورات FAR CALL تغییر داده شوند.

* در برنامه‌های همپوشایی که کامپایلرهای بورلند تحت داس ایجاد می‌کنند، یک "Overlay Manager" به برنامه متصل می‌شود که کارهای مدیریتی مورد نیاز را انجام می‌دهد و نشانیها را پس از بارکردن در حافظه تصحیح می‌کند.

* تابع Exec در کامپایلرهای پاسکال و C بورلند تحت داس وجود دارد:

Borland Pascal: DOS Unit

```
procedure Exec(Path, CmdLine : String);
```

Borland C: <process.h>

exec... functions:

Enable your program to load and run other files (child processes)

Declaration:

```
int execl  (char *path, char *arg0, ..., NULL);
int execle (char *path, char *arg0, ..., NULL, char **env);
int execlp (char *path, char *arg0, ...);
int execlpe(char *path, char *arg0, ..., NULL, char **env);
```

spawn... functions:

Enable your programs to run other files (child processes). spawn... functions return control to your program when the child processes finish.

Declaration:

```
int spawnl  (int mode, char *path, char *arg0, ..., NULL);
int spawnle (int mode, char *path, char *arg0, ..., NULL, char *envp[]);
int spawnlp (int mode, char *path, char *arg0, ..., NULL);
int spawnlpe(int mode, char *path, char *arg0, ..., NULL, char *envp[]);
```

system:

Issues a DOS command

Declaration:

```
int system(const char *command);
```

[۱] entry address

فصل ۲۵: گردانندههای وسیله
(Device Drivers)

۲۵-۱ مقدمه

یکی از پیچیدهترین کارهای برنامهنویسی سیستم، نوشتن گردانندههای وسیله[۱](DD) است. گردانندههای وسیله در داس برای دسترسی به لوازم خارجی مورد استفاده قرار میگیرند و برنامههای کوچکی هستند که وسایل مختلفی، از صفحهکلید گرفته تا CD-ROM را پشتیبانی میکنند. به دلیل ساختار و محدودیتهای خاص، گردانندههای وسیله را در داس تنها با زبان اسمبلی میتوان نوشت. در این فصل، ساختار گردانندههای وسیله را مورد مطالعه قرار میدهیم.

۲۵-۲ گردانندههای وسیله تحت داس

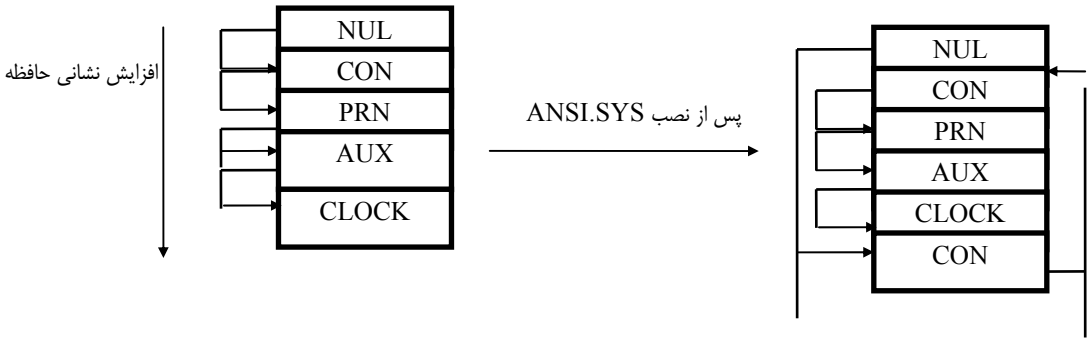
DD بخشی از همهی سیستمهای عامل است و مسئول کنترل و برقراری ارتباط با سختافزار بوده و سطحپایینترین بخش سیستم عامل است و به سایر بخشها، ازقبیل هسته و برنامههای کاربردی اجازه میدهد که مستقل از سختافزار باشند. برای آنکه یک سیستم عامل قابل حمل[۲] بر روی کامپیوترهای مختلف باشد، باید مبتنی بر DD باشد. برای حمل یک سیستم عامل به کامپیوترها و معماریهای مختلف کافی است که بخش DD بازنویسی شود. در سیستمهای عامل قدیمی، DD جزیی از سیستم عامل بود و از اینرو با تغییر سختافزار، سیستم عامل باید عوض میشد. البته داس، با دید قابل حمل بودن ایجاد نشده است، زیرا بسیاری از بخشهای آن مبتنی بر BIOS هستند و از این نظر به معماری PC محدود است.

در DOS 2.0، مفهوم DD معرفی شد و به این ترتیب امکان استفاده از EMS و HD فراهم شد. یک DD شامل اطلاعات وضعیت و روالهایی بهعنوان توابع گرداننده است. این روالها مسئول انجام وظایف مورد نیاز DOS برای دسترسی به وسیله هستند. برای نمونه، گردانندهی یک دیسک سخت، دارای توابعی برای خواندن، نوشتن، و وارسی بخشها بر روی دیسک است.

گردانندههای نوشته شده بوسیلهی کاربر: ارتباط داس با DD مبتنی است بر یک ساختمان داده و برخی فراخوانی توابع ساده، برنامهنویسان زبان اسمبلی میتوانند برای هروسیلهای یک DD بنویسند. متأسفانه تحت داس نمیتوان DD را با زبانهای سطح بالا نوشت. DDها از برخی جهات مشابه برنامههای COM هستند(نشانیهای جابجاپذیر ندارند)، اما از نشانی 00H شروع میشوند نه 100H، یعنی قبل از آنها PSP قرار نمیگیرد.

DDهای نوشته شده بوسیلهی کاربر توسط داس و در هنگام بوتشدن آن، از طریق اطلاعات فایل CONFIG.SYS نصب میشوند و همانند برنامههای COM و EXE از خط فرمان قابل اجرا نیستند. برخی DDها نیز در هستهی داس وجود دارند و بهطور خودکار نصب میشوند. اینها عبارتند از: $CLOCK، AUX،CON، PRN. به علاوه، گردانندههای HD و FD هم بهطور خودکار نصب میشوند.

DDها بهطور ترتیبی در حافظه بارشده و به همدیگر مرتبط میشوند و تشکیل یک زنجیره را میدهند. در شکل زیر این زنجیره نشان داده شده است:



اگر بخواهیم گردانندهی جدیدی نصب کنیم، باید دستوری بهشکل زیر در CONFIG.SYS قرار دهیم:

DEVICE=NEWDD.DRV

---

۱ device drivers
۲ portable

برای مثال ANSI.SYS که گردانندهی جدیدی را برای ورودی-خروجی کاراکتر، و در واقع یک CON جدید را فراهم میکند، به صورت زیر نصب میشود:

```
DEVICE=ANSI.SYS
```

با قرار گرفتن آن در زنجیرهی DDها، هرگاه که از طریق داس بخواهیم کاراکتری را بر روی صفحه بنویسیم، داس چون زنجیرهی DDها را دنبال میکند، ابتدا تابع تعریف شده در CON جدید را پیدا میکند و آنرا اجرا میکند.

همهی DDها را نمیتوان همانند CON تعویض نمود. مثلاً NUL همیشه نخستین DD بوده و اگر DD جدیدی برای آن ایجاد شود، داس از آن صرفنظر خواهد نمود. همین مسأله در مورد گردانندههای HD و FD هم صادق است. میتوان گردانندهی جدیدی برای یک وسیلهی ذخیرهسازی انبوه اضافه نمود، ولی نمیتوان گردانندهی C: یا A: را تعویض نمود.

۳-۲۵ انواع گردانندهها

دو نوع DD در داس وجود دارد: کاراکتری(CDD)[۱] و بلوکی(BDD)[۲]

الف)گردانندهی وسیلهی کاراکتری:

این نوع DD دارای ساختار سادهای است و با هر بار فراخوانی یک کاراکتر را منتقل میکنند و با وسایلی نظیر صفحهکلید، صفحه نمایش، چاپگر و مودم کار میکنند. یک CDD فقط یک وسیله را میتواند کنترل کند. بنابر این برای وسایل مختلف باید CDDهای متفاوت در داس نصب شود. این نوع DD دارای دو حالت عملیاتی است:

- **حالت Cooked:** کاراکترها قبل از انتقال پردازش میشوند و آنگاه تحویل برنامهها میشوند. برای مثال یک گردانندهی صفحهکلید، کلیدهایی نظیر <ENTER> یا <CTRL+P> را پردازش و کنترل میکند و با زدن اولی، از کلیدهای بعدی زده شده، صرفنظر میکند و در مورد دومی، همهی حرفهای زدن شده را علاوه بر نمایش، بر روی چاپگر هم چاپ میکند.
- **حالت Raw:** در این حالت کاراکترهای کنترلی پردازش نمیشوند و تمامی کاراکترها از وسیله به برنامه و برعکس منتقل میشوند. اگر برنامه بخواهد ۱۰ کاراکتر بخواند، دقیقاً همان ۱۰ کاراکتر به او تحویل داده میشود و هیچ پردازشی بر روی کاراکترها انجام نمیشود.

ب) گردانندهی وسیلهی بلوکی:

این نوع گردانندهها عموماً با لوازم ذخیرهسازی انبوه ارتباط برقرار میکنند، که FDD و HDD دو نمونهی مشخص از آن هستند. این نوع گردانندهها هر بار یک یا چند بلوک از دادهها را منتقل میکنند. اندازهی بلوک در مورد وسایل مختلف متفاوت است، برای نمونه در مورد HD و FD یک بخش یا 512 بایت است. گردانند وسیله در مورد وسایل بلوکی وظیفهی تبدیل شماره بلوکهای منطقی به بلوکهای فیزیکی را دارد. برای مثال در مورد HD باید شمارهی بخش منطقی[۳] را به شمارهی هد، سیلندر، و بخش فیزیکی تبدیل کند. برخلاف CDD، یک BDD میتواند در هر زمان به چند وسیلهی هم نوع، سرویس دهد. مثلاً ولومها[۴] یا درایوهای مختلف یک HD یک چند HD را میتوان با یک گردانندهی HD کنترل نمود.

نحوهی تشخیص وسایل مدیریت شونده با یک BDD به این صورت است که از نام وسیله(مثل CON یا PRN) استفاده نمیشود، بلکه با یک حرف، نظیر A، B، C، D و غیره مشخص میشوند، که نمیتوانند تکراری باشند، ولی باید بهصورت پیدرپی باشند. هر BDD باید یک FAT و یک دایرکتوری ریشه داشته باشد و بر خلاف CDD حالتهای عملیاتی Cooked و Raw ندارند.

دسترسی به گردانندهها:

راههای مختلفی برای دسترسی به گردانندهها وجود دارند. به CDD میتوان با استفاده از توابع FCB یا handle دسترسی پیدا نمود، که اسم وسیله بهجای اسم فایل مورد استفاده قرار میگیرد. اما به BDDها میتوان با استفاده از توابع داس برای فهرست، فایل و غیره دسترسی پیدا نمود. روش دیگر دسترسی، فراخوانی تابع 44H داس است که به IOCTL[۵] معروف است و دارای تعدادی زیر تابع است که در مورد آن در ادامهی این فصل صحبت میشود.

۴-۲۵ ساختار گردانندههای وسیله

گرچه CDD و BDD با هم متفاوتند، اما هر دو دارای ساختار واحدی هستند. هر DD دارای اجزاء زیر است:

---

۱ Character Device Driver
۲ Block Device Driver
۳ logical sector
٤ volume
٥ I/O Control

- **Device header:** در ابتدای هر DD وجود دارد و شامل اطلاعات زیر برای نصب وسیله توسط داس است:

Address	Contents	Type
00H	Offset address of next driver	1 word
02H	Segment address of next driver	1 word
04H	Device attribute	1 word
06H	Offset address of strategy routine	1 word
08H	Offset address of interrupt routine	1 word
0AH	Driver name(for CDD) or Number of device(for BDD)	8 bytes

* فیلد سوم، یک word است که بیت ۱۵ آن، نوع DD را مشخص میکند و برای CDD=1 و برای BDD=0 است.

- **Strategy routine:** برای مقداردهی اولیهی DD توسط داس در هنگام فراخوانی توابع گرداننده و قبل از اجرای روال وقفه، فراخوانی میشود.
- **Interrupt routine:** با روالهای معمولی وقفه متفاوت است و در حقیقت توابع گرداننده توسط آن انجام میشود و پس از روال استراتژی فراخوانی میشود.

۲۵-۵ توابع گردانندههای وسیله

تحت DOS 2.0، هر DD قابل نصب میبایست ۱۳ تابع را پشتیبانی میکرد، که دارای شمارههای 00H تا 0CH بودند، حتی اگر مجبور بودند که دارای بدنهی خالی باشند و یک DONE flag را مقداردهی کنند. اما تحت نگارشهای بعدی داس، توابع دیگری نیز میتوانند بهطور اختیاری وجود داشته باشند.

در ادامه انواع توابع و کاربرد آنها تشریح میشود:

Function 00H: Driver initialization

در هنگام نصب گرداننده توسط داس فراخوانی شده و DD را مقداردهی اولیه میکند و برای این منظور کارهایی چون (۱)مقداردهی اولیهی سختافزار (۲)مقداردهی اولیهی متغیرهای داخلی DD (۳)تعویض وقفهها را انجام میدهد.

از آنجایی که در زمان نصب DD، داس بهطور کامل بالا نیامده است، روال مقدار دهی میتواند تنها توابع 01H تا 1CH داس (برای ورودی-خروجی کاراکتر)، تابع 30H (تعیین نگارش داس)، و توابع 25H و 35H (Get/Set Int. Vect) را فراخوانی کند، و این دلیل اصلی عدم امکان نوشتن DD با زبانهای سطح بالا است.

Function 01H: Media check

فقط با BDD کاربرد دارد و برای نمونه در مورد FD، تعویض دیسکت را بررسی میکند.

Function 02H: Build BIOSParameter Block (BPB)

فقط با BDD کاربرد دارد و یک جدول ایجاد میکند که اطلاعات موردنیاز BIOS برای دسترسی به وسیله در آن قرار میگیرد. این اطلاعات شامل Media Descriptor Byte و نشانی بافر شامل FAT است.

Function 03H: I/O control read

ارتباط مستقیم بین DD و کاربردها را امکانپذیر نموده و انتقال اطلاعات از وسیله به برنامهی کاربردی را میسر میکند. بیت ۱۴ در attribute word باید به 1 مقداردهی شود، تا وسیله فقط از طریق تابع 44H وقفهی 21H (IOCTL) قابل دسترسی شود.

Function 04H: Read

فقط با BDD کاربرد دارد و دادهها را از وسیله خوانده و در یک بافر که نشانی آن داده میشود، قرار میدهد.

Function 05H: Non-destructive read

فقط با CDD کاربرد دارد و داده را از وسیله خوانده، ولی آنرا از بافر وسیله برنمیدارد و با فراخوانی مجدد این تابع یا تابع 04H قابل خواندن است.

Function 06H: Input status

برای تعیین آماده بودن داده در بافر ورودی یک CDD بهکار میرود.

Function 07H: Flush input buffers

بافرهای ورودی یک CDD را خالی میکند و دادههای موجود در آنرا دور میریزد(مثل کلیدهای بافر صفحهکلید).

Function 08H: Write

داده‌ها را از بافر در وسیله می‌نویسد و خطای احتمالی را در Status word نشان می‌دهد.

Function 09H: Write with verify

با BDD کاربرد و دارد داده‌های نوشته شده در وسیله را دوباره خوانده و وارسی می‌کند.

Function 0AH: Output status

بررسی می‌کند که آیا آخرین کاراکتر نوشته شده در CDD تکمیل شده است یا نه؟

Function 0BH: Flush output buffers

کاراکترهای نوشته شده در بافر خروجی را که هنوز به وسیله منتقل نشده‌اند را پاک می‌کند.

Function 0CH: I/O control write

انتقال اطلاعات از برنامه‌ی کاربردی به وسیله را به‌طور مستقیم و با تابع IOCTL اجازه می‌دهد.

Function 0DH: Open

در مورد CDD هربار فراخوانی می‌شود، ولی درمورد BDD در موقع دسترسی اولیه، مثلاً بازکردن فایل فراخوانی شده و وسیله را برای نقل و انتقال آماده می‌کند.

Function 0EH: Device close

بستن یک وسیله‌ی و انتقال فیزیکی داده‌های موجود در بافرهای خروجی به وسیله.

Function 0FH: Removable media

در مورد BDD کاربرد دارد و قابل تعویض بودن آنرا کنترل می‌کند(نظیر FD).

Function 10H: Output until busy

اطلاعات را تا زمانی که وسیله مشغول نشده از بافر به وسیله منتقل می‌کند.

Function 11H: Get logical device

با BDD به‌کار می‌رود و اطلاعات منطقی در مورد وسیله را بدست می‌دهد. مثلاً در مورد FD تعیین می‌کند که فرمت آن double-density است یا high-density.

Function 12H: Set logical device

با BDD به‌کار می‌رود و اطلاعات منطقی در مورد وسیله را تنظیم می‌کند. مثلاً در مورد FD، نوع فرمت آنرا تنظیم می‌کند.

۲۵-۶ گرداننده‌ی ساعت

یکی از گردانندههای اصلی داس است که نام آن $CLOCK بوده و به‌طور خودکار نصب می‌شود و یک CDD است. می‌توان از نام دیگری نیز به‌جای $CLOCK برای آن استفاده نمود، اما باید در device header، بیت دوم attribute word به 1 مقداردهی شود. همچنین چون CDD است، بیت ۱۵ آن نیز 1 است.

توابع 2AH تا 2DH داس، برای تنظیم و نمایش تاریخ و زمان توابع این گرداننده را فراخوانی می‌کنند. این CDD فقط دارای توابعی برای نقل و انتقال تاریخ است، و از اینرو توابع 04H (Read)، 08H (Write) و 00H (Initialization) باید فراهم شوند و مابقی باید خالی باشند. با فراخوانی تابع 04H، تاریخ و زمان از گرداننده به داس منتقل می‌شود و داس نیز با فراخوانی تابع 08H، تاریخ و زمان جدید را تنظیم می‌کند. پارامترهای زمان و تاریخ، در یک بافر که ۶ بایت است، منتقل می‌شوند:

Address	Contents	Type
+00H	Number of days since Jan. 1, 1980	1 word
+02H	Minutes	1 byte
+03H	Hours	1 byte
+04H	Hundredth of seconds	1 byte
+05H	Seconds	1 byte

* توابع 04H و 08H گردانند هم از توابع 00H و 01H وقفه‌ی 1AH برای تنظیم زمان استفاده می‌کنند.

* نحوه‌ی نگهداری تاریخ (بر اساس تعداد روزها از اول ژانویه ۱۹۸۰) غیر معمول است و داس باید روز/ماه/سال را با درنظر گرفتن تمامی جوانب تبدیل، از قبیل سالهای کبیسه به تعداد روز و برعکس تبدیل نماید.

۷-۲۵ انواع دسترسی به گرداننده

۱) دسترسی غیرمستقیم از طریق داس

وقتی یک گرداننده نصب می‌شود، یک وسیله‌ی فیزیکی با یک نام منطقی به سیستم متصل شده و ارتباط با آن میسّر می‌شود. مثلاً اگر CD-ROM نصب کرده‌ایم، درایو :E به آن منسوب می‌شود و دسترسی به :E انجام می‌دهیم، داس بر اساس نوع دسترسی، یکی از توابع تشریح شده را که در کد DD وجود دارد، فراخوانی نموده و اجرا می‌کند. مثلاً اگر دسترسی، از نوع خواندن است، تابع read مخصوص گرداننده که در کد DD وجود دارد فراخوانی می‌شود.

۲) دسترسی مستقیم از طریق IOCTL

روش دیگری برای ارتباط با DD است و درصورتی که بیت ۱۴ در attribute word به 1 مقداردهی شده باشد، امکان‌پذیر می‌شود و از طریق تابع 44H داس که به IOCTL معروف است، می‌توان به DD دسترسی پیدا نمود و کارهای زیر را انجام داد:

- پیکره‌بندی وسیله[1]
- انتقال داده‌ها[2]
- وضعیت گرداننده

IOCTL دارای تعداد زیادی زیرتابع است که شماره‌ی آن باید در AL پاس شود و همانند همه‌ی توابع داس، پس از اجرا CF نشان‌دهنده‌ی وقوع یا عدم وقوع خطا خواهد بود. در ادامه مختصری در مورد این توابع توضیح داده می‌شود:

* زیر توابع 06 و 07 وضعیت CDD را نشان می‌دهند. اولی می‌گوید که آیا وسیله قادر به ارسال داده است یا نه؟ و دومی می‌گوید که آیا وسیله قادر به دریافت داده است یا نه؟ handle وسیله باید در BX ارسال شود و اگر وسیله آماده باشد، مقدار FFH در AL برگردانده می‌شود.

* زیرتابع 01 برای انتخاب حالتهای عملیاتی Cooked و Raw در CDD استفاده می‌شود.

* زیرتابع 02 داده‌های کنترلی را از CDD می‌خواند.

* زیرتابع 03 داده‌های کنترلی را بر روی CDD می‌نویسد.

* زیرتابع 04 داده‌های کنترلی را از BDD می‌خواند.

* زیرتابع 05 داده‌های کنترلی را بر روی BDD می‌نویسد.

* زیرتابع 00 برای بدست آوردن اطلاعات وسیله برای یک handle در BDD است. چون ممکن است چند handle در وسیله فعال باشد.

۸-۲۵ نکاتی درباره‌ی نوشتن DD

نوشتن DD از مشکل‌ترین بخشهای برنامه‌سازی سیستم است و مشکلات زیر ممکن است در موقع تست آن پیش آید:

- نشانی بارشدن DD در حافظه توسط داس تعیین می‌شود و در زمان تست و اشکالزدایی برای برنامه‌نویس نامشخص است.
- گردانندهی جدید برای CON را نمی‌توان با برنامه‌ی debug داس، تست کرد. چون خود debug از CON استفاده می‌کند.
- وقتی یک DD می‌نویسید، یک برنامه‌ی تست برای تک تک توابع با روشی مشابه داس بنویسید و قبل از نصب DD همه‌ی توابع آنرا تحت کنترل خود تست کنید.
- BDDها را تنها پس از تست کامل در سیستم نصب کنید. زیرا ممکن است مشکلاتی مثل تداخل با سایر لوازم و مثلاً خراب کردن partition table دیسک سخت و غیره پیش آید.

۹-۲۵ برنامه‌های EXE به عنوان DD

طی سالهای اخیر، برخی گردانندهها به‌صورت فایلهای EXE ایجاد شده‌اند و به وسیله‌های مربوطه هم می‌توان از طریق گردانندههای نصب شده در زمان بوت شدن داس و معرفی شده در CONFIG.SYS، دسترسی پیدا نمود و هم از طریق برنامه‌های EXE که تحت خط فرمان داس و پس از بوت شدن داس اجرا می‌شوند. برای مثال گردانندههای ماوس یک EMS این‌گونه هستند. این گونه گردانندهها، گردانندههای واقعی نیستند و از ساختار گفته شده برای DD تبعیت نمی‌کنند، بلکه یک برنامه‌ی TSR هستند. اغلب این گونه گردانندهها، CDD هستند و داس حرف درایو به آنها منسوب نمی‌کند. برای آنکه داس آنها را به عنوان گرداننده بشناسد، باید دارای تابع 00H باشند تا گرداننده را مقداردهی اولیه نمایند.

---

[1] device configurartion
[2] data transmision

۱۰-۲۵ گرداننده‌های CD-ROM

از نظر داس، CD-ROM که یک وسیله‌ی ذخیره‌سازی انبوه است، باید یک گرداننده‌ی وسیله‌ی بلوکی داشته و دارای FAT باشد. این در حالی‌است که فرمت ذخیره‌سازی داده‌ها در CD-ROM به صورتهای خاصی بوده و برای خواندن ونوشتن داده‌ها، بلوکهایی با اندازه‌ی متفاوت با دیسک سخت، و مثلاً بلوکهای 2KB استفاده می‌شود. اطلاعات فایلها هم به صورت FAT نگهداری نمی‌شود، بلکه از روشهای خاص، نظیر استانداردهای ISO استفاده می‌شود. از اینرو، باید به‌جای BDD از CDD برای دسترسی به CD-ROM استفاده نمود، که در این صورت یک حرف درایو به آن منسوب نخواهد شد. اما، برای آنکه بتوان از این وسیله به عنوان یک وسیله‌ی ذخیره‌سازی انبوه و همانند وسایل بلوکی نظیر HD استفاده نمود، و یک حرف درایو به آن منسوب کرد، باید واسطهایی وجود داشته باشد.

برای نصب گردانندهی CD-ROM، باید خطی مشابه زیر به CONFIG.SYS اضافه شود:

```
DEVICE=HITACHI.SYS  /D:CDR1
```

که گردانندهٔ برای CD-ROM ساخت HITACHI را نصب می‌کند و این CDD با نام CDR1 به سیستم شناسانده می‌شود. اما همانگونه که گفته شد، ترجیح می‌دهیم که از طریق داس و همانند یک ولوم HD به آن دسترسی داشته باشیم. برای این منظور، باید برنامه‌ی MSCDEX[1] را اجرا کنیم:

```
MSCDEX.EXE /D:CDR1 /V /M:8 /L:E
```

/L:E حرف درایو :E را به CDR1 منسوب می‌کند. در این صورت، برنامه‌ی MSCDEX، دسترسی CDD را به صورت BDD امکان‌پذیر می‌کند.

[1] Microsoft CD Extension

فصل ۲۶: سیستم فایل داس
(DOS File System)

۲۶-۱ ساختار مبنایی سیستم فایل

سیستم فایل داس مبتنی بر ساختار ولوم[۱] است. از دیدگاه کاربر، داس به لوازم ذخیره‌سازی انبوه[۲] تحت عنوان ولوم دسترسی پیدا نموده و به هر ولوم مجزا یک حرف درایو منسوب می‌کند، که برای دیسکتها، A: و B: و دیسکهای سخت، C: و D: است. قبل از DOS 3.3، هر ولوم می‌توانست دارای اندازه‌ای معادل حداکثر 32MB باشد. از اینرو، دیسکهای سخت با ظرفیت بالاتر، به تعدادی ولوم منطقی 32MB تقسیم‌بندی[۳] می‌شدند.

هر ولوم دارای ساختار منطقی خاصی بوده، که به دیسکت یا دیسک بودن و میزان ظرفیت آن بستگی ندارد و فقط اندازه‌داده‌های مدیریتی ساختار، تفاوت خواهد داشت. به هر ولوم می‌توان یک نام برچسب ولوم[۴] منسوب نمود. هر ولوم حتماً دارای یک فهرست ریشه و احتمالاً تعدادی زیرفهرست است.

بخشها[۵]: داس هر ولوم را یک سری بخش پشت‌سر هم می‌بیند. هر بخش ۵۱۲ بایت است و دارای یک شماره‌ی بخش منطقی است که به‌صورت پی‌درپی و از 0 شماره‌گذاری می‌شوند. برای نمونه، یک ولوم ۱۰ مگابایتی 20,480 بخش دارد، که شماره‌هایش از 0 تا 20,479 هستند. داس کنترلی بر مدیریت فیزیکی بخشها ندارد و این کار توسط گرداننده‌های وسیله دیسک سخت[۶] انجام می‌شود و کار تبدیل بخشهای منطقی به فیزیکی توسط HDDD انجام می‌شود.

FAT: توابع داس برای مدیریت فایلها، دسترسی به فایل را تبدیل به دسترسی به بخشهای منطقی می‌کنند و برای این منظور از ساختار جدول تخصیص فایلها[۷] یا FAT استفاده می‌شود، که در ادامه در باره‌ی آن مفصلاً صحبت می‌شود.

ساختار لوازم ذخیره‌سازی انبوه: ساختار اغلب لوازم ذخیره‌سازی انبوه به‌صورت زیر است:

Sector 0:
- Manufacturer's name, device drivers, boot routine
- First FAT
- One or more copies of FAT
- Root directory with volume label names
- Data for files directories
- ...
- ...

ساختار فوق توسط فرمان Format ایجاد می‌شود.

Boot Sector: بخش صفر، بخش بوت‌کننده است و هر ولوم دارای یک بخش صفر است و در صورتی که این بخش حاوی روالهای بوت‌کننده‌ی سیستم عامل باشد، به آن بوت‌کننده[۸] گفته می‌شود. این بخش حاوی اطلاعات موردنیاز برای دسترسی به ناحیه‌های مختلف و ساختارهای داده‌ای ولوم است، و به شکل زیر است:

Address	Contents	Type
00H	Jump to boot routine (E9xx or EBxx90)	3 bytes
03H	Manufacturer's name and version no.	8 bytes
0BH	Bytes per sector	1 word

---

۱ volume
۲ mass storage devices
۳ partitioning
٤ volume label name
٥ sectors
٦ HD device drivers
۷ File Allocation Table
۸ bootable

0DH	Sector per cluster	1 byte
0EH	Number of reserved sectors	1 word
10H	Number of FATs	1 byte
11H	Number of enteries in root directory	1 word
13H	Number of sectors in volume	1 byte
15H	Media descriptor	1 byte
16H	Number of sectors per FAT	1 word
18H	Sectors per track	1 word
1AH	Number of read/writes heads	1 word
1CH	Number of hidden sectors	1 word
1EH-1FFH	Boot routine	483 bytes

اولین فیلد سه‌بایت بوده و دستورالعمل پرش است که کد آن، EBxx (short jump) یا EBxx90 (normal jump) است که اولی دو بایتی است و دومی سه‌بایتی است و در حالت اول پس از EBxx، کد دستورالعمل یک بایتی NOP در بایت سوم قرار می‌گیرد. در فیلد دوم، نام سازنده‌ی برنامه‌ی فرمت، نظیر MS یا PCTools و غیره قرار می‌گیرد.

FAT ۲-۲۶

ساختاری است که اطلاعات مربوط به بخشهای یک فایل در آن نگهداری می‌شوند. هر ورودی[۱] در FAT، که ۱۶بیت یا یک word است، متناظر با یک کلاستر[۲] یا واحد تخصیص[۳] است که شامل تعدادی بخش پشت سر هم است. نشانی 0DH در بخش صفر، تعداد بخشها در هر کلاستر را معین می‌کند و باید توانی از ۲، نظیر 1، 2، 4 یا 8. باشد:

Device	Sectors per cluster
Single sided FD	1
Double sided FD	2
AT HD	4
XT HD	8

ولی علی‌رغم جدول فوق، برنامه‌های فرمت به مقادیر فوق محدود نیستند و در مورد ولومهای بزرگتر از 32MB باید مقادیر بیشتری در نظر گرفته شود. زیرا تعداد کلاسترها به 65536 محدود بوده و با آن نمی‌توان ولومهای بزرگ را با FAT ۱۶بیتی مشخص نمود. به این دلیل در DOS 4.0 به بعد، مفهوم فوق گسترش داده شد، بدون آنکه منطق کار عوض شود، و به این ترتیب در هر کلاستر می‌توان تعداد بخشهای بیشتری را مشخص نمود، که برای ولومهای 32MB تا 2GB به‌صورت زیر است:

Volume cluster sizes of DOS 4.0

Volume size:	128M	256M	520M	1GB	2G
Cluster size:	2K	4K	8K	16K	32K
Sectors per cluster:	4	8	16	32	64

زیاد شدن تعداد بخشها در هر کلاستر باعث می‌شود که برای فایلهای کوچک، حجم حافظه‌ی زیادی تلف شود و این امر در صورتی که تعداد فایلهای کوچک زیاد باشد، قابل ملاحظه‌تر خواهد بود.

تکه‌تکه‌شدن فایل[۴]: روش داس برای نگهداری فایلها در قالب تعدادی بخش بر روی دیسک، مشکلاتی را در زمینه‌ی سرعت دسترسی به فایلها ایجاد می‌کند. زیرا پس از مدتی و با بهنگام سازی فایل، کلاسترهای یک فایل پشت سر هم قرار نخواهند داشت و برای دسترسی به کلاسترهای مختلف، نیاز به حرکت زیاد هِد HD خواهد بود. برنامه‌هایی نظیر Defrag یا Norton Speed Disk کاری که انجام می‌دهند، قرار دادن این کلاسترها به صورت پشت سر هم است.

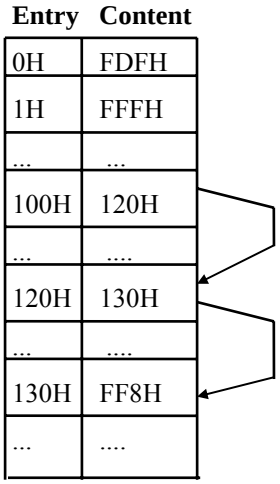
ساختار FAT: در DOS 2.0، هر ورودی FAT، ۱۲بیتی بود که 4,096 کلاستر را مشخص می‌کند و اگر هر کلاستر حاوی ۴ بخش باشد، با استفاده از این نوع FAT می‌توان به ولومی با ظرفیت 32MB =4096*512*4 دسترسی پیدا نمود. از اینرو، در نگارشهای بعدی داس، می‌توان برای ولومهای بزرگتر، از FAT16 نیز استفاده کرد، که هر ورودی FAT16، ۱۶ بیت است و با آن می‌توان 65,536 کلاستر را مشخص نمود. اگر هر کلاستر 4 بخش داشته

---

۱ entry
۲ cluster
۳ allocation unit
٤ file fragmentation

باشد، ظرفیت ولوم می‌تواند تا 132MB =65536*512*4 باشد، و بر اساس ستون آخر جدول فوق، برای کلاسترهای ۶۴ بخشی، می‌توان به ولومی با ظرفیت 64*512*65536=2GB دسترسی پیدا نمود، که در این حالت مشکل بزرگ بودن کلاسترها و اتلاف فضای HD پیش می‌آید. به این دلیل در Win98، FAT32 معرفی شده است که با استفاده از آن می‌توان کلاسترهای کوچکتر داشت و درنتیجه مشکل اتلاف فضای HD برطرف می‌شود.

اما ساختار FAT به شکل زیر است:



دو ورودی اول FAT، رزرو هستند و نشان‌دهنده‌ی توصیف‌گر وسیله[1] هستند، که برای FD مقدار F8H و برای HD، FFH است. مابقی فضای دو ورودی اول با FFH پر می‌شود.

در فهرست فایلها، برای هر فایل، شماره‌ی نخستین کلاستر در FAT نگهداری می‌شود. مثلاً برای یک فایل خاص که دارای سه کلاستر است، در شکل فوق، شماره‌ی اولین کلاستر، که در فهرست فایلها نگهداری می‌شود، 100H است و شماره‌ی دومین کلاستر 120H و شماره‌ی سومین کلاستر، 130H است. در ورودی شماره‌ی 100H، عدد 120H نوشته شده و در ورودی 120H، عدد 130H نوشته شده و در ورودی 120H، کد FFF8H نوشته شده که مشخص کننده‌ی پایان لیست پیوندی کلاسترها است.

محتوی هر ورودی FAT یک کد ۱۲ یا ۱۶ بیتی است، که انواع آن طبق جدول زیر است:

FAT12	FAT16	Meaning
000H	0000H	Cluster is available
FF0H-FF6H	FFF0H-FFF6H	Reserved cluster
FF7H	FFF7H	Cluster damaged, not used
FF8H-FFFH	FFF8H-FFFFH	Last file cluster
xxxH	xxxxH	Next file cluster

پس عملاً ورودیها در FAT12 بین 002H-FEFH و اندکی کمتر از ۴۰۹۶ و در FAT16 بین 0002H-FFEFH و اندکی کمتر از 65536 خواهد بود.

مشکلی که در مورد FAT در اغلب اوقات اتفاق می‌افتد، خراب شدن لیست پیوندی ورودیها یا کلاسترهای گم‌شده[2] است، به نحوی که برخی ورودیها حاوی کدهای خاص نبوده و کد موجود در آن، مربوط به هیچکدام از فایلهای سیستم نیست. برنامه‌هایی مثل ChkDsk سا ScanDisk اینگونه کلاسترها را یافته و تحت عنوان available علامت می‌زنند.

۳-۲۶ فهرست ریشه

فهرست ریشه[3]، بلافاصله پس از اولین نسخه‌ی FAT در ولوم ذخیره می‌شود. فهرست ریشه همانند زیرفهرست، یک ساختار ۳۲بیتی است که حاوی اطلاعات فایلها یا زیرفهرستها یا برچسب ولوم است. این اطلاعات عبارتند از:

---

۱ media descriptor
۲ lost clusters
۳ root directory

Addr.	Contents	Type
00H	File name	8 bytes
08H	File extention	3 bytes
0BH	File attribute	1 byte
0CH	Reserved	10 bytes
16H	Time of loast update	1 word
18H	Date of loast update	1 word
1AH	First cluster of file	1 word
1CH	File size	2 words

اگر اطلاعات ورودی مربوط به فایل نباشد، اولین بایت فیلد نام فایل یا همان اولین بایت ورودی، حاوی یک کد خاص است:

Code	Meaning
00H	Last directory entry
05H	First character of file name has ASCII code E5H
2EH	File applies to current directory (?!)
E5H	File is deleted

بیتهای مختلف فیلد file attribute در اطلاعات ورودی فهرست، مشخص‌کننده‌ی اطلاعات زیر است:

Bit	Meaning
0	1=Write-Protected (or Read-Only)
1	1=Hidden
2	1=System file
3	1=Volume name
4	1=Subdirectory
5	1=Archive
6	Reserved
7	Reserved

* بیت پنجم مشخص‌کننده Archive بودن یا نبودن فایل است. این بیت توسط دستور Backup و پس از کپی‌کردن فایل در رسانه‌ی Backup، به صفر مقداردهی می‌شود و با اولین دستیابی توسط توابع داس، به ۱ مقداردهی می‌شود. به این ترتیب در اجرای بعدی دستور Backup، تنها از فایلهایی Backup تهیه می‌شود که در این فاصله بهنگام شده‌اند.

* فیلد File Size حاوی اندازه‌ی فایل است که به‌صورت زیر در دو Word مجزا نگهداری می‌شود:

File Size = Word1 + Word2 * 65535

* Volume label name را تنها برای فهرست ریشه‌ی هر ولوم می‌توان معرفی نمود.

* برای زیرفهرستها، ساختار فوق در قالب یک فایل در سیستم نگهداری می‌شود و برخلاف فهرست ریشه، محدودیت تعداد فایل را ندارد.

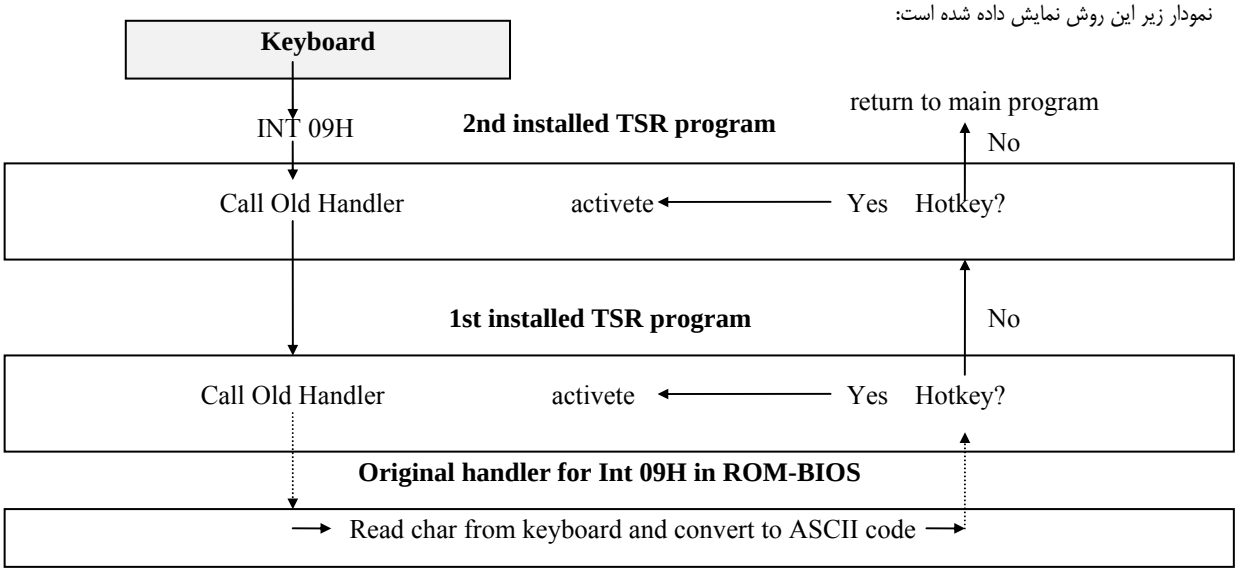
فصل ۳۲: برنامه‌های مقیم در حافظه
(Terminate and Stay Resident Programs)

۳۲-۱ مقدمه

داس یک سیستم عامل چندکاره[۱] نیست و نمی‌تواند بیش از یک برنامه را در هر زمان اجرا کند. اما برنامه‌های مقیم در حافظه(TSR) می‌توانند برخی از مزایای چندکاره‌بودن را برای داس به‌همراه بیاورند. اغلب برنامه‌های TSR به این صورت هستند که با زدن یک کلید، فعال شده و کاری را انجام می‌دهند و دوباره به زمینه[۲] می‌روند. مثالی از کاربرد این گونه برنامه‌ها، برنامه‌های تقویم، ماشین حساب و غیره است. در حقیقت برنامه‌های TSR، برنامه‌هایی چندکاره نیستند، بلکه یک روش چندبرنامه‌ای[۳] هستند. زیرا برنامه‌ها به‌طور همزمان اجرا نمی‌شوند. گرچه برنامه‌های TSR ممکن است کارهای متفاوتی انجام دهند، ولی مفاهیم و ساختار یکسانی دارند. در ادامه با این مفاهیم و ساختار آشنا می‌شوید.

۳۲-۲ فعال‌سازی برنامه‌های TSR

عموماً، برنامه‌های TSR با زدن یک کلید که Hotkey نامیده می‌شود، فعال می‌شوند. برخی دیگر از برنامه‌های TSR، از جمله برخی از برنامه‌های داس، با استفاده از وقفه‌ی 2FH که به تسهیم‌کننده[۴] معروف است، فعال می‌شوند، که در ادامه‌ی این فصل در مورد آن صحبت می‌شود. برای فراهم‌سازی قابلیت فعال‌شده با Hotkey، برنامه‌ی TSR باید روال سرویس وقفه‌ی صفحه‌کلید(09H) را تعویض کند، و در این روال Hotkey مورد نظرش را کنترل نماید. در نمودار زیر این روش نمایش داده شده است:



---

۱ multitask
۲ background
۳ multiprogramming
٤ multiplexer

بر اساس شکل فوق که برای دو برنامه‌ی TSR رسم شده است، هر برنامه‌ی TSR دارای یک روال سرویس وقفه‌ی 09H است که در آن ابتدا روال قبلی را فراخوانی نموده و سپس بررسی می‌کند که Hotkey موردنظرش زده شده است یا نه؟ در صورتی که زده شده باشد، توابع مربوط به خود را انجام می‌دهد. با این روش هر برنامه‌ای که ابتدا نصب شود الویت دریافت و پردازش کلید با او خواهد بود. با این روش، نمی‌توان یک برنامه‌ی TSR نوشت که زدن کلیدهای Alt+Ctrl+Del را کنترل نماید، زیرا این کار در روال اصلی سرویس وقفه‌ی 09H که در ROM-BIOS قرار دارد کنترل می‌شود و سیستم restart می‌شود. اگر بخواهیم این کار را انجام دهیم، باید ابتدا و قبل از فراخوانی روال قبلی، صفحه کلید را با روش دسترسی مستقیم بخوانیم(خواندن درگاه 60H و کنترل بایت وضعیت صفحه کلید از ثباتهای آن).

فراخوانی توابع داس: برنامه‌های TSR ممکن است توابع داس را فراخوانی کنند. این مسأله علی‌الخصوص در برنامه‌های سطح بالا، قابل کنترل بوسیله‌ی برنامه‌نویس نیست. مشکلی که وجود دارد، عدم قابلیت اجرای مجدد[۱] کدهای داس است. چون داس برای حالت تک‌کاره نوشته شده است ولی وقتی در داخل یک برنامه، با زدن یک Hotkey یک برنامه‌ی TSR را فعال می‌کنیم، این مشکل پیش می‌آید که ثباتهای مورد استفاده‌ی برنامه‌ی در حال اجرا، توسط برنامه‌ی TSR دستکاری می‌شوند و لذا پس از برگشت، برنامه‌ی درحال اجرا دچار اختلال می‌شود. به این دلیل در برنامه‌های TSR بهتر است از توابع داس استفاده نشود و نیز همانند روالهای سرویس وقفه‌ای که در کامپایلرهای بورلند تولید می‌شود، ابتدای ISR همه‌ی ثباتها در پشته push و در انتهای کار همه‌ی ثباتها pop شوند.

۳-۳۲ عملیات بحرانی نسبت به زمان

برخی اعمال باید در زمان کوتاهی تکمیل شوند و قابل وقفه نیستند، نظیر دسترسی به FD یا HD، که در سطح پایین بوسیله‌ی BIOS INT 13H مدیریت می‌شوند. اگر این‌گونه اعمال در زمان کوتاهی تکمیل نشوند، با مشکل مواجه می‌شوند. یک مثال خوب حالتی است که یک برنامه‌ی TSR، در خلال عمل دسترسی به دیسک برنامه‌ی اصلی، بخواهد به دیسک دسترسی پیدا نماید. در چنین مواقعی اغلب سیستم دچار اختلال شده و crash می‌کند، و اگر هم این‌طور نشود، داده‌ها خراب می‌شوند. یک راه اجتناب از این امر، تعویض وقفه‌ی 13H است. وقتی که ISR آن فعال می‌شود، یک Flag را مقداردهی می‌کند تا نشان‌دهنده‌ی فعال بودن آن باشد و به این وسیله کنترل می‌شود تا برنامه‌ی TSR به این ISR دسترسی نداشته باشد و آنرا اجرا نکند. در این صورت این برنامه‌ی TSR در صورت فعال شدن، نمی‌تواند کارهایش را درست انجام دهد، اما حداقل با این روش سیستم دچار اختلال نخواهد شد. در حقیقت کد مربوط به INT 13H، یک کد بحرانی است و باید به‌طریقی محافظت شود.

فعال‌سازی با تأخیر: با توجه به مشکل فوق، باید قبل از فعال‌سازی یک برنامه‌ی TSR کنترل شود که آیا در صورت فعال‌سازی قادر به انجام وظایفش است یا نه؟ به این دلیل اغلب برنامه‌های TSR روال سرویس جدیدی برای INT 08H (تایمر) نیز نصب می‌کنند. در صورت فراهم نبودن شرایط اجرای برنامه‌ی TSR، یک Flag را در ISR وقفه‌ی 09H، مقداردهی می‌کند و در ISR وقفه‌ی تایمر، flag را کنترل می‌کنند و در صورت 1 بودن بررسی می‌کنند که آیا شرایط آماده است یا نه؟ در صورت آماده بودن شرایط، برنامه‌ی TSR را فعال می‌کند. در این‌صورت، فعال‌سازی برنامه‌ی TSR با تأخیر انجام می‌شود.

* باید از اجرای مجدد برنامه‌های TSR جلو گیری شود. این کار باید در قسمت اصلی برنامه‌ی TSR انجام شود.

۴-۳۲ تعویض زمینه[۲]

به فرآیند فعال‌سازی یک برنامه‌ی TSR، تعویض زمینه گفته می‌شود. زمینه یا محیط یک برنامه، اطلاعات مورد نیازش برای عملیات و شامل موارد زیر است:

- **محتوی ثباتهای پردازنده:** اطلاعات ثباتهای پردازنده و از جمله ثباتهای قطعه باید در متغیرهای داخلی TSR یا در پشته ذخیره شوند و قبل از تعویض زمینه،ثباتهای پردازنده با ثباتهای ذخیره شده، بار شوند.
- **اطلاعات مهم سیستم عامل:** شامل PSP و DTA[۳] است. با PSP آشنا هستید، اما DTA بافری است که برای خواندن بلوکهای بزرگ داده از دیسک مورد استفاده قرار می‌گیرد. باید نشانی آنها هم ذخیره شود و در تعویض زمینه، آنها نیز تعویض شوند. برای خواندن و تنظیم نشانی DTA توابع 1AH و 2FH داس قابل استفاده هستند و برای PSP نیز تابع 50H برای تنظیم و توابع 62H و 51H برای خواندن نشانی قابل استفاده هستند.
- **حافظه‌ی اشغال‌شده بوسیله‌ی برنامه:** که به دلیل مقیم بودن برنامه در حافظه، مشکلی از این نظر نداریم.

۵-۳۲ اجزای برنامه‌های TSR

در زبان اسمبلی: باید ماجولهای زیر نوشته شوند:

---

۱ re-enterant
۲ context switch
۳ Disk Transfer Area

- TsrInit: برنامه را تبدیل به یک برنامه‌ی TSR نموده، مدیر وقفه را نصب می‌کند، برنامه را درحافظه بارمی‌کند و آنرا خاتمه می‌دهد.
- TsrIsInst: تعیین می‌کند که آیا بخش مقیم در حافظه‌ی برنامه، قبلاً در حافظه مقیم شده است یا نه؟
- TsrCanUnInst: تعیین می‌کند که آیا بخش مقیم در حافظه قابل خارج شدن است یا نه؟
- TsrUnInst: برنامه‌ی TSR را از حافظه خارج می‌کند.
- TsrSetPtr: یک اشاره‌گر به نشانی روالی که قرار است در داخل برنامه اجرا شود، ایجاد می‌کند.
- TsrCall: روال علامت‌زده‌شده را فراخوانی می‌کند.
- TsrSetHotKey: کلید فعال‌سازی برنامه را تنظیم می‌کند.

در زبان بورلند پاسکال: باید اعمال زیر انجام شوند:

- روال جدیدی برای ISR های تعویضی، تعریف شود:

```
procedure NewISRn(...); Interrupt;
```

- متغیری برای نگهداری نشانی روال قبلی تعریف شود:

```
var  OldISRn : procedure;
```

- در برنامه‌ی اصلی نشانی روال قبلی در OldISRn ذخیره شود:

```
OldISRn := GetIntVec (n);
```

- ISR جدید نصب شود:

```
SetIntVec(NewISRn, n);
```

- مقیم‌سازی برنامه در حافظه:

```
Keep(0);
```

* مقدار پارامتر Keep، همیشه مقدار 0 است که کد برگشتی به داس است.

در زبان بورلند C: باید اعمال زیر انجام شوند:

- روال جدیدی برای ISR های تعویضی، تعریف شود:

```
void interrupt NewISRn();
```

- متغیری برای نگهداری نشانی روال قبلی تعریف شود:

```
void interrupt (*OldISRn)();
```

- در برنامه‌ی اصلی نشانی روال قبلی در OldISRn ذخیره شود:

```
OldISRn = getvect(n);
```

- ISR جدید نصب شود:

```
setvect(NewISRn, n);
```

- مقیم‌سازی برنامه در حافظه:

```
keep(0, ProgSize);
```

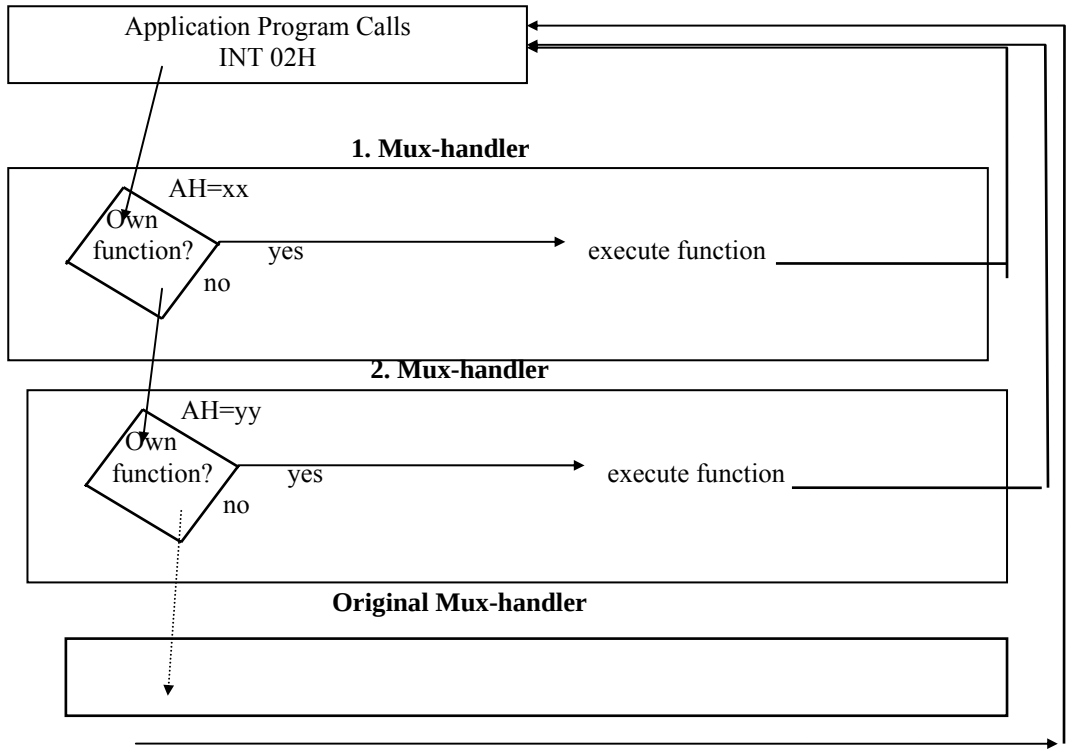
* مقدار پارامتر اول keep، همیشه مقدار 0 است که کد برگشتی به داس است.

* پارامتر دوم keep، اندازه‌ی برنامه در حافظه است که وابسه به مدل برنامه بوده و برای مدل small، طبق فرمول زیر بدست می‌آید:

```
ProgSize = (_SS + _SP / 16) - _PSP)+1
```

۶-۳۲ تسهیم‌کننده

برخی برنامه‌های داس، نظیر Print، Share، Assign، Append و Doskey به‌صورت TSR هستند و در صور نیاز، فعال شده و عملی را انجام می‌دهند و برای این منظور، از وقفه‌ی 2FH استفاده می‌کنند که به تسهیم‌کننده[۱] (Mux) معروف است. Mux در حقیقت یک واسط ارتباطی بین برنامه‌های TSR و سایر برنامه‌ها است و هر برنامه‌ی TSR می‌تواند از طریق Hotkey یا امکانات این وقفه فعال شود. برنامه‌های TSR استفاده کننده از Mux باید این نکته را رعایت کنند که در هر زمان برنامه‌های متعددی از Mux استفاده می‌کنند. برای این منظور به این روش عمل شود که قبل از آنکه برنامه از Mux استفاده کند، یک id هشت بیتی که به آن Mux Code گفته می‌شود به خود منسوب کند. Mux Codeهای 00H تا BFH به برنامه‌های داس تخصیص داده شده‌اند و از کدهای BFH تا FFH سایر برنامه‌ها می‌توانند استفاده کنند. وقتی این نوع برنامه‌ی TSR نصب می‌شود، باید یک روال سرویس وقفه برای Mux،علاوه بر سایر ISRهای مورد نیازش نصب کند. یعنی باید INT 2FH را نیز تعویض نماید و در ISR، چک کند که اگر Mux Code قرار داده شده در AH، قبل از فراخوانی INT 2FH، id آن است، توابع خود را اجرا کند، وگرنه روال قبلی را که نشانی آنرا حفظ نموده، فراخوانی کند:



در ابتدا باید تعیین شود که آیا برنامه‌ی فراخواننده می‌خواهد آن Handler را فراخوانی کند یا نه؟ برای این منظور Mux Code باید در ثبات AH قرار داده شود. در صورتی که کد مخصوص برنامه در AH باشد، توابع مربوط به آن برنامه فراخوانی می‌شوند، که باید شماره تابع در ثبات دیگری، غیر از AH قرار داده شده و سایر پارامترها نیز در ثباتهای دیگر قرار داده شده باشند. برخی از Mux Codeهای برنامه‌های داس به‌صورت هستند:

DOS Program	Mux Code
Print	01H
Assign	06H
Share	10H
ANSI.SYS	1AH
GRAFTABL	B0H
HIMEM.SYS	43H
DOSKEY	48H
DBLSPACE.BIN	4AH
KeyB	ADH
Append	B7H

[۱] Multiplexer

فصل ۳۳: حالت محافظت‌شده و داس
(Protected Mode and DOS)

۳۳-۱ مقدمه

پردازنده‌ی 80286 امکان استفاده از حالت محافظت‌شده[۱](PM) را فراهم نمود. اما داس سیستم عاملی برای حالت واقعی[۲](RM) است و نمی‌تواند از مزایای PM استفاده کند. مشکل اصلی این است که داس بر اساس BIOS است و BIOS هم برای RM است و در PM قابل استفاده نیست. چرا؟

زیرا روالهای وقفه‌ی BIOS براساس مدیریت حافظه‌ی منطقی (مبدأ، قطعه)[۳] عمل می‌کنند که تنها در RM معتبر است، نه مدیریت حافظه‌ی فیزیکی و براساس توصیف‌گر قطعه[۴] که در PM استفاده می‌شود. به این دلیل به محض سوئیچ‌کردن به PM روالهای BIOS دیگر قابل استفاده نخواهند بود.

گرچه 80286، PM را فراهم نمود، ولی دارای یک نقطه ضعف عمده بود و آن مدیریت حافظه به‌صورت قطعه‌بندی‌شده[۵] بود، نه صفحه بندی[۶] و از این نظر بر روی آن سیستمهای عمل privilidged نظیر Unix قابل پیاده‌سازی نبودند. اما 80386 و i486 تمامی قدرت PM را در اختیار قرار می‌دهند و پس از آمدن آنها افراد و شرکتهای زیادی شروع به دورزدن مشکلات داس و استفاده از مزایای PM تحت داس نمودند و برای این منظور کاربردهای زیر پیشنهاد شد:

- EMS Emulator: نظیر UMB در XMS یا QEMM386
- برنامه‌های مدیریت حافظه: نظیر EMB در XMS
- بسطهای داس[۷]: مثل DPMI و VCPI
- Multitaskers: مثل DesqView

۳۳-۲ حالت محافظت‌شده

PM در سال ۸۲ و همزمان با 80286 و برای استفاده در سیستمهای عمل چندکاره[۸] (MTOS) معرفی شد. اما داس در حالت واقعی کار می‌کند و تک‌کاره[۹] است و اگر می‌خواست یک MTOS بشود، باید بازنویسی می‌شد، که این مشکلات عدم سازگاری با برنامه‌های موجود را داشت. به این دلیل میکروسافت و IBM سیستم عامل جدیدی را به نام OS/2 معرفی کردند، که به‌دلیل نیاز به سخت‌افزار قدرتمند و حافظه‌ی زیاد، چندان مرسوم نشد. اما میکروسافت بعداً، ابتدا اقدام به عرضه‌ی گونه‌ای از سیستم عامل Unix برای کامپیوترهای PC، با نام XENIX نمود، ولی بعداً با فروش آن به شرکت SCO، کار بر روی آن را متوقف نموده و اقدام به عرضه‌ی Windows 3.1 و سپس Windows NT و آنگاه سایر ویندوزهای 32 بیتی نمود. تمامی این سیستمهای عامل نام برده شده، MTOS بوده و در PM کار می‌کنند.

مشخصات یک MTOS: نیازهای پردازنده‌ای در محیط MT به‌صورت زیر است:

- **محافظت:** پردازنده باید امکانات کاملی را برای فراهم‌سازی محافظت به‌منظور جلوگیری از تداخل کارها در محیط و حافظه‌ی همدیگر فراهم نماید.
- **امکانات مربوط به تعویض کارها[۱۰]:** برای این منظور باید چندکاره‌بودن غیرانحصاری[۱۱] و برش زمان[۱۲] فراهم شوند.
- **امتیازات سیستم عاملی[۱۳]:** اینکه هسته‌ی سیستم عامل در یک حالت دارای امتیاز کار کند و پردازه‌های کاربر[۱] دارای امتیاز کمتری بوده و برنامه‌های دارای امتیاز کمتر نتوانند حافظه‌ی سایرین یا هسته‌ی سیستم عامل را بازنویسی کنند.

---

۱ Protcted Mode
۲ Real Mode
۳ Segment:Offset
۴ Segment Descriptor
۵ segmented
۶ paging
۷ DOS extentions
۸ multitasking OSs
۹ single-task
۱۰ task switching
۱۱ preemptive multutasking
۱۲ time slicing
۱۳ OS privilidges

- **حافظه‌ی مجازی[2] (VM):** نیازهای حافظه‌ای در محیطهای چندکاره‌ای زیاد است و بنابر این سیستم عامل باید بتواند از حافظه‌ی اصلی و دیسک به‌عنوان swap استفاده کند و مسائل مدیریتی را پردازنده تسهیل کند.

۳-۳۳ حالت محافظت‌شده در ۸۰۲۸۶

این ریزپردازنده علاوه بر ثباتهای ۸ و ۱۶ بیتی ۸۰۸۶، ثباتهای زیر را دارد:

- **MSW:** Machine Status Word
- **TR:** Task Register
- **GDTR, LDTR, IDTR:** Pointers to Descriptor Tables

همچنین دو flag دیگر به ثبات FALGS اضافه شده است: NT و IOPL

ثبات MSW: این ثبات برای سوئیچ کردن به PM قابل استفاده است و دارای بیتهای زیر است:

Bit	Name	Description
0	PE	Operating mode selection
1	MB	Co-processor available
2	EM	Emulate co-processor
3	TS	Task Switch command

وقتی PC روشن شده و داس بوت می‌شود، ریزپردازنده در RM است و برای سوئیچ‌کردن به PM باید بیت PE به 0 مقداردهی شود، یعنی PE(RM)=1 و PE(PM)=0 است. اما برای برگشت به RM در 80286 یک مشکل وجود داشت و آن این بود که دستورالعمل مشخصی برای این کار و 1 کردن PE نداشت. اگر هم با یک trick بتوان این کار را نمود، منجر به مقداردهی اولیه‌ی ROM-BIOS شده و همانند reset شدن سیستم است و لذا برگشت به برنامه‌ی فراخواننده که در داس و RM عمل می‌کند، میسر نخواهد شد، و این مشکل اساسی در استفاده از PM تحت داس در 80286 بود. اما پردازنده‌های بعدی، این مشکلات را مرتفع نموده و ایجاد کاربردهای ذکر شده در مقدمه، امکان‌پذیر گردید.

۳-۳۳ مدیریت حافظه در ۸۰۲۸۶

مبتنی بر جدول توصیف‌گر قطعه‌ی سراسری[3](GDT) و جدول توصیف‌گر محلی[4] (LDT) است. GDT در زمان سوئیچ‌کردن کارها مورد استفاده قرار می‌گیرد و اطلاعات کلی در مورد قطعات سراسری در آن توصیف می‌شود. اما LDT پس از آنکه به یک کار سوئیچ شد، مورد استفاده قرار می‌گیرد و اطلاعات قطعات مورد استفاده‌ی برنامه در آن توصیف می‌شوند. دستورالعملهای LGDT[5] و LLDT[6] برای این منظور استفاده می‌شود.

حافظه‌ی مجازی: هر توصیف‌گر در GDT می‌تواند توصیف‌گر یک LDT باشد. همچنین هر کدام از این دو جدول می‌تواند ۸۱۹۲ توصیف‌گر داشته باشد. همچنین هر توصیف‌گر LDT می‌تواند یک قطعه‌ی 64K را توصیف کند. به این ترتیب، پردازنده می‌تواند به فضای حافظه‌ی مجازی 8192*8192*64KB = 1GB دسترسی داشته باشد، که می‌تواند شامل حافظه‌ی اصلی و دیسک(به‌عنوان swap) باشد.

۴-۳۳ حالت مجازی۸۶

حالت مجازی 86 یا V86[7] برای ایجاد سازش بین PM و RM ایجاد شد. بسیاری از تسهیلات مربوط به EMS، نظیر EMM386.SYS و محیطهای چندکاره، نظیر DescView386 از V86 استفاده می‌کنند. وقتی یک سیستم در این حالت کار می‌کند،کامپیوتر در حالت اجرا در RM است، در حالیکه مدیریت حافظه، مدیریت کارها، و قواعد مربوط به امتیازات، در PM عمل می‌کنند. وقتی یک برنامه را در V86 اجرا می‌کنید، آن برنامه نظیر یک کار مستقل در RM عمل می‌کند و PM در حقیقت از دید آن پنهان است. برنامه فکر می‌کند که یک مگابات فضای نشانی دارد و نشانی‌دهی در آن طبق همان فرمول تبدیل نشانی منطقی به فیزیکی SegmentAddress * 16 + OffsetAddress است. برای مثال در ویندوز، پنجره‌های داسی که در DOS-prompt در اختیار قرار می‌گیرند، کاربردی از این حالت است.

---

۱ user processes
۲ vitual memory
۳ Global Descriptor Table
۴ Local Descriptor Table
۵ Load GDT
۶ Load LDT
۷ Virtual-86 mode

۳۳-۵ واسطهای برنامه‌نویسی حالت محافظت‌شده

برنامه‌نویسی در حالت محافظت‌شده بسیار مشکل است و کارهای اولیه‌ی سوئیچ کردن به PM باید حتماً با زبان اسمبلی انجام شود. اما واسطهایی نظیر DPMI[۱] و VCPI[۲] معرفی شده‌اند که این کار را تسهیل می‌کنند. این واسطها، استفاده از PM را تحت داس آسان می‌کنند. از نظر مقایسه، DPMI از VCPI بهتر است و بیشتر از آن استفاده شده و پیشرفته‌تر از آن است.

در ادامه شرح مختصری از این دو نوع واسط ارائه می‌شود.

VCPI: در سال ۱۹۸۹ بوسیله‌ی گروهی از شرکتهای کامپیوتری تحت رهبری Phar Lap و Quadrateck معرفی شد. این واسط هدف برطرف نمودن مشکلات ناشی از وجود همزمان بسطهای DOS، multitaskers، و برنامه‌های مدیریت حافظه در ماشینهای 80386 و i486 را دنبال می‌نمود.

VCPI به توابع خودش، تحت نام Server و به برنامه‌های استفاده کننده از این توابع، تحت نام Client ارجاع می‌کند. این واسط سرویسهای زیر را فراهم می‌کند:

- سه تابع مخصوص مقداردهی اولیه VCPI
- چهار تابع ابتدایی برای مدیریت حافظه‌ی بسط‌یافته
- سه تابع برای دسترسی به CR0 (اولین ثبات کنترل) و ثباتهای اشکالزدایی پردازنده
- دو تابع برای مدیریت کنترلر وقفه
- یک تابع برای سوئیچ‌کردن بین حالت V86 و PM
- **DPMI:** این واسط ابتداً به‌عنوان استانداردی برای برنامه‌نویسی PM معرفی نشده بود، بلکه در طراحی Win 3.0 ایجاد شد و به دلایل نامعلوم، میکروسافت آنرا منتشر نمود. اما بعداً در سال ۱۹۹۰، شرکتهای معروفی چون میکروسافت، اینتل، بورلند، IBM و غیره تشکیل گروهی را برای استاندارد نمودن آن دادند. DPMI پوشش کاملتری را از امکانات PM در اختیار قرار می‌دهد، که شامل موارد زیر است:
- مدیریت جدولهای توصیف‌گر یک برنامه‌ی PM
- مدیریت و تخصیص حافظه‌ی بسط‌یافته
- مدیریت وقفه‌ها و استثنائات[۳]
- ارتباط با برنامه‌ها و مدیریت‌کننده‌های وقفه‌ی RM
- دسترسی به ثباتهای گوناگون پردازنده
- مجازی‌سازی DMA[۴]

---

۱ DOS Protected Mode Interface
۲ Virtual Control Programming Interface
۳ exceptions
٤ DMA Virtualization