МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

КАФЕДРА «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ»

МЕТОДИЧНІ ВКАЗІВКИ

нормативної навчальної дисципліни циклу професійної та практичної підготовки

«Системне програмне забезпечення»

(Опорний конспект лекцій)

Галузь знань: **0501**

Напрям(и) підготовки: *6.050102 "Комп'ютерна інженерія"*

Затверджено
на засіданні кафедри КІ
Протокол № 1 від
30.08.2010

Рекомендовано до видання
методичною комісією
спеціальностей 7.091502
Протокол № від

Донецьк – 2010 р.

УДК 681.3

Методичні вказівки до виконання лабораторних робіт з дисципліни "Системне програмне забезпечення" (для студентів спеціальностей 7.091501: "Комп'ютерні системи і мережі",7.091502*:* “ Системне програмне забезпечення ”)/ Автори: О.Ю.Іванов , О.Г.Шевченко –Донецьк :ДонНТУ, 2010, с.-155

В курсі **“** Системне програмне забезпечення**”** студенти спрямовані на опанування основних принципів, методів, систем і засобів, що складають системне програмне забезпечення: теоретичні основи і формальні моделі; способи оцінки систем;; засоби проектування і реалізації програмно-апаратних засобів програм і даних; методи застосування програмно-апаратних засобів в операційних системах і обчислювальних мережах.

В кожному розділі розглядаються можливі практичного застосування, в основному до проблем інформатики. В усіх розділах приділяється значна увага побудові алгоритмів для розв'язування задач. Поняття, факти, алгоритми, що вивчаються у курсі використовують знання набуті у курсах " Програмування", "Дискретна математика", "Системне програмування", "Мікропроцесорні системи".

Укладачі ***Іванов Олександр Юрієвич, Шевченко Ольга Георгіївна***

Рецензент ***Турупалов Віктор Володимирович***

Література.

1. Б.Страуструп. Язык программирования С++, 2-е изд./Пер. с англ. Часть первая. – Киев.: “ДиаСофт”, 1993. – 264 с. Часть вторая. – Киев.: “ДиаСофт”, 1993. – 296 с.
2. М.Эллис, Б.Страуструп. Справочное руководство по языку программирования С++ с комментариями./Пер. с англ. – М.: Мир, 1992.
3. Бабз Б. Просто и ясно о Borland С++./Пер. с англ. – М.: Бином, 1995. - 400 с.
4. Дьюхарст С., Старк К. Программирование на С++/Пер. с англ. – Киев, ДиаСофт, 1993. – 272 с.
5. Романов В.Ю. Программирование на языке С++: Практический подход. – М.: Компьютер, 1993. – 160 с.
6. С++. Язык программирования. – М.: ИВК СОФТ, 1991. – 315 с.
7. Г.Шилдт. Самоучитель С++, 3-е изд./Пер. с англ. – СПб.: БХВ-Петербург, 2001. – 688 с.
8. Б.Страуструп. Язык программирования С++, 3-е изд./Пер. с англ. – СПб.: М.: «Невский Диалект» – «Издательство БИНОМ», 1999. – 991 с., ил.
9. Том Сван. Программирование для Windows в Borland С++./Пер. с англ. – М.: БИНОМ, 1995. – 480 с. (Только особенности программирования в среде Windows, использование библиотеки OWL. Предполагает знание уже С++).
10. Г.Шилдт. Теория и практика С++ (серия Мастер, руководство для профессионалов). /Пер. с англ. – СПб.:BHV – Санкт-Петербург, 1996. – 416 с.
11. Г.Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на С++, 2-е изд./Пер. с англ. – М.: «Издательство Бином», СПб.: «Невский диалект», 1999 г. – 560 с., ил.

ВСТУП

Область інтересів фахівця КС перебуває на границі апаратного й програмного забезпечення. Він може бути системний програміст, що займається написанням BIOS або діагностичного програмного забезпечення, якому потрібно налагодити процес взаємодії процедур програми з апаратурою. Також він може бути розроблювач мікросхем чипсета, якому потрібно виконати ряд випробувань: наприклад програмно сформувати деякий керуючий вплив (записати дані в регістри керування) і проконтролювати результат (прочитати дані з регістрів статусу). На відміну від прикладного програмування, для такого завдання важливе розуміння процесу виконання програми на рівні принципової електричної схеми, генерації сигналів і виконання транзакцій на системній шині процесора при виконанні асемблерних команд.

Для перерахованих вище прикладів зручна система DOS тим, що в силу своєї простоти й компактності, вона, у відмінність, наприклад від Windows, має високі шанси запуститися на частково працездатній системі. Вона не вимагає процедури повторної інсталяції, якщо ми заміняємо налагоджуваєму плату, що важливо у випадках, коли через налагодження проходять десятки й сотні плат різних моделей. Але головне не в цьому.

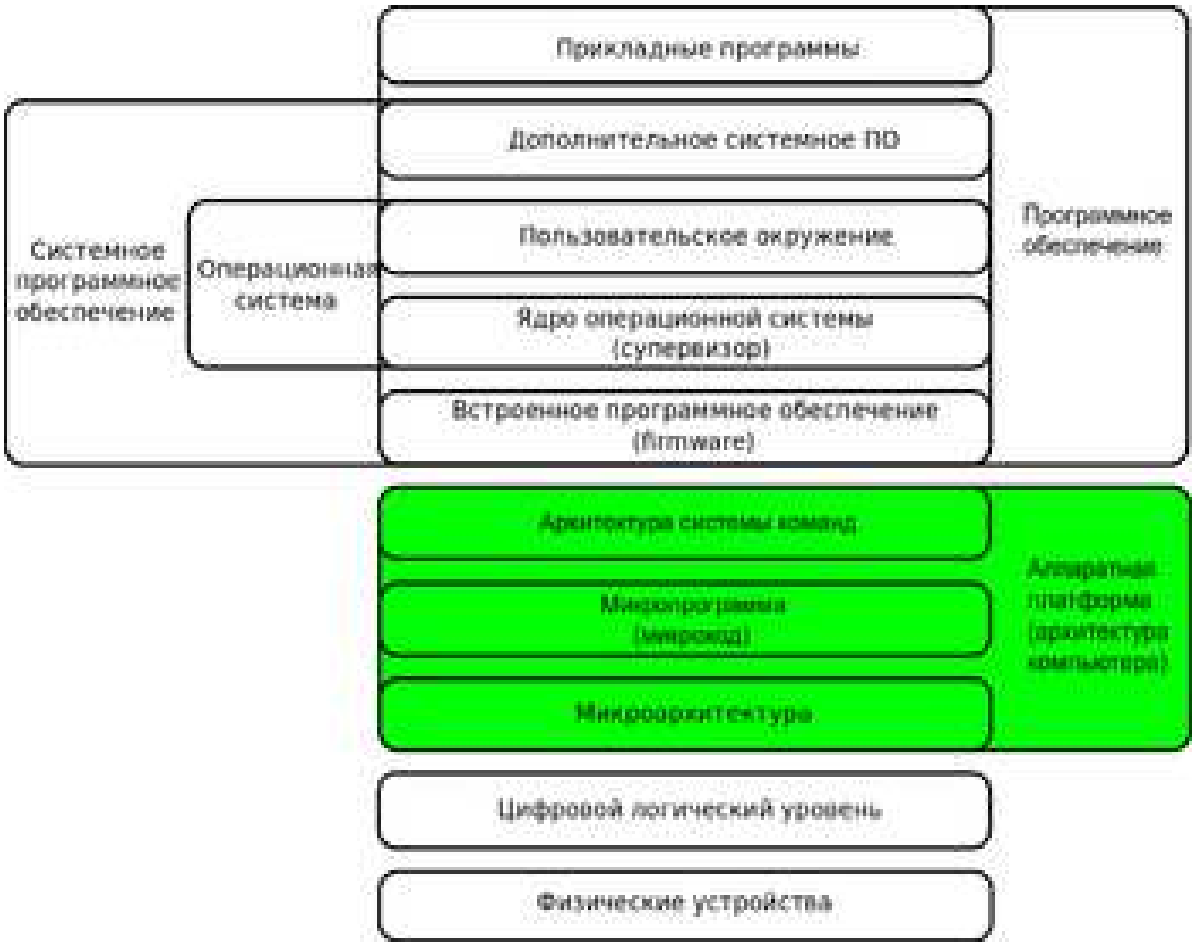
Відомо, що одним з найважливіших властивостей багатозадачних ОС є здатність зберігати стійкість ОС при різних збоях у середині додатків. Для цього ОС істотно обмежує набір операцій, які дозволено виконувати додаткам. Наприклад, якщо користувальницька програма спробує прямо звернутися до регістрів системних пристроїв, операційна система перехопить даний обіг і запитану операцію не відбудеться. При цьому ОС виконає емуляцію або видасть повідомлення про помилку. Обидва варіанти неприйнятні, якщо наше завдання - налагодження взаємодії програми з регістрами чипсета або системних регістрів процесора.

У свою чергу, якщо ми працюємо під ”чистим” DOS, у нас немає необхідності обходити які-небудь захисти, наша програма сама включає Protected Mode й є супервізором, вона може монопольно взаємодіяти з будь-яким програмно-доступним ресурсом. Ми можемо контролювати виконання програми на рівні асемблерних команд і машинного коду. У силу своєї простоти й компактності, DOS має істотно меншу ймовірність “зависання” при виконанні різних нестандартних експериментів з устаткуванням.

Тема 1. Апаратна платформа й основні компоненти комп'ютера.

Апаратна платформа комп'ютера (архітектура комп'ютера) - рівень, утворений мікроархітектурою, мікропрограмою керування ядром мікропроцесора й архітектурою набору команд на апаратній базі конкретних мікросхем процесора, чипсета, інших фізичних компонентів, які в сукупності становлять апаратну модель обчислювальної системи.

Призначений для запуску певних сімейств програмних продуктів (операційна система, прикладне програмне забезпечення), які, у свою чергу, розроблені виходячи з можливостей і для запуску на даних апаратурах.



Конкретно, апаратні платформи відрізняються друг від друга сукупністю апаратури (процесором, чипсетом), а також розробленими (і що запускають) програмними компонентами.

Однієї з найпоширеніших офісних платформ і персональних комп'ютерів є IBM PC. На ринку персональних комп'ютерів також поширені комп'ютери Apple. Ці платформи є широко відомими брендами.

Апаратна платформа	Розроблювач	Розрядність, біт	Типи систем	Примітки
IA-32 1985м	Intel	32	ПК, сервер, ноутбук, кластер	Домінуюча архітектура у світі Windows
x86-64 2003м	AMD	64	ПК, сервер, ноутбук, кластер	Зворотна сумісність із i386. Широко вироблена, але, через відсутність 64 bit драйверів для деяких пристроїв, рідше використовувана архітектура.
SPARCv9 1994м	Sun Microsystems	64	робоча станція, сервер	
IA-64 2001м	Intel й Hewlett Packard	64	сервер	Не сумісна з i386. Втратила важливість із появою x86-64

Несумісність коду, виконуваного процесором

Корпорація Intel, розвиваючи свої сімейства процесорів, наполняет процессоры дополнительными командами:

1.команди математичного співпроцесора (FPU);

2.команди для обробки мультімедіа (MMX);

3.серії команд SSE (SSE, SSE2, SSE3, SSE4і SSE5), запозичені в AMD команди 3DNow!,

4. а також 64-бітний набір команд AMD64.

Нові команди впливають на сумісність процесорів, тому розроблювачам програмного забезпечення доводиться орієнтуватися на дві платформи, більше стару й «численну» IA-32 і нову x86-64. Проблема сумісності коду — ситуація, коли процесори різних сімейств не можуть виконувати той самий машинний код. Наприклад, між двома 32-бітними процесорами того самого виробника, Intel — Pentium й Pentium 2, може виникнути несумісність через властиве обмеження (апаратної відсутності MMX команд).

Сімейство	**Кодове ім'я Intel**	**Короткий опис і технічні характеристики**
Desktop CPU		
486	**P24**	Перший повністю 32-х розрядний процесор. 1,25 млн. транзисторів; тактова частота: 50-66 МГц; кеш першого рівня: 8 Кб; кеш другого рівня на материнській платі (до 512 Кб); процесор 32-розрядний; шина даних 32-розрядна (25-33 МГц); адресна шина

		32-розрядна; загальна розрядність: 32.
	P24C	Остання "четвірка" зі збільшеним до 16 Кб кешем першого рівня. 1,6 млн. транзисторів; тактова частота: 75-100 МГц; кеш першого рівня: 16 Кб; кеш другого рівня на материнській платі (до 512 Кб); процесор 32-розрядний; шина даних 32-розрядна (25-33 МГц); адресна шина 32-розрядна; загальна розрядність: 32.
Pentium MMX	**P5**	Перший процесор із двухконвейерной структурою, випускався під Socket 4. Кеш-пам'ять уперше була розділена на 8 Кб для даних й 8 Кб - для інструкцій. 3,1 млн. транзисторів; технологія виробництва: 0,8 мкм; тактова частота: 60-66 МГц; кеш першого рівня: 16 Кб (8 Кб для даних й 8 Кб для інструкцій); кеш другого рівня на материнській платі (до 1 Мб); процесор 64-розрядний; шина даних 64-розрядна (60-66 МГц); адресна шина 32-розрядна; загальна розрядність: 32; рознімання Socket 4.
	Pentium P6	Створювався, як процесор для серверів і робочих станцій, має об'єднаний в одному корпусі кеш другого рівня обсягом 256Кб (у серпні 1997 р. з'явилася версія з 1Мб кеша другого рівня). Анонсований у листопаді 1995 р. 5.5 млн транзисторів, технологія виробництва: 0.35 мкм, тактова частота 150 MHz (виготовлена за технологією 0.6 мкм), 166 MHz, 180 MHz, 200 MHz.
	Klamath	Перший процесор лінійки Pentium II і перша модель із рознімання Slot 1. Травень 1997. Klamath виготовлявся по старої 0,35-мікронної технології - тому він працював тільки на частотах 233-300 Мгц й, в добавок до цьому, сильно нагрівався. Менше ніж через рік його зняли з виробництва. Він мав L1-кеш обсягом 32 Кбайт (16 Кбайт для даних + 16 Кбайт для інструкцій), L2-кеш обсягом 512 Кбайт працював на половинній частоті процесора, частота системної шини 66 МГц, підтримував MMX, многопроцессорность - до 2 процесорів. Напруга на ядрі - 2,8 У, на L2-кеше - 3,3 У. CPUID для процесорів

		сімейства Klamath дорівнює 63х.
Pentium Pro	**Deschutes**	З'явився в січні 1998, і це стало подальшим етапом у розвитку лінійки Pentium II. Процесор виготовлявся по 0,25-мікронної технології, мав тактову частоту 266-450 МГц, частоту системної шини 66/100 МГц, L1-кеш обсягом 32 Кбайт (16 Кбайт для даних + 16 Кбайт для інструкцій), L2-кеш обсягом 512 Кбайт. Розміщений на одній друкованій платі із процесором і працював на половинній частоті ядра, рознімання Slot 1, підтримував MMX, многопроцессорность - до 2 процесорів, мав CPUID рівний 65х. Вироблявся до II кварталу 1999 р. Використання 0,25-мікронної технології дозволило знизити не тільки собівартість процесора, але й напруга на ядрі - 2,0 У, що, у свою чергу, привело до зниження теплової потужності, що розсіює процесором. Архітектура ядра процесора залишилася такою же, як й в Klamath.
	Katmai	32-розрядні мікросхеми випускаються з першої половини 1999 р. Виконує 70 нових команд MMX. Вони поліпшать роботу додатків, що застосовують операції із плаваючою крапкою й тривимірною графікою, а також нові технології користувальницького інтерфейсу. Спочатку процесори виготовлялися по 0,25-мікронної технології, а згодом виробництво перейшло на 0,18-мкм технологічний процес. Для персональних комп'ютерів вони убудовані в картридж Slot 1; для мобільних систем - у модулі MMO (mobile module), а для потужних робочих станцій і серверів - у картридж Slot 2.
	Coppermine	Більше дешевий варіант Coppermine у формі-факторі FlipChip PGA 370, розрахований на використання з Socket-370 материнськими платами й частоту системної шини 100 й 133 Мгц. FC-PGA Coppermine нижче 600 МГц офіційно не підтримують режим мультипроцессорности - SMP. Тактова частота починається з 500 МГц, подальше збільшення швидкості

		відбувається в рамках всієї лінійки Coppermine до 1.13 ГГц. Живлення - 1.65 В. Протягом першої половини 2000 року існує спільно з Slot1 варіантом процесора.
	Tualatin-256K	Кодове найменування ядра й процесорів Socket 370 Pentium III, зроблених по 0,13 мкм техпроцессу. Це останні Pentium III. Відрізняються від Coppermine більше зробленими архітектурою й технологією виробництва. Характеризуються зниженою напругою живлення й меншим енергоспоживанням. Робоча частота моделей для Desktop з FSB 100 МГц - 1,0, 1,1 ГГц, а з FSB 133 МГц - 1,13 ГГц і вище.
	Covington	Перший процесор лінійки Celeron, з'явився на ринку у квітні 1998 року. Побудований на ядрі Deschutes і випускався по 0,25-мікронної технології. Тактова частота - 266-300 МГц, частота системної шини - 66 МГц, L1-кеш - 32 Кбайт. Для зменшення собівартості випускався без L2-кеша й захисного картриджа - у так називаному S.E.P.P. виконанні (Single-Edge Processor Package). Фізичний інтерфейс - Slot 1.
	Coppermine 128ДО	Новий етап розвитку лінійки Celeron. Починаючи із частоти 533 МГц, Celeron обзавівся новим процесорним ядром - Coppermine з урізаним до 128 Кбайт кешем L2. Відповідно, по своїх характеристиках процесор максимально близький до Pentium III, побудованим на базі Coppermine, у тому числі вперше для Celeron включає підтримку SSE. Очікується ріст частот до 900 МГц і вище, перехід на 0.13 мкм і частоту системної шини 100 МГц (уже здійснений у моделях із частотами 800 й 850 МГц).
Celeron	**Timna**	Кодове найменування процесорів, створених на основі ядра Coppermine з кеш-пам'яттю L2 128 Кбайт, інтегрованими на чипі графічним ядром і контролером оперативної пам'яті запозиченим із чипсетів сімейства Intel 800. Орієнтовані на сверхдешевые PC і

		телеприставки. Випуск відмінний фірмою Intel через безперспективність виробу.
	Willamette 423	Процесор з гіперконвеєризацією (hyperpipelining) - з конвеєром, що складається з 20 щаблів. Застосована 400 МГц системна шина (Quad-pumped), що забезпечує пропускну здатність в 3,2 ГБайта в секунду проти 133 МГц шини із пропускною здатністю 1,06 ГБайт в Pentium. Тих. характеристики: технологія виробництва: 0,18 мкм; тактова частота: 1.3-2 ГГц; кеш першого рівня: 8 Кб; кеш другого рівня 256 Кб (полноскоростной); процесор 64-розрядний; шина даних 64-розрядна (400 МГц); рознімання Socket 423.
	Willamette 478	Цей процесор виконаний по 0.18 мкм процесі. Установлюється в нове рознімання Socket 478, тому що попередній форма-фактор Socket 423 був "перехідним" й Intel надалі не збирається його підтримувати. Тих. характеристики: технологія виробництва: 0,18 мкм; тактова частота: 1,3-2 ГГц; кеш першого рівня: 8 Кб; кеш другого рівня 256 Кб (повноскоростний); процесор 64-розрядний; шина даних 64-розрядна (400 МГц); рознімання Socket 478.
	Northwood	Pentium 4 Northwood (попередня назва - Willamette-478) виробляється з дотриманням технологічних норм 0,13 мікронного процесу в 478 контактному корпусі форми-фактора mPGA478 (FC-PGA2). Особливістю процесора є спеціальна алюминевая пластина над клисталлом, що одночасно виконує функції теплоотвода й елемента, що екранує.
	Prescott	Спадкоємець ядра Northwood, буде виготовлятися по 90 нм технології, частота FSB=667 MHz (166 MHz QPB), підтримка Hyper-Threading, Socket 478, має кеш другого рівня обсягом 1 Мб, і буде представлений у першому кварталі 2004 року.
Pentium 4	**Nehalem**	Принципово нове ядро, на відміну від чипа Prescott - поліпшеної версії

		Pentium 4, і наступного за ним чипа Tejas. Nahalem буде виробляється в другій половині 2004 року по 90 нм техпроцессу, а пізніше, наприкінці 2005 - буде перехід на 65 нм техпроцесс.
	Cascades	Кодове найменування Pentium III Xeon, створеного на базі технологічного процесу 0,18 мкм. Є серверним варіантом Coppermine. На чипі втримується кеш L2 256 Кбайт, тактова частота від 600 МГц, частота шини процесора - 133 Мгц. Перші варіанти працюють тільки у двухпроцессорных конфігураціях і тільки на частоті системної шини 133 Мгц. Наприкінці 2000 року обсяг кеш-пам'яті L2 на чипі був збільшений до 2 Мбайт. Фінальна тактова частота - 900 МГц для повноцінної версії, 1 ГГц - для версії з 256 Кбайт L2. Форма-фактор - Slot 2.
	Foster	Кодове найменування ядра й процесорів Pentium 4 у серверному варіанті, побудованих по ідеології й архітектурі Willamette. Тактова частота - 100 МГц при передачі даних із частотою 400 Мгц. Як й у випадку з Cascades, обсяг кеша L2 залишився тим же, що в Willamette. Основні відмінності Foster від звичайних Pentium 4 на ядрі Willamette полягають у підтримці двухпроцессорных конфігурацій і використанні рознімання Socket 603. Тактова частота перших процесорів Xeon на ядрі Foster починається від 1,7 ГГц. Основу систем складуть чипсети i860 й GC-HE від ServerWorks. В 2002 р. планується переклад архітектури на технологію 0,13 мкм. Тоді ж буде випущена й нова версія Foster, що містить додатковий кеш третього рівня.
	Gallatin	Процесор Pentium 4 Xeon MP (кодове ім'я Gallatin), що відрізняється збільшеним обсягом кеш-пам'яті, буде випущений уже в четвертому кварталі поточного року. Раніше Intel планувала представити цей процесор у першому кварталі 2003 року. Чипи будуть працювати з тактовими частотами 1,50, 1,90 й 2,0 ГГц і будуть оснащуватися кешем обсягом 1 або 2 Мб.

	Prestonia	Кодове найменування ядра й процесорів Pentium 4 у серверному варіанті, створених за технологією 0,13 мкм. Продовження лінійки Xeon. Мікроархітектура NetBurst. Розробка ведеться на основі ядра Foster, що і буде замінено цим новим ядром у майбутніх процесорах Xeon. Основу систем складе спеціальний чипсет Plumas. Випуск запланований на першу половину 2002 року. Частота перших моделей процесора - 2,20 ГГц.
	Merced	Кодове найменування ядра й першого процесора архітектури IA-64, апаратно сполучимо з архітектурою IA-32. Включає трехуровневую кеш-пам'ять обсягом 2-4 Мбайт. Продуктивність приблизно в три рази вище, ніж в Tanner. Технологія виготовлення - 0,18 мкм, частота ядра - 667 МГц і вище, частота шини - 266 Мгц. Перевершує Pentium Pro по операціях FPU в 20 разів. Фізичний інтерфейс - Slot M. Підтримує MMX й SSE. Офіційне найменування - Itanium.
Server & Workstation CPU		
P II / III	**Banias**	Процесор Banias буде підтримувати три нові технології - Aggressive Clock Gating, Special Sizing Techniques й MicroOps Fusion. Весь дизайн процесора був спеціально розроблений з метою максимального зменшення споживання енергії.
	Tualatin	Новий Celeron має кеш другого рівня розміром 256 Кб і працює на 100 МГц системній шині, тобто перевершує по характеристиках перші моделі Pentium!!! (Coppermine). Тих. характеристики: 28.1 млн. транзисторів; технологія виробництва: 0,13 мкм; тактова частота: 1-1.4 ГГц; кеш перші рівні: 32 Кб (16 Кб на дані й 16 Кб на інструкції); кеш другого рівня 256 Кб (полноскоростной); процесор 64-розрядний; шина даних 64-розрядна (100 МГц); адресна шина 64-розрядна; загальна розрядність: 32; рознімання FC-PGA2 370.
Pentium 4	**Whitney** (i810)	Перший чипсет 800-й серії. Був

		анонсований у квітні 1999 року. Проектувався розраховуючи на нижню цінову категорію під процесори Celeron на ядрах Mendocino й Coppermine. Крім досить низької швидкості чипсет i810 і його старший брат i810E мали один істотний недолік - відсутність можливості підключити AGP-видеокарту, тому сфера їхнього застосування обмежувалася слабкими офісними машинами.
	Almador (i830)	Повинен був вийти в 2-3 кварталі 2001 року. У підсумку вийшов тільки варіант для мобільних комп'ютерів. Almador підтримував системну шину AGTL+ не тільки 133МГц, але й 200(!) Мгц. Максимальна кількість системної пам'яті становило 1.5Гб, підтримувалася пам'ять PC133 SDRAM і що більше важливо DDR SDRAM. У південному мосту ICH3 уведена підтримка 6 портів USB 2.0 й ATA-100.
	Tehama (i850)	Найперший набір логіки для новітніх тоді процесорів Pentium 4. Випущений одночасно із процесором 20 жовтня 2000 року. Чипсет підтримує 400МГц FSB шину (у модифікації E и 533 МГц FSB), має в наявності 4 RIMM рознімання й підтримує до 2Гб PC600/800 пам'яті Rambus RDRAM, AGP4x. Наявність морально застарілого ICH2 давало стандартний набір технологій: 4 USB порти, ATA-100 і т.д. Досить успішний продукт, але ймовірно тому, що інших тоді просто не було. Мав винятково високу швидкість роботи внаслідок загальної збалансованості архітектури. Цей набір логіки був останнім набором від Intel, що підтримував RDRAM.
	Springdale-P й **Springdale-G**	Чипсети будуть мати підтримку процесорів Pentium 4 із частотою шини 400, 533 й 667 МГц і технології Hyper-Threading. Мати двухканальный DDR333 SDRAM інтерфейс пам'яті, підтримувати AGP 8x і комплектуватися новим південним мостом ICH5, що підтримує Serial ATA. Springdale-P й Springdale-G сумісні між собою по виводах, а розходження між ними буде складатися в наявності

		нового високопродуктивного інтегрованого графічного ядра в Springdale-G.
	i865PE (Springdale-PE)	Набір логіки для масового ринку, що повинен буде замінити i845PE. Цей чипсет буде підтримувати процесори Pentium 4 й Celeron із частотами шини 800/533/400 МГц, а також працювати із двухканальной DDR400/DDR333/DDR266 SDRAM. ECC не підтримується. Також, у чипсеті реалізована шина AGP 8x. Чипсет буде комплектуватися новим південним мостом ICH5 з підтримкою Serial ATA-150.
IA - 64	**Grantsdale**	Grantsdale, очікуваний в 2004 році, буде поставлятися з південним мостом ICH6, FSB до 1066 МГц, двухканальную пам'ять DDR2, графічний інтерфейс PCI Express x16, шину PCI Express, що випливає покоління інтегрованого аудио. Примітно, що як шина між південним MCH і північним ICH мостами нового чипсета Grantsdale буде використатися аж ніяк не PCI Express, але новий інтерфейс DMI (Direction Media Interface) із пропускною здатністю до 2 Гб/с. Чипсет буде підтримувати як FSB 800 МГц, так й 1066 Мгц. Чипсет буде підтримувати процесори в конструктиве LGA755. За попередніми даними, буде підтримуватися як DDR2, так і пам'ять нинішнього покоління, DDR-I. Основні характеристики чипсета підкреслюють роботу контролера пам'яті у двухканальном режимі, однак, не виключено, що будуть представлені й одноканальні low-end версії чипсета.
	Odem (i855PM)	Odem (i855PM). Набір логіки Odem призначений для роботи із процесором Pentium M (Banias) входить до складу компонентів під загальною назвою Centrino. Вийшов у світло в березні 2003 року. Чипсет підтримує оперативну пам'ять типу DDR 266 обсягом до 1Гб. Чип оснащений поліпшеною технологією енергозбереження Enhanced Speedstep, що має 3 режими роботи. Odem планується використати в

		полноразмерных ноутбуках. В одній з наступних ревізій заплановане введення підтримки Hyper Threading.
	Montana-GM (855GM)	Montana-GM має свій убудований 3D прискорювач. Обоє північних моста підтримують DDR пам'ять, мають оптимизированный під мобільне застосування контролер пам'яті й "спілкуються" з Pentium-M процесором по 400 МГц (100 МГц QDR, як в P4) шині зі зниженою потужністю.
	Montara-GML	Montara-GML для Banias. Цей набір логіки буде використати DDR SDRAM підсистему пам'яті й комплектуватися південним мостом ICH4M, що підтримує USB 2.0. Північний і південний міст буде з'єднуватися за допомогою Hub Link із пропускною здатністю 266 Мбайт у секунду. Завдяки тому, що Banias й Pentium 4 сумісні по шині, чипсет Montara-GML може бути використаний і із процесорами Pentium 4-M.
	Plumas (E7500)	Серверний чип для процесорів Intel Xeon на ядрі Prestonia. Вийшов в 3-м кварталі 2002 року. Підтримує двухканальный доступ до DDR SDRAM із загальною смугою до 3.2 GBytes/s і роботу із двома процесорами. Крім цього, набір логіки підтримує технологію Chipkill, що дозволяє виключити вихід з ладу сервера через дефект пам'яті. Це перший за останні два роки набір логіки від Intel для двухпроцессорных серверів, все це час у цьому секторі використалася продукція компанії Server Works. Також була випущена модифікація чипсета Plumas 533 (E7501), що підтримував частоту системної шини 133 MHz QPB.
P II / III	**Granite Bay**	Аналог Lindenhurst для серверів середнього рівня. Підтримує до 4 процесорів Xeon MP на ядрі Potomac, шину PCI Express і пам'ять DDR II.
	Twin Castle	Аналог Lindenhurst для ринку робочих станцій середнього рівня.
Celeron	**Turnwater**	Цей чипсет буде націлюватися на використання у високопродуктивних

		робочих станціях на базі Pentium 4, а тому буде підтримувати два канали DDR333 SDRAM, ECC, AGP 8x, процесори із частотою шини 400/533/667 МГц і технологією Hyper-Threading, а також комплектуватися південним мостом ICH5 з підтримкою Serial ATA. Фактично, CanterWood є більше просунутою версією Granite Bay, призначеної для Prescott.
P III / Celeron	**CanterWood** або **Hance Rapids**	Mercury призначений для процесорів Pentium з напругою живлення 5 У, установлюваних у рознімання Slot 4 й имеющих тактову частоту 60 або 66 Мгц.
	Mercury	Neptune призначався для процесорів P54C з напругою живлення 3,3 У и тактовою частотою 100-133 Мгц. Neptune також підтримував двухпроцессорную конфігурацію, і до появи Triton II (430HX) це був єдиний чипсет, що забезпечує SMP. Mercury підтримував до 192 Мбайт повністю кешируемой оперативної пам'яті, а Neptune - до 512 Мбайт, що більше, ніж у последовавшего за ними Triton I. Обидва чипсети мали підтримку шини EISA, тому навіть після появи Triton I виробники використали їх для серверних плат.
	Neptune	Це був найбільш значний крок Intel у створенні чипсетів, і на довгий час Triton затвердився як стандарт для комп'ютерів на основі процесора Pentium. У той час як інші виробники намагалися забезпечити сумісність шин VLB й PCI, не одержуючи при цьому достатньої швидкодії, Intel повністю відмовилася від підтримки VLB. На той час Triton включав всі новітні досягнення: підтримку пам'яті EDO, кеш-пам'яті другого рівня типу Pipelined Burst і синхронної SRAM, підтримку PCI версії 2.0. Чипсет підтримував чотири банки SIMM для оперативної пам'яті загальним обсягом до 128 Мбайт, хоча кешируемый обсяг

		становив тільки 64 Мбайт. У той час цього обсягу було досить, тому що навіть найбільш серйозні додатки вимагали 8-16 Мбайт. У первісних версіях чипсета було багато помилок, але вони були виправлені в наступних модифікаціях. На сьогоднішній день всі функції 430FX підтримуються більше новими 430VX й 430TX.
	Triton	Це був найбільш значний крок Intel у створенні чипсетів, і на довгий час Triton затвердився як стандарт для комп'ютерів на основі процесора Pentium. У той час як інші виробники намагалися забезпечити сумісність шин VLB й PCI, не одержуючи при цьому достатньої швидкодії, Intel повністю відмовилася від підтримки VLB. На той час Triton включав всі новітні досягнення: підтримку пам'яті EDO, кеш-пам'яті другого рівня типу Pipelined Burst і синхронної SRAM, підтримку PCI версії 2.0. Чипсет підтримував чотири банки SIMM для оперативної пам'яті загальним обсягом до 128 Мбайт, хоча кешируемый обсяг становив тільки 64 Мбайт. У той час цього обсягу було досить, тому що навіть найбільш серйозні додатки вимагали 8-16 Мбайт. У первісних версіях чипсета було багато помилок, але вони були виправлені в наступних модифікаціях. На сьогоднішній день всі функції 430FX підтримуються більше новими 430VX й 430TX.
	Triton	Це був найбільш значний крок Intel у створенні чипсетів, і на довгий час Triton затвердився як стандарт для комп'ютерів на основі процесора Pentium. У той час як інші виробники намагалися забезпечити сумісність шин VLB й PCI, не одержуючи при цьому

		достатньої швидкодії, Intel повністю відмовилася від підтримки VLB. На той час Triton включав всі новітні досягнення: підтримку пам'яті EDO, кеш-пам'яті другого рівня типу Pipelined Burst і синхронної SRAM, підтримку PCI версії 2.0. Чипсет підтримував чотири банки SIMM для оперативної пам'яті загальним обсягом до 128 Мбайт, хоча кешируемый обсяг становив тільки 64 Мбайт. У той час цього обсягу було досить, тому що навіть найбільш серйозні додатки вимагали 8-16 Мбайт. У первісних версіях чипсета було багато помилок, але вони були виправлені в наступних модифікаціях. На сьогоднішній день всі функції 430FX підтримуються більше новими 430VX й 430TX.
	Triton	Це був найбільш значний крок Intel у створенні чипсетів, і на довгий час Triton затвердився як стандарт для комп'ютерів на основі процесора Pentium. У той час як інші виробники намагалися забезпечити сумісність шин VLB й PCI, не одержуючи при цьому достатньої швидкодії, Intel повністю відмовилася від підтримки VLB. На той час Triton включав всі новітні досягнення: підтримку пам'яті EDO, кеш-пам'яті другого рівня типу Pipelined Burst і синхронної SRAM, підтримку PCI версії 2.0. Чипсет підтримував чотири банки SIMM для оперативної пам'яті загальним обсягом до 128 Мбайт, хоча кешируемый обсяг становив тільки 64 Мбайт. У той час цього обсягу було досить, тому що навіть найбільш серйозні додатки вимагали 8-16 Мбайт. У первісних версіях чипсета було багато помилок, але вони були виправлені в наступних модифікаціях. На сьогоднішній день всі функції 430FX підтримуються більше новими 430VX й 430TX.

Несумісність пристроїв і материнських плат

Конкретно материнська плата персонального комп'ютера, також вносить свій внесок у несумісність платформ. На сучасній материнській платі розташована безліч убудованих пристроїв, для яких, на відміну від певних у дистрибутиві сімейства операційних систем Windows NT восьми альтернативних і мультіплатформених драйверів, потрібні специфічні драйвери. Тому, при установці операційної системи Windows 9x або NT, вона за допомогою установки драйверів специфічного встаткування, «привязуються» до конкретної материнської плати. Наступний перенос операційної системи на іншу материнську плату сполучений зі складністю забезпечення апаратної сумісності нової апаратної платформи.

Для рішення цієї проблеми в корпоративному сегменті, нова техніка (материнська плата, периферійні пристрої) проходить ретельне припасування під існуючий HAL, або під неї створюється новий HAL, погоджений зі сторонніми розроблювачами програмного й апаратного забезпечення.

Кроссплатформене програмне забезпечення

Запуск програмного забезпечення на більш ніж одній апаратній платформі й/або операційній системі є важливим завданням, як для розроблювачів нових апаратур, так і для програмістів.

- Debian компілює свої пакети для GNU/Linux для трьох архітектур процесорів Intel: IA-32 (x86-32), x86-64, IA64. Також офіційно створюються пакети для ще 8 апаратних платформ.
- Microsoft розробляє спеціальні вітки своєї операційної системи Microsoft Windows: Windows CE і Windows Embedded.
- Запуск на різних архітектурах того самого прикладного програмного забезпечення без необхідності забезпечувати сумісність на рівні ОС реалізується шляхом стандартизації мов, компіляторів, бібліотек і середовища виконання (див., наприклад, POSIX), а також шляхом переходу на виконання ПО у віртуальній машині й стандартному оточенні, які реалізуються для кожної платформи й гарантують однакове виконання ПО незалежно від платформи.

Чіпсет - це набір мікросхем системної логіки. Комп'ютер складається з деякої кількості пристроїв, які підключені до материнської плати. Логічною організацією цієї роботи й займаються чипсети. За давніх часів, коли МСЛ ще не існувало, материнські плати несли на собі до ста мікросхем, які займалися логічною організацією роботи пристроїв. От деякі з них: контролери переривань, контролер прямого доступу, годинники, системний таймер, контролер шини та інше та інше. Таке положення проіснувало до 1986 року, коли фірма Chip and Technologies запропонувала воістину

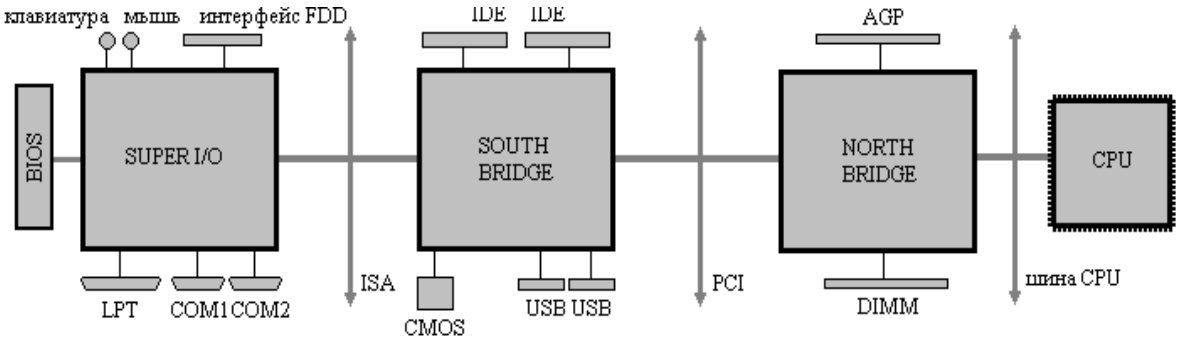
революційне рішення. Мікросхема називалася 82C206 і стала основною частиною набору системної логіки. Вона виконувала такі функції як:

1. контролер шин
2. генератор тактової частоти
3. системний таймер
4. контролер переривань
5. контролер прямого доступу до пам'яті
6. CMOS

Крім її на материнській платі використалися ще чотири мікросхеми як буфери й контролерів. Разом, виходить, п'ять мікросхем. Весь цей набір мікросхем системної логіки був названий CS8220. Потім на ринку з'явився наступний набір 82C836 Single Chip (SCAT), що взагалі складався з однієї мікросхеми. Але Chip and Technologies не залишилася на самоті: Suntac, Acer, Opti, UMC, Erso. Всі вони колись робили набори системної логіки, але в 1994 році на цей ринок вийшов Intel. Навіть прабатько чипсетів, Chip and Technologies, був видалений із цього ринку, а потім і зовсім викуплений Intel'ом. На даний момент чипсети випускають Intel, VIA Technologies й Si.

В 1989 році Compaq розробила шину EISA і в Intel за короткий час самі розробили контролер. Випустивши 486 процесор фірмі довелося чекати, коли виробники чипсетів підготують підходящий набір системної логіки. Що б більше не гаяти час і гроші, при випуску процесора Pentium, в 1995 році, Intel відразу ж розробила й випустила власний набір логіки. З тих пор і пішло: вийшов новий процесор - готовий до нього чипсет.

Розглянемо спрощену схему материнської плати на основі i440BX:



Набір системної логіки складається із двох мікросхем (має дворівневу архітектуру): North Bridge й South Bridge. North Bridge, крім усього іншого, містить: кеш, контролери оперативної пам'яті, інтерфейс між шиною процесора й PCI, AGP. Все це реалізовано на одному кристалі. Частота роботи цієї мікросхеми дорівнює тактовій частоті материнської плати. Сучасні North Bridge працюють на високих тактових частотах і тому додатково обладнані пристроями охолодження.

South Bridge є більше повільною мікросхемою, цей компонент відповідає за роботу шини ISA (у наявності є контролер прямого доступу й контролер переривань цієї шини), контролерів IDE й USB, а також реалізує функції пам'яті CMOS і годин і т.д. Слід зазначити, що той самий тип мікросхеми South Bridge може використатися, як правило, у

декількох наборах системної логіки, тобто може працювати з декількома типами North Bridge.

На кожному компоненті набору мікросхем системної логіки є номер (назва), що відповідає номеру (назві) усього набору. Наприклад: 82443BX North Bridge й 82371EX South Bridge - не що інше, як i440BX.

Інструкція CPUID, доступна, починаючи з Pentium і деяких моделей 486, викликається з параметром, зазначеним у регістрі EAX:

CPUID(0) - у регістрі EAX повертається максимально припустиме значення параметра виклику; у регістрах EBX, EDX й ECX процесор повертає символьний рядок, специфічну для виробника. Символи рядка розміщаються в регістрах у зазначеному порядку, починаючи з молодших байт.

Процесори Intel повертають рядок "GenuineIntel":

- EBX=756E6547h - "Genu", символ "G" у регістрі BL,
- EDX=49656E69h - "ine", символ "i" у регістрі DL,
- ECX=6C65746Eh - "ntel", символ "n" у регістрі CL.

Процесори AMD повертають рядок "AuthenticAMD":

- EBX=68747541h
- ECX=444D4163h
- EDX=69746E65h

CPUID(1) - у молодшому слові регістра EAX процесор повертає код ідентифікації (див. таблицю 1) він же сигнатура процесора й старший елемент 96-бітного серійного номера. Це ж значення втримується в регістрі DX після апаратного скидання:

- EAX[3:0] - степпінг;
- EAX[7:4] - модель;
- EAX[11:8] - сімейство;
- EAX[13:12] - тип;
- EAX[31:14] - зарезервовано (0);
- Регістри EBX=0, ECX=0 (резерв).

Регістр EDX містить список наявних розширень базової архітектури - відображає регістр властивостей (Feature Flags register). Призначення біт регістра наведений у таблиці 2.

Таблиця 2

Біт	Название	Призначення
0	FPU	Floating Poin Unit - наявність математичного співпроцесора
1	VME	Virtual-8086 Mode Enhancements - розширення режиму V86 (віртуалізація прапора переривань)
2	DE	Debugging Extensions - розширення налагодження

		(можливість зупинки по звертанню до портів)
3	PSE	Page Size Extension - можливість застосування розміру сторінки в 4 Мбайт
4	TSC	Time Stamp Counter - наявність лічильника міток реального часу
5	MSR	Model Specific Register - підтримка модельно-специфічних регістрів у стилі Pentium (інструкції RDMSR, WRMSR)
6	PAE	Physical Address Extension - можливість розширення фізичної адреси до 36 біт
7	MCE	Machine Check Exception - підтримка виключення машинного контролю #MC
8	CX8	Підтримка інструкції CMPXCHG8B
9	APIC	Наявність убудованого програмно^-доступного контролера переривань APIC
10	-	Зарезервовано
11	SEP	SYSENTER Present - підтримка інструкцій швидких системних викликів SYSENTER й SYSEXIT
12	MTRR	Memory Type Range Registers - наявність регістра керування кешуванням MTRRcap
13	PGE	Page Global Enable - підтримка біт глобальності в елементах каталогу й таблиць сторінок, а також біта PGE у регістрі CR4
14	MCA	Machine Check Architecture - підтримка архітектури машинного контролю
15	CMOV	Conditional Move - підтримка інструкцій умовного пересилання CMOVcc, а якщо є FPU, те і інструкцій FCMOVCC й FCOMI
16	PAT	Page Attribute Table - підтримка таблиць атрибутів сторінок (PAT)
17	PSE-36	36-bit Page Size Extension - можливість використання 36-бітної фізичної адресації для сторінок в 4 Мб
18	PN	Processor Number - підтримка повідомлення 96-бітного серійного номера по інструкції CPUID(3)
18-22	-	Зарезервовано
23	MMX	Підтримка MMX
24	FXSR	Fast floating point save and restore - підтримка інструкцій швидкого збереження й відновлення контексту FPU (інструкцій FXSAVE і FXRSTOR). Указує й на доступність індикатора використання цих інструкцій операційною системою (CR4.OSFXSR)
25	XMM	Наявність блоку XMM (підтримка нових інструкцій розширення SSE)
24... - 31		Зарезервовано

CPUID(2)- у регістрах EAX, EBX, ECX, EDX повертаються параметри конфігурації процесора. Молодші 8 біт EAX повідомляють, скільки разів потрібно підряд викликати інструкцію (з EAX=2) для одержання повної інформації про процесор. Інші байти регістра EAX й інших регістрів містять дескриптори окремих вузлів, які розшифровуються по спеціальних таблицях. Ознакою використання кожного з регістрів EAX, EBX, ECX, EDX є нуль у його біті 31. Виклик CPUID(2) з'явився із процесорами 6-го покоління. Поки що по ньому повідомляються тільки дескриптори елементів кешуваням (таблиця 3).

Таблиця 3. Призначення прапорів розширення архітектури

00h Нульовий дескриптор (у невикористовуваних байтах)
01h TLB інструкцій: сторінки 4ДО, 4WSA, 32 входження
02h TLB інструкцій: сторінки 4М, FA, 2 входження
03h TLB даних: сторінки 4ДО, 4WSA, 64 входження
04h TLB даних: сторінки 4М, 4WSA, 8 входжень
06h Кеш інструкцій (LI): 8ДО, 4WSA, розмір рядка 32 байта
08h Кеш інструкцій (LI): 16ДО, 4WSA, розмір рядка 32 байта
0Ah Кеш даних (LI): 8ДО, 2WSA, розмір рядка 32 байта
0Ch Кеш даних (LI): 16ДО, 2WSA, розмір рядка 32 байта
40h Немає вторинного кеша
41h Вторинний кеш 128ДО, 4WSA, розмір рядка 32 байта
42h Вторинний кеш 256ДО, 4WSA, розмір рядка 32 байта
43h Вторинний кеш 512ДО, 4WSA, розмір рядка 32 байта
44h Вторинний кеш 1М, 4WSA, розмір рядка 32 байта
45h Вторинний кеш 2М, 4WSA, розмір рядка 32 байта

Наприклад, для Pentium Pro по CPUID(2) повернулися: EAX=03020101h, EBX=0, ECX=0, EDX=06040A42h. Це означає, що виклик потрібно робити однократно (AL=1); TLB інструкцій для сторінок 4До має 32 входження (01h), для сторінок 4М - 2 входження; TLB даних для сторінок 4До на 64 входження (03h), для сторінок 4М - на 8 входжень (04h); первинний кеш інструкцій - 8ДО (06h), даних - 8ДО (0Ah); вторинного кеш 256ДО (42h). Ця таблиця використає характеристики L2-кеша для більше точної ідентифікації процесора. Якщо L2-кеш відсутній, то це Intel Celeron, якщо розмір L2-кеша дорівнює 1 Мбайт або 2 Мбайт - Pentium II Xeon, в всіх інших випадках - Pentium II model 5 або Pentium II Xeon с L2-кешем обсягом 512 Кбайт.

CPUID(3) - одержання молодших 64 біт серійного номера процесора (Intel Processor Serial Number), доступно починаючи з Pentium III (сімейство 6, модель 7 і старше). EDX - середні 32 біта ідентифікатора; ECX - молодші 32 біта ідентифікатора. Повний ідентифікатор має довжину 96 біт. Старші 32 біта - код ідентифікації процесора, що повертає в EAX по CPUID(1). Доступність виклику визначається по біті PN регістра властивостей (після CPUID(1) біт EDX.18=1). Після апаратного скидання в процесорів, що підтримують повідомлення ідентифікатора, цей виклик дозволений. Заборонити повідомлення ідентифікатора до наступного апаратного скидання можна установкою в одиницю біта 21 регістра.

Фірма AMD розширила виклики CPUID. Для перевірки наявності розширень викликається CPUID з EAX=8000_0000h. При наявності розширень в EAX результатом буде число, більше 8000_0000h, - максимальний параметр розширеного виклику. Викликом EAX=8000_0001h можна визначити специфічні розширення архітектури від AMD. Наприклад, підтримка 3DNow! визначається по встановленому біті 31 регістра EDX.

Наведені вище фрагменти програм є основою для створення ассемблерних процедур або програм, у вигляді.asm файлів які транслюються в здійсненний.exe файл.

Інший спосіб: ці ж фрагменти можна скомпонувати в основний С -програмі, у вигляді ассемблерних вставок - з наступною повною компіляцією.

```
// Compile by BC++version 3.1
#include <stdio.h>
#include <string.h>
#include <conio.h>

void main() {
 char VendorSign[13];  // for to store our vendorstring
 unsigned long MaxEAX; //This will be used to store the maximum EAX
 char* Comp1[32];//This is the array that will hold the short names
                    //for our features bitmap.
  unsigned long REGEAX, REGEBX, REGECX, REGEDX;
  int dFamily, dModel, dStepping, dFamilyEx, dModelEx;
  char dType[10];
  int dComp1Supported[32];
  int dBrand, dCacheLineSize, dLogicalProcessorCount, dLocalAPICID;
  asm {
    XOR EAX, EAX
    //An efficient alternatvie to MOV EAX, 0x0
    db 0fh,0a2h
    //=cpuID,This instruction will load our registers.
    MOV dword ptr [VendorSign], EBX
    //Copy the first 4 bytes in the VendorString from EBX.
    MOV dword ptr [VendorSign+4], EDX
    //Copy the next 4 bytes.
    MOV  dword ptr [VendorSign+8],   ECX
    //Copy the next 4 bytes.
    MOV  dword ptr MaxEAX,           EAX
    //EAX contains the maximum value to call CPUID with. Copy
    //it to the MaxEAX variable.
  }
  VendorSign[12]=0;
//The last character in the VendorSign can be anything.
//To make sure that it stops at the last character we add
//a zero character at the end
  printf("Vendor string: %s\n", VendorSign);
  printf("Maximum EAX value: %i\n", MaxEAX);
  if(strcmp(VendorSign, "GenuineIntel")==0) {
    Comp1[0]="FPU";   //Floating Point Unit
    Comp1[1]="VME";   //Virtual Mode Extension
    Comp1[2]="DE";    //Debugging Extension
    Comp1[3]="PSE";   //Page Size Extension
    Comp1[4]="TSC";   //Time Stamp Counter
    Comp1[5]="MSR";   //Model Specific Registers
    Comp1[6]="PAE";   //Physical Address Extesnion
    Comp1[7]="MCE";   //Machine Check Extension
    Comp1[8]="CX8";   //CMPXCHG8 Instruction
    Comp1[9]="APIC";  //On-chip APIC Hardware
    Comp1[10]="";     //Reserved
    Comp1[11]="SEP";  //SYSENTER SYSEXIT
    Comp1[12]="MTRR"; //Machine Type Range Registers
    Comp1[13]="PGE";  //Global Paging Extension
    Comp1[14]="MCA";  //Machine Check Architecture
    Comp1[15]="CMOV"; //Conditional Move Instrction
    Comp1[16]="PAT";  //Page Attribute Table
    Comp1[17]="PSE-36"; //36-bit Page Size Extension
    Comp1[18]="PSN";  //96-bit Processor Serial Number
    Comp1[19]="CLFSH"; //CLFLUSH Instruction
    Comp1[20]="";     //Reserved
```

```
  Comp1[21]="DS";   //Debug Trace Store
  Comp1[22]="ACPI"; //ACPI Support
  Comp1[23]="MMX";  //MMX Technology
  Comp1[24]="FXSR"; //FXSAVE FXRSTOR (Fast save and restore)
  Comp1[25]="SSE";  //Streaming SIMD Extensions
  Comp1[26]="SSE2"; //Streaming SIMD Extensions 2
  Comp1[27]="SS";   //Self-Snoop
  Comp1[28]="HTT";  //Hyper-Threading Technology
  Comp1[29]="TM";   //Thermal Monitor Supported
  Comp1[30]="IA-64"; //IA-64 capable
  Comp1[31]="";     //Reserved
}
else if(strcmp(VendorSign, "AuthenticAMD")==0) {
  Comp1[0]="FPU";   //Floating Point Unit
  Comp1[1]="VME";   //Virtual Mode Extension
  Comp1[2]="DE";    //Debugging Extension
  Comp1[3]="PSE";   //Page Size Extension
  Comp1[4]="TSC";   //Time Stamp Counter
  Comp1[5]="MSR";   //Model Specific Registers
  Comp1[6]="PAE";   //Physical Address Extesnion
  Comp1[7]="MCE";   //Machine Check Extension
  Comp1[8]="CX8";   //CMPXCHG8 Instruction
  Comp1[9]="APIC";  //On-chip APIC Hardware
  Comp1[10]="";     //Reserved
  Comp1[11]="SEP";  //SYSENTER SYSEXIT
  Comp1[12]="MTRR"; //Machine Type Range Registers
  Comp1[13]="PGE";  //Global Paging Extension
  Comp1[14]="MCA";  //Machine Check Architecture
  Comp1[15]="CMOV"; //Conditional Move Instrction
  Comp1[16]="PAT";  //Page Attribute Table
  Comp1[17]="PSE-36"; //36-bit Page Size Extension
  Comp1[18]="";     //?
  Comp1[19]="MPC";  //MultiProcessing Capable
  Comp1[20]="";     //Reserved
  Comp1[21]="";     //?
  Comp1[22]="MIE";  //AMD Multimedia Instruction Extensions
  Comp1[23]="MMX";  //MMX Technology
  Comp1[24]="FXSR"; //FXSAVE FXRSTOR (Fast save and restore)
  Comp1[25]="SSE";  //Streaming SIMD Extensions
  Comp1[26]="";     //?
  Comp1[27]="";     //?
  Comp1[28]="";     //?
  Comp1[29]="";     //?
  Comp1[30]="3DNowExt"; //3DNow Instruction Extensions
  Comp1[31]="3DNow"; //3DNow Instructions
}
if(MaxEAX>=1) {
  asm {
    MOV     EAX,                    1
    db 0fh,0a2h
    MOV     [REGEAX],               EAX
    MOV     [REGEBX],               EBX
    MOV     [REGECX],               ECX
    MOV     [REGEDX],               EDX
  }
  dFamily=((REGEAX>>8)&0x);
  dModel=((REGEAX>>4)&0x);
  dStepping=(REGEAX&0x);
```

```
      dFamilyEx=((REGEAX>>20)&0xFF);
      dModelEx=((REGEAX>>16)&0x);
      switch(((REGEAX>>12)&0x7)) {
        case 0:
          strcpy(dType, "Original");
          break;
        case 1:
          strcpy(dType, "OverDrive");
          break;
        case 2:
          strcpy(dType, "Dual");
          break;
      }

      for(unsigned long C=1, Q=0;Q<32;C*=2, Q++) {
        dComp1Supported[Q]=(REGEDX&C)!=0?1:0;
      }


      dBrand=REGEBX&0xFF;
      dCacheLineSize=((strcmp(Comp1[19], "CLFSH")==0)
     &&(dComp1Supported[19]==1))?((REGEBX>>8)&0xFF)*8:-1;
      dLogicalProcessorCount=((strcmp(Comp1[28], "HTT")==0)
     &&(dComp1Supported[28]==1))?((REGEBX>>16)&0xFF):-1;
      dLocalAPICID=((REGEBX>>24)&0xFF); //This works on P4 or later
    }
    printf("%s\n", dType);
    printf("Family %i, Model %i, Stepping %i\n", dFamily, dModel,
dStepping);
    printf("Extended Family %i, Extended Model %i\n", dFamilyEx,
dModelEx);
    printf("Supported flags: ");
    for(unsigned long Q=0;Q<27;Q++) {
      if(dComp1Supported[Q]) {
        printf("%s ", Comp1[Q]);
      }
    }
    printf("\n");
    printf("CacheLineSize: %i\n", dCacheLineSize);
    printf("Logical processor count: %i\n", dLogicalProcessorCount);
    printf("Local APIC ID: %i\n", dLocalAPICID);
  }
```

Стратегії відновлення даних у кеш пам'яті.

WriteBack

У схемі відновлення зі зворотним записом використається біт "зміни" у поле тэга. Цей біт установлюється, якщо блок був обновлений новими даними і є більше пізнім, чим його оригінальна копія в основній пам'яті. Перед тим як записати блок з основної пам'яті в кеш-пам'ять, контролер перевіряє стан цього біта. Якщо він установлений, то контролер переписує даний блок в основну пам'ять перед завантаженням нових даних у кеш-пам'ять (тобто тільки тоді, коли вже не потрібні обновлені дані в кеш і вони заміщаються). Зворотний запис швидше наскрізний, тому що звичайно число випадків, коли блок

змінюється й повинен бути переписаний в основну пам'ять, менше числа випадків, коли ці блоки зчитуються й перезаписуються. Однак зворотний запис має кілька недоліків. По-перше, всі змінені блоки повинні бути переписані в основну пам'ять перед тим, як інший пристрій зможе одержати до них доступ. По-друге, у випадку катастрофічної відмови, наприклад, відключення живлення, коли вміст кеш-пам'яті губиться, але вміст основної пам'яті зберігається, не можна визначити, які місця в основній пам'яті містять застарілі дані. Нарешті, контролер кеш-пам'яті для зворотного запису містить більше (і більше складних) логічних мікросхем, чим контролер для наскрізного запису. Наприклад, коли система зі зворотним записом здійснює запис зміненого блоку на згадку, то вона формує адресу запису з тэга й виконує цикл зворотного запису точно так само, як і знову запитуваний доступ.

Writethrough

При відновленні кеш-пам'яті методом наскрізного запису контролер кеш-пам'яті одночасно обновляє вміст основної пам'яті. Інакше кажучи, основна пам'ять відбиває поточний уміст кеш-пам'яті. Швидке відновлення дозволяє перезаписувати будь-який блок у кеш-пам'яті в будь-який час без втрати даних. Система з наскрізним записом простий, але час, необхідний для запису в основну пам'ять, знижує продуктивність і збільшує кількість обігів по шині (що особливо помітно в мультізадачній системі).

Буферизованная наскрізний запис.

У схемі відновлення з буферізованій наскрізним записом будь-який запис в основну пам'ять буферизується, тобто інформація затримується в кеш-пам'яті перед записом в основну пам'ять (схеми кеш-пам'яті управляють доступом до основної пам'яті асинхронно стосовно роботи процесора). Потім процесор починає новий цикл до завершення циклу запису в основну пам'ять. Якщо за записом треба читання, то це кеш-попадание, тому що читання може бути виконане в той час, коли контролер кеш-пам'яті зайнятий відновленням основної пам'яті. Ця буферизація дозволяє уникнути зниження продуктивності, характерного для системи з наскрізним записом (запис виробляється в той момент, поки процесор читає з кеша й зайнятий чим чимсь крім відновлення даних). У цього методу є один істотний недолік. Тому що звичайно буферизується тільки одиночний запис, те два послідовні записи в основну пам'ять вимагають циклу очікування процесора. Крім цього, запис із пропущеним наступним читанням також вимагає очікування процесора. Стан очікування - це внутрішній стан, у яке входить процесор при відсутності синхронізуючих сигналів. Стан очікування використається для синхронізації процесора з повільною пам'яттю.

Запис із розміщенням і без.

Попередні підходи описують тільки випадки кеш-попадання при записі результату процесором. Однак випадки, коли обновлювані дані в Кеше відсутні, також можливі. Тоді дані пишуться в ОЗУ й потім копіюються в кеш. Запис без розміщення - дані не копіюються в кеш. Звичайно при використанні стратегії WriteThru розміщення не робиться, а при використанні зворотного запису робиться - є надія, що не прийде знову лізти на згадку для запису наступного разу.

Випадок запису нових даних з пам'яті в кеш при кеш-промаху.

Для випадку прямого відображення стратегія тривіальна - заміщається рядок, у якій може розташовуватися даних блок ОЗУ. Для асоціативної організації кеша (повністю або частково) треба вибирати, яку з рядків заміщати новими даними. Дві стратегії: випадкова або LRU (Least Recently Used) - заміняється та, котру довше в

Архітектура досягла таких успіхів, що при виконанні декількох команд можна вважати всю інформацію про комп'ютер, аж до виробника й апаратних можливостей. Також з'явилася можливість управляти апаратною конфігурацією процесора. У сучасних процесорах їсти можливість виправляти помилки шляхом перепрошивання мікрокоду процесора. Завдяки цьому можна виправити помилки, допущені при проектуванні мікропроцесора.

Починаючи із процесора Pentium, фірма Intel увела специфічні модельні регістри, які дозволяють дізнаватися про апаратні несправності процесора, його конфігурацію, а також управляти апаратними ресурсами процесора.

МОДЕЛЬНО СПЕЦЕФИЧЕСКИЕ РЕГІСТРИ

MSR розшифровуються як model-specific registers - модельно-специфічні регістри. Msr регістри підтримуються в моделях pentium, intel xeon, p6 родини, pentium 4. Фірми виробники не гарантують підтримку цих регістрів у майбутніх версіях ia-32 архітектури або що ці процесори будуть мати ті ж функції.

Ці регістри надають кошти програмного зв'язку з мікроархітектурою процесора, всі найважливіші настроювання які можна міняти програмно перебувають саме там.

Існує два рівні архітектури процесора: модельно-незалежний і модельно-модельно-залежний. До першого ставиться все те, що забезпечує сумісність процесорів від покоління до покоління: регістри загального призначення, система команд і т.д.

Модельно-модельно-залежний рівень становлять засоби, які можуть підтримуватися в одному процесорі, і не підтримуватися в наступних. Ці засоби потрібні, наприклад, для програмної взаємодії з якими-небудь функціями процесора, які присутні тільки в даному процесорі, ці функції підтримуються за допомогою особливих 64-розрядних регістрів msr.

Ці регістри призначені для керування розмаїтістю апаратних пристроїв і пов'язаних з ним програмним забезпеченням:

- лічильники для контролю роботи процесора. Ці лічильники дозволяють вибирати параметри роботи процесора, які можна перевірити або змінити. Цю інформацію можна використати для настроювання системи;

- способи налагодження. У родині p6 була уведена можливість установлювати контрольні крапки зупинки переривань, виключень, переходів.

- контроль машини. Процесор відтворить машинну перевірку архітектури, це забезпечує механізм для того, щоб знаходити й сповіщати про апаратні помилки: помилки системної шини, помилки кеша, tlb помилки, ecc помилки.

За допомогою msr регістрів також можна поміняти конфігурацію процесора. Множник частоти також можна змінити за допомогою msr регістрів, що приведе до зменшення енергоспоживання й падінню частоти роботи. Можна настроїти й тепловий моніторинг, що служить для того, що б при визначенні температури більше заданої автоматично включається механізм зменшення частоти, а разом з тим і тепловиділення.

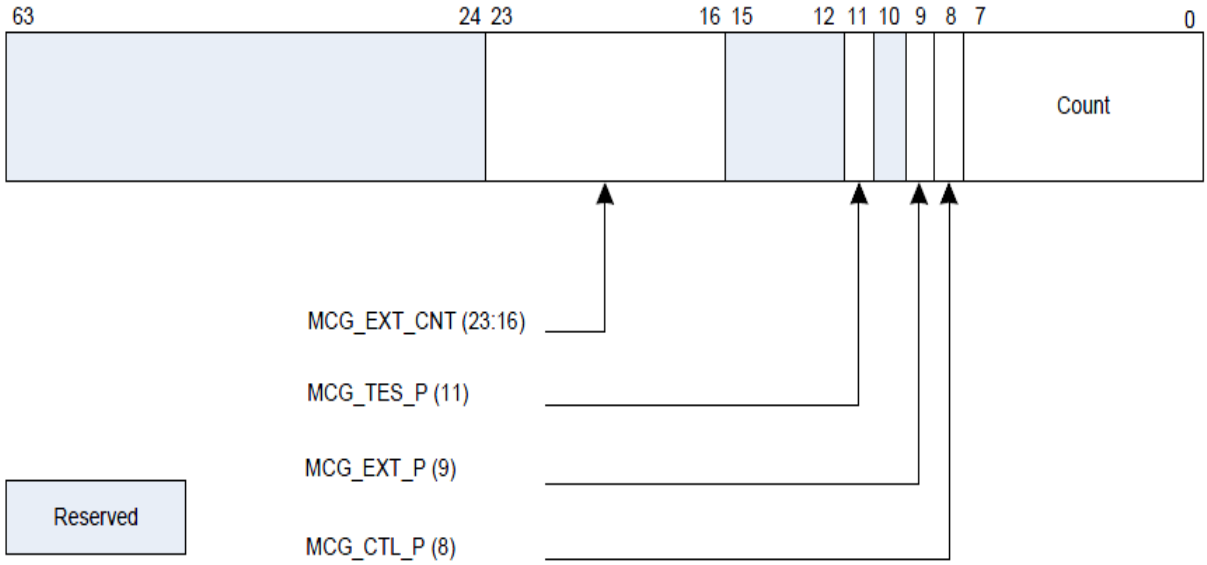
Msr-регістри мають свої власні імена (наприклад, ia32_time_stamp_counter), однак звернеться до них по іменах не можливо, а тому доводиться використати "безликі" номери, перераховані в додатку "model-specific registers" керівництва intel "ia-32 architecture software developer's manual volume 3: system programming guide", звідки, зокрема, можна довідатися, що msr-регістр ia32_time_stamp_counter має номер 10h.

Звернеться до msr регістра можна за допомогою команд rdmsr й wrmsr. Команда rdmsr (read from model specific register) виконує читання з msr-регістра. Дія команди полягає в перевірці двох умов: по-перше, перевіряється наявність нульового рівня привілейованості коду, по-друге, перевіряється наявність у регістрі ecx значення, що адресує один з msr-регістрів. Якщо хоча б одне із цих умов не виконується, то виконання команди rdmsr закінчується. Якщо виконуються обоє умови, то значення msr-регістра, адресуемого вмістом регістра ecx, міститься в парі 32-бітних регістрів edx:eax. Команда wrmsr (write to model specific register) робить запис значення в один з 64-розрядних msr-регістрів. Дія команди полягає в перевірці тих же двох умов: по-перше, перевіряється наявність нульового рівня привілейованості коду, по-друге, перевіряється наявність у регістрі ecx значення, що адресує один з msr-регістрів. Якщо хоча б одне із цих умов не виконується, то робота команди wrmsr закінчується. Якщо виконуються обоє

умови, то значення пари 32-бітних регістрів edx:eax пересилається в 64-бітний msr-регістр, номер якого заданий у регістрі ecx.

Msr регістри машинної перевірки містять інформацію, одержувану з регістрів глобального контролю, регістрів керування й декількох банків регістрів, які повідомляють про помилку. Кожен банк пов'язаний з відповідним апаратним модулем.

Модельний регістр mcg_cap(номер регістра 179) тільки для читання. Він містить наступні поля:



Малюнок 3.3 - загальний формат регістра mcg_cap

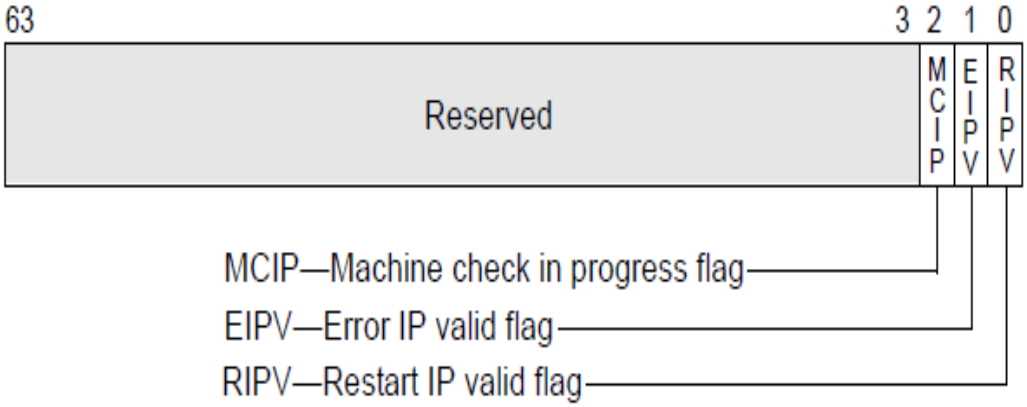
- count - містить кількість доступних банків на цьому процесорі;

- mcg_ctl_p - показує присутність регістра mcg_ctl;

- msg_etx_p - показує, чи підтримуються розширені регістри, починаючи з адреси 180h;

- msg_etx_ctl - указує число розширених регістрів машинної перевірки. Дійсно, тільки, якщо msg_etx_p установлений в 1. Це поле було уведено починаючи із процесора pentium 4.

Модельний регістр mcg_status (номер регістра 17a) описує початкове положення процесора, після того як виконалося виключення (мал. 3.3).



Малюнок 3.3 - загальний формат регістра mcg_status

- ripv - якщо цей прапор установлений в 1, те це означає, що при генерації виключення машинної помилки. Тобто є можливість запустити знову виконання програми;

- eipv - якщо дорівнює 1, то адреса помилки, що була записана в стек після генерації виключення, адреса прямо асоціюється з помилкою. Якщо цей прапор очищений, то адреса не асоціюється з помилкою;

- mcip - якщо цей біт, установлений в 1, те це означає що було сгенерировано виключення. Цей біт можна програмно як звести, так й очистити.

Регістр mcg_ctl(номер 17b) є присутнім, якщо біт msg_ctl_p регістра mcg_cap установлений в 1. Msg_ctl управляє повідомленням про генерації виключень машинної перевірки. Якщо в цей регістр записані одиниці, тобто особливості машинної перевірки, якщо записані всі нулі, то особливості машинної перевірки відключені. Інші комбінації не визначені.

Розширені регістри машинної перевірки - у процесорах pentium 4 й xeon кількість розширених регістрів машинної перевірки змінюється. Якщо прапор mcg_cap_p у модельному регістрі mcg_cap зведений в 1, то це вказує на наявність розширених регістрів. А поле mcg_cap_cnt - указує на їхню кількість у даній моделі процесора.

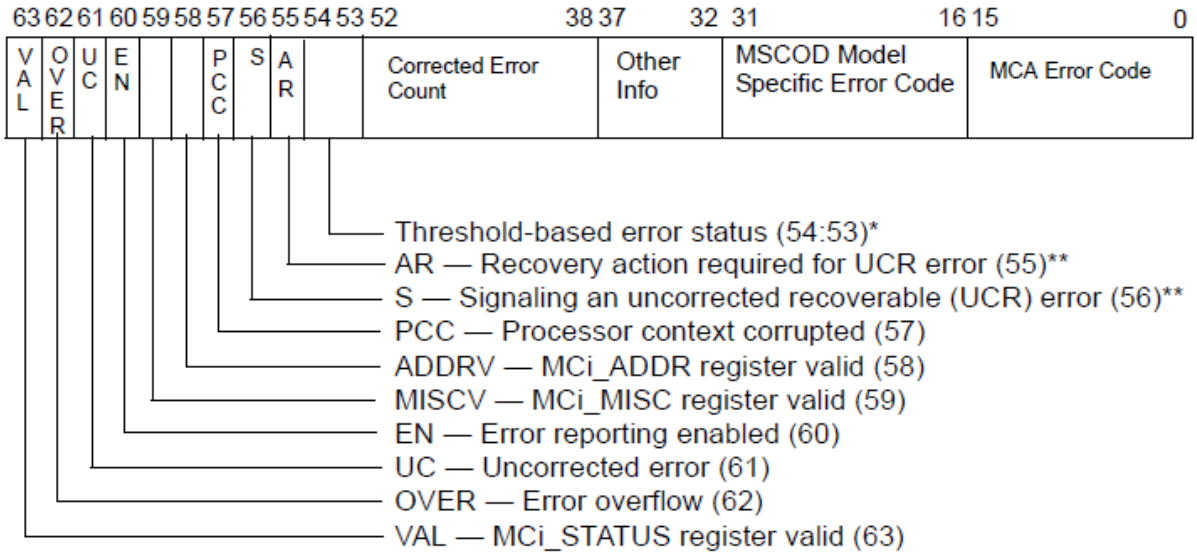
Регістри mcg_eax(180h), mcg_ebx(181h), mcg_ecx(182h), mcg_edx(184h), mcg_esi(184h), mcg_edi(185h), mcg_ebp(186h), mcg_esp(187h), mcg_eflags(188h), mcg_eip(189h) підтримуються, починаючи з моделі pentium 4.

У ці регістри заноситься відповідній назві регістрів дані (у випадку виникнення машинної помилки). Ці регістри доступні як для запису, так і для читання. Але при спробі запису в них не нульового значення генерується виключення загального захисту(#gp).

Кожний з банків регістрів може містити регістри: mci_ctl, mci_status, mci_addr, mci_misc(де i – це номер банку регістрів). Процесори pentium 4 містять 5 банків, які

сповіщають про помилки. Перший регістр нульового банку(mc0_ctl) завжди стартує з адреси 400_{16}.

Регістр mci_status містить наступні поля:



Малюнок 3.4 - загальний формат регістра mci_status

- mca error code - код машинної помилки (він однаковий для всіх моделей 32-х розрядної архітектури);

- model specific error code - унікальний код машинної помилки(він може бути різним для різних моделей процесорів);

- pcc - коли цей біт (біт 57) рівняється 1, те це повідомляє про те, що дана помилка, можливо, порушила процесор, що не гарантує надійного перезапуску процесора;

- addrv(біт 58) - ознака наявності адреси, на якій відбулося виключення машинної помилки, у регістрі mci_addr. Якщо addrv дорівнює 1, то в регістрі перебуває правильна адреса, якщо 0 - те, або адреса не вірний, або регістр mci_addr не підтримується на даному процесорі;

- miscv(біт 59) - якщо цей біт дорівнює 1, то значить що регістр mci_misc містить інформацію про помилку. Якщо 0 - те, або адреса не вірний, або регістр mci_addr не підтримується на даному процесорі;

- en(біт 60) - ознака того, що механізм пошуку машинних помилок працює;

- uc (біт 61) - якщо дорівнює 1, то процесор у стані сам виправити дану помилку. Якщо 0 - то не в змозі;

- over (біт 62) - ознака того, що виникла помилка, у той час коли попередня помилка усе ще перебуває в банку помилок (тобто Якщо біт val був уже встановлений (див. далі));

- val (біт 63) - ознака того, що інформація в цьому регістрі вірна.

Модельний регістр mci_misc - містить адреса команди або даних, які привели до помилки.

Для того щоб запустити даний механізм пошуку апаратних помилок необхідно його инициализировать. Для цього треба в регістр mcg_ctl заслати всі одиниці, якщо біт msg_ctl_p = 1 у регістрі msg_cap.

Також якщо родина процесора більше 6, те треба для банків від 0 до (mcg_cap_count - 1) виконати слід дії:

- у регістр mci_ctl заслати всі 1;
- у регістр mci_status заслати всі 0.

При виникненні помилок оброблювач виключення #mc процесора зобов'язаний аналізувати біти val кожного регістра mci_status, і якщо цей біт установлений в одиницю, то вважати поле mca регістра.

На процесорах pentium підтримують тільки два регістри машинної перевірки, а саме mc_addr (номер 0h) і mc_type (номер 1h). Де зберігаються адреса помилки й тип, відповідно.

Модельний регістр time stamp counter - лічильник тактів. Цей регістр входить у банк моніторингу продуктивності. Це 64-х бітний лічильник. Він стартує з початку запуску процесора й збільшується на 1 при кожному такті процесора.

Модельний регістр 79 служить для відновлення мікрокоду процесора. Якщо записати в нього значення лінійної адреси даних (за допомогою функції wrmsr), то це приведе до відновлення мікрокоду процесора.

==========

Тема 2. Операційна система обчислювальної системи.

Все програмне забезпечення прийняте ділити на дві частини: прикладне й системне. До прикладного програмного забезпечення, як правило, ставляться ігри, текстові процесори й т.п. Під системним програмним забезпеченням звичайно розуміють програми, що сприяють функціонуванню й розробці прикладних програм. Компілятор мови Си для звичайного програміста - системна програма, а для системного - прикладна.

Основні властивості операційної системи.

1. Операційна система - віртуальна машина

При розробці ОС широко застосовується абстрагування, що є важливим методом спрощення й дозволяє сконцентруватися на взаємодії высокорівневих компонентів системи, ігноруючи деталі їхньої реалізації. У цьому змісті ОС являє собою інтерфейс між користувачем і комп'ютером.

Архітектура більшості комп'ютерів на рівні машинних команд дуже незручна для використання прикладними програмами. Наприклад, робота з диском припускає знання внутрішнього пристрою його електронного компонента - контролера для уведення команд обертання диска, пошуку й форматування доріжок, читання й записи секторів і т.д. Середній програміст не в змозі враховувати всі особливості такої роботи, а повинен мати высокорівневу абстракцію, представляючи інформаційний простір диска як набір файлів. Файл можна відкривати для читання або запису, використати для одержання або скидання інформації, а потім закривати. Це простіше, ніж піклуватися про деталі переміщення головок дисків або організації роботи мотора. Аналогічним образом приховуються від програміста всі подробиці організації переривань, роботи таймера, керування пам'яттю й т.д. Всім цим займається операційна система. Таким чином, операційна система представляється користувачеві віртуальною машиною, з якою простіше мати справу, чим безпосередньо з устаткуванням комп'ютера.

2. Операційна система - менеджер ресурсів

Операційна система призначена для керування всіма частинами досить складної архітектури комп'ютера. Приміром, кілька програм, що працюють на одному комп'ютері, будуть намагатися одночасно здійснювати вивід на принтер. Ми одержали б мішанину рядків і сторінок, виведених різними програмами. Операційна система запобігає такого за рахунок буферизации інформації, призначеної для печатки, на диску й організації черги на печатку. Для багатокористувачевих комп'ютерів необхідність керування ресурсами і їхнього захисту ще більш очевидна. Отже, операційна система, як менеджер ресурсів, здійснює впорядкований і контрольований розподіл процесорів, пам'яті й інших ресурсів між різними програмами.

3. Операційна система - захисник користувачів і програм

Якщо обчислювальна система допускає спільну роботу декількох користувачів, то виникає проблема організації їхньої безпечної діяльності. Необхідно забезпечити схоронність інформації на диску, щоб ніхто не міг видалити або зашкодити чужі файли. Всю цю діяльність здійснює операційна система як організатор безпечної роботи користувачів й їхніх програм.

4. Операційна система - постійно функціонуюче ядро

Нарешті, операційна система - це програма, що постійно працює на комп'ютері й взаємодіюча з усіма прикладними програмами. Здавалося б, це абсолютно правильне визначення, але, як ми побачимо далі, у багатьох сучасних операційних системах постійно працює на комп'ютері лише частина операційної системи, що прийнято називати її ядром.

Нам простіше сказати не що є операційна система, а для чого вона потрібна й що вона робить. Для з'ясування цього питання розглянемо історію розвитку обчислювальних систем.

Еволюції обчислювальних систем

Ми будемо розглядати історію розвитку саме обчислювальних, а не операційних систем, тому що hardware і програмне забезпечення еволюціонували спільно, впливаючи один на одного. Поява нових технічних можливостей приводило до прориву в області створення зручних, ефективних і безпечних програм, а свіжі ідеї в програмній області стимулювали пошуки нових технічних рішень. Саме ці критерії - зручність, ефективність і безпека - відігравали роль факторів природного добору при еволюції обчислювальних систем.

Перший період (1945-1955 р.). *Лампові машини.*

У середині 40-х були створені перші лампові обчислювальні пристрої й з'явився принцип програми, що зберігається в пам'яті машини (John Von Neumann, червень 1945 р.). У той час група людей брала участь й у проектуванні, і в експлуатації, і в програмуванні обчислювальної машини. Це була скоріше науково-дослідна робота в області обчислювальної техніки, а не регулярне використання комп'ютерів як інструмент рішення яких-небудь практичних завдань із інших прикладних областей. Програмування здійснювалося винятково машинною мовою. Про операційні системи не було й мови, всі завдання організації обчислювального процесу вирішувалися вручну кожним програмістом з пульта керування. За пультом міг перебувати тільки один користувач. Програма завантажувалася в машину в найкращому разі з колоди перфокарт, а звичайно за допомогою панелі перемикачів.

Обчислювальна система виконувала одночасно тільки одну операцію (ввід-вивід або властиво обчислення). Налагодження програм велося з пульта керування за допомогою вивчення стану пам'яті й регістрів машини. Наприкінці цього періоду з'являється перше системне програмне забезпечення: в 1951-1952 р. виникають прообрази перших компіляторів із символічних мов (Fortran й ін.), а в 1954 р. Nat Rochester розробляє Асемблер для IBM-701.

Істотна частина часу йшла на підготовку запуску програми, а самі програми виконувалися строго послідовно. Такий режим роботи називається послідовною обробкою даних. У цілому перший період характеризується вкрай високою вартістю обчислювальних систем, їхньою малою кількістю й низькою ефективністю використання.

Другий період (1955 р.-почало 60-х)*. Комп'ютери на основі транзисторів. Пакетні операційні системи*

Із середини 50-х років почався наступний період в еволюції обчислювальної техніки, пов'язаний з появою нової технічної бази - напівпровідникових елементів. Застосування транзисторів привело до підвищення надійності комп'ютерів. Розміри комп'ютерів зменшилися. Знизилася вартість експлуатації й обслуговування обчислювальної техніки. Почалося використання ЕОМ комерційними фірмами. Одночасно спостерігається бурхливий розвиток алгоритмічних мов (LISP, COBOL, ALGOL-60, PL-1 і т.д.). З'являються перші дійсні компілятори, редактори зв'язків, бібліотеки математичних і службових підпрограм. Спрощується процес програмування. Пропадає необхідність звалювати на тих самих людей весь процес розробки й використання комп'ютерів. Саме в цей період відбувається поділ персоналу на програмістів й операторів, фахівців з експлуатації. Змінюється сам процес прогону програм. Тепер користувач приносить програму із вхідними даними у вигляді колоди перфокарт і вказує необхідні ресурси. Така колода одержує назву завдання. Оператор завантажує завдання в машину й запускає його на виконання. Отримані вихідні дані друкуються на принтері, і користувач одержує їх назад через якийсь час. Для підвищення ефективності використання комп'ютера завдання зі схожими ресурсами починають збирати разом, створюючи пакет завдань. З'являються перші системи пакетної обробки, які просто автоматизують запуск однієї програми з пакета за іншою і тим самим збільшують коефіцієнт завантаження процесора. Системи пакетної обробки стали прообразом сучасних операційних систем, вони були

першими системними програмами, призначеними для керування обчислювальним процесом.

Третій період (початок 60-х - 1980 р.). Комп'ютери на основі інтегральних мікросхем.

Наступний важливий період розвитку обчислювальних машин ставиться до початку 60-х - 1980 р. У цей час у технічній базі відбувся перехід від окремих напівпровідникових елементів типу транзисторів до інтегральних мікросхем. Росте складність і кількість завдань, розв'язуваних комп'ютерами. Підвищується продуктивність процесорів.

Підвищенню ефективності використання процесорного часу заважає низька швидкість роботи механічних пристроїв вводу-виводу. Спочатку операції вводу-виводу здійснювалися в режимі off-line. Надалі вони починають виконуватися на тім же комп'ютері, що робить обчислення, тобто в режимі on-line. Такий прийом одержує назву spooling (скорочення від Simultaneous Peripheral Operation On Line) або підкачування даних. Введення техніки підкачування в пакетні системи дозволило получити реальні операції вводу-виводу одного завдання з виконанням іншого завдання, але зажадало розробки апарата переривань для повідомлення процесора про закінчення цих операцій. Магнітні стрічки були пристроями послідовного доступу, тобто інформація зчитувалася з них у тім порядку, у якому була записана. Поява магнітного диска, для якого не важливий порядок читання інформації, тобто пристрою прямого доступу, привело до подальшого розвитку обчислювальних систем. При обробці пакета завдань на магнітній стрічці черговість запуску завдань визначалася порядком їхнього уведення. При обробці пакета завдань на магнітному диску з'явилася можливість вибору чергового виконуваного завдання. Пакетні системи починають займатися плануванням завдань: залежно від наявності запитаних ресурсів, терміновості обчислень і т.д. на рахунок вибирається те або інше завдання.

Подальше підвищення ефективності використання процесора було досягнуто за допомогою мультипрограмування. Ідея мультипрограмування полягає в наступному: поки одна програма виконує операцію вводу-виводу, процесор не простоює, як це відбувалося при однопрограмному режимі, а виконує іншу програму. Коли операція вводу-виводу закінчується, процесор повертається до виконання першої програми. Ця ідея нагадує поводження викладача й студентів на іспиті. Поки один студент (програма) обмірковує відповідь на питання (операція вводу-виводу), викладач (процесор) вислухує відповідь іншого студента (обчислення). Природно, така ситуація вимагає наявності в кімнаті декількох студентів. Точно так само мультипрограмування вимагає наявності в пам'яті декількох програм одночасно. При цьому кожна програма завантажується у свою ділянку оперативної пам'яті, називана розділом, і не повинна впливати на виконання іншої програми. (Студенти сидять за окремими столами й не підказують один одному.) Поява мультипрограмування вимагає дійсної революції в будові обчислювальної системи. Особливу роль тут грає апаратна підтримка (багато апаратних нововведень з'явилися ще на попередньому етапі еволюції), найбільш істотні особливості якої перераховані нижче.

- Реалізація захисних механізмів. Програми не повинні мати самостійного доступу до розподілу ресурсів, що приводить до появи привілейованих і непривілейованих команд. Привілейовані команди, наприклад команди вводу-виводу, можуть виконуватися тільки операційною системою. Говорять, що вона працює в привілейованому режимі. Перехід керування від прикладної програми до ОС супроводжується контрольованою зміною режиму. Крім того, це захист пам'яті, що дозволяє ізолювати конкуруючі користувальницькі програми друг від друга, а ОС - від програм користувачів.
- Наявність переривань. Зовнішні переривання сповіщають ОС про те, що відбулася асинхронна подія, наприклад завершилася операція вводу-виводу. Внутрішні переривання (зараз їх прийнято називати винятковими ситуаціями) виникають,

коли виконання програми привело до ситуації, що вимагає втручання ОС, наприклад розподіл на нуль або спроба порушення захисту.
- Розвиток паралелізму в архітектурі. Прямий доступ до пам'яті й організація каналів вводу-виводу дозволили звільнити центральний процесор від рутинних операцій.

Не менш важлива в організації мультипрограмування роль операційної системи. Вона відповідає за наступні операції.

- Організація інтерфейсу між прикладною програмою й ОС за допомогою системних викликів.
- Організація черги із завдань у пам'яті й виділення процесора одному із завдань зажадало планування використання процесора.
- Перемикання з одного завдання на інше вимагає збереження вмісту регістрів і структур даних, необхідних для виконання завдання, інакше кажучи, контексту для забезпечення правильного продовження обчислень.
- Оскільки пам'ять є обмеженим ресурсом, потрібні стратегії керування пам'яттю, тобто потрібно впорядкувати процеси розміщення, заміщення й вибірки інформації з пам'яті.
- Організація зберігання інформації на зовнішніх носіях у вигляді файлів і забезпечення доступу до конкретного файлу тільки певним категоріям користувачів.
- Оскільки програмам може знадобитися зробити санкціонований обмін даними, необхідно їх забезпечити засобами комунікації.
- Для коректного обміну даними необхідно дозволяти конфліктні ситуації, що виникають при роботі з різними ресурсами й передбачити координацію програмами своїх дій, тобто постачити систему засобами синхронізації.

Мультипрограмні системи забезпечили можливість більше ефективного використання системних ресурсів (наприклад, процесора, пам'яті, периферійних пристроїв), але вони ще довго залишалися пакетними. Користувач не міг безпосередньо взаємодіяти із завданням і повинен був передбачити за допомогою керуючих карт всі можливі ситуації. Налагодження програм як і раніше займали багато часу й вимагала вивчення многостраничных роздруківок умісту пам'яті й регістрів або використання отладочной печатки.

Поява електронно-променевих дисплеїв і переосмислення можливостей застосування клавіатур поставили на чергу рішення цієї проблеми. Логічним розширенням систем мультипрограмування стали time-sharing системи, або системи поділу часу[1)]. У них процесор перемикається між завданнями не тільки на час операцій вводу-виводу, але й просто по закінченні певного часу. Ці перемикання відбуваються так часто, що користувачі можуть взаємодіяти зі своїми програмами під час їхнього виконання, тобто интерактивно. У результаті з'являється можливість одночасної роботи декількох користувачів на одній комп'ютерній системі. У кожного користувача для цього повинна бути хоча б одна програма в пам'яті. Щоб зменшити обмеження на кількість працюючих користувачів, була впроваджена ідея неповного знаходження виконує программы, що, в оперативній пам'яті. Основна частина програми перебуває на диску, і фрагмент, якому необхідно в цей момент виконувати, може бути завантажений в оперативну пам'ять, а непотрібний - викачаний назад на диск. Це реалізується за допомогою механізму віртуальної пам'яті. Основним достоїнством такого механізму є створення ілюзії необмеженої оперативної пам'яті ЕОМ.

У системах поділу часу користувач одержав можливість ефективно робити налагодження програми в інтерактивному режимі й записувати інформацію на диск, не використовуючи перфокарти, а безпосередньо із клавіатури. Поява on-line-файлів привело до необхідності розробки розвинених файлових систем.

Паралельно внутрішньої еволюції обчислювальних систем відбувалася й зовнішня їхня еволюція. До початку цього періоду обчислювальні комплекси були, як правило,

несумісні. Кожний мав власну операційну систему, свою систему команд і т.д. У результаті програму, що успішно працює на одному типі машин, необхідно було повністю переписувати й заново налагоджувати для виконання на комп'ютерах іншого типу. На початку третього періоду з'явилася ідея створення сімейств програмно сумісних машин, що працюють під керуванням однієї й тієї ж операційної системи. Першим сімейством програмно сумісних комп'ютерів, побудованих на інтегральних мікросхемах, стала серія машин IBM/360. Розроблене на початку 60-х років, це сімейство значно перевершувало машини другого покоління за критерієм ціна/продуктивність. За ним пішла лінія комп'ютерів PDP, несумісних з лінією IBM, і кращою моделлю в ній стала PDP-11. Сила "однієї родини" була одночасно і її слабістю. Широкі можливості цієї концепції (наявність всіх моделей: від міні-комп'ютерів до гігантських машин; достаток різноманітної периферії; різне оточення; різні користувачі) породжували складну й громіздку операційну систему. Мільйони рядків Асемблера, написані тисячами програмістів, містили безліч помилок, що викликало безперервний потік публікацій про їх і спроб виправлення. Тільки в операційній системі OS/360 утримувалося більше 1000 відомих помилок. Проте ідея стандартизації операційних систем була широко впроваджена у свідомість користувачів і надалі одержала активний розвиток.

Четвертий період (з 1980 р. по теперішній час). Персональні комп'ютери. Класичні, мережні й розподілені системи

Наступний період в еволюції обчислювальних систем пов'язаний з появою більших інтегральних схем (БІС). У ці роки відбулося різке зростання ступеня інтеграції й зниження вартості мікросхем. Комп'ютер, що не відрізняється по архітектурі від PDP-11, за ціною й простотою експлуатації став доступний окремій людині, а не відділу підприємства або університету. Наступила ера персональних комп'ютерів. Спочатку персональні комп'ютери призначалися для використання одним користувачем в однопрограмному режимі, що спричинило деградацію архітектури цих ЕОМ й їхніх операційних систем (зокрема, пропала необхідність захисту файлів і пам'яті, планування завдань і т.п.).

Комп'ютери стали використатися не тільки фахівцями, що зажадало розробки "дружнього" програмного забезпечення.

Однак ріст складності й розмаїтості завдань, розв'язуваних на персональних комп'ютерах, необхідність підвищення надійності їхньої роботи привели до відродження практично всіх рис, характерних для архітектури більших обчислювальних систем.
У середині 80-х стали бурхливо розвиватися мережі комп'ютерів, у тому числі персональних, працюючих під керуванням мережних або розподілених операційних систем.

У мережних операційних системах користувачі можуть одержати доступ до ресурсів іншого мережного комп'ютера, тільки вони повинні знати про їхню наявність і вміти це зробити. Кожна машина в мережі працює під керуванням своєї локальної операційної системи, що відрізняється від операційної системи автономного комп'ютера наявністю додаткових засобів (програмною підтримкою для мережних интерфейсных пристроїв і доступу до вилучених ресурсів), але ці доповнення не міняють структуру операційної системи.

Розподілена система, навпроти, зовні виглядає як звичайна автономна система. Користувач не знає й не повинен знати, де його файли зберігаються - на локальній або вилученій машині - і де його програми виконуються. Він може взагалі не знати, чи підключений його комп'ютер до мережі. Внутрішня будова розподіленої операційної системи має істотні відмінності від автономних систем.

Надалі автономні операційні системи ми будемо називати класичними операційними системами.

Переглянувши етапи розвитку обчислювальних систем, ми можемо виділити шість основних функцій, які виконували класичні операційні системи в процесі еволюції:

- Планування завдань і використання процесора.
- Забезпечення програм засобами комунікації й синхронізації.
- Керування пам'яттю.
- Керування файловою системою.
- Керування вводом-виводом.
- Забезпечення безпеки

Кожна з наведених функцій звичайно реалізована у вигляді підсистеми, що є структурним компонентом ОС. У кожній операційній системі ці функції, звичайно, реалізовувалися по-своєму, у різному обсязі. Вони не були споконвічно придумані як складові частини операційних систем, а з'явилися в процесі розвитку, у міру того як обчислювальні системи ставали усе більше зручними, ефективними й безпечними. Еволюція обчислювальних систем, створених людиною, пішла по такому шляху, але ніхто ще не довів, що це єдино можливий шлях їхнього розвитку. Операційні системи існують тому, що на даний момент їхнє існування - це розумний спосіб використання обчислювальних систем. Розгляд загальних принципів й алгоритмів реалізації їхніх функцій і становить зміст більшої частини нашого курсу, у якому будуть послідовно описані перераховані підсистеми.

Основні поняття, концепції ОС

У процесі еволюції виникло кілька важливих концепцій, які стали невід'ємною частиною теорії й практики ОС. Розглянуті в даному розділі поняття будуть зустрічатися й роз'яснятися протягом усього курсу. Тут дається їхній короткий опис.

Системні виклики

У будь-якій операційній системі підтримується механізм, що дозволяє користувальницьким програмам звертатися до послуг ядра ОС. В операційних системах найбільш відомої радянської обчислювальної машини БЭСМ-6 відповідні засоби "спілкування" з ядром називалися экстракодами, в операційних системах IBM вони називалися системними макрокомандами й т.д. В ОС Unix такі засоби називають системними викликами.

Системні виклики (system calls) - це інтерфейс між операційною системою й користувальницькою програмою. Вони створюють, видаляють і використають різні об'єкти, головні з яких - процеси й файли. Користувальницька програма запитує сервіс в операційної системи, здійснюючи системний виклик. Є бібліотеки процедур, які завантажують машинні регістри певними параметрами й здійснюють переривання процесора, після чого керування передається оброблювачеві даного виклику, що входить у ядро операційної системи. Ціль таких бібліотек - зробити системний виклик схожим на звичайний виклик підпрограми.

Основна відмінність полягає в тому, що при системному виклику завдання переходить у привілейований режим або режим ядра (kernel mode). Тому системні виклики іноді ще називають програмними перериваннями, на відміну від апаратних переривань, які частіше називають просто перериваннями.

У цьому режимі працює код ядра операційної системи, причому виконується він в адресному просторі й у контексті його завдання, що викликало. Таким чином, ядро операційної системи має повний доступ до пам'яті користувальницької програми, і при системному виклику досить передати адреси однієї або декількох областей пам'яті з параметрами виклику й адреси однієї або декількох областей пам'яті для результатів виклику.

У більшості операційних систем системний виклик здійснюється командою програмного переривання (INT). Програмне переривання - це синхронна подія, що може бути повторене при виконанні того самого програмного коду.

Переривання

Переривання (hardware interrupt) - це подія, генерируемое зовнішнім (стосовно процесора) пристроєм. За допомогою апаратних переривань апаратури або інформує

центральний процесор про те, що відбулася яка-небудь подія, що вимагає негайної реакції (наприклад, користувач нажав клавішу), або повідомляє про завершення асинхронної операції вводу-виводу (наприклад, закінчене читання даних з диска в основну пам'ять). Важливий тип апаратних переривань - переривання таймера, які генеруються періодично через фіксований проміжок часу. Переривання таймера використаються операційною системою при плануванні процесів. Кожен тип апаратних переривань має власний номер, що однозначно визначає джерело переривання. Апаратне переривання - це асинхронна подія, тобто воно виникає поза залежністю від того, який код виконується процесором у цей момент. Обробка апаратного переривання не повинна враховувати, який процес є поточним.

Виняткові ситуації

Виняткова ситуація (exception) - подія, що виникає в результаті спроби виконання програмою команди, що з якихось причин не може бути виконана до кінця. Прикладами таких команд можуть бути спроби доступу до ресурсу при відсутності достатніх привілеїв або звертання до відсутньої сторінки пам'яті. Виняткові ситуації, як і системні виклики, є синхронними подіями, що виникають у контексті поточного завдання. Виняткові ситуації можна розділити на поправні й непоправні. До поправного ставляться такі виняткові ситуації, як відсутність потрібної інформації в оперативній пам'яті. Після усунення причини поправної виняткової ситуації програма може виконуватися далі. Виникнення в процесі роботи операційної системи поправних виняткових ситуацій уважається нормальним явищем. Непоправні виняткові ситуації найчастіше виникають у результаті помилок у програмах (наприклад, розподіл на нуль). Звичайно в таких випадках операційна система реагує завершенням програми, що викликала виняткову ситуацію.

Файли

Файли призначені для зберігання інформації на зовнішніх носіях, тобто прийнято, що інформація, записана, наприклад, на диску, повинна перебувати усередині файлу. Звичайно під файлом розуміють іменовану частину простору на носії інформації. Головне завдання файлової системи (file system) - сховати особливості вводу-виводу й дати програмістові просту абстрактну модель файлів, незалежних від пристроїв. Для читання, створення, видалення, запису, відкриття й закриття файлів також є велика категорія системних викликів (створення, видалення, відкриття, закриття, читання й т.д.). Користувачам добре знайомі такі пов'язані з організацією файлової системи поняття, як каталог, що текет каталог, кореневий каталог, шлях. Для маніпулювання цими об'єктами в операційній системі є системні виклики. Файлова система ОС описана в лекціях 11-12.

Монолітне ядро

По суті справи, операційна система - це звичайна програма, тому було б логічно й організувати її так само, як улаштована більшість програм, тобто скласти із процедур і функцій. У цьому випадку компонента операційної системи є не самостійними модулями, а складовими частинами однієї великої програми. Така структура операційної системи називається монолітним ядром (monolithic kernel). Монолітне ядро являє собою набір процедур, кожна з яких може викликати кожну. Всі процедури працюють у привілейованому режимі. Таким чином, монолітне ядро - це така схема операційної системи, при якій всі її компоненти є складовими частинами однієї програми, використають загальні структури даних і взаємодіють один з одним шляхом безпосереднього виклику процедур. Для монолітної операційної системи ядро збігається з усією системою.

У багатьох операційних системах з монолітним ядром зборка ядра, тобто його компіляція, здійснюється окремо для кожного комп'ютера, на який установлюється операційна система. При цьому можна вибрати список устаткування й програмних протоколів, підтримка яких буде включена в ядро. Тому що ядро є єдиною програмою, перекомпіляція - це єдиний спосіб додати в нього нові компоненти або виключити невикористовувані. Слід зазначити, що присутність у ядрі зайвих компонентів украй

небажано, тому що ядро завжди повністю розташовується в оперативній пам'яті. Крім того, виключення непотрібних компонентів підвищує надійність операційної системи в цілому.

Монолітне ядро - найстарший спосіб організації операційних систем. Прикладом систем з монолітним ядром є більшість Unix-систем.

Навіть у монолітних системах можна виділити деяку структуру. Як у бетонній брилі можна розрізнити вкраплення щебінки, так й у монолітному ядрі виділяються вкраплення сервісних процедур, що відповідають системним викликам. Сервісні процедури виконуються в привілейованому режимі, тоді як користувальницькі програми - у непривілейованому. Для переходу з одного рівня привілеїв на іншій іноді може використатися головна сервісна програма, що визначає, який саме системний виклик був зроблений, коректність вхідних даних для цього виклику й передавальне керування відповідній сервісній процедурі з переходом у привілейований режим роботи. Іноді виділяють також набір програмних утиліт, які допомагають виконувати сервісні процедури.

Багаторівневі системи (Layered systems)

Продовжуючи структуризацію, можна розбити всю обчислювальну систему на ряд більше дрібних рівнів з добре певними зв'язками між ними, так щоб об'єкти рівня N могли викликати тільки об'єкти рівня N-1. Нижнім рівнем у таких системах звичайно є hardware, верхнім рівнем - інтерфейс користувача. Чим нижче рівень, тим більше привілейовані команди й дії може виконувати модуль, що перебуває на цьому рівні. Уперше такий підхід був застосований при створенні системи THE (Technishe Hogeschool Eindhoven) Дейкстрой (Dijkstra) і його студентами в 1968 р. Ця система мала наступні рівні:

5	Интерфейс пользователя
4	Управление вводом-выводом
3	Драйвер устройства связи оператора и консоли
2	Управление памятью
1	Планирование задач и процессов
0	Hardware

Рис. 1.2. Листкова система THE

При використанні операцій нижнього шару не потрібно знати, як вони реалізовані, потрібно лише розуміти, що вони роблять. Листкові системи добре тестируются. Налагодження починається з нижнього шару й проводяться послойно. При виникненні помилки ми можемо бути впевнені, що вона перебуває в тестируемом шарі. Листкові системи добре модифікуються. При необхідності можна замінити лише один шар, не торкаючи інші. Але листкові системи складні для розробки: важко правильно визначити порядок шарів і що до якого шару ставиться. Листкові системи менш ефективні, чим монолітні. Так, наприклад, для виконання операцій вводу-виводу програмі користувача прийде послідовно проходити всі шари від верхнього до нижнього.

Віртуальні машини

На початку лекції ми говорили про погляд на операційну систему як на віртуальну машину, коли користувачеві немає необхідності знати деталі внутрішнього пристрою комп'ютера. Він працює з файлами, а не з магнітними головками й двигуном; він працює з величезною віртуальною, а не обмеженою реальною оперативною пам'яттю; його мало

хвилює, єдиний він на машині чи користувач ні. Розглянемо трохи інший підхід. Нехай операційна система реалізує віртуальну машину для кожного користувача, але не спрощуючи йому життя, а, навпаки, ускладнюючи. Кожна така віртуальна машина з'являється перед користувачем як голе залізо - копія всього hardware в обчислювальній системі, включаючи процесор, привілейовані й непривілейовані команди, пристрої вводу-виводу, переривання й т.д. І він залишається із цим залізом один на один. При спробі звернутися до такого віртуального заліза на рівні привілейованих команд у дійсності відбувається системний виклик реальної операційної системи, що і робить всі необхідні дії. Такий підхід дозволяє кожному користувачеві завантажити свою операційну систему на віртуальну машину й робити з нею все, що душа побажає.

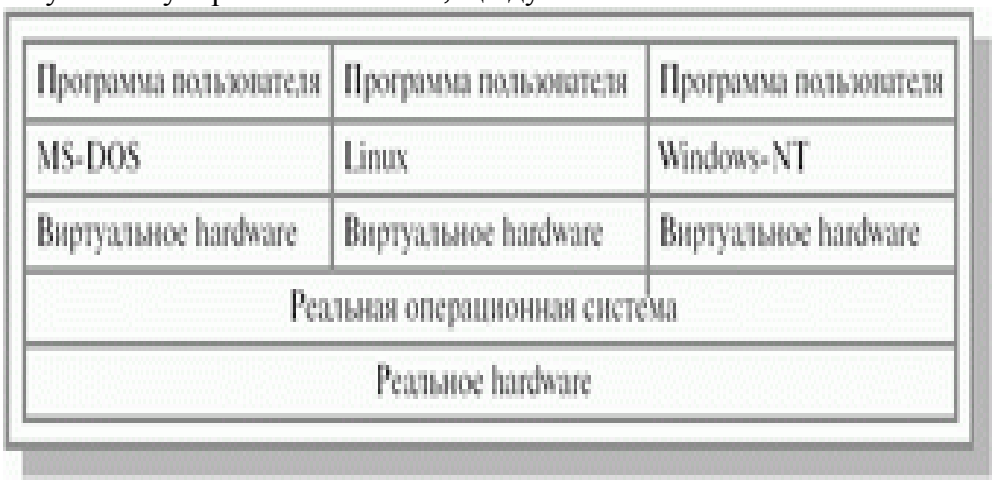


Рис. 1.3. Варіант віртуальної машини

Першою реальною системою такого роду була система CP/CMS, або VM/370, як неї називають зараз, для сімейства машин IBM/370.

Недоліком таких операційних систем є зниження ефективності віртуальних машин у порівнянні з реальним комп'ютером, і, як правило, вони дуже громіздкі. Перевага ж полягає у використанні на одній обчислювальній системі програм, написаних для різних операційних систем.

Мікроядерна архітектура

Сучасна тенденція в розробці операційних систем складається в перенесенні значної частини системного коду на рівень користувача й одночасної мінімізації ядра. Мова йде про підхід до побудови ядра, називаній мікроядерною архітектурою (microkernel architecture) операційної системи, коли більшість її складових є самостійними програмами. У цьому випадку взаємодія між ними забезпечує спеціальний модуль ядра, називаний мікроядром. Мікроядро працює в привілейованому режимі й забезпечує взаємодію між програмами, планування використання процесора, первинну обробку переривань, операції вводу-виводу й базове керування пам'яттю.

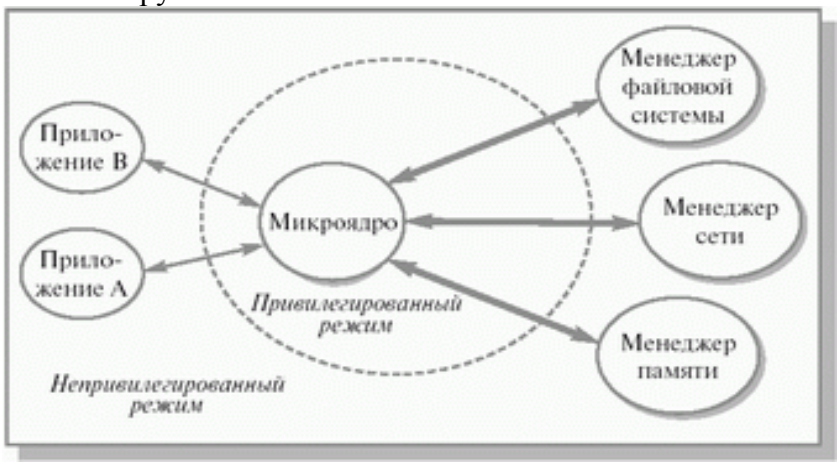


Рис. 1.4. Мікроядерна архітектура операційної системи

Інші компоненти системи взаємодіють один з одним шляхом передачі повідомлень через мікроядро.

Основне достоїнство мікроядерної архітектури - високий ступінь модульности ядра операційної системи. Це істотно спрощує додавання в нього нових компонентів. У мікроядерній операційній системі можна, не перериваючи її роботи, завантажувати й вивантажувати нові драйвери, файлові системи й т.д. Істотно спрощується процес налагодження компонентів ядра, тому що нова версія драйвера може завантажуватися без перезапуску всієї операційної системи. Компоненти ядра операційної системи нічим принципово не відрізняються від користувальницьких програм, тому для їхнього налагодження можна застосовувати звичайні засоби. Мікроядерна архітектура підвищує надійність системи, оскільки помилка на рівні непривілейованої програми менш небезпечна, чим відмова на рівні режиму ядра.

У той же час мікроядерна архітектура операційної системи вносить додаткові накладні витрати, пов'язані з передачею повідомлень, що істотно впливає на продуктивність. Для того щоб мікроядерна операційна система по швидкості не уступала операційним системам на базі монолітного ядра, потрібно дуже акуратно проектувати розбивку системи на компоненти, намагаючись мінімізувати взаємодію між ними. Таким чином, основна складність при створенні мікроядерних операційних систем - необхідність дуже акуратного проектування.

Змішані системи

Всі розглянуті підходи до побудови операційних систем мають свої достоїнства й недоліки. У більшості випадків сучасні операційні системи використають різні комбінації цих підходів. Так, наприклад, ядро операційної системи Linux являє собою монолітну систему з елементами мікроядерної архітектури. При компіляції ядра можна дозволити динамічне завантаження й вивантаження дуже багатьох компонентів ядра - так званих модулів. У момент завантаження модуля його код завантажується на рівні системи й зв'язується з іншою частиною ядра. Усередині модуля можуть використатися будь-які експортовані ядром функції.

Іншим прикладом змішаного підходу може служити можливість запуску операційної системи з монолітним ядром під управлінням мікроядра. Так улаштовані 4.4BSD й MkLinux, засновані на мікроядрі Mach. Мікроядро забезпечує керування віртуальною пам'яттю й роботу низкоуровневых драйверів. Всі інші функції, у тому числі взаємодія із прикладними програмами, здійснюється монолітним ядром. Даний підхід сформувався в результаті спроб використати переваги мікроядерної архітектури, зберігаючи по можливості добре налагоджений код монолітного ядра.

Найбільше тісно елементи мікроядерної архітектури й елементи монолітного ядра переплетені в ядрі Windows NT. Хоча Windows NT часто називають мікроядерною операційною системою, це не зовсім так. Мікроядро NT занадто велик (більше 1 Мбайт), щоб носи приставк "мікро". Компоненти ядра Windows NT розташовуються у витісняє памяти, що, і взаємодіють один з одним шляхом передачі повідомлень, як і покладено в мікроядерних операційних системах. У той же час усе компоненти ядра працюють в одному адресному просторі й активно використають загальні структури даних, що властиво операційним системам з монолітним ядром. На думку фахівців Microsoft, причина проста: чисто мікроядерний дизайн комерційно невигідний, оскільки неефективно.

Таким чином, Windows NT можна з повним правом назвати гібридною операційною системою.

Класифікація ОС

Існує кілька схем класифікації операційних систем. Нижче наведена класифікація по деяких ознаках з погляду користувача.

Реалізація многозадачности

По числу одночасно виконуваних завдань операційні системи можна розділити на два класи:

- многозадачные (Unix, OS/2, Windows);
- однозадачные (наприклад, MS-DOS).

Многозадачная ОС, вирішуючи проблеми розподілу ресурсів і конкуренції, повністю реалізує мультипрограмний режим відповідно до вимог роздягнула "Основні поняття, концепції ОС".

Многозадачный режим, що втілює в собі ідею поділу часу, називається що витісняє (preemptive). Кожній програмі виділяється квант процесорного часу, після закінчення якого керування передається іншій програмі. Говорять, що перша програма буде витиснута. У режимі, що витісняє, працюють користувальницькі програми більшості комерційних ОС.

У деяких ОС (Windows 3.11, наприклад) користувальницька програма може монополізувати процесор, тобто працювати в режимі, що не витісняє. Як правило, у більшості систем не підлягає витисненню код властиво ОС. Відповідальні програми, зокрема завдання реального часу, також не витісняються. Більш докладно про це розказано в лекції, присвяченої плануванню роботи процесора.

По наведених прикладах можна судити про приблизність класифікації. Так, в ОС MS-DOS можна організувати запуск дочірнього завдання й наявність у пам'яті двох і більше завдань одночасно. Однак ця ОС традиційно вважається однозадачной, головним чином через відсутність захисних механізмів і комунікаційних можливостей.

Підтримка многопользовательского режиму

По числу одночасно працюючих користувачів ОС можна розділити на:

- однокористувальницькі (MS-DOS, Windows 3.x);
- многопользовательские (Windows NT, Unix).

Найбільш істотна відмінність між цими ОС полягає в наявності в многопользовательских систем механізмів захисту персональних даних кожного користувача.

Многопроцессорна обробка

Аж до недавнього часу обчислювальні системи мали один центральний процесор. У результаті вимог до підвищення продуктивності з'явилися многопроцессорные системи, що складаються із двох і більше процесорів загального призначення, що здійснюють паралельне виконання команд. Підтримка мультипроцессирования є важливою властивістю ОС і приводить до ускладнення всіх алгоритмів керування ресурсами. Многопроцессорная обробка реалізована в таких ОС, як Linux, Solaris, Windows NT, і ряді інших.

Многопроцессорные ОС розділяють на симетричні й асиметричні. У симетричних ОС на кожному процесорі функціонує те саме ядро, і завдання може бути виконана на будь-якому процесорі, тобто обробка повністю децентралізована. При цьому кожному із процесорів доступна вся пам'ять.

В асиметричних ОС процесори нерівноправні. Звичайно існує головний процесор (master) і підлеглі (slave), завантаження й характер роботи яких визначає головний процесор.

Системи реального часу

У розряд многозадачных ОС, поряд з пакетними системами й системами поділу часу, включаються також системи реального часу, що не згадувалися дотепер.

Вони використаються для керування різними технічними об'єктами або технологічними процесами. Такі системи характеризуються гранично припустимим часом реакції на зовнішню подію, протягом якого повинна бути виконана програма, що управляє об'єктом. Система повинна обробляти дані, що надходять, швидше, ніж вони можуть надходити, причому від декількох джерел одночасно.

Настільки тверді обмеження позначаються на архітектурі систем реального часу, наприклад, у них може отсутствовать віртуальна пам'ять, підтримка якої дає непередбачені затримки у виконанні програм. (Див. також розділи, пов'язані із плануванням процесів і реалізацією віртуальної пам'яті.)

Висновок

Ми розглянули різні погляди на те, що таке операційна система; вивчили історію розвитку операційних систем; з'ясували, які функції звичайно виконують операційні системи; нарешті, розібралися в тім, які існують підходи до побудови операційних систем. Наступну лекцію ми присвятимо з'ясуванню поняття "процес" і питанням планування процесів.

Дисципліни розподілу ресурсів й основні режими роботи мультипрограмної ЕОМ

Дисципліни розподілу ресурсів мультипрограмної ЕОМ

Дисципліни розподілу ресурсів (ДРР) - досить важливий показник, що впливає на ефективність роботи ЕОМ. Застосування тієї або іншої дисципліни розподілу залежить від особливостей використання даного ресурсу, критеріїв оцінки ефективності роботи системи, а також від складності реалізації даної ДРР.

Одночергові дисципліни

1. FIFO (**First In - First Out**) - перший прийшов - перший обслужений (рис. 13.1).

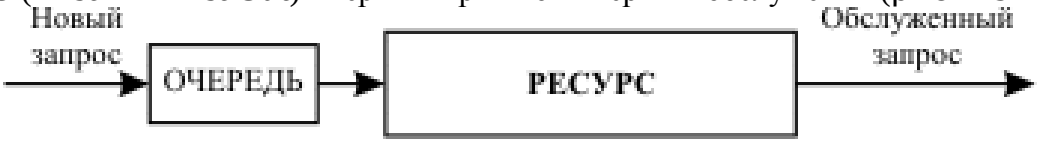


Рис. 1. Схема розподілу ресурсу по дисципліні FIFO

Схема доступу - черга. Широко використається як самостійна дисципліна розподілу, так і складовій частини більше складних дисциплін. Час знаходження в черзі довгих (тобто потребуючого великого часу обслуговування) і коротких запитів залежить тільки від моменту їхнього надходження.

2. LIFO (**Last In - First Out**) - останній прийшов - перший обслужений (рис. 13.2).

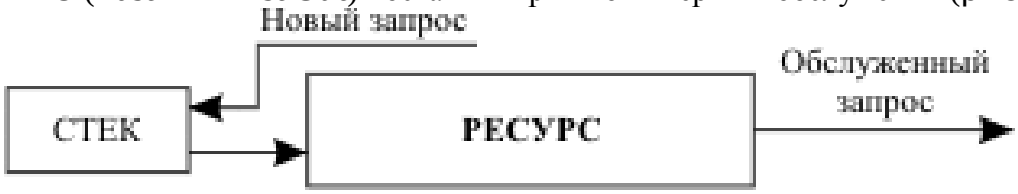


Рис. 13.2. Схема розподілу ресурсу по дисципліні LIFO

Схема доступу - стек.

3. Круговий циклічний алгоритм (рис. 13.3).

Рис. 13.3. Схема розподілу ресурсу по круговому циклічному алгоритмі

Запит обслуговується протягом кванта часу t_k. Якщо за цей час обслуговування не завершене, то запит передається в кінець вхідної черги на дообслуговування.

Тут короткі запити перебувають у черзі менший час, чим довгі.

Багаточергові дисципліни

Базовий варіант багаьочергової дисципліни обслуговування представлений на рис.

13.4.

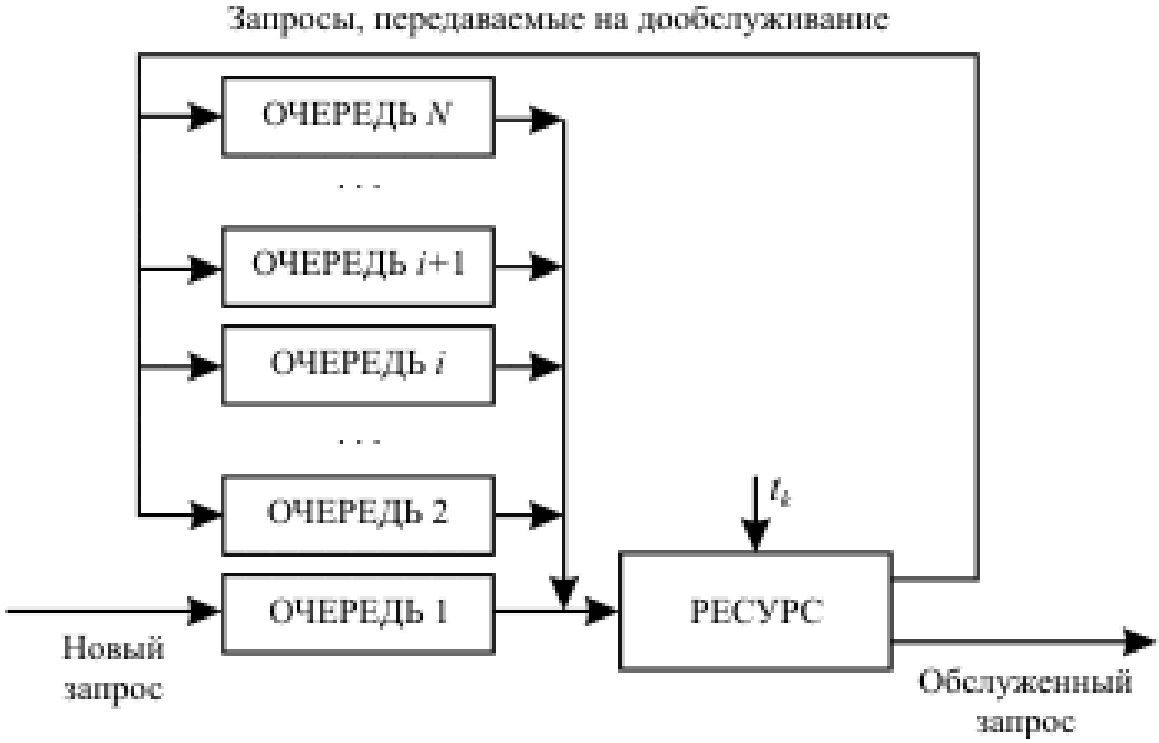


Рис. 13.4. Схема розподілу ресурсу при багаьочергової дисципліні обслуговування

Основа дисципліни - круговий циклічний алгоритм. Всі нові запити надходять у чергу 1. Час, виділюваний на обслуговування будь-якого запиту, дорівнює тривалості кванта t_k. Якщо запит обслужений за цей час, то він залишає систему, а якщо ні, те після закінчення виділеного кванта часу він надходить у кінець черги i+1.

На обслуговування вибирається запит із черги i, тільки якщо черги 1,..., i-1 порожні. Таким чином, довгі запити надходять спочатку в чергу 1, потім поступово доходять до черги N і тут обслуговуються до кінця або по дисципліні FIFO, або по круговому циклічному алгоритмі.

Модифікації базового варіанта багаьочергової дисципліни обслуговування запитів.

1. Виділюваний програмі квант часу на обслуговування зростає зі збільшенням номера черги звичайно за правилом

$$t_{ki} = 2^{i-1} \cdot t_k$$

де t_k - квант часу, виділюваний для програм із черги 1.

Така дисципліна обслуговування найбільш сприятлива коротким програмам, хоча явної вказівки пріоритетів програм тут немає. Ступінь сприяння тим вище, чим менше t_k. Однак зменшення тривалості кванта веде до збільшення накладних витрат, необхідних для перерозподілу ресурсу між програмами.

Дана ДРР може працювати як з відносними, так і з абсолютними пріоритетами програм.

- **Обслуговування програм з відносними пріоритетами**. Заявка, що входить у систему, не викликає переривання обслуговує заявки, що, навіть якщо остання й менш пріоритетна. Тільки після закінчення обслуговування поточної заявки починається обслуговування більше пріоритетної.
- **Обслуговування програм з абсолютними пріоритетами**. Якщо під час обслуговування програми із черги i у чергу з більшим пріоритетом надходить нова програма, то після закінчення поточного кванта t_k оброблювана програма переривається й повертається в початок своєї черги, для того щоб згодом обслугуватися на час, недібраний до $2^{i-1} \cdot t_k$.

2. **Система з динамічною зміною пріоритетів програм**. Щоб уникнути неприпустимо довгого очікування для більших програм, пріоритет робиться

залежним від часу очікування в черзі. Якщо очікування перевищить деякий установлений час, програма переводиться в чергу з меншим номером.

3. **Система зі статичною вказівкою пріоритетів програм**. Уважається, що тривалість виконання програми приблизно пропорційна її довжині. Принаймні, від довжини програми прямо залежить час, затрачуване на передачу програми між ОЗУ й зовнішнім ЗУ при її активізації.
   Визначення номера черги, у яку надходить програма при первісному завантаженні, здійснюється по алгоритму планування Корбато: програма відразу надходить у чергу $i = [\log_2 l_p/l_{tk} + 1]$, де l_p - довжина програми в байтах; l_{tk} - число байт, які можуть бути передані між ОЗУ й зовнішньою пам'яттю за час t_k (рис. 13.5).

Рис. 13.5. Схема розподілу ресурсу при багаьочергової дисципліні обслуговування зі статичною вказівкою пріоритетів програм

Ця дисципліна дозволяє скоротити кількість системних перемикань за рахунок того, що програмам, що вимагають більшого часу рішення, будуть надаватися досить більші кванти часу вже при першому занятті ними ресурсу (нераціонально програмі, що вимагає для свого рішення 1 година часу, спочатку виділяти квант в 1 мс).

Основні режими роботи мультипрограмної ЕОМ

Мультипрограмна ЕОМ може працювати в різних режимах, використання того або іншого з них визначається областю її застосування. Серед основних режимів роботи мультипрограмної ЕОМ виділимо наступні:

1. пакетний;
2. поділу часу;
3. реального часу.

Пакетний режим

Суть пакетного режиму полягає в тім, що ЕОМ обробляє попередньо сформований пакет завдань без втручання користувача в процес обробки.

Пакетний режим використається, як правило, на високопродуктивних ЕОМ. Основна вимога до організації обчислювального процесу на комп'ютері, що працює в пакетному режимі, - це мінімізація часу рішення всього пакета завдань за рахунок ефективного завантаження встаткування ЕОМ.

При пакетному режимі основним показником ефективності служить **пропускна здатність ЕОМ** - число завдань, виконаних в одиницю часу.

Кількісна оцінка виграшу при мультипрограмній роботі з порівняння з однопрограмним використанням ЕОМ представляється у вигляді **коефіцієнта збільшення пропускної здатності**:

$k_{пс} = T_{ОПР}/T_{МПР}$

де $T_{ОПР}$ і $T_{МПР}$ - час виконання пакета завдань при однопрограмному й

мультипрограмному режимі роботи відповідно.

У прикладі роботи мультипрограмної ЕОМ kпс = 36/24 = 1,5 при Км = 2 й kпс = 36/22 1,64 при Км = 3.

Збільшення пропускної здатності ЕОМ досягається належним плануванням надходження завдань пакета на обробку в складі мультипрограмної суміші завдань, а також оптимальним призначенням пріоритетів завданням у цих сумішах, що ґрунтується на поданнях розроблювачів про важливість обліку тих або інших аспектів функціонування ЕОМ і властивостей кожного завдання вхідного пакета.

Основні етапи обробки пакета завдань:

1. Підготовка програм до рахунку. При цьому кожна програма пакета може бути розроблена окремим програмістом.
2. Передача програм і вихідних даних на ЕОМ, що буде обробляти їх у пакетному режимі.
3. Формування пакета завдань із переданих програм по одному з евристичних алгоритмів.
4. Обробка пакета завдань на мультипрограмної ЕОМ.

Особливості пакетного режиму роботи:

1. Користувач відсторонений від безпосереднього доступу до ЕОМ.
2. Результати роботи користувач одержує через певне (іноді досить велике) час одночасно для всіх завдань пакета.
3. Збільшується час налагодження програм.
4. Істотно зростає пропускна здатність ЕОМ у порівнянні з послідовним рішенням завдань пакета.

Таким чином, пакетний режим найбільш ефективний при обробці більших налагоджених програм.

Режим поділу часу

Призначення - обслуговування кінцевого числа користувачів із прийнятним для кожного користувача часом відповіді на їхні запити (рис. 13.6).

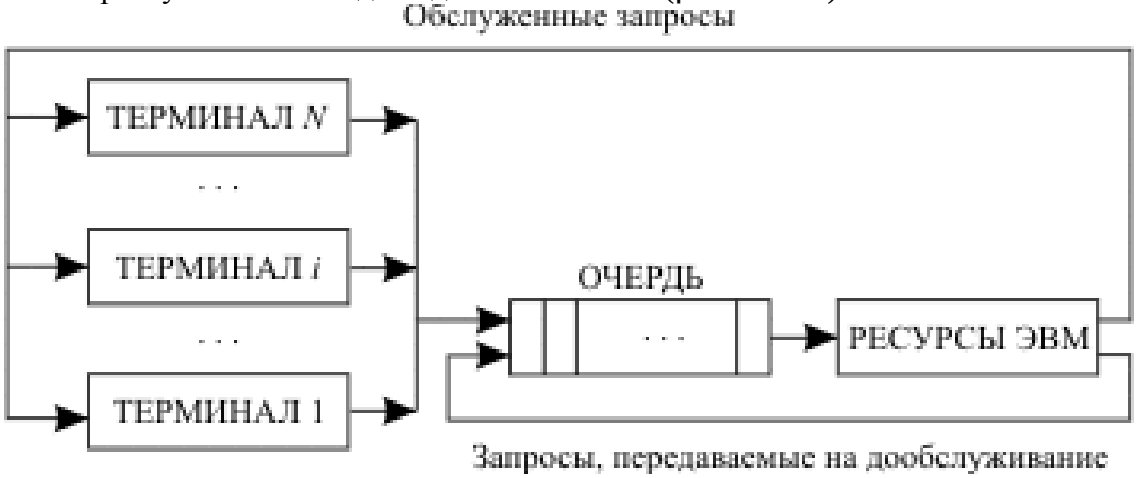


Рис. 13.6. Організація роботи ЕОМ у режимі поділу часу

Основні характеристики:

1. Багатотермінальна багатопользовательска система.
2. Любою користувач зі свого термінала може звернутися до будь-яких ресурсів ЕОМ.
3. У користувача створюється враження, що він один працює на ЕОМ.

Реалізація.

Час роботи машини розділяється на кванти t_k. Кожен квант виділяється для відповідного термінала. Термінали можуть бути активними й пасивними: активний реально включений в обслуговування (за ним працює користувач), пасивний - немає (квант не виділяється). Після обслуговування всіх терміналів послідовність квантів

повторюється.

Єдиного способу вибору часу кванта не існує. Іноді воно вибирається по кількості команд, що повинна виконати ЕОМ за цей час.

В основі реалізації режиму поділу часу лежить одночергова дисципліна обслуговування користувачів.

Режим реального часу

Цей режим роботи мультипрограмних ЕОМ використається, як правило, у системах автоматичного керування об'єктом (рис. 13.7).

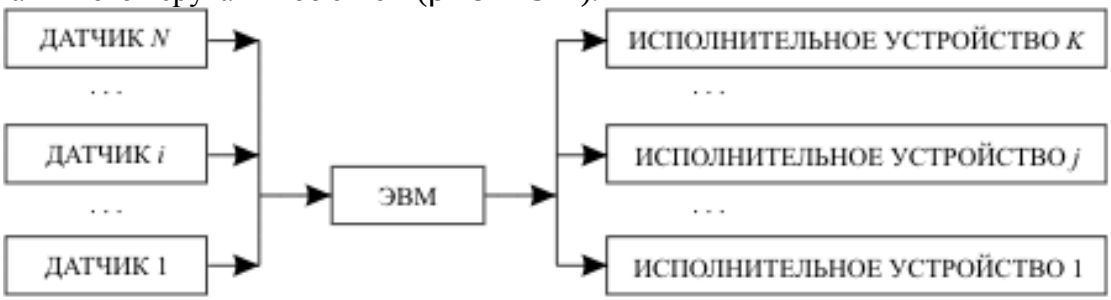


Рис. 13.7. Організація роботи ЕОМ у режимі реального часу

Призначення - забезпечити виконання завдання за час, що не перевищує максимально припустимого для даного завдання. Більшу роль грають дисципліни розподілу ресурсів, особливе призначення пріоритетів завданням.

Режим реального часу має багато загального із системою поділу часу:

- багато терміналів - багато датчиків,
- багато терміналів - багато виконавчих пристроїв.

Особлива увага при побудові систем реального часу приділяється питанням забезпечення надійності функціонування системи.

Тема №3. ОРГАНІЗАЦІЯ ОБРОБКИ ПЕРЕРИВАНЬ

Апаратні переривання це фізичні сигнали, якими контролер пристрою повідомляє процесор про необхідність обробити запит від будь-якого пристрою ПК. Коли виникає переривання, процесор переключається з поточної програми на модуль (у подальшому підпрограму) обробки переривання (у загальному випадку драйвер), а після його завершення повертається до перерваної програми. При переході на підпрограму обробки переривання в стек заноситься вміст регістра прапорів мікропроцесора, а також адреса повернення у форматі сегмент: зсув. Початкові адреси підпрограм обробки переривань прийнято називати векторами. Кожен вектор у реальному режимі роботи процесора має довжину чотири байти і представляється у формі сегмент: зсув. Молодші 1024 байта оперативної пам'яті містять таблицю векторів переривань. Вектор для переривання 0 починається з адреси 0000:0000, переривання 1 – з адреси 0000:0004, переривання 2 - з адреси 0000:0008 і т.д.

Програмні переривання генеруються наявно або командою процесора INT.

1. Виклик програмного переривання з прикладної програми

Мова програмування **C** дозволяє використовувати ряд функцій для роботи з програмними перериваннями.

1.1.Функція **int86()** викликає будь-яке системне або BIOS переривання

int int86(int intno, union REGS *inr, union REGS *outr),

де **intno** - номер переривання;
inr - покажчик на об'єднання вхідних регістрів;
outr - покажчик на об'єднання вихідних регістрів.

union REGS - визначене у файлі dos.h і є перекриттям двох типів структур: **struct WORDREGS x**; - забезпечує доступ до 16-бітових регістрів AX,BX,CX,DX, SI,DI, і до регістрів прапорів, а **struct BYTEREGS h**; - для доступу до відповідних 8-бітним половинам регістрів: AH,AL, BH,BL, CH,DH і DH,DL.

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

Функція **int86()** спочатку копіює значення регістрів з структури ***inr** у відповідні регістри мікропроцесора, а потім генерує програмне переривання з номером **intno.** Після повернення з переривання функція копіює вміст регістрів мікропроцесора у відповідні елементи структури ***outr**.

Функція **int86()** повертає значення регістра **AX**, що той має після виходу з переривання. У випадку помилки встановлюється ненульове значення перемінної **outr.x.cflag**, і в глобальній змінній **_doserrno** запам'ятовується код помилки.

Приклад . Визначити значення поточного часу.

```
#include<stdio.h>
#include<dos.h>
void main(void)
{     union REGS inr, outr;
```

```
   inr.h.ah = 0x2C; /*  Номер функції  */
   int86(0x21, &inr, &outr);
   printf("Поточний час %u:%u:%u.%u\n", outr.h.ch, outr.h.cl,
          outr.h.dh, outr.h.dl);
}
```

Для генруванії програмного переривання з передачею параметрів через сегментні регістри використовується функція **int86x()**:

int int86x(int intno, union REGS *inr, union REGS *outr, struct SREGS *segr),

де **intno** - номер переривання;
inr - покажчик на об'єднання вхідних регістрів;
outr - покажчик на об'єднання вихідних регістрів;
segr - покажчик на структуру сегментних регістрів.

Значення сегментних регістрів ES, CS, SS і DS передаються через структуру SREGS, що визначена у файлі dos.h:

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

Функція **int86x()** спочатку копіює значення регістрів з структури ***inr** у відповідні регістри мікропроцесора, потім зберігає вміст регістра **DS** і записує нові значення з структури ***segr** у регістри **DS** і **ES**. Після цього вона генерує програмне переривання з номером **intno**. При поверненні з переривання функція **int86x()** копіює вміст регістрів мікропроцесора у відповідні елементи структури ***outr** і відновлює значення регістра **DS**. Функція **int86x** повертає значення так само як і **int86()**.

Приклад. Призначити заданий каталог поточним.

```
#include <stdio.h>
#include <dos.h>
int main(void)
{  union REGS reg;
   struct SREGS sreg;
   char  *buffer = “C:\\WINDOWS”;
   reg.h.ah = 0x3B;
   sreg.ds = FP_SEG(buffer);
   reg.x.dx = FP_OFF(buffer);
   int86x(0x21, &reg, &reg, &sreg);
      if(reg.x.cflag) { perror("Помилка intdosx"); return 1; }
   printf("Тепер поточний каталог - C:\\WINDOWS\n");
   return 0;
}
```

Для виклику функцій з набору переривання 21h використовується функція **intdos()**.

int intdos(union REGS *inr, union REGS *outr),

де **inr** - покажчик на об'єднання вхідних регістрів;
outr - покажчик на об'єднання вихідних регістрів.

Її варто використовувати, коли системна функція не вимагає передачі параметрів через сегментні регістри **ES** і **DS**. Будь-яке програмне переривання можна викликати за

допомогою функції int86(), а функцію intdos() завжди можна замінити на int86() з номером переривання 21h.

```
#include <dos.h>
int main(void)
{    union REGS rr;
   ir.h.ah = 0x2C;
   intdos(&rr, &rr);
   printf("Поточний час %u:%u:%u.%u\n", rr.h.ch, rr.h.cl,
          rr.h.dh, rr.h.dl);
}
```

Для генерування програмного переривання 21h, з передачею параметрів через сегментні регістри використовується функція **intdosx()**, що має наступний синтаксис:

int intdosx(union REGS *inr, union REGS *outr,struct SREGS *segr)

Робота цієї функції цілком відповідає раніше розглянутому виклику функції: int86x(0x21, &inr, &outr, &segr);

Альтернативний спосіб виклику програмних переривань надає функція **intr()**, що має наступний синтаксис:

void intr(int intno, struct REGPACK *preg),

де **intno** - номер переривання;
preg - покажчик на структуру регістрів і прапорів.

Функція **intr()** генерує програмне переривання так, як це робить функція **int86x().** При цьому вміст регістрів мікропроцесора передається через **структуру REGPACK**, визначену в файлі **dos.h** у такий спосіб:

```
struct  REGPACK {
     unsigned      r_ax, r_bx, r_cx, r_dx;
     unsigned      r_bp, r_si, r_di, r_ds, r_es, r_flags;
};
```

Функція **intr()** копіює значення з ***preg** у відповідні регістри, а потім використовує ассемблерну команду **INT** для генерування програмного переривання з номером **intno**. Після повернення з переривання функція **intr()** копіює вміст регістрів мікропроцесора у відповідні елементи структури ***preg**.

Розглянемо приклад. Функція 3Bh переривання 21h виконує установку нового поточного каталогу. Ім'я каталогу задається рядком символів у форматі [дисковод:]шлях. Функція повинна одержати в якості аргументу покажчик на цей рядок символів, переданий через пару регістрів DS:DX. У випадку виникнення помилки при зміні каталогу встановлюється прапор переносу CF у регістрі прапорів мікропроцесора.

Для виклику підмножини функцій переривання **21h**, що не вимагають аргументів чи потребують передачі параметрів тільки через регістри DX і AL, можна використовувати функцію **bdos().** Її синтаксис:

int bdos(int dosfun, unsigned dosdx, unsugned dosal),

де **dosfun** - номер функції DOS;
dosdx - значення регістра DX;
dosal - значення регістра Al.

Функція **bdos()** спочатку копіює значення своїх аргументів **dosdx** і **dosal** у відповідні регістри мікропроцесора, а потім викликає функцію **dosfun** переривання 21h. Після виходу з переривання функція **bdos()** повертає значення регістра AX.

Для виклику підмножини функцій переривання 21h, що вимагають передачі адреси (покажчика) через регістри DS:DX і аргумента через регістр AL, можна використовувати функцію **bdosptr().** Її синтаксис:

int bdosptr(int dosfun, void *argument, unsugned dosal),

де **dosfun** - номер функції DOS;
argument - адреса аргумента, що поміщається в DS:DX;
dosal - значення регістра Al.

Функція **bdosptr()** спочатку заносить адресу аргументу argument у регістри DS:DX, копіює значення **dosal** у регістр мікропроцесора AL, а потім викликає функцію **dosfun** переривання 21h. Після виходу з переривання функція **bdosptr()** повертає значення регістра AX. У випадку помилки значення, що повертається, дорівнює -1, а глобальні змінні errno та _doserrno містять код помилки.

Нарешті, макрос **void geninterrupt(int intno)** викликає асемблерну команду INT, що генерує програмне переривання з номером **intno**. При цьому для передачі аргументів і одержання результатів можна використовувати псевдозмінні

_AX _AL _AH _SI _ES
_BX _BL _BH _DI _SS
_CX _CL _CH _BP _CS
_DX _DL _DH _SP _DS
_FLAGS,

які відповідають регістрам загального призначення, спеціальним і сегментним регістрам мікропроцесора. Наприклад, у наступної програми після повернення з переривання 12h через псевдоперемінну _AX зчитується вміст регістра AX, що показує обсяг звичайної пам'яті комп'ютера в кілобайтах:

```
#include<stdio.h>
#include<dos.h>
int main(void)
{
   int size_memory;
   geninterrupt(0x12);
   size_memory = _AX;
   printf("Обсяг ОЗП %d Кбайт\n", size_memory);
   return 0;
}
```

3. Заборона/дозвіл апаратних переривань

Для керування апаратними перериваннями в IBM PC використовується мікросхема програмувального контролера переривань Intel 8259. Оскільки в кожен момент часу може надійти не один запит, мікросхема має схему пріоритетів (16 рівнів). Рівні позначаються скороченнями від IRQ0 до IRQ15.

Максимальний пріоритет відповідає рівню 0. Запити на переривання 0-7 відповідають векторам переривань від 8h до Fh; запити на переривання 8-15 обслуговуються векторами від 70h до 77h. У табл. 1 наведені призначення цих переривань.

Таблиця 1

Номер запиту	Апаратні переривання в порядку пріоритету
IRQ 0	таймер (номер переривання завжди зайнятий)
1	клавіатура, -//-
2	канал вводу/виводу=другий контролер
8	годинник реального часу з автономним живленням(RTC),-
9	програмно переводиться в IRQ2 (тільки AT)
10	вільний
11	вільний
12	контролер миші PC-2
13	співпроцесор (тільки AT), завжди зайнятий
14	контролер IDE HDD (перший канал)
15	контролер IDE HDD (другий канал)
3	COM1 (COM2 для AT),можна відключити. і номер забрати
4	COM2 (модем, COM1 для AT)
5	фіксований диск НДД, LPT2
6	контролер НГМД
7	LPT1

Переривання від таймеру відбувається 18,2 рази в секунду. Йому наданий максимальний пріоритет, оскільки, якщо він буде постійно губитися, то будуть невірними показання системних годин. Переривання від клавіатури викликається при натисканні чи відпусканні клавіші; воно викликає ланцюг подій, що звичайно закінчується тим, що код клавіші міститься в буфер клавіатури, звідки він потім може бути отриманий через програмне переривання.

Програмно можна заборонити апаратні переривання, перераховані в табл. 1. Ці переривання та інші апаратні переривання, що виникають при деяких помилках (таких, як ділення на нуль), не можуть бути масковані. Існують дві причини для заборони апаратних переривань. У першому випадку блокуються всі переривання для того, щоб критична частина коду була виконана цілком, перш ніж машина зробить яку-небудь іншу дію. Наприклад, переривання забороняють при зміни вектора апаратного переривання, щоб уникнути виконання переривання, коли вектор змінений не повністю. В другому випадку маскуються тільки визначені апаратні переривання. Це робиться, коли деякі переривання можуть взаємодіяти з операціями, критичними до часових інтервалів. Наприклад, точно розрахована за часом процедура вводу/виводу не може собі дозволити бути перерваною тривалим дисковим перериванням.

Обслуговування переривання процесором залежить від значення прапора переривання IF у регістрі прапорів. Коли цей прапор дорівнює 0, дозволені усі переривання, що дозволяє регістр маски контролера переривання. Коли прапор дорівнює 1, усі апаратні переривання заборонені. Для роботи з прапором переривання використовуються наступні функції:

void disable()
void enable()

Треба уникати відключення переривань на тривалий період. При виклику обробників переривань прапор IF встановлюється в 1, тому при написанні власного обробника треба викликати функцію enable(), якщо можна допустити апаратні переривання.

Для маскування визначених апаратних переривань потрібно послати необхідний ланцюжок бітів у порт з адресою 21h, що відповідає регістру маски переривань (IMR). Регістр маски на другій мікросхемі 8259 для AT (IRQ8-15) має адресу порту A1h.

У фрагменті, що нижче приводиться, спочатку блокується дискове переривання IRQ5, а потім наприкінці програми воно відновлюється:

```
#include<dos.h>
int main(void)
{
     outportb(0x21, 0x40); /* Пересилання в порт 21h байта 40h */
     outportb(0x21, 0);    /* Пересилання в порт 21h байта  0  */
return 0;
}
```

Після реалізації ACPI и IRQ Sharing на всіх материнських платах, починаючи з плат для Pentium 1 (чіпсети Intel), склалась розкладка переривань, яка існує до цього часу:

Лінія IRQ	Пристрій
0	Системний таймер
1	Клавіатура
2	Cascad (вивід на другу мікросхему контролеру ліній перериваннь)
8	Годинник реального часу
9	ACPI Controller
10	Вільний
11	USB
12	PS2
13	Сопроцессор
14	IDE Primary (Контроллер жорстких дисків)
15	IDE Secondary (Контроллер жорстких дисків)
3	Com Port 1 (Миша)
4	Com Port 2 (Модем)
5	Вільний
6	Floppy
7	LPT (Принтер)

Довідатися, як розподілені номери переривань, можна декількома способами:

- На початку завантаження комп'ютера з'являється текстова таблиця конфігурації. Після її йде перелік PCI-пристроїв та з призначених їм номерів IRQ.
- Для Windows 9x. У "панелі керування" є опція "Система", у ній - "Пристрої". Вибираємо властивості пристрою "Комп'ютер", і в ньому будуть перераховано всі пристрої і призначених їм IRQ.
- Для Windows 2000. Для перегляду списку IRQ потрібно скористатися стандартною утилітою (Панель керування/Адміністрування/Керування комп'ютером/Відомості про систему/Ресурси апаратури).
- Для Windows XP - /Пуск/Програм./Стандарт./Служб./ Відомості про систему/Ресурси апаратури

4. Власний оброблювач переривання

Може існувати кілька причин для написання власної підпрограми обробки переривання – доповнити або змінити стандартний обробник, обслуговувати новий пристрій тощо.

На мові С підпрограма обробки переривання повинна бути описана як функція з ключовим словом **interrupt**, наприклад:

void interrupt get_out(void)

```
{
        // Алгоритмічна частина
}
```

Для установлення нового вектора переривання використовується функція **setvect**, що має синтаксис:

void setvect(int intno, void interrupt(*handler())),

де **intno** - номер переривання;

handler- покажчик на новий оброблювач переривання.

Дана функція встановлює адресу оброблювача переривання, задану аргументом **handler**, як новий вектор для переривання **intno**. Програма користувача перед заміною вектора переривання повинна зберегти старе значення вектора, а перед закінченням своєї роботи відновити його. Адресу обробника переривання повертає функція getvect().

void interrupt (*getvect(int intno))(),

де **intno** - номер переривання.

Змінна, якій буде привласнене значення, що повертається, повинна бути описана як покажчик на функцію типу interrupt,

void interrupt (*oldfunc)(void);
oldfunc = getvect(5);

Для відновлення вектора переривання варто викликати описану вище функцію setvect():

setvect(5, oldfunc);

Наступний приклад ілюструє установку на початку програми нового оброблювача переривання 5h і відновлення системного оброблювача наприкінці її:

```
/*  Файл програми має розширення .C*/
#include <stdio.h>
#include <dos.h>
void interrupt get_out(void);  /*Прототип функції типу
interrupt*/
void interrupt (*oldfunc)(void); /*Покажчик на функцію
interrupt*/
int looping = 1;
int main(void)
{
  oldfunc  = getvect(5); /* Збереження старого вектора int 5*/
  setvect(5, get_out);   /* Установка нового оброблювача int 5*/
  printf("Зациклення...\n");
  printf("Натисніть <Shift><PrtSc> для виходу з циклу\n");
  while (looping);
  setvect(5, oldfunc);   /* Відновлення старого вектора */
  printf("Тепер <Shift><PrtSc> викликає печатку копії
екрана\n");
  return 0;
}
void interrupt get_out(void) // Новий оброблювач переривання 5
{
  looping = 0; /*зміна глобальної перемінної для виходу з циклу
*/
}
```

ПРИМІТКА: повинні бути встановлені опції компілятора:
Compiler/CodeGeneration - Test stack overflow =off;
Compiler/CodeGeneration/Optimization - Use register variables =off;
Linker - Stack Warning =off.

Користувальницький оброблювач переривання може не тільки цілком замінити системний оброблювач, але і доповнити його. Загальна структура програми в цьому випадку має вигляд:

```
#include <dos.h>
.................
void interrupt handler(void); /* Прототип функції типу
interrupt*/
void interrupt (*oldfunc)(void);/* Покажчик на функцію
interrupt*/
int main(void)
{
  oldfunc  = getvect(номер_переривання);// Збереження старого
                                         //вектора переривання
  setvect(номер_переривання, handler); // Установка нового
                                       // оброблювача
переривання   setvect(номер_переривання, oldfunc);   //
Відновлення старого
                                       // вектора
  return 0;
}
void interrupt handler(void)   /* Новий оброблювач переривання
*/
{
/* Перша частина користувальницького оброблювача */
  oldfunc();  // Виклик системного оброблювача з наступним
              //поверненням  у це місце
/* Друга частина користувальницького оброблювача */
}
```

При цьому нова процедура обробки переривання handler() може містити будь-який код як до, так і після виклику системного оброблювача.

4.1. Створення процедури обробки Ctrl-Break

При одночасному натисканні клавіш <Ctrl> і <Break> встановлюється в 1 старший біт байта BIOS_BREAK з адресою 0040:0071 в області даних BIOS. Будь-яка програма може перевірити значення цього біта для визначення стану Ctrl-Break. Потім викликається переривання 1Bh. Звичайно вектор переривання 1Bh указує на системний оброблювач DOS, але будь-яка програма може перехопити цей вектор і тим самим самостійно обробляти Ctrl-Break. Якщо переривання 1Bh викликає системний оброблювач, то останній установлює внутрішній прапор Clrl-Break системи. Прапор указує на те, що повинна бути виконана процедура обробки Ctrl-Break. Керування передається цій процедурі тільки у той момент, коли програма звертається до системної функції, здатної розпізнати цей прапор. Звичайно тільки стандартні функції вводу/виводу розпізнають цей прапор (функції від 1 до С переривання 21h, за винятком функцій 6 і 7). Однак, якщо в файл AUTOEXEC.BAT чи CONFIG.SYS помістити рядок BREAK=ON, тоді звертання до будь-якої системної функції приведе до виклику процедури обробки Ctrl-Break.

Процедура обробки Ctrl-Break дає можливість завершити програму в будь-який момент часу. Коли системна функція розпізнає статус Ctrl-Break, то керування передається процедурі, на яку указує вектор переривання 23h. Система використовує цю процедуру для завершення працюючої програми.

Встановлення нового оброблювача для Ctrl-Break з модифікацією вектора 23h виконує функція:

void ctrlbrk(int (*handler)(void)),

де **handler** - покажчик на функцію, що стане новим оброблювачем Ctrl_Break.

Для забезпечення виходу в систему по закінченні своєї роботи оброблювач повинний завершуватися оператором

return 0;

При цьому перед поверненням у систему автоматично відновлюється первісний вектор переривання 23h.

Приклад установки програмного оброблювача Ctrl-Break:

```
#include<stdio.h>
#include<dos.h>
#define ABORT 0
int c_break(void)
  {
   printf("\nControl-Break включений. Програма перервана
...\n");
   return(ABORT);
  }
int main(void)
  {
    ctrlbrk(c_break);
    for(;;)
      printf("Зациклення ... Для виходу"
             " натисніть <Ctrl><Break>\n");
    return 0;
  }
```

Якщо код повернення в операторі **return**, що завершує оброблювач, відмінний від нуля, то після виконання процедури обробки переривання відбудеться повернення до перерваної програми. Переривання 1Bh іноді називають апаратним перериванням Ctrl-Break на відміну від програмного переривання Ctrl-Break з номером 23h

Приклад. /* Оброблювач апаратного переривання по натисканню Cntr-Break. */

```
#include<dos.h>
#include<stdio.h>
#include<conio.h>
#include<time.h>
void interrupt myprint(void)
  {
   disable();
   cprintf("ВИВІД З ОБРОБЛЮВАЧА ПЕРЕРИВАННЯ... ");
   enable();
  }
int  main(void)
  {
```

```
   int i;
   void interrupt (*oldvect)(void);
   oldvect = getvect(0x1B);
   setvect(0x1B, myprint);
   for(i = 0; i <= 30; i++)
    {
     cprintf("\n\r для виклику оброблювача переривання"
             " натисніть <Ctrl><Break>\n\r");
     sleep(1);
    }
   setvect(0x1B, oldvect);
   cprintf("\n\rпрограма з а в е р ш е н а\n\r");
   return 0;
  }
```

4.2. Особливості користувальницьких оброблювачів апаратних переривань

Програма обробки апаратних переривань, перерахованих у табл. 1, перед закінченням своєї роботи повинна посилати код 20h у порт 20h програмувального контролера переривань Intel 8259, попередньо заборонивши всі переривання:

disable();
outportb(0x20, 0x20);

Якщо оброблювач апаратного переривання не закінчується цими операторами, то мікросхема 8259 не очистить інформацію регістра обслуговування для того, щоб була дозволена обробка переривань з більш низькими рівнями, чим тільки що оброблене. Ці оператори не потрібні якщо власний оброблювач доповнює системний.

5. Створення резидентного оброблювача переривання

Програма, що знаходиться у пам’яті ПК у межах всього сеансу роботи, називається резидентною. Для того, щоб залишити програму у пам’яті, використовується функція

keep(статус, розмір)

Параметр "**розмір**" задає число параграфів оперативної пам'яті, що залишаються програмі. Функція **keep()** передає код завершення процесу через параметр **"статус".**
Розглянемо як приклад програму, що для переривання 71h встановлює резидентний оброблювач mybeep() :

```
/*     Резидентний оброблювач ПРОГРАМНОГО переривання 0x71
#include<dos.h>
 void interrupt mybeep()
{ int register i;
    for(i=0;i<=9900;i++) sound(1000 - i/20);
    nosound();
}

 void install(void interrupt (*faddr)(),  int inum)
  {  setvect(inum, faddr);
  }
  void resident(void)
   {  int status;
```

```
    keep(status,0x2000);
   }
 main()
  { install(mybeep, 0x71);
    resident();      /* Залишає в пам'яті ВСЮ ПРОГРАМУ  main */
  }
```

У даній програмі зміна вектора переривання здійснюється функцією install(), а залишення програми резидентною-функцією resident(), що у свою чергу викликає функцію keep() за допомогою оператора

```
keep(status, 0x2000);
```

При цьому другий параметр 0x2000 задає розмір пам'яті (у параграфах), що залишається програмі. Це число, що відповідає 128Кб, обрано з запасом як максимальний можливий розмір програми на С при роботі з малою (small) моделлю пам'яті. При роботі з маленькою (tiny) моделлю пам'яті можна задавати 0x1000, що відповідає 64К байтів. Тип моделі пам'яті встановлюється шляхом вибору в головному меню С пункту Options, а потім підпунктів Compiler і Model. У реальних задачах на відміну від даного навчального приклада розмір пам'яті, що залишається програмі, повинний ретельно розраховуватися і, імовірно, буде значно менше.

Для виклику резидентного оброблювача програмного переривання, установленого за допомогою приведеної програми, варто запустити іншу програму з функцією geninterrupt().

Наприклад:

```
#include<dos.h>
main()
{
 geninterrupt(0x71);
}
```

При цьому буде генеруватися звуковий сигнал з перемінною частотою, що змінюється в діапазоні від 1000 Гц до 505 Гц.

Тема №4. СИСТЕМНІ РЕСУРСИ КОМП'ЮТЕРА

1. Тип IBM PC і тип мікропроцесора.

Програма може визначити на якому типі IBM PC вона завантажена, прочитавши вміст байта в ПЗП BIOS за адресою F000:FFFE. Деякі типи комп'ютерів, частково сумісні з IBM PC, мають нуль у цьому байті чи інші коди. Більш повну інформацію про тип комп'ютера можна одержати з функції C0h переривання 15h, яка через пару регістрів ES:BX повертає покажчик на початок таблиці системного середовища.

Приклад. Визначити тип IBM PC, код підмоделі і рівень ревізії BIOS.

```
#include<stdio.h>
#include<dos.h>
void main(void)
 { union REGS r;
   struct SREGS s;
   unsigned char tip;
   r.h.ah = 0xC0;
   int86x(0x15, &r, &r, &s);
   tip = peekb(s.es, r.x.bx + 2);
   printf("МОДЕЛЬ  КОМП'ЮТЕРА:\n");
   switch(tip)
     { case 0xFF: printf("Оригінальний IBM PC\n"); break;
       case 0xFE: printf("IBM PC XT чи портативний PC\n");
break;
       case 0xFD: printf("IBM PC jr\n"); break;
       case 0xFC: printf("IBM PC AT (чи XT моделі 286"
                         " чи PS/2 моделі 50/60)\n"); break;
       case 0xFB: printf("IBM PC XT з 640K пам'яті на "
                        "материнській платі\n"); break;
       case 0xFA: printf("PS/2 моделі 30\n"); break;
       case 0x9: printf("Convertible PC\n"); break;
       case 0x8: printf("PS/2 моделі 80\n"); break;
       default:   printf("Невідомий тип IBM PC\n");
      }
   printf("\n КОД  ПІДМОДЕЛІ %X\n", peekb(s.es, r.x.bx + 3));
   printf("\n РІВЕНЬ РЕВІЗІЇ BIOS %X\n", peekb(s.es, r.x.bx +
4));
 }
```

1.1. Визначення типу мікропроцесора

Комп'ютери сімейства IBM PC звичайно використовують як центральний процесор мікропроцесори Intel та AMD. Стандартні для всіх мікросхем оцінки наступні.

Технологічний параметр (товщина пластини)- це товщина пластини, на якій виконується напилювання процесора. Виміряється в мікрометрах, чи мікронах [мкм].

Споживаний струм – це струм, що споживає процесор при максимальному завантаженні. Якщо помножити струм на напругу процесора, можна одержати споживану процесором потужність. Струм виміряється в амперах [А].

Напруга ядра - напруга, що подається на процесор. Визначається при виготовленні. На відміну від струму, подану на процесор напругу можна змінити, що приведе до зміни споживаної ним потужності. Вимірюється у вольтах [В].

Частота процесора - кількість циклів, виконуваних в одну секунду. У залежності від конструкції процесора, може бути змінено користувачем. Вимірюється в мегагерцах [Мгц].

Площа ядра - площа процесорного ядра. Вимірюється в квадратних міліметрах [мм^2].

Ступінь інтеграції - кількість інтегрованих елементів (транзисторів) на одиницю площі. Виміряються в мільйонах транзисторів на квадратний міліметр [млн./мм^2].

Тепловиділення - показує потужність, що виділяється у вигляді тепла . Прямо залежить від споживаної потужності і товщини пластини. Це ключовий параметр, що визначає температуру, до якої нагрівається процесор. Вимірюється у ватах [Вт].

1.2 Визначення частоти процесора

1.2.1. Визначення частоти CPU методом порівняння циклів

Для програмного визначення частоти CPU можна врахувати, що час роботи команд визначається або в тіках - адреса **0x40:0x63**, або в сотих частках секунди з переривання **21h,** функції **2Ch**. У цьому випадку алгоритм складається з етапів:

- емпірично визначити кількість циклів тесту - до 10**7;
- знайти час роботи еталонного порожнього циклу, як різницю часів початку і кінця циклу;
- визначити час роботи циклу, у тілі якого знаходиться одна асемблерна команда, наприклад dec;
- знайти чистий час роботи тестуємої команди, як різницю між результатами попередніх двох пунктів;
- знайти частоту процесора як величину зворотну часу виконання однотактної команди в секундах.

1.2.2. Визначення частоти CPU методом завдання інтервалу часу.

Інтервал часу в мікросекундах формується годинником реального часу - від переривання **int 70h**. У програмі користувача інтервал задається перериванням **int 15h** у двох видах:

- функцією 86h int 15h, після чого програма користувача переходить у режим чекання на зазначений час;
- функцією 83h int 15h - тут керування передається наступному оператору програми, а кінець заданого інтервалу часу відзначається установкою в одиницю байта з адресою 0:04A0.

Для реалізації алгоритму в регістрах CX:DX задається інтервал, наприклад, 60000мкс і викликається переривання 0x15, при _AX=0x8300. Далі організується цикл до кінця заданого інтервалу часу. У тілі циклу, крім команд перевірки на його завершення, повинний бути лічильник циклів і біля сотні асемблерних команд dec dx, заданих повторювачем DUP. Визначається час роботи однотактної команди, як частка від розподілу заданого інтервалу часу на кількість тактів у циклі. Частота процесора - це величина зворотна часу виконання однотактної команди в секундах.

Приклад.

```
{union REGS rr;
 unsigned int i;
```

```
 unsigned long max,c;
 float fq;
printf("Уведіть час визначення частоти процесора***\n");
printf("у секундах (рекомендується в межах 1..4294)\n");
printf("при завданні більш тривалого часу\n");
printf("підвищується точність визначення частоти.\n");
scanf("%u",&i);
printf("визначення частоти. Будь ласка, почекайте...\n");
max=i*1000000.0;
rr.x.ax=0x8300;
rr.x.cx=ceil(max/65536);
rr.x.dx=fmod(max,65536);
int86(0x15,&rr,&rr);
_ES=0;
_SI=0x4a0;
asm mov ecx,0
mm1:    asm{
        inc ecx
        db 120 DUP(4Ah)
        cmp al,byte ptr es:[si]
        jne mm1
        mov dword ptr c,ecx
    };
fq=max;
fq/=c;
fq/=127.0;
fq=1.0/fq;
printf(" Частота процесора %8.5f Мгц\n",fq);
}
```

2. Визначення характеристик BIOS

Basic Input/Output System – це основна система вводу/виводу, розміщена в ПЗП (звідси назва ROM BIOS). Вона являє собою набір програм перевірки й обслуговування апаратури комп'ютера, і виконує роль посередника між операційною системою і апаратурою. BIOS одержує керування при включенні і скиданні системної плати, тестує саму плату й основні блоки комп'ютера - відеоадаптер, клавіатуру, контролери дисків і портів вводу/виводу, набудовує chipset плати і завантажує зовнішню операційну систему.

Звичайно на системній платі встановлено тільки ПЗП із системним (Main, System) BIOS, що відповідає за саму плату і контролери FDD, HDD, портів і клавіатури. У системний BIOS практично завжди включений System Setup - програма настроювання системи. Відеоадаптери і контролери HDD мають власні BIOS в окремих ПЗП; їх також можуть мати й інші плати - інтелектуальні контролери дисків і портів, мережні карти і т.п. BIOS для сучасних системних плат розробляється однією з фірм, що спеціалізуються на цьому. Іноді для однієї і тієї ж плати маються версії BIOS від різних виробників - у цьому випадку допускається копіювати прошивання чи заміняти мікросхеми ПЗП; у загальному ж випадку кожна версія BIOS прив'язана до конкретної моделі плати.

Інформація, що характеризує BIOS зберігається у виді текстових рядків по відповідним адресам у ROM BIOS. Основні дані починаються з таких базових адрес:

Виробник 0x000:0x091.

Версія 0x000:0x061.

Дата видання 0x000:0xFFF5.

Дані відеосистеми:

Тип VideoBIOS 0x000:0x001E.
Тип відеокарти і версія VideoBIOS 0x000:0x004B.
Крім того, характеристики інтерфейсу:
Типи шини даних 0x000:0xFFD9.
Тип чіпсету 0x000:0xEC71.

Приклад. Визначити основні характеристики BIOS

```
. . . . . .
char far *ptr  = MK_FP(0xF000, 0xE091);
 char far *ptr1 = MK_FP(0xF000, 0xE061);
 char far *ptr2 = MK_FP(0xF000, 0xFFF5);
 char far *ptr3 = MK_FP(0xC000, 0x001E);
 char far *ptr4 = MK_FP(0xC000, 0x004B);
 char far *ptr5 = MK_FP(0xF000, 0xFFD9);
 char far *ptr6 = MK_FP(0xF000, 0xEC71);

        ...
   printf("Виробник: ");
   for(; *ptr == 0; ptr++) ;
   while(*ptr) printf("%c", *ptr++);
   printf("\n");
   printf("Версія: ");
   for(; *ptr1 == 0; ptr1++) ;
   while(*ptr1) printf("%c", *ptr1++);
   printf("\n");
   printf("Дата видання: ");
   for(; *ptr2 == 0; ptr2++) ;
   while(*ptr2) printf("%c", *ptr2++);
   printf("\n");
   printf("Тип VideoBIOS: ");
   for(; *ptr3 == 0; ptr3++) ;
   while(*ptr3) printf("%c", *ptr3++);
   printf("\n");
   printf("Тип відеокарти і версія VideoBIOS: ");
   for(; *ptr4 == 0; ptr4++) ;
   while(*ptr4) printf("%c", *ptr4++);
   printf("\n");
   printf("Тип шини даних: ");
   for(; *ptr5 == 0; ptr5++) ;
   while(*ptr5) printf("%c", *ptr5++);
   printf("\n");
   printf("Тип чіпсета: ");
   for(; *ptr6 == 0; ptr6++) ;
   while(*ptr6) printf("%c", *ptr6++);
   printf("\n");
}
```

3. Визначення обсягу основної пам'яті

Для визначення розміру оперативної пам'яті можна використовувати переривання **12h BIOS.** Це переривання повертає в регістр AX мікропроцесора кількість кілобайт звичайної пам'яті в системі (без обліку розширеної і додаткової відображуваної пам'яті).

Переривання **12h** використовується функцією C biosmemory(), яка повертає обсяг ОЗП в кілобайтах. Прототип функції знаходиться у файлі bios.h.
Приклад. Визначити обсяг ОЗП IBM PC

```
#include<stdio.h>
#include<bios.h>
int main(void)
 {
  printf("Обсяг звичайної пам'яті %d Кбайт\n", biosmemory());
  return 0;
 }
```

4. Визначення числа і типу периферійних пристроїв

При включенні ПЕОМ BIOS перевіряє приєднане устаткування і повідомляє про результати в двохбайтну змінну з адресою 0x0040:0x0010. Переривання **11h BIOS** повертає в регістр AX значення цієї змінної. Значення інтерпретується як набір бітових полів, що характеризують устаткування:

біт 0 якщо "1", то присутній накопичувач на гнучкому магнітному диску (НГМД);

біт 1 якщо "1", то є співпроцесор;

біти 2-3 розмір базової пам'яті на системній платі (11 - 64 К и більш);

біти 4-5 активний відеоадаптер:

00 - не використовується,
01 - кольоровий (CGA) 40 символів * 25 рядків,
10 - кольоровий (CGA) 80 символів * 25 рядків,
11 - монохромний 80 символів * 25 рядків;

біти 6-7 кількість НГМД (тільки, якщо біт 0 дорівнює "1"):

00 - один,
01 - два,
10 - три,
11 - чотири;

біт 8 використовується тільки для IBM PCjr: якщо "0", то є контролер прямого доступу до пам'яті (ПДП);

біти 9-11 кількість послідовних портів RS232;

біт 12 якщо "1", те є присутнім ігровий адаптер;

біт 13 використовується тільки для IBM PCjr: якщо "1", то приєднано послідовний принтер;

біти 14-15 кількість портів для принтерів.

Приклад .Перевірити наявності устаткування (переривання BIOS 11h - функція biosequip() в мові C)

```
#include<stdio.h>
#include<bios.h>
#include<dos.h>
int main(void)
{
 union
   {
     int status;
     struct
       {
```

```
        unsigned drives_present:1;
        unsigned coprocessor   :1;
        unsigned unused1       :4;
        unsigned num_drives    :2;
        unsigned unused2       :1;
        unsigned RS232_ports   :3;
        unsigned unused3       :2;
        unsigned num_printers  :2;
      }fields;
   }ob;
   ob.status = biosequip();
   if (!ob.fields.drives_present) printf("Немає НГМД\n");
    else printf("Усього %u НГМД\n",ob.fields.num_drives + 1);
   if (ob.fields.coprocessor)
       printf("Є співпроцесор із ПЗ\n");
   else printf("Немає співпроцесора з ПЗ\n");
   printf("Кількість послідовних портів RS232 %u\n",
          ob.fields.RS232_ports);
   printf ("Кількість портів для принтерів  %u\n",
           ob.fields.num_printers);
   return 0;
}
```

Приклад. Визначити адреси портів (перший варіант)

```
{void far *p;
 long com;
 int flag;
 p=MK_FP(0x40,0);
 com=*(long far *)p;
 printf("\n\nAdress COM1 : %ph",com);
 if(flag<2)return;
 p=MK_FP(0x40,2);
 com=*(long far *)p;
 printf("\nAdress COM2 : %ph",com);

}
```

Приклад . Визначити адреси портів (другий варіант)

```
printf("* Послідовні порти :\n");
int86(0x11, &regs, &regs);
 regs.h.ah=regs.h.ah>>1;
 regs.h.ah&=7;
printf("  - Кількість :  %u\n", regs.h.ah);
int k=regs.h.ah;

printf("  - Адреси:\n    N        Адреса\n");
int addr=0x400;
for (int i=0;i<k;i++)
 {
  unsigned long far *comadr= (unsigned long far* )MK_FP(0,addr);
  printf ("    %d         %x\n",i,*comadr);
  addr+=2;
```

```
}
```

5. Визначення типу магнітних накопичувачів

Для визначення виду і конфігурації магнітних накопичувачів існує більш точний спосіб за допомогою функцій переривання **21h** (компілятор Borland):

- функція **4408h**, повертає регістр AL=0, якщо це FD Drv;
- функція **4409h**, повертає в регістрі DX атрибути накопичувача;
  біт 15 дорівнює 1, якщо це Substituted Drv,
  біти 12 і **9** рівні 1, якщо це Network Drv;
- функція **440Dh,** під функція 0660h визначає тип накопичувача;

Число і тип FDD і HDD можна довідатися з байтів 10h, 12h енергонезалежної пам'яті. Розмір дисків визначає функція **08h**, переривання **13h** - у регістрі DH вона повертає кількість голівок, у регістрі CH повертає молодші цифри кількості циліндрів, у перших 2 бітах регістра CL - старші цифри кількість циліндрів, останній 6 біт регістра CL – кількість секторів на доріжці.

5.1. Визначення числа і розмірів дисків

Наявність HDD у комп'ютері визначається шляхом посилки в порт **70h** коду байту - визначника **12h.** Ненульовий результат приймається з порту **71h**. Функція **8** переривання **13h** повертає в регістри мікропроцесора характеристики HDD (це максимальні номери, починаючи з 0):

CH - молодша частина max номера циліндра;
CL - у перших двох бітах - старша частина max номера циліндрів, а в інших бітах - max номер сектора на доріжці;
DH - max номер голівки (сторони).

Добуток цих величин (з врахуванням того, що сектор=512 байт) дає розмір HDD у байтах. Наявність FDD у комп'ютері визначається шляхом посилки в порт **70h** коду байту-визначника **10h.** Ненульовий результат приймається з порту **71h** і визначає дисководи:

1- 360 Кб; 2- 1.2 Мб; 0x30- 720 Кб; 0x40- 1.44 Мб;

Якщо дисководів два:

0x42- 1.2Мб і 1.44Мб; 0x24- 1.44Мб і 1.2Мб;
0x34- 1.44Мб і 720Кб; 0x44- 1.44Мб і 1.44Мб і т.д.

6. Визначення типу клавіатури

При натисканні клавіші викликається переривання клавіатури 9h, і код символу міститься в буфер клавіатури, що є областю пам'яті здатної запам'ятати до 15 символів. Взаємодію з клавіатурою забезпечує переривання BIOS **16h**. Функція **10h** даного переривання вибирає символ з буферу клавіатури; код символу повертається в регістрі AX. Функція **11h** перевіряє, чи мається в буфері клавіатури символ для зчитування. Якщо є, то прапор нуля мікропроцесора (ZF) встановлюється в 0, і в регістрі AX передається код символу, хоча сам символ залишається в буфері клавіатури. Якщо символу для зчитування немає, то прапор ZF встановлюється в 1. Функція **05h** переривання 16h дозволяє програмі занести визначений код символу в буфер клавіатури. Код передається через регістр CX. Якщо після повернення з переривання регістр AL містить 0, то занесення символу завершилося успішно. У противному випадку регістр AL містить 1, що говорить про те, що буфер клавіатури повний.

За допомогою описаних функцій переривання **16h** можна виконати перевірку типу клавіатури за методикою, рекомендованою в технічному описі BIOS. Для цього спочатку робиться спроба з допомогою функції **05h** занести в буфер клавіатури код **FFFFh** (цьому

коду не відповідає ніяка клавіша), потім за допомогою функції 11h перевіряється наявність символу для зчитування, а за допомогою функції **10h** витягається символ з буфера. Якщо код витягнутого символу дорівнює **FFFFh**, то функції **05h, 10h** і **11h** функціонують коректно, що свідчить про наявність 101-клавішної клавіатури. Якщо ж хоча б одна з функцій працює некоректно, то на комп'ютері встановлена 83-клавішна клавіатура. Описана методика реалізована в наступній програмі:

```
#include<dos.h>
#include<process.h>
#define SPEC_CODE 0xFFFF
void retrieve(void);
union REGS r;
int main(void)
 {
   int i;
   for(i = 0; i <= 1; i++)
    { /*  Виконати дві спроби запису коду 0xFFFF у буфер  */
      r.x.cx = SPEC_CODE;
      r.h.ah = 0x5;
      int86(0x16, &r, &r);
      if(r.h.al)
        { /*   Буфер повний чи функція 5h не підтримується */
           r.h.ah = 0x10;
           int86(0x16, &r, &r);   /*  Витяг символу  */
         }else
           {
              retrieve();
              return 0;
           }
     }
   printf("83-кл. клавіатура\n ");
   return 0;
}

void retrieve(void)
 {
   int j;
   for(j = 0; j < 15; j++)  /* Цикл по всій довжині буфера  */
     {
       r.h.ah = 0x11;
       int86(0x16, &r, &r); /* Перевірка наявності
                               символу для зчитування      */
       if(r.x.flags & 0x40) /* Перевірка прапора ZF (у регістрі
                               прапорів це 6-й розряд)      */
         {
           printf("83-кл. клавіатура\n ");
           exit(0);
         }
         r.h.ah = 0x10;
         int86(0x16, &r, &r);  /*  Зчитування символу   */
         if(r.x.ax == SPEC_CODE)
          {
            printf("Розширена 101-кл. клавіатура\n");
```

```
          exit(0);
        }
    }
printf("83-кл. клавіатура\n ");  }
```

Тема №5. ДОСЛІДЖЕННЯ ОРГАНІЗАЦІЇ ОПЕРАТИВНОЇ ПАМ'ЯТІ

1. Організація пам'яті реального режиму

Для сучасних процесорів максимальний обсяг пам'яті, що адресується, дорівнює 64 Гбайт. Адресний простір комп'ютерів IBM PC при роботі їх у реальному режимі являє собою одномірний масив байт, кожний з яких має 20-розрядну фізичну адресу. Зручно вважати, що кожна комірка пам'яті має дві адреси: фізичну та логічну. Фізична адреса являє собою 20- розрядне значення в діапазоні від 0 до FFFFF, що визначає положення кожного байта в просторі пам'яті 1 Мб. Логічний адрес складається з двох 16-розрядних без знакових значень: базового (початкового) адресу сегмента і зсуву усередині сегмента. Сегмент являє собою логічну одиницю пам'яті розміром 64 Кбайт. Усі сегменти починаються на 16-байтних границях пам'яті, які називаються межами параграфів. Для будь-якої комірки пам'яті база ідентифікує перший байт її сегмента, тобто початок сегмента, а зсув визначає відстань у байтах від початку сегмента до цього осередку.

Адресний простір комп'ютерів IBM PC у межах 1 Мбайт розподіляється відповідно до табл. 1.

2. Область даних BIOS

Область основної пам'яті розміром 256 байт, розташована безпосередньо за таблицею векторів переривань, - починаючи з адреси 0040:0000 і закінчуючи 004F:0000 - призначена для використання програмами BIOS. У табл. 2 приведений опис частини інформації, що поміщається системою BIOS у зазначену область пам'яті. Звертаючись до неї, прикладні програми одержують важливу інформацію про стан системи. Наприклад, прочитавши байт за адресою 0040:0013, програма може визначити обсяг оперативної пам'яті в кілобайтах.

У моделях комп'ютерів IBM PC використовується операція "затінення" (Shadow) для системного ПЗП BIOS і BIOS відеоадаптерів VGA. Shadow Memory - це так звана "тіньова" пам'ять. В адресах пам'яті від 640 Кб до 1 Мб (A0000h - FFFFFh) знаходяться "вікна", через які "видно" зміст різних системних ПЗП. Наприклад, адреси F0000h-FFFFFh займають системне ПЗП, що містить BIOS системи, вікно C0000h - C7FFFh - ПЗП відеоадаптера (відео BIOS) і т.п. При включенні режиму Shadow для яких-небудь адресних діапазонів, що відповідають системним ПЗП або картам розширення, зміст їх ПЗП копіюється у ділянки основної пам'яті, що потім підключаються до цих же адрес замість ROM, "затінюючи" їх. Для реалізації операції "затінення" в оперативній пам'яті, крім основної ділянки області даних BIOS, що починається по адресі 0040:0000, виділяється додаткова ділянка, що має назву розширеної області даних BIOS (табл.1).

Характеристики розширеної області даних BIOS можна визначити шляхом виклику функції **C0h** переривання **15h.** Якщо після повернення з переривання прапор переносу містить 0, то розширена область BIOS даних підтримується. Через пару регістрів ES:BX повертається покажчик на початок таблиці системного середовища, яке містить характеристики розширеної області даних BIOS.

Таблиця 1

Адреса (сегмент зсув)	Довжина (у байтах)	Залежність від	Найменування і опис
0000:0000	1024		Таблиця векторів переривань 256 4-байтових адрес
0040:0000	256		Область даних BIOS
0050:0000	256		Область даних DOS
0060:0000	256		Область даних DOS, використовувана тільки при завантаженні
XXXX:0000	8928	версії MS DOS	Код BIOS (прочитаний з IO.SYS завантажувального диску)
XXXX:0000	29920	версії MS DOS	Оброблювачі переривань DOS, включаючи INT 21h (MSDOS.SYS)
XXXX:0000		конфігу-рації	DOS: буфера, область даних і встановлювані драйвери
XXXX:0000			Резидентна частина COMMAND.COM, включаючи оброблювач INT 22h,23h, 24h
XXXX:0000			Резид. програми і дані
XXXX:0000			Поточна виконувана програма (типу .COM чи .EXE).
XXXX:0000		конфігу-рації	Транзитна частина COMMAND.COM. перезавантажена якщо була чимось перекрита
XXXX:0000			Розширена область даних BIOS для комп'ютера PS/2
			Границя 640 К зони
A000:0000			EGA-, VGA-пам'ять для деяких відеорежимів
B000:0000			Відеопам'ять монохромного адаптера й адаптера Hercules
B800:0000			Відеопам'ять CGA (Hercules)
C000:0000 до E000:0000			Зовнішній код ПЗП. ПЗП BIOS шукає тут (у 2 К-блоках) код, виконує мий при завантажен.
також			Сторінковий блок EMS
E000:0000 до E000:FFFF			Модулі ПЗП материнської плати у блоках по 64К (тільки системної плати IBM PC AT).
F600:0000			ПЗП-резидентний інтерпретатор BASIC (для PC фірми IBM)
FE00:0000			ПЗП - BIOS: програма автоматичного тестування POST і ін.
F000:FFF0			Команда JMP на програму, виконувану при вмик/скиданні
F000:FFF5:8	8		Дата видання BIOS
F000:FFFE	1		Ідентифікаційний код IBM PC

Таблиця 2.

Адреса (сегмент зсув)	Довжина (у байтах)	Найменування і опис
		ОБЛАСТЬ ДАНИХ ПОРТІВ
0040:0000	8	Базова адреса вводу-виводу для COM1-COM4
0040:0008	8	Базова адреса вводу-виводу для LPT1-LPT4
		ЗМІШАНА ОБЛАСТЬ ДАНИХ
0040:0010	2	Прапори устаткування
0040:0013	2	Обсяг пам'яті в кілобайтах
		ОБЛАСТЬ ДАНИХ КЛАВІАТУРИ
0040:0017	1	Прапори 1 стану регістрів клавіатури
0040:0018	1	Прапори 2 стану регістрів клавіатури
0040:0019	1	Уведення з додаткового клавіатурного поля
0040:001A	2	Адреса початку буфера клавіатури
0040:001C	2	Адреса кінця буфера клавіатури
0040:001E	32	Буфер клавіатури
		ОБЛАСТЬ НАКОПИЧУВАЧА FDD
0040:003E	1	Стан повторного калібрування
0040:003F	1	Стан двигуна
0040:0040	1	Лічильник числа відключень двигуна
0040:0041	1	Стан останньої операції
0040:0042	7	Байти стану контролера
		ОБЛАСТЬ ДАНИХ ВІДЕОАДАПТЕРА
0040:0049	1	Поточний відеорежим
0040:004A	2	Кількість стовбців у відображуваному тексті
0040:004C	2	Довжина буфера регенерації в байтах
0040:004E	2	Адреса зсуву активної відеосторінки
0040:0050	16	Положення курсору (відеосторінки 0-7)
0040:0060	2	Тип курсору (початок. і кінець рядка розгорнення)
0040:0062	1	Активна відеосторінка
0040:0063	2	Базова адреса відеоконтролера
0040:0065	1	Поточна установка регістра 3x8
0040:0066	1	Поточна установка регістру
		ОБЛАСТЬ ДАНИХ СИСТЕМНОГО ТАЙМЕРА
0040:006C	2	Молодше слово вмісту таймера
0040:006E	2	Старше слово вмісту таймера
0040:0070	1	Індикатор заповнення таймера
		СИСТЕМНА ОБЛАСТЬ ДАНИХ
0040:0071	1	Байт BIOS_BREAK
0040:0072	2	Прапор скидання
		ОБЛАСТЬ ДАНИХ ТВЕРДОГО ДИСКА
0040:0074	1	Стан останньої операції
0040:0075	1	Число підключених накопичувачів HDD
		ЧАСИ ЧЕКАННЯ РЕАКЦІЇ (ТАЙМАУТИ)
0040:0078	4	Значення часів чекання для LPT1-LPT4
0040:007C	4	Значення часів чекання для COM1-COM4
		ОБЛАСТЬ ДАНІ КЛАВІАТУРИ
0040:0080	2	Адреса початку буфера клавіатури
00040:0082	2	Адреса кінця буфера клавіатури
		ОБЛАСТЬ ДАНИХ ВІДЕОАДАПТЕРА
0040:0084	1	Число відображуваних рядків тексту мінус 1
0040:0085	2	Висота символу в рядках розгорнення
0040:0087	1	Параметри відеорежиму
0040:0088	1	Параметри відеорежиму
		ОБЛАСТЬ ЗВ'ЯЗКУ МІЖ ПРОГРАМАМИ
0040:00F0	16	Область, де програма може чи записати введені дані (наприклад, статус).

3. Енергонезалежна пам'ять

Персональні комп'ютери класів IBM PC містять годинник реального часу і енергонезалежну пам'ять (CMOS) ємкістю 64 байта, що одержують живлення від батарейки чи акумулятора. CMOS - база даних, призначена для збереження інформації про конфігурацію ПК. CMOS зберігає свої дані на мікросхемі багаторазового запису (write many-read many). Програма установки BIOS SETUP при записі зберігає в ній свою системну інформацію, яку згодом сама ж і зчитує при завантаженні ПК. Кожна частка має розмір у 1 байт. У даній пам'яті зберігається різна інформація, що включає поточну дату і поточний час, конфігурацію устаткування і т. і. (табл. 3).

Таблиця 3

Адреса	Опис
00H-0d	використовується годинник реального часу (RTC)
0e	байт стану після автоматичного тестування POST
0f	байт стану при поверненні в реальний режим
10H	тип накопичувача на гнучких дисках
11H	(зарезервовано)
12H	тип накопичувачів на твердих дисках (якщо < 15)
13H	(зарезервовано)
14H	байт конфігурації устаткування
15H-16H	розмір базової пам'яті
17H-18H	обсяг вище 1M
19H	тип твердого диска C (якщо > 15)
1a	тип твердого диска D (якщо > 15)
1bH-20H	(зарезервовано)
21H-2d	(зарезервовано)
2eH-2f	пам'ять для контрольної суми вмісту адрес від 10h до 20h
30H-31H	обсяг розширеної пам'яті вище 1M
32H	поточний, в двоїчно-десятковому форматі (наприклад, 20) BCD
33H	допоміжна інформація
34H-3f	(зарезервовано)

Контрольна сума являє собою 16-бітну суму всіх значень, записаних в осередки CMOS з 10H по 20H. В осередок 2EH пишеться старший байт суми, а в 2FH - молодший. Більш докладно RTC:

Адреса (HEX)	Опис
00H	Поточна секунда
01H	Сигнальна секунда
02H	Поточна хвилина
03H	Сигнальна хвилина
04H	Поточна година
05H	Сигнальна година
06H	Поточний день тижня (1 - Неділя)
07H	Поточний день місяця
08H	Поточний місяць
09H	Поточний рік (тільки 2 останні цифри, напр. 98)

Усі значення RTC змережуються в BCD форматі як 2 напівбайти але в десятковому форматі. Наприклад, 31 (dec) зберігається як 31 (hex). Докладніше окремі байти CMOS:
Адреса Опис
- -

0EH Байт діагностики завантаження (POST Byte):
Біти 0 і 1 завжди рівні 0.
Біт 2 - Час вірний (1=вірно, що сьогодні не 30 лютого)
Біт 3 - Невірний завантажувальний твердий диск (1=не можна завантажитися з вінчестеру)
Біт 4 - Помилка розміру RAM (1=POST знайшла невірний розмір RAM)
Біт 5 - Невірний запис про устаткування (1=невірне устаткування)
Біт 6 - Невірна контрольна сума (1=невірна сума CMOS)
Біт 7 - Утрата живлення батареї CMOS (1=утрата живлення)

0FH Байт статусу завершення роботи ПК. Застосовується найчастіше після перезавантаження ПК процедурою SETUP. Значення можуть бути наступні:
0 - якщо було перезавантаження по натисканні Ctrl-Alt-Del чи несподіваний перезапуск. У будь-якому випадку процедура POST не виконується
1 - перезапуск після визначення розміру пам'яті
2 - перезапуск після тесту пам'яті
3 - перезапуск після виявлення помилки пам'яті
4 - перезапуск по запиті завантажника ОС
5 - перезапуск унаслідок далекого переходу (FAR JMP) на адресу 0:0467H
6,7,8 - перезапуск після перевірки захищеного режиму 80286
9 - перезапуск після перепризначення блоку пам'яті (функція 0x87 переривання 0x15)

10H Байт типу дисковода:
Біти 0-3: перший дисковод
Біти 4-7: другий дисковод
0100 = 4 = 1,44 Мб

12H Тип вінчестера (для дисків C: і D:, коли байт знаходиться у проміжку від 1 до 14). Біти 0-3: перший вінчестер. Біти 4-7: другий вінчестер У будь-якому випадку, значення бітів можуть бути наступними: 0000 = 0 - диск не встановлений, інше_значення = тип диска 1111 - дивись адреси 19H и 1AH.

14H Байт устаткування:
Біт 0 = 1, якщо є присутнім дисковод(и)
Біт 1 = 1, якщо є присутнім математичний співпроцесор
Біти 2, 3 не використовуються і рівні 0
Біти 5, 4 - основний відеоадаптер: · 00 - немає чи EGA ·
01 - 40*25 EGA, CGA, VGA ·
10 - 80*25 EGA, CGA, VGA ·
11 - монохромний (ч/б)
Біти 6, 7 - кількість дисководів - 1 (00=1, 01=2, 10=3, 11=4)

15H, 16H Базова пам'ять: 15H - молодший байт 16H - старший байт.

17H, 18H Додаткова пам'ять понад 1 Мб (17H - молодший байт, 18H - старший байт). Розмір записаний у Кб.

19H Тип диска № 0 (3:), якщо значення адреси (12H & 0FH) = 0FH

20H Тип диска № 1 (D:), якщо значення адреси (12H & F0H) = F0H

2EH, 2FH Контрольна сума значень адрес від 10H по 20H 2EH - старший байт · 2FH - молодший байт

34H-3FH РЕЗЕРВ.

Для того, щоб прочитати байт із енергонезалежної пам'яті, необхідно спочатку послати необхідну адресу байта в порт **70h**, а потім виконати команду читання байта з порту **71h.**

Приклад: Визначити розмір розширеної пам'яті

```
/*  Визначення наявності і розміру розширеної   пам'яті */
#include<stdio.h>
#include<dos.h>
#define AT          0xFC
#define PS_2__30    0xFA
#define PS_2__80    0xF8
int main(void)
{
   unsigned char type_ibm, high_byte, low_byte;
   type_ibm = peekb(0xF000, 0xFFFE);
   if(type_ibm == AT || type_ibm == PS_2__30 ||
      type_ibm == PS_2__80)
     {
       outportb(0x70, 0x17);
       low_byte=inportb(0x71);
       outportb(0x70, 0x18);
       high_byte=inportb(0x71);
       printf("Обсяг розширеної пам'яті %u Кбайт\n",
              high_byte * 256 + low_byte);
     }
    else printf("Розширена пам'ять відсутня\n");
   return 0;
}
```

Програмувати CMOS бажано з реального режиму ОС. Для ОС Windows це робиться за допомогою спеціальних драйверів VxД чи SYS.

4. Визначення наявності і розміру додаткової відображаємої пам'яті

Для задоволення специфікації LIM EMS 4.0 менеджер розширеної пам'яті реалізує 28 різних функцій, у багатьох з яких є багато підфункцій. При програмуванні необхідно:

- помістити код функції для функції специфікації, що вимагається, розширеної пам'яті в регістр AH;
- помістити інші аргументи, необхідні для обраної функції, і/чи структури даних у пам'ять;
- передати керування менеджеру розширеної пам'яті шляхом видачі програмного переривання 67h;
- менеджер розширеної пам'яті повертає керування програмі, що видала запит, перезаписуючи код функції, поміщений у регістр AH, кодом для запитаної операції. Код стану 00h сигналізує про успішне завершення функції. Будь-яке інше значення показує, що менеджер розширеної пам'яті наштовхнувся на які-небудь проблеми. Значення кодів помилок і їхній зміст перераховані далі.

Для маніпулювання додатковою відображуваною пам'яттю використовується переривання **67h.** Функція **42h** даного переривання повідомляє:

- у регістрі DX сумарна кількість сторінок EMS по 16Кб кожна;
- у регістрі BX число наявних у даний момент вільних сторінок;

- у регістрі AH статус після виконання функції (якщо 0, операція завершилася успішно, інакше - помилка).

Однак використовувати переривання **67h** можна тільки за умови, що в комп'ютері забезпечена апаратна і програмна підтримка додаткової відображуваної пам'яті. Тому перед викликом переривання **67h** необхідно переконатися в тім, що в ланцюжку драйверів є драйвер додаткової відображуваної пам'яті (у поле ім'я_пристрою заголовка драйвера ДОП повинне бути записане ім'я EMMXXXX0).

Приклад. Визначення наявності і розміру додаткової відображуваний пам'яті (встановити значення Alignment рівне byte):

```
#include<stdio.h>
#include<dos.h>
int main(void)
{
  union REGS regs;
  struct SREGS segregs;
  typedef struct _DRIVER
   {
     struct _DRIVER far *pNext;
     int DevAttr;
     unsigned int Strategy;
     unsigned int Interrupt;
     char sName[8];
   };
  struct _DRIVER far *ptr;
  unsigned int offset;
  regs.h.ah = 0x52;
  intdosx(&regs, &regs, &segregs);
  ptr = MK_FP(segregs.es, regs.x.bx + 0x22);
  offset = FP_OFF(ptr);
  while(offset != 0xFFFF)
    ( if(ptr->DevAttr < 0)
      if(!strcmp(ptr->sName, "EMMXXXX0"))
       {
         regs.x.ax = 0x4200;
         int86(0x67, &regs, &regs);
         if(regs.h.ah)
               printf("Помилка при звертанні до адміністратора
               EMS\n");
          else  printf("Обсяг додаткової пам'яті %lu Кбайт\n",
                        (unsigned long)regs.x.dx * 16);
         return 0;
       }
     ptr=ptr->pNext;
     offset = FP_OFF(ptr);
    }
  printf("Додаткова пам'ять не підтримується\n");
  return 0;
 }
```

Якщо менеджер розширеної пам'яті існує і готовий обслуговувати запити, можна видати функцію 7 "Одержати версію", для того щоб упевнитися, що версія менеджера розширеної пам'яті, з яким програма спілкується, підтримує версію специфікації

розширеної пам'яті, що їй потрібна. Ця функція повертає число в двоічно-десятковому коді з двох цифр у регістрі AL. Старші чотири біти числа показують основний номер версії. Молодші чотири біти чи дробова частина цього числа можуть використовуватися постачальниками для позначення специфіки виробників менеджерів розширеної пам'яті. Отже, програмі краще виконати порівняння на "більше чи дорівнює".

5. Визначення наявності й адрес зовнішніх ПЗП

У комп'ютері IBM PC існує єдиний адресний простір для оперативної і постійної пам'яті. Між адресами C000:0000 і E000:0000 можуть бути встановлені зовнішні ПЗП на платах ві-деоадаптерів EGA, VGA, контролера твердого диска, адаптера мережі і т.п. Адресний простір ПЗП можна вважати розбитим на дві частини: область від C000H до C7FFFH сканується раніш виконання більшості тестів, інший адресний простір C8000H - EFFFFH сканується після виконання більшості тестів.

Перша область може містити ПЗП адаптерів використовуваних процедурами ініціалізації (наприклад, ПЗП дисплея). Обидві області скануються з кроком рівним 2Кбайта в пошуках коректних модулів ПЗП адаптерів.

Коректний модуль ПЗП містить наступну інформацію:

байт 0 - 55H,

байт 1 - 0AAH,

байт 2 - довжина модуля в 512-байтових одиницях (максимально 64К);

байт 3 - крапка входу в модуль ініціалізації, що викликається із BIOS при виконанні тестів включення живлення.

Модуль ініціалізації викликається інструкцією між сегментного виклику і повинний повернути керування BIOS інструкцією міжсегментного повернення.

Таким чином необхідно виконати сканування даної ділянки пам'яті з метою виявлення зовнішніх ПЗП. Кожен блок розміром 2 Кбайт у зазначеному діапазоні перевіряється на сигнатуру AA55h, з якої починаються зовнішні ПЗП . При встановленні сигнатури AA55h перевіряється контрольна сума модуля ПЗП: усі байти підсумовуються по модулю 100h, у результаті чого повинне бути отримане значення 0.

Приклад. Визначити наявності і початкові адреси додаткових ПЗП

```
#include<stdio.h>
#include<dos.h>
unsigned char check_sum(unsigned);
int main(void)
 {
   unsigned segment, signature;
   for(segment = 0x000; segment < 0x000; segment += 128)
    {   /* 128 параграфів складає 2 Кбайт */
     signature = (unsigned)peek(segment, 0);
     if(signature == 0xAA55)
      if(!check_sum(segment))
        printf("Мається ПЗП за адресою %04X:0000\n", segment);
    }
   return 0;
 }

unsigned char check_sum(unsigned segment)
 {
  int i;
  unsigned char sum = 0;
```

```
  long length = (long)peekb(segment, 2) * 512L;
  for (i = 0; i < length; i++)
    sum += (unsigned char)peekb(segment, i);
  return sum;
 }
```

6. Робота з XMS - пам'яттю

Робота з XMS реалізується при наявності в ОЗП драйвера himem.sys.- шляхом визначення адреси функції - диспетчера XMS. Режими для XMS-manager є int **2fh**, funcnion **4300h, 4310h.**

Приклад . Використання функцій по роботі зXMS пам’яттю

```
#include <stdio.h>
#include <dos.h>
union REGS rr;
  /*Структура для обміну блоками пам'яті*/
struct XMS_MOVE
 { unsigned long length;
   unsigned int  source_handle;
   unsigned long source_off;
   unsigned int  dest_handle;
   unsigned long dest_off;
 };
struct XMS_MOVE xmov;
/*     Покажчик на функцію - менеджер XMS пам'яті:*/
void far (*xms_entry_point)(void);
/*     Функція, що організує доступ до XMS:*/
 int xms_install(void)
 { _AX=0x4300;
  geninterrupt(0x2F);
  if (_AL==0)  return 1;
  _AX=0x4310;
  geninterrupt(0x2F);
  xms_entry_point=(void far (*)(void))MK_FP(_ES,_BX);
  return 0;
 }
/*      Функція обробки входу в XMS - пам'ять:*/
 int xms_go(char a, int d)
 { _AH = a;
   _DX = d;
  xms_entry_point();
  rr.x.ax = _AX;
  rr.x.bx = _BX;
  rr.x.dx = _DX;
  if (rr.x.ax==0)  return 1;
  return 0;
 }
/*      Функція маніпулювання з блоками XMS і ConvertMem:*/
 int xms_movblk(long len, int hnd1, long off1,
                          int hnd2, long off2)
 { xmov.length = len;           /*  Exactly byte*/
```

```
   xmov.source_handle = hnd1;    /*from   (0)*/
   xmov.source_off = off1;       /*offset (abs adr)*/
   xmov.dest_handle = hnd2;      /*to     (0)*/
   xmov.dest_off = off2;         /*offset (abs adr)*/
   _SI = FP_OFF(&xmov);
   if (xms_go(0x0B,0))  return 1;
 return 0;
 }
```

Приклад організації роботи з XMS пам’яттю.

```
 int main(void)
 {     unsigned int  handle;
       unsigned long r;
       unsigned char far *txt="0123456789";
/*Інсталяція*/
    if (xms_install()) { printf("\nError XMS"); return 1; }
/*Захоплення блоку  XMS - 3kb*/
    if (!xms_go(0x09,3))  printf("\nAllocXMS= %X", rr.x.dx);
    handle=rr.x.dx;
/*Перенести блок 10 байт  і XMS у ConvertMem*/
    r = (unsigned long)txt;     //abs address
    xms_movblk(10, handle,0, 0,r);
/*Захоплення ще одного блоку XMS  - 12kb*/
    if (!xms_go(0x09,12))  printf("\nAllocXMS= %X", rr.x.dx);
    handle=rr.x.dx;
/*Звільнити останній блок XMS -12Kb*/
    if (!xms_go(0x0B,handle)) printf("\n Free");
}
```

7. Тестування областей пам'яті

Приклади тестування.

Приклад тестування відеопам'яті.

```
void V_MemWin( void )
{  char res,i;
   union REGS rr;
   int size, j;
/* Визначення розміру відеопам'яті int10 Fun12*/
  rr.h.ah = 0x12;
  rr.h.bl = 0x10;
  int86(0x10, &rr, &rr);
  res = rr.h.bl;    //Розмір відеопам'яті
  switch(res)
    { case 0:   size=64;   break;
      case 1:   size=128;  break;
      case 2:   size=192;  break;
      case 3:   size=256;  break;
    }
  rr.h.ah = 0x0f;
  int86(0x10, &rr, &rr);
```

```
/* Запам'ятати номер активної відео сторінки*/
  res = rr.h.bh;
  i=0;
  for(j=0;j<size/64;j++)
/* Читати усі відео сторінки*/
    { rr.h.ah = 0x05;
      rr.h.al = i;     // це її номер
      int86(0x10, &rr, &rr);
      aindow(1,1,24,77);
      textbackground(i);  clrscr();
      printf("\n VIDEO_PAGE N %d ",i);
      getch();
      i++;
    }

/* Відновити номер активної відео сторінки*/
  rr.h.ah = 0x05;
  rr.h.al = res;
  int86(0x10, &rr, &rr);
}
```

Тестування DMA і контролера переривань

```
 char Test1,Test2, Flag,i,j;
  Flag=0;          //*** Тест ПДП
  for(i=0;i<3;i++)
    { for(j=0;j<8;j++)
        { Test1=inportb(j);
          outportb(j,Test1);
          Test2=inportb(j);
          if (Test1!=Test2)  {  Flag=1;  break; }
        }
  ...

  Flag=0;  /* Тест КОНТРОЛЕРА  ПЕРЕРИВАНЬ*/
  disable();
  for(i=0;i<3;i++)
  for(j=0x20;j<0x22;j++)
    { Test1=inportb(j);
      outportb(j,Test1);
      Test2=inportb(j);
      if (Test1!=Test2)  {  Flag=1;  break; }
    }
  enable();
  ...
```

Перевірка роботи DMA і контролера переривань. Обсяг XMS,

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

#define AT          0xFC
```

```
#define PS_2__30    0xFA
#define PS_2__80    0x8

/* Покажчик на функцію - менеджер XMS пам'яті:*/
void far (*xms_entry_point)(void);
union REGS rr;

void test_DMA();
int XMS_size();
void A20_check();
int xms_install(void);
int xms_go(char a, int d);

void main()
{
 clrscr();
 test_DMA();
 XMS_size();
 A20_check();
 getch();
}

/* Test DMA and interrupt controller*/
void test_DMA()
{
  char Test1,Test2, Flag,i,j;
  Flag=0;          // Test DMA
  for(j=0;j<8;j++)
      {
          Test1=inportb(j);
          outportb(j,Test1);
          Test2=inportb(j);
          if (Test1!=Test2)
          {
          Flag=1;  break;
          }
      }
  if(Flag) printf("DMA is absent\n");
  else     printf("DMA present\n");

  Flag=0;  // Test interrupt controller
  disable();
  for(j=0x20;j<0x22;j++)
      { Test1=inportb(j);
          outportb(j,Test1);
          Test2=inportb(j);
          if (Test1!=Test2)  {  Flag=1;  break; }
      }
  enable();
  if(Flag) printf("Interrupt controller is absent\n");
  else     printf("Interrupt controller is present\n");
}
```

```
/* Check for XMS size*/
int XMS_size()
{
  union REGS reg;
  unsigned char type_ibm, high_byte, low_byte;

  /* Check for XMS present*/
  printf("\nTESTING XMS MEMORY:");
  reg.x.ax=0x4300;
  int86(0x2f,&reg,&reg);
  if(reg.h.al==0x80)  printf(" OK.\n");
  else
      {
       puts(" Memory error, or HIMEM is absent.");
       return 0;
      }

  /* Check for XMS size*/
  type_ibm = peekb(0xF000, 0xFFFE);
  if(type_ibm == AT || type_ibm == PS_2__30 ||
       type_ibm == PS_2__80)
      {
          outportb(0x70, 0x17);
          low_byte=inportb(0x71);
          outportb(0x70, 0x18);
          high_byte=inportb(0x71);
          printf("Size of XMS %u Кбайт\n",
                      high_byte * 256 + low_byte);
      }
     else printf("XMS is absent\n");
}

/* Check for enabled line A20*/
void A20_check()
{
  _AX=0x4300;
  geninterrupt(0x2F);
  if(_AX) printf("\nHIMEM is enabled\n");
  else    printf("\nHIMEM  is disabled\n");

     if(xms_install()) printf("Cannot install XMS\n");
     else
          if(xms_go(0x07,0x002f))
               printf("A20 is disabled\n");
          else
               printf("A20 is phisically enabled\n");
}

/*Функція, що організує доступ до XMS:*/
 int xms_install(void)
 {
     _AX=0x4300;
     geninterrupt(0x2F);
```

```
    if (_AL==0)  return 1;
  _AX=0x4310;
  geninterrupt(0x2F);
  xms_entry_point=(void far (*)(void))MK_FP(_ES,_BX);
  return 0;
 }

/*Функція обробки входу в XMS - пам'ять:*/
 int xms_go(char a, int d)
 {
    _AH = a;
    _DX = d;
  xms_entry_point();
  rr.x.ax = _AX;
  rr.x.bx = _BX;
  rr.x.dx = _DX;
  if (rr.x.ax==0)  return 1;
  return 0;
 }
```

7.1. Методи тестування ОЗУ

Тест ОЗП методом послідовного запису/читання. У кожний блок пам'яті послідовно записуються нулі. Потім байти блоку зчитуються і порівнюються з нулем. Далі, у той же блок, пишуться одиниці (FFh) і ті ж дії повторюються.

Тест пам'яті методом шахового коду. З початку блоку по два байти послідовно пишуться коди AAh і 55h, потім перевіряється правильність їхнього зчитування.

Тест пам'яті методом зчитування/запису. У блок пам’яті послідовно пишуться нулі, потім кожен байт блоку зчитується, порівнюється з нулем і туди пишуться одиниці (FFh). Коли досягнутий кінець блоку, від кінця до початку зчитуються раніше записані одиниці і пишуться нулі.

Тест пам'яті методом парної/непарної адреси**.** З початку блоку послідовно у байти з парною адресою записуються нулі, а з непарною адресою - одиниці (FFh). Потім байти з початку блоку зчитуються і парні порівнюються з нулем, а непарні з кодом FFh.

Тест пам'яті методом «бігучої» одиниці**.** У блок спочатку записуються нулі, потім у перший байт блоку записується одиниця. Відбувається читання всіх байтів від кінця блоку до його початку і їхнє порівняння з кодами 00h і FFh. Коли досягнутий початок блоку, у другий байт пишеться одиниця, і ті ж дії повторюються.

Алгоритм попарного зчитування забезпечує будь-які адресні переходи з різною зміною інформації при зчитуванні. У початкову адресу записується одиниця на фоні всіх інших нулів, а потім зчитуються адреси перший із другим, перший із третім і так далі до останнього, потім друга адреса з першим, другий з третім і так далі.

Приклад .Визначити обсяг та протестувати пам'ять.

```
#include <dos.h>
#include <stdio.h>
#include <string.h>
union REGS rr;
struct SREGS srr;
unsigned char far *ll;
unsigned int fmcb,dos_seg;
struct mcb
```

```
{
 char type;
 unsigned int owner,size;
 char unnved[3];
 char name[9];
}mcb;

char zbl=0,tot=0;
void main (void)
{
 int p,i,n;
 unsigned int temp,d_s;
 rr.x.ax=0x5200;
 intdosx(&rr,&rr,&srr);
 ll=MK_FP(srr.es,rr.x.bx);
 dos_seg=srr.es;
 fmcb=ll[-1]<<8 | ll[-2];
 temp=fmcb;d_s=dos_seg;
 while (zbl<2)
   {
    movedata(temp,0,_DS,(unsigned)&mcb.type,0x10);
    mcb.name[8]=0;
    if((mcb.type!='M')&&(mcb.type!='Z')&&(zbl>0))
      break;//not UMB
    printf("\n %c",mcb.type);
    printf("   %0.4x",temp+1);
    printf("   %0.4x",mcb.owner);
    printf("   %0.4x",mcb.size);
    if(mcb.owner==0)
      {printf("    Free Block");
       n=mem_test(temp,temp+mcb.size+1);
       if(n)
      printf(" Test Ok");
       else
      printf(" Trouble");
      }
    else
      if(mcb.owner==temp+1)
     printf("  %s",mcb.name);
      else
     if(mcb.owner<=dos_seg)
       printf("    System\n");
     else
       printf("    Enviroment\n");
    temp+=mcb.size+1;
    tot++;
    if(mcb.type=='Z')
      zbl++;
   }
}
int mem_test (unsigned int beg,unsigned int end)
{
 int n=1;
```

```
 unsigned long i,j,k,z;
 unsigned char test,test1;
 for(i=beg+1;i<end;i++)
   for(j=0;j<16;j++)
     {
      if(!(z=j%2))
     pokeb(i,j,0);
      else
     pokeb(i,j,0xff);
     }
 for(i=beg+1;i<end & n ;i++)
   for(j=0;j<16;j++)
     {
      if(!(z=j%2))
     {
      test=peekb(i,j);
      if(test!=0)
         {n=0;return;}
     }
      else
     {
      test=peekb(i,j);
      if(test!=0xff)
        {n=0;return;}
      }
     }
 return n;
}
```

Тема № 6. ІНТЕРФЕЙС ПРОГРАМИ з OS. КЕРУВАННЯ ПРОЦЕСАМИ. ОБМІН ДАНИМИ МІЖ ПРОЦЕСАМИ.

1. Префікс програмного сегмента (PSP)

Коду програми, що завантажується в пам'ять, передує область у 256 байт, що називається префіксом програмного сегмента (PSP). При завантаженні програми в пам'ять, PSP розміщено за блоком керування пам'яттю MCB. Сегментна частина початкової адреси PSP записана в полі "Власник" блоку керування пам'яттю MCB. Префікс програмного сегмента має структуру (мал. 6.):

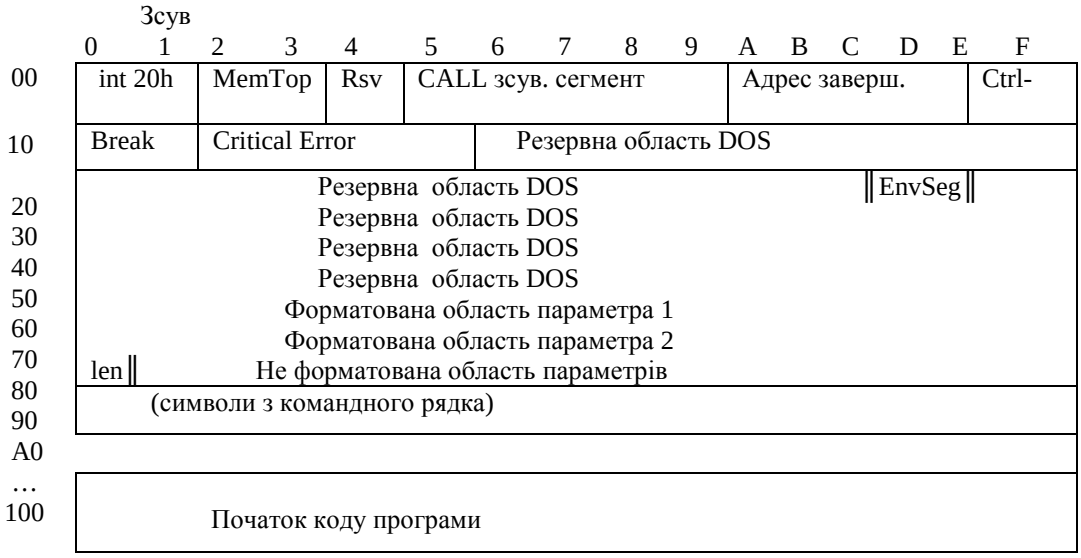


Рис. 6

Першим полем префікса програмного сегмента є команда INT 20h - програми можуть виконувати на неї перехід для виходу в MS DOS. Поле MemTop містить адресу вершини доступної пам'яті системи в параграфах (один параграф = 16 байт). Далі один байт резервується - поле Rsv. Наступне поле CALL... містить команду переходу до диспетчеру функцій DOS. Для COM-файлів байти 6-7 PSP показують кількість доступних байтів у програмному сегменті. Наступні - три чотирьохбайтові поля у форматі *сегмент:зсув*, що містять відповідно адресу завершення, адресу обробки Ctrl-Break і оброблювач критичних помилок. За ними йде резервна область DOS.

Поле EnvSeg містить сегментний адрес оточення DOS. Далі випливає ще одна резервна область DOS, а також форматовані та не форматовані області параметрів. Поле len указує довжину не форматованної області параметрів. Сама ж область містить символи, отримані з командного рядка (крім директив перепризначення).

Завантажена для виконання програма може читати інформацію з PSP чи змінювати деякі поля.

У мові С у файлах dos.h, process.h і stdlib.h визначена глобальна перемінна _psp, значення якої дорівнює сегментній частині адреси префікса програмного сегмента, виконуваної в даний момент програми. Ця перемінна у програмі з'являється в такий спосіб:

```
extern unsigned int _psp;
```

Зрозуміло, що сегментна адреса блоку керування пам'яттю MCB виконуваної програми складає _psp - 1 .

Крім того, сегментний адрес PSP повертає функція getpsp(), прототип якої знаходиться у файлі dos.h. Ця функція використовує переривання DOS 62h.

У мові С мається функція peekb(сегмент, зсув), що повертає вміст байта з адресою сегмент:зсув.

Наприклад, у результаті виконання операторів

```
unsigned char tip_mcb;
tip_mcb = peekb(_psp - 1, 0);
```

перемінна tip_mcb одержить значення початкового байта блоку керування пам'яттю.

Уміст 16-розрядного слова з адресою сегмент:зсув повертає функція peek(сегмент, зсув). Так, у результаті виконання операторів

```
unsigned int memory_size;
memory_size = peek(_psp, 2);
```

у перемінну memory_size буде записаний зміст поля MemTop префікса програмного сегмента виконуваної в даний момент програми.

Нижче приводиться текст функції mcb(), що дозволяє довідатися адресу сегмента виділеного програмі блоку пам'яті; визначити, чи є блок останнім у ланцюжку чи ні; вивести на дисплей адреса сегмента власника блоку; з'ясувати розмір блоку пам'яті, адреса сегмента PSP, а також обсяг доступної пам'яті в системі:

```
#include<math.h>
#include<conio.h>
void mcb(void)
 {
  extern unsigned int _psp;
  unsigned char tip_mcb;
  unsigned int segment, owner, size, memory_size;
  segment = _psp - 1;
  printf("Адреса сегмента блоку MCB %X\n", segment);
  tip_mcb = peekb(segment, 0);
  if(tip_mcb == 'M')
    printf("Блок не останній \n");
   else if(tip_mcb == 'Z')
    printf("Блок останній\n");
   else printf("Поле \"тип\" невірне \n");
  owner = peek(segment, 1);
  if(owner == 0)
    printf("Блок вільний\n");
   else printf("Адреса сегмента власника блоку %X\n", owner);
  size = peek(segment, 3);
  printf("Розмір блоку %X параграфів\n", size);
  printf("   чи %lu байтів\n",((unsigned long)size) * 16);
  printf("Адреса сегмента PSP %X\n", _psp);
  memory_size = peek(_psp, 2);
  printf("Вершина пам'яті %X параграфів\n", memory_size);
  printf("    чи %u Кбайт.\n",
        (unsigned)floor( (double)memory_size / 64.0 + 0.5) );
  printf("ДЛЯ  ПРОДОВЖЕННЯ  НАТИСНІТЬ  КЛАВІШУ...\n");
  getch(); /* Чекання натискання клавіші  */
 }
```

2. Зв'язок програми з командним рядком і оточенням

Мінімальний користувальницький інтерфейс забезпечує командний рядок, що крім імені файлу з програмою, що завантажується, може містити параметри і ключі. Функція main() у програмі, якій необхідний доступ до командного рядка, повинна бути описана наступним чином:

```
int main(int argc, char *argv[])
  { ...  /*  Тіло  функції */ ...  }
```

Перемінна argc приймає значення, рівне числу аргументів командного рядка (розділених пробілами), а масив argv містить покажчики на адреси, по яких зберігаються самі аргументи у виді символьних рядків. При цьому ім'я програми теж вважається аргументом і його адреса зберігається в пам'яті як argv[0].
Точніше, argv[0] містить також шлях до програми, що виконується, незалежно від того, чи був він зазначений у командному рядку чи ні. Масив argv може бути описаний і в іншому виді:

```
char **argv;
```

Тоді ім'я програми визначається за допомогою операції *argv .

У програмі, що нижче приводиться, продемонстровані елементарні способи доступу до аргументів командного рядка:

```
/*  Вивод списку аргументів  */
#include<dos.h>
#include<stdio.h>
int main(int argc, char *argv[])
 {
  int num = 0;
  printf("\n Усього %d аргументів\n", argc);
  if(_osmajor < 3)
    { /*  Пропуск argv[0]  */
      printf("argv[0] не визначений\n");
      num++;
    }
  for(; num < argc; num++)
     printf("argv[%d] =>   %s\n", num, argv[num]);
  return 0;
 }
```

У файлі dos.h C оголошені глобальна перемінна _argc, еквівалентна argc, і масив *_argv[], еквівалентний *argv[]. Їх використання дозволяє описувати функцію main() стандартним образом:

```
int main(void)
```

Якщо застосувати дані глобальні змінні, то попередня програма буде приведена до наступного виду:

```
/*  Вивод списку аргументів  */
#include<dos.h>
#include<stdio.h>
int main(void)
 {
  int num = 0;
  printf("\n Усього %d аргументів\n", _argc);
  if(_osmajor < 3)
    { /*  Пропуск _argv[0]  */
      printf("_argv[0] не визначений\n");
      num++;
    }
  for(; num < _argc; num++)
   printf("_argv[%d] =>   %s\n", num, _argv[num]);
  return 0;
 }
```

Можна також представити функцію main() функцією трьох аргументів, додавши до argc і argv ще один масив char *envp[] (чи char **envp), що містить покажчики на строкові змінні поточного оточення MS DOS. Оточення представляє собою послідовність текстових рядків, кожний з яких закінчується нульовим байтом. Наприкінці оточення

записан додатковий байт, що має значення 0. Програма, що приводиться нижче, виводить на екран рядка оточення, визначаючи кінець оточення по додатковому нульовому байті (чи по порожньому рядку в термінах С):

```
/* Вивід рядків оточення */
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char *argv[], char *envp[])
 {
  for( ; *envp; ++envp)printf("%s\n", *envp);
  return 0;
 }
```

У файлі stdlib.h С оголошена глобальна перемінна *environ[], еквівалентна *envp[]. Її використання дозволяє виключити з опису функції main() оголошення char *envp[] :

```
/* Вивід рядків оточення */
#include<stdlib.h>
#include<stdio.h>
int main(void)
 {
  int i = 0;
  while(environ[i]) printf("%s\n", environ[i++]);
  return 0;
 }
```

MS DOS поміщає додатковий рядок після формального оточення. Цей рядок містить найменування дисковода, шлях і ім'я файлу, з якого була завантажена програма (значення цього рядка і одержує елемент argv[0]). Відразу за останнім рядком оточення знаходиться нульовий байт, що вказує на кінець формального оточення. Наступні два байти містять число додаткових рядків (звичайно 0001h). Слідом за значенням лічильника розташован додатковий рядок з повним ім'ям файлу, що закінчується нульовим байтом.

3. Запуск програми

MS DOS може завантажувати і виконувати програмні файли двох типів - COM і EXE. Через сегментацію адресного простору мікропроцесора обидва типи програм можуть виконуватися в будь-якому місці пам'яті. Програми ніколи не пишуться в припущенні, що вони будуть завантажуватися з визначеної адреси (за винятком деяких самозавантажних, захищених від копіювання ігрових програм).

4. Завершення програми

Оператор функції int main() С-програми

```
return код_повернення;
```

забезпечує завершення програми і вихід у MS DOS з передачею коду завершення, обумовленого параметром код_повернення. Код завершення програми можна проаналізувати в командному файлі з допомогою команди if ERRORLEVEL. Звичайно, якщо програма завершилася без помилок, вона повертає код 0; при наявності помилок це значення відмінне від нуля. При налагодженні програми в інтегрованому середовищу С++ чи Borland++ код завершення програми (Program exit code) можна подивитися, обравши пункт Get info... у меню File.

У програмі можна установити одну чи кілька функцій, які будуть виконуватися перед виходом у MS DOS. Для установки (чи реєстрації) функції завершення програми використовується функція atexit(), що має наступний синтаксис:

```
#include<stdlib.h>
```

```
int atexit(atexit_t func);
```
Перемінна func являє собою покажчик на встановлювану функцію виходу з програми. Функція atexit() повертає 0 при успішному завершенні і ненульове значення при невдалому завершенні (наприклад, якщо для установки функції завершення не вистачає пам'яті).

Кожен виклик функції atexit() реєструє різні функції завершення. Усього може бути встановлене до 32 функцій. При завершенні програми і виході в MS DOS за допомогою оператора

return код_повернення

спочатку виконується остання зареєстрована функція, потім передостання і т.д. Наприклад, результатом виконання програми

```
#include   <stdlib.h>
#include   <stdio.h>
void exit_fn1(void)
{
   printf("Викликана функція exit_fn1\n");
}
void exit_fn2(void)
{
   printf("Викликана функція exit_fn2\n");
}
int main(void)
{
   atexit(exit_fn1);
   atexit(exit_fn2);
   return 0;
}
```

буде наступна інформація:

Викликано функцію exit_fn2

Викликано функцію exit_fn1

Для завершення програми і виходу в MS DOS можна використовувати також функції exit(), _exit() і abort().

Функція

```
#include<stdlib.h>
void exit(int status);
```

закриває усі відкриті в програмі файли, викликає всі зареєстровані за допомогою atexit() функції завершення і забезпечує вихід у MS DOS з кодом повернення status. У якості аргументу функції exit() можна використовувати ціле число, наприклад

exit(2);

чи одну з наступних констант:

EXIT_SUCCESS - нормальне завершення,

EXIT_FAILURE - завершення з помилкою,

наприклад

exit(EXIT_SUCCESS);

Константи EXIT_SUCCESS і EXIT_FAILURE визначені у файлі stdlib.h і мають значення 0 і 1 відповідно.

Функція

```
#include<stdlib.h>
void _exit(int status);
```

робить вихід у MS DOS з кодом повернення status без закриття яких-небудь файлів і виклику функцій завершення.

Функція

```
#include<stdlib.h>
```

```
void abort(void);
```
виводить повідомлення про аварійне завершення програми

Abnormal program termination

і перериває програму за допомогою виклику функції _exit() з кодом завершення 3. Приклад використання функції abort():

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   printf("Виклик abort()\n");
   abort();
   return 0; /* Цей рядок ніколи не буде досягнут */
}
```

5. Керування процесами

MS DOS дає можливість запустити одну програму з іншої. При цьому програма, що запускає, називається процесом-батьком, а що запускається - процесом-нащадком.

У С створення і запуск процесу-нащадка реалізується за допомогою функцій сімейства spawn(), прототипи яких знаходяться у файлі process.h.

Функція int spawnl(режим, ім'я_файлу, аргумент0, ..., аргумент, NULL); має наступні аргументи:

1) "режим" (типу int) визначає характер взаємин між процесом-нащадком і процесом-батьком і може приймати одне з трьох значень:

P_WAIT - призупинити виконання процесу-батька до тих пір, поки не завершиться виконання процесу-нащадка;

P_OVERLAY - перекрити процес-батько процесом-нащадком;

P_NOWAIT - продовжити роботу процесу-батька паралельно з процесом-нащадком (у компіляторах фірми Borland до Borland++ 3.1 включно даний режим не підтримується і зарезервований для подальших розробок. Режим P_NOWAIT реалізований у компіляторах фірми Microsoft, починаючи з Microsoft C Optimizing Compiler 5.1).

2) "ім'я_файлу" (типу char*) - це символьний рядок, що містить ім'я файлу з програмою процесу-нащадка.

3) "аргумент 0" ..., "аргумент N" (типу char*) - це символьні рядки, призначені для передачі аргументів у процес-нащадок (за згодою "аргумент0" являє собою копію параметра "ім'я_файлу"). Загальна довжина списку аргументів не повинна перевищувати 128 байт.

4) параметр NULL - це термінатор, що сигналізує закінчення списку аргументів.

При успішному закінченні функція spawnl() повертає код завершення процесу-нащадка, а при виникненні помилки повертається -1, і глобальна перемінна errno одержує одне з наступних значень, що конкретизують помилку:

E2BIG - список аргументів занадто довгий;

EINVAL - невірний аргумент;

ENOENT - шлях чи ім'я файлу не знайдені;

ENOEXEC - помилка формату EXEC;

ENOMEM - не вистачає пам'яті.

Перемінна errno і перераховані константи визначені у файлі errno.h. Функція

```
#include<stdio.h>
void perror(const char *s);
```

виводить на екран повідомлення про помилку, що відповідає значенню перемінної error, випереджаючи його символьним рядком *s.

Приклад програми, що спочатку викликає описану вище функцію mcb(), потім запускає процес-нащадок з файлу CHILD.EXE, використовуючи функцію spawnl() у режимі P_WAIT, а після завершення процесу-нащадка генерує звуковий сигнал частотою 1000 Гц і знову викликає функцію mcb():

```
/* Неоверлейний виклик нащадка з процесу-батька
функцією spawnl(). Тестувати після виходу з Turbo */
#include<process.h>
#include<stdio.h>
#include<dos.h>
#include"mcb.c"
int main(void)
 {
  int i, stat;
  printf("\n РОЗПОДІЛ ПАМ'ЯТІ ДО ЗАВАНТАЖЕННЯ"
         " ПРОЦЕСУ-НАЩАДКА\n");
  mcb();
  printf("\n ВИКЛИК ПОРОДЖЕНОГО ПРОЦЕСУ\n");
  stat = spawnl(P_WAIT, "\\TC2\\CHILD.EXE", "\\TC2\\CHILD.EXE",
                "15", "-67", NULL);
  if (stat == -1)
      {
        perror("Помилка при неоверлейному виклику");
        return 1;
      }
     else
      {
        printf("\n ЗВУК  ІЗ ПРОЦЕСУ-БАТЬКА...\n");
        for(i = 0; i <= 30000; i++)
          sound(1000);
        nosound();
        printf("\n РОЗПОДІЛ ПАМ'ЯТІ ПІСЛЯ ПОВЕРНЕННЯ"
               " У ПРОЦЕС-БАТЬКО\n");
        mcb();
       }
  return 0;
 }
```

У С включення звукового сигналу - sound(частота), а вимикання звуку - nosound(). Прототипи - у файлі dos.h .

Припустимо, що модуль даної програми, що виконується знаходиться у файлі PARENT.EXE. Після завантаження програми з цього файлу спочатку буде викликана функція mcb(), у результаті чого на екрані дисплея з'явиться інформація про розподіл пам'яті в системі. Потім дана програма викликає процес-нащадок з файлу \TC2\CHILD.EXE, передавши йому як аргумент 0 рядок "\TC2\CHILD.EXE", як аргумент1 - рядок "15", у якості аргументу2 - рядок "-67".

Текст програми процесу-нащадка приводиться нижче:

```
#include<stdio.h>
#include<dos.h>
#include<stdlib.h>
#include"mcb.c"
int main(int argc, char *argv[])
{
 int sum;
 if(_osmajor >= 3)
```

```
   printf("ПРОЦЕС - ПОТОМОК З ФАЙЛУ %s ЗАПУЩЕНИЙ\n", argv[0]);
 sum = atoi(argv[1]) + atoi(argv[2]);
 printf("Сума %s і %s дорівнює %d\n", argv[1], argv[2], sum);
 printf("\n РОЗПОДІЛ ПАМ'ЯТІ ПІСЛЯ ЗАВАНТАЖЕННЯ "
        "ПРОЦЕСУ-НАЩАДКА\n");
 mcb();
 return 0;
 }
```

Програма виконує підсумовування чисел 15 і -67, переданих їй як аргументи з процесу-батька, попередньо перетворивши їх зі строкового представлення у формат цілого числа за допомогою функції atoi(), що має синтаксис:

int atoi(const char *s);

Прототип функції atoi() знаходиться у файлі stdlib.h, тому в текст даної програми включена директива #include<stdlib.h> .

Наприкінці виконання процесу-нащадка знову викликається функція mcb(), що дозволяє одержати інформацію про розподіл оперативної пам'яті.

Після завершення процесу-нащадка керування повертається процесу-батьку. При цьому спочатку генерується звук частотою 1000 Гц, а потім третій раз викликається функція mcb().

Аналіз інформації, виведеної " !o5~a++й mcb(), дозволяє оцінити розподіл пам'яті до виклику процесу-нащадка (мал. 7), після виклику процесу-нащадка (мал. 8) і, нарешті, після повернення керування процесу-батьку (мал. 9). При цьому передбачається, що після компіляції програм і створення EXE-файлів був здійснений вихід із середовища C в MS DOS і уже відтіля запущений процес-батько PARENT.EXE.

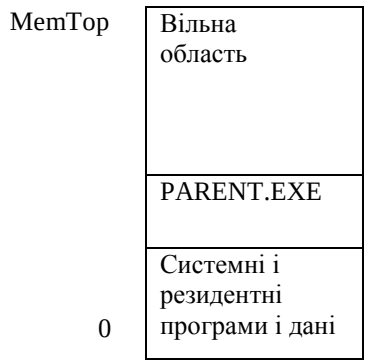


Рис. 7

Рис. 8

Рис. 9

Другий варіант тієї ж задачі: процес-нащадок викликається за допомогою функції spawnl() у режимі P_OVERLAY:

```
/*  Оверлейний виклик процесу-нащадка з процесу-батька
   функцією spawnl(). Тестувати після виходу з Turbo */
#include<process.h>
#include<stdio.h>
#include<dos.h>
#include"mcb.c"
int main(void)
 {
  int stat;
  printf("\n РОЗПОДІЛ ПАМ'ЯТІ ДО ЗАВАНТАЖЕННЯ"
         " ПРОЦЕСУ-НАЩАДКА\n");
  mcb();
```

```
  printf("\n ВИКЛИК ПОРОДЖЕНОГО ПРОЦЕСУ\n");
  stat = spawnl(P_OVERLAY, "\\TC2\\CHILD.EXE",
"\\TC2\\CHILD.EXE",
                "15", "-67", NULL);
  if (stat == -1)
      {
        perror("Помилка при оверлейному виклику");
        return 1;
      }
  return 0;
 }
```

У цьому випадку розподіл пам'яті до завантаження процесу-нащадка має вид, показаний на мал. 10. При запуску процесу-нащадка CHILD.EXE він завантажується в область пам'яті, займану процесом-батьком PARENT.EXE, у результаті чого процес-батько знищується (мал. 11).

Рис. 10 Рис. 11

По завершенні виконання процесу-нащадка керування передається MS DOS. Такий спосіб запуску однієї програми з іншої називається оверлейним.

У сімейство функцій spawn входять функції spawnl, spawnlp, spawnle, spawnlpe, а також spawnv, spawnvp, spawnve, spawnvpe.

У приведених вище прикладах аргумент ім'я_файлу, рівний \TC2\CHILD.EXE, однозначно визначає місце розташування файлу процесу-нащадка за рахунок завдання шляху до нього. Якщо у функції spawnl() задати тільки ім'я файлу без указівки до нього шляху, те функція буде шукати файл у поточному каталозі. Функція spawnlp(), яка має такий же синтаксис, що і spawnl() :

int spawnlp(режим, ім'я_файлу, аргумент0, ..., аргумент, NULL);

у цьому випадку буде шукати файл нащадка спочатку в поточному каталозі, а потім у всіх каталогах, зазначених у перемінної оточення PATH. Приклад використання функції pawnlp():

```
/*  Неоверлейний виклик процесу-нащадка з процесу-батька
   функцією spawnlp(). Тестувати після виходу з Turbo */
#include<stdio.h>
#include<dos.h>
#include<process.h>
#include"mcb.c"
int main(void)
 {
  int i, stat;
```

```
  printf("\n РОЗПОДІЛ ПАМ'ЯТІ ДО ЗАВАНТАЖЕННЯ"
         " ПРОЦЕСУ-НАЩАДКА\n");
  mcb();
  printf("\n ВИКЛИК ПОРОДЖЕНОГО ПРОЦЕСУ\n");
  stat = spawnlp(P_WAIT, "CHILD.EXE", "CHILD.EXE",
                 "15", "-67", NULL);
  if (stat == -1)
      {
        perror("Помилка при неоверлейному виклику");
         return 1;
      }
     else
      {
        printf("\n ЗВУК  ІЗ ПРОЦЕСУ-БАТЬКА...\n");
        for(i = 0; i <= 30000; i++)
          sound(1000);
        nosound();
        printf("\nрозподіл ПАМ'ЯТІ ПІСЛЯ ПОВЕРНЕННЯ"
               " У ПРОЦЕС-БАТЬКО\n");
        mcb();
       }
  return 0;
 }
```

При використанні функцій spawnl() і spawnlp() процес-нащадок успадковує оточення процесу-батька. Функція spawnle(), що має синтаксис:

int spawnle(режим, ім'я_файлу, аргумент0, ..., аргумент,
NULL, char *envp[]);

передає додатково породженому процесу покажчик envp на нове оточення. У прикладі, що нижче приводиться, у процесі-батьку в оточення MS DOS додається рядок

OUT_FILE=result.txt

Який повідомляє процесу-нащадку ім'я файлу, у який потрібно записати результат підсумовування переданих йому аргументів.

```
/*  Неоверлейний виклик процесу-нащадка з процесу-батька
  функцією spawnle(). Тестувати після виходу з Turbo */
#include<process.h>
#include<stdio.h>
#include<dos.h>
#include<stdlib.h>
#include"mcb.c"
int main(void)
 {
  int i = 0, stat;
  printf("\nрозподіл ПАМ'ЯТІ ДО ЗАВАНТАЖЕННЯ"
         " ПРОЦЕСУ-НАЩАДКА\n");
  mcb();
  printf("\n СТАРЕ ОТОЧЕННЯ\n");
  while(environ[i]) printf("%s\n", environ[i++]);
  if(putenv("OUT_FILE=result.txt") == -1)
   {
    printf("Помилка putenv()\n");
    return 1;
   }
  printf("\nвиклик ПОРОДЖЕНОГО ПРОЦЕСУ\n");
```

```
  stat = spawnle(P_WAIT, "\\TC2\\CHILD1.EXE",
"\\TC2\\CHILD1.EXE",
                 "15", "-67", NULL, environ);
  if (stat == -1)
      {
        perror("Помилка при неоверлейному виклику");
        return 1;
      }
     else
      {
        printf("\n ЗВУК  ІЗ ПРОЦЕСУ-БАТЬКА...\n");
        for(i = 0; i <= 30000; i++)
          sound(1000);
        nosound();
        printf("\nрозподіл ПАМ'ЯТІ ПІСЛЯ ПОВЕРНЕННЯ"
               " У ПРОЦЕС-БАТЬКО\n");
        mcb();
       }
  return 0;
 }
```

Програма процесу-нащадка має вид:

```
#include<stdio.h>
#include<dos.h>
#include<stdlib.h>
#include"mcb.c"
int main(int argc, char *argv[])
{
 FILE* out;
 char* file_name;
 int sum, i = 0;
 if(_osmajor >= 3)
   printf("ПРОЦЕС - ПОТОМОК З ФАЙЛУ %s ЗАПУЩЕНИЙ\n", argv[0]);
 printf("\n НОВЕ ОТОЧЕННЯ\n");
 while(environ[i]) printf("%s\n", environ[i++]);
 sum = atoi(argv[1]) + atoi(argv[2]);
 file_name = getenv("OUT_FILE");
 if(file_name)
   {
     out = fopen(file_name, "wt");
     fprintf(out, "Сума %s і %s дорівнює %d\n",
             argv[1], argv[2], sum);
     fclose(out);
   }
   else printf("Сума %s і %s дорівнює %d\n", argv[1], argv[2],
sum);
 printf("\nрозподіл ПАМ'ЯТІ ПІСЛЯ ЗАВАНТАЖЕННЯ "
        "ПРОЦЕСУ-НАЩАДКА\n");
 mcb();
 return 0;
}
```

Функція spawnlpe(), що має синтаксис:

int spawnlpe(режим, ім'я_файлу, аргумент0, ..., аргумент,
NULL, char *envp[]);

поєднує особливості функцій spawnlp() і spawnle().

Функції spawnl, spawnlp, spawnle, spawnlpe використовуються, коли заздалегідь відоме число переданих нащадку аргументів. У противному випадку варто застосовувати функції spawnv, spawnvp, spawnve чи spawnvpe. Синтаксис функції spawnv наступний:

int spawnv(режим, ім'я_файлу, char *argv[]);

Вона передає процесу-нащадку аргументи у виді масиву покажчиків *argv[]. Суфікси -p, -e і -pe, що додаються до spawnv, видозмінюють роботу функцій також, як це відбувається для функцій spawnl, spawnlp, spawnle, spawnlpe. Синтаксис функції spawnvp:

int spawnvp(режим, ім'я_файлу, char *argv[]);

Синтаксис функції spawnve:

int spawnve(режим, ім'я_файлу, char *argv[], char *envp[]);

Синтаксис функції spawnvpe:

int spawnvpe(режим, ім'я_файлу, char *argv[], char *envp[]);

При неоверлейному запуску після повернення з процесу-нащадка у процес-батько функції сімейства spawn повертають код завершення програми-нащадка. Його можна використовувати для визначення того, як завершився процес-нащадок. Звичайно код завершення 0 свідчить про нормальне завершення; код же, відмінний від нуля, говорить про помилку. У прикладі, що нижче приводиться, у програмі нащадка перевіряється число переданих їй аргументів і, якщо воно недостатньо для обчислення суми argv[1] і argv[2], то процес-нащадок закінчується з кодом завершення 1. Після повернення у процес-батько ця ситуація виявляється, після чого виводиться повідомлення

******У нащадок було передано НЕДОСТАТНЄ ЧИСЛО АРГУМЕНТІВ******

```
/*  Неоверлейний виклик процесу-нащадка з процесу-батька
   за допомогою функції spawnl() з аналізом коду повернення.
   Тестувати після виходу з Turbo.     */
#include<process.h>
#include<stdio.h>
#include<dos.h>
#include"mcb.c"
int main(void)
 {
  int i, stat;
  printf("\nрозподіл ПАМ'ЯТІ ДО ЗАВАНТАЖЕННЯ"
         " ПРОЦЕСУ-НАЩАДКА\n");
  mcb();
  printf("\nвиклик ПОРОДЖЕНОГО ПРОЦЕСУ\n");
  stat = spawnl(P_WAIT, "\\TC2\\CHILD2.EXE",
"\\TC2\\CHILD2.EXE",
                "15", NULL);
  if (stat == -1)
      {
        perror("Помилка при неоверлейному виклику");
        return 1;
      }
     else
      {
        printf("\n ЗВУК  ІЗ ПРОЦЕСУ-БАТЬКА...\n");
        for(i = 0; i <= 30000; i++)
          sound(1000);
        nosound();
        if(stat == 1)printf("\n******У нащадок було передано "
           "НЕДОСТАТНЄ  ЧИСЛО  АРГУМЕНТІВ******\n");
```

```
        printf("\nрозподіл ПАМ'ЯТІ ПІСЛЯ ПОВЕРНЕННЯ"
               " У ПРОЦЕС-БАТЬКО\n");
        mcb();
       }
  return 0;
 }
```

Текст програми нащадка має вид:

```
#include<stdio.h>
#include<dos.h>
#include<stdlib.h>
#include"mcb.c"
int main(int argc, char *argv[])
{
 int sum;
 if(_osmajor >= 3)
   printf("ПРОЦЕС - ПОТОМОК З ФАЙЛУ %s ЗАПУЩЕНИЙ\n", argv[0]);
 if(argc < 3) return 1;
 sum = atoi(argv[1]) + atoi(argv[2]);
 printf("Сума %s і %s дорівнює %d\n", argv[1], argv[2], sum);
 printf("\nрозподіл ПАМ'ЯТІ ПІСЛЯ ЗАВАНТАЖЕННЯ "
        "ПРОЦЕСУ-НАЩАДКА\n");
 mcb();
 return 0;
 }
```

Для тестування даного приклада в аргументах функції spawnl спеціально залишене тільки один доданок "15" і опущений другий ("-67" у попередніх прикладах).

Для передачі коду завершення з процесу-нащадка в процес-батько при неоверлейному виклику можна використовувати також описані вище функції exit(), _exit(), abort(). Застосування оператора return чи функцій exit(), _exit(), abort() для виходу з процесу-нащадка при оверлейному виклику забезпечить передачу коду завершення в MS DOS.

В області даних BIOS мається 16-байтне поле, що починається з адреси 0040:00F0 (див. табл. 2), що може використовуватися для обміну даними між програмами. У наступному прикладі процес-нащадок записує результат підсумовування аргументів argv[1] і argv[2] у перші чотири байти цього поля у форматі float, а процес-батько бере їх відтіля і виводить на екран у виді числа з крапкою, що плаває:

```
/*  Неоверлейний виклик процесу-нащадка з процесу-батька
   за допомогою функції spawnl() з передачею результату.
   Тестувати після виходу з Turbo.      */
#include<process.h>
#include<stdio.h>
#include<dos.h>
#include"mcb.c"
int main(void)
 {
  int i, stat;
  printf("\nрозподіл ПАМ'ЯТІ ДО ЗАВАНТАЖЕННЯ"
         " ПРОЦЕСУ-НАЩАДКА\n");
  mcb();
  printf("\nвиклик ПОРОДЖЕНОГО ПРОЦЕСУ\n");
  stat = spawnl(P_WAIT, "\\TC2\\CHILD3.EXE",
"\\TC2\\CHILD3.EXE",
                "15.78", "-56", NULL);
  if (stat == -1)
```

```
      {
        perror("Помилка при неоверлейному виклику");
        return 1;
      }
     else
      {
        float far *result = (float far *)MK_FP(0x0040, 0x00F0);
        printf("\n ЗВУК  ІЗ ПРОЦЕСУ-БАТЬКА...\n");
        for(i = 0; i <= 30000; i++)
          sound(1000);
        nosound();
        printf("СУМА  15.78  і  -56  ДОРІВНЮЄ  %f\n", *result);
        printf("\nрозподіл ПАМ'ЯТІ ПІСЛЯ ПОВЕРНЕННЯ"
               " У ПРОЦЕС-БАТЬКО\n");
        mcb();
       }
  return 0;
 }
```

Текст програми-нащадка:

```
#include<stdio.h>
#include<dos.h>
#include<stdlib.h>
#include"mcb.c"
int main(int argc, char *argv[])
{
 float far *sum = (float far *)MK_FP(0x0040, 0x00F0);
 if(_osmajor >= 3)
   printf("ПРОЦЕС - ПОТОМОК З ФАЙЛУ %s ЗАПУЩЕНИЙ\n", argv[0]);
 *sum = (float)(atof(argv[1]) + atof(argv[2]));
 printf("\nрозподіл ПАМ'ЯТІ ПІСЛЯ ЗАВАНТАЖЕННЯ "
        "ПРОЦЕСУ-НАЩАДКА\n");
 mcb();
 return 0;
 }
```

Програма-нащадок виконує підсумовування чисел 15.78 і -56.00, переданих їй як аргументи з процесу-батька, попередньо перетворивши їх зі строкового представлення у формат числа з крапкою, що плаває, за допомогою функції atof(), що має синтаксис:

double atof(const char *s);

Прототип функції atof() знаходиться у файлі stdlib.h .

Оверлейний виклик процесу-нащадка поряд з функціями сімейства spawn у режимі P_OVERLAY можуть здійснювати також функції сімейства exec, що мають наступний синтаксис:

```
int execl(ім'я_файлу, аргумент0, ..., аргумент, NULL);
int execlp(ім'я_файлу, аргумент0, ..., аргумент, NULL);
int execle(ім'я_файлу, аргумент0, ..., аргумент, NULL,
           char *envp[]);
int execlpe(ім'я_файлу, аргумент0, ..., аргумент, NULL,
            char *envp[]);
int execv(ім'я_файлу, char *argv[]);
int execvp(ім'я_файлу, char *argv[]);
int execve(ім'я_файлу, char *argv[], char *envp[]);
int execvpe(ім'я_файлу, char *argv[], char *envp[]);
```

Зміст суфіксів -p, -e, -pe для цих функцій той же, що і для функцій сімейства spawn.

Якщо скомпілювати наступні три файли одержимо працюючий приклад керування процесами:

/* файл mcb.c*/

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

 union REGS reg;
 struct SREGS sregs;
 typedef struct
    {
      unsigned char marker;
      unsigned int owner;
      unsigned int sizePara;
      unsigned char dummy[3];
      unsigned char name[8];
    }mcb;
 mcb far *ptr;

void mcbfun(void)
 { unsigned int segm, i;
   reg.h.ah = 0x52;
   intdosx(&reg, &reg, &sregs);
   segm = peek(sregs.es, reg.x.bx - 2);
   printf("  Адреса MCB  | Тип |  Розмір блоку   |  Ім'я
власник\n"
      "             |     |                 |  пам'яті
(байт)\n");
   printf("-----------------------------------------------
\n");

   ptr = MK_FP(segm, 0);
   printf("  %04X:0000  |  %c  |    %8lu     |", segm,ptr-
>marker,
                     (long)ptr->sizePara*16);
   if(!ptr->owner){printf("Блок вільний");delay(100);}
   else
     if(( _osmajor >= 4) &&  (segm + 1) == ptr->owner)
      { for(i=0; i<=7; i++)
     printf("%c", ptr->name[i]); }
   delay(100);
   printf("\n");

   while( ptr->marker == 'M')
    { segm = segm + ptr->sizePara + 1;
      ptr = MK_FP(segm, 0);
   printf("  %04X:0000  |  %c  |    %8lu     |", segm,ptr-
>marker,
                      (long)ptr->sizePara * 16);
   if(!ptr->owner){printf("Блок вільний");delay(300);}
```

```
     else if((_osmajor >= 4) && ( segm + 1) == ptr->owner)
       { for(i=0; i<=7; i++) printf("%c", ptr->name[i]);}
     delay(300);printf("\n");
   }getch();
 }
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
/* нащадок FUNC.c*/
#include<stdio.h>
#include<dos.h>
#include<stdlib.h>
#include"mcb.c"

int main(int argc, char *argv[])
{
 int f,f1;
 if(_osmajor >= 3)
   printf("Процес - нащадок з файлу %s запущений\n", argv[0]);
 if(argc < 5) return 1;
 f = (atoi(argv[1]))/(atoi(argv[2]));
 f1 = f + ((atoi(argv[3]))*(atoi(argv[4])));
 printf("Дані: a = %d, b = %d, x = %d, y = %d\n",
      atoi(argv[1]),atoi(argv[2]),atoi(argv[3]),atoi(argv[4]));
 printf("Функція F = %d\n",f1);getch();
 printf("\nрозподіл пам'яті після завантаження процесу-
нащадка\n");
 mcbfun();
 return 0;
 }
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
/*  Неоверлейний виклик процесу-нащадка з процесу-батька
  за допомогою функції spawnl()*/
#include<process.h>
#include<stdio.h>
#include<dos.h>
#include"mcb.c"

int main(void)
 { int i, stat,a,b,x,y;
   printf("\nрозподіл пам'яті до завантаження процесу-
нащадка\n");
   mcbfun();
   printf("\nвиклик породженого процесу\n");
   stat = spawnlp(P_WAIT, "FUNC.EXE",
"FUNC.EXE","10","2","2","4",NULL);
   if (stat == -1)
     {    perror("Помилка при неоверлейному виклику");
     return 1;
     }
   else
     { if (stat == 1) printf("\nнедостатньо аргументів\n");
       printf("\nрозподіл пам'яті після повернення в процес-
батько\n");
       mcbfun();
```

```
    }
  return 0;
 }
```

6. Запуск команди MS DOS із програми

Функція С:

```
#include<stdlib.h>
   int system(const char *command)
```

дозволяє запустити неоверлейним способом із програми користувача будь-яку команду MS DOS чи командний файл. Функція викликає копію командного процесора COMMAND.COM і передає йому рядок *command з ім'ям команди чи командного файлу. Для пошуку файлу COMMAND.COM використовується перемінна оточення COMSPEC. Функція повертає 0 при успішному завершенні і -1 у випадку помилки. Приклад використання функції system():

```
// Виклик команди MS DOS із прикладної програми.
// Тестувати після виходу з компілятора
#include<stdio.h>
#include<stdlib.h>
int main(void)
 {
  printf("Запуск команди MS DOS\n");
  if(!system("dir")) return 0;
    else return 1;
 }
```

7. Програмування під Windows MS C++ (MS Developer Studio)

Програма внесення тексту в діалогове вікно і перенос цього тексту в іншу область вікна.

```
// Основна програма для MS C++
#include <windows.h>
#include "resource.h"
HINSTANCE ghInstance;    //Handle додатка
#define WM_INITDIALOGPARAMS   (WM_USER+1)  //Повідомлення
  // посилає діалоговому вікну при ініціалізації
#define AP_LINELEN 30    // Довжина рядка введення

LRESULT CALLBACK dialog0handle
  (   // Оброблювач подій діалогового вікна
             HWND hDlg,  // Handle вікна
             UINT uMsg,  // Повідомлення
             WPARAM wParam,   // 1-й параметр
             LPARAM lParam) { // 2-й параметр
   switch(uMsg)
    { case WM_INITDIALOG:
        if (lParam==WM_INITDIALOGPARAMS)
          { //Ініціалізація
       SendDlgItemMessage(hDlg,IDC_EDIT1,
            EM_SETLIMITTEXT,  // Уст. MAX довжину рядка, що
вводиться
          (WPARAM)AP_LINELEN,(LPARAM)0);
```

```
       SendDlgItemMessage(hDlg,IDCANCEL,BM_SETIMAGE,
             IMAGE_BITMAP     //Вуст. BITMAP на кнопку
           ,(LPARAM)LoadBitmap(ghInstance,
                               MAKEINTRESOURCE(RC_BITMAP0)));
          }
      case WM_COMMAND:
       switch(wParam)
        { case IDCANCEL:   // ESC  або закриваюча кнопка
            EndDialog(hDlg,FALSE);   // Завершити діалог
            return TRUE;
          case IDINSERT:   // Натиснута кнопка "перенести"
            char str[AP_LINELEN+1];
            GetDlgItemText(hDlg,IDC_EDIT1,(LPTSTR)&str,
                   AP_LINELEN+1);  // Узяти текст рядка введення
            SetDlgItemText(hDlg,IDC_STATIC1,
             (LPCTSTR)&str); // Уст. текст статичного об'єкта
            return TRUE;
        } // end of wParam switch
          break; // case WM_COMMAND break
    } // end of uMSG switch
      return FALSE;
  }

int WINAPI WinMain(    // Головна функція
           HINSTANCE hInstance,     // Handle додатка
           HINSTANCE hPrevInstance, // Handle попередн. прилож.
- NULL
           LPSTR lpszCmdLine,       // Командний рядок
          int nCmdShow)             // Як показувати головн.
вікно
  {   ghInstance=hInstance;
      DialogBoxParam(hInstance,MAKEINTRESOURCE(DIALOG0),
         NULL,(DLGPROC)dialog0handle,
         WM_INITDIALOGPARAMS); // Завантажити діал. вікно з
ресурсу
      return  0;
  }

// Resurce.h - згенерований MS Developer Studio -
{{NO_DEPENDENCIES}}
   // Used by prac.rc
#define DIALOG0                         102
#define RC_BITMAP0                      104
#define IDINSERT                        1000
#define IDC_EDIT1                       1001
#define IDC_STATIC1                     1002
   // Next default values for new objects
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        111
#define _APS_NEXT_COMMAND_VALUE         40001
#define _APS_NEXT_CONTROL_VALUE         1003
#define _APS_NEXT_SYMED_VALUE           101
```

```
#endif
#endif
```

7.1. Робота програми

Формується діалогове вікно з 4-мя елементами керування
STATIC - текст
EDIT - рядок уведення
BUTTON - кнопка з текстом "Перенести"
BUTTON - кнопка з графіч. зображенням - вихід із програми
При натисканні кнопки "Перенести" - текст, що раніше введений у рядок уведення EDIT - відображається в елементі STATIC.

7.2. Структура програми

Аналог функції main() при програмуванні під Windows є функція WinMain(). Вона одержує 4 параметри.

HINSTANCE hInstance - Handle додатка;
HINSTANCE hPrevInstance - Handle попередньої копії цього додатка - для Win3.x. (NULL);
LPSTR lpszCmdLine - покажчик на командний рядок;
int nCmdShow - вид головного вікна - SS_MAXIMIZED, SS_MINIMIZED, SS_NORMAL.

Після старту WinMain() - глобальної перемінної типу HINSTANCE привласнюється Handle додатки. Потім, виконується функція DialogBoxParam(), що завантажує інформацію про діалогове вікно з ресурсу, що знаходиться в файлі, що виконується, формується діалогове вікно і виповнюється.

Функція DialogBoxParam() має 5 параметрів.

hInstance - Handle додатки для завантаження ресурсу;
MAKEINTRESOURCE(DIALOG0) - ідентифікатор ресурсу;
NULL - Handle вікна - діалог не відноситься до жодного вікну;
(DLGPROC)dialog0handle - покажчик на функцію обробки подій вікна;
WM_INITDIALOGPARAMS - повідомлення при якому діалоговому вікну варто зробити власну ініціалізацію.

Функція DialogBoxParam() завершиться тільки тоді, коли буде закрите діалогове вікно - тому що це діалогове вікно не породжує нових вікон і задач.

Функція обробки подій діалогового вікна dialog0handle() має 4 параметри.

HWND hDlg - Handle діалогового вікна - в одному додатку можна відкрити кілька діалогових вікон;
UINT uMsg - 4 байти для опису основного стандартного повідомлення Windows;
WPARAM wParam - 16байт- 1-й параметр для додатк. повідомлення;
LPARAM lParam - 32байта- 2-й параметр для додатк. повідомлення.

Приклад: при натисканні на кнопку діалогов. вікна, генерується подія з uMsg = WM_COMMAND; і wParam установлюється рівним номеру кнопки, що задається на етапі створення ресурсу діалогового вікна - файл Resurce.h.

Тема № 7. ШИНИ ДАНИХ, КЕРУВАННЯ МИШЕЮ, ВІДЕОДОСТУП.

1. ШИНИ комп'ютера

Шини - це канали зв'язку, застосовувані для організації взаємодії між пристроями комп'ютера. Слоти, куди вставляються плати розширення - зовнішня частина, за допомогою яких здійснюються підключення до шин.

Ієрархія шин PC виражається в тім , що кожна більш повільна шина з'єднана з більш швидкою. Сучасні комп'ютерні системи включають кілька шин. Кожен системний пристрій з'єднано з якою-небудь шиною, причому деякі пристрої (набори мікросхем) виконують роль моста між шинами. Розходження між цими шинами в основному зв'язані з обсягом одночасно переданих даних (розрядністю) і швидкістю передачі (швидкодією). Кожна шина будується на основі спеціальних мікросхем, що підключаються до шини процесора. Звичайно ці мікросхеми використовуються і для керування шиною пам'яті. Існує три основних показники роботи шини. Це тактова частота, розрядність і швидкість передачі даних.

Робота будь-якого цифрового комп'ютера залежить від тактової частоти, що визначає кварцовий резонатор. Ця частота коливання і називається тактовою частотою. Усі зміни логічних сигналів у будь-якій мікросхемі комп'ютера відбуваються через визначені інтервали, які називаються тактами. Найменшою одиницею виміру часу для логічних пристроїв комп'ютера є такт чи по іншому - період тактової частоти. Якщо системна шина вашого комп'ютера працює на частоті в 100 Мгц, то значить вона може виконувати до 100 000 000 операцій у секунду.

Розрядність. Шина складається з декількох каналів для передачі електричних сигналів. Якщо говорять, що шина тридцятидвухрозрядна, то це означає, що для передачі даних виділено 32 канали, а додаткові канали призначені для передачі специфічної інформації. Швидкість передачі даних. Вона визначається по формулі:

тактова частота * розрядність = швидкість передачі даних

Зробимо розрахунок швидкості передачі даних для 64 розрядної системної шини, що працює на тактовій частоті в 100 Мгц.

100 * 64 = 6400 Мбит/сек

6400 / 8 = 800 Мбайт/сек

За роботою кожної шини стежать спеціально для цього призначені контролери набору системної логіки (чіпсет).

Шини, що присутні на материнській платі.

Основна - системна шина FSB (Front Side Bus). По цій шині передаються дані між процесором і оперативною пам'яттю, між процесором і іншими пристроями. Розрядність FSB дорівнює розрядності CPU. Якщо використовується 64 розрядний процесор і тактова частота системної шини 100 Мгц, то швидкість передачі даних буде дорівнювати 800 Мбайт/сек. Включення кеш -пам'яті другого рівня в процесор дозволило значно збільшити її швидкість. У сучасних процесорах кеш -пам'ять розташована безпосередньо в кристалі процесора, тобто працює з частотою процесора. У більш ранніх версіях кеш - пам'ять другого рівня знаходилася в окремій мікросхемі, інтегрованої в корпус процесора.

Шина пам'яті призначена для передачі інформації між процесором і основною пам'яттю системи. Ця шина з'єднана з набором мікросхем плати North Bridge чи мікросхемою Memory Controller Hub. У залежності від типу пам'яті, використовуваної набором мікросхем (і , отже, системною платою), шина пам'яті може працювати з різними швидкостями. Найкращий варіант, якщо робоча частота шини пам'яті буде збігатися зі швидкістю шини процесора. Розрядність шини пам'яті завжди дорівнює розрядності шини процесора.

ISA (Industrial Standard Architecture - промислова стандартна архітектура). Перша 8 розрядна шина ISA з'явилася в 1981 році, а в 1984 року з'явився її 16 розрядний варіант. 8 розрядна ISA застосовувалася в комп'ютерах класу XT і працювала на частоті рівної 4,77

Мгц, а 16 розрядна - в AT з частотою в 8,33 Мгц. На інтерфейс 8 розрядної ISA було виведено 8 каналів даних і 20 каналів адреси. Це дозволяло адресувати до 1 Мбайт пам'яті. З появою 80286 процесора, що міг обробляти 16 біт даних, з'явилася 16 розрядна ISA. Слот було доповнено ще 36 каналами, 8 з яких були виведені під дані, а 7 - під адресу. У 1985 році фірма Intel розробила 32 розрядний 80386 процесор. Замість ISA, у IBM створили нову шину MCA (Micro Channel Architecture - мікроканальна архітектура)

EISA (Extended Industry Standard Architecture - розширена промислова стандартна архітектура). Основна її відмінність полягала в 32 розрядної технології, хоча і створювалася вона на основі архітектури усе тієї ж ISA (тактова частота залишилася колишньої - 8,33 Мгц). Переваги нової технології: як і в MCA, використовується арбітраж запитів ISP (Integrated System Peripheral), підвищилася швидкість обміну даними, потужність споживаєма адаптерами. При цьому була збережена сумісність із платами, розрахованими для роботи з ISA. Швидкість передачі даних дорівнювала 33 Мбайт/сек. Підтримує Bus Mastering - режим керування шиною з боку будь-якого з пристроїв на шині, дозволяє автоматично встановити параметри пристроїв, можливий поділ каналів IRQ і DMA

З підвищенням тактових частот і розрядності процесорів настала проблема в підвищенні швидкості передачі даних у шинах. Одні пристрої могли працювати зі швидкістю, що шини не могли надати, клавіатура - навпаки. Рішення було прийнято таке: операцій по обміну даними здійснювати не через стандартні слоти шини вводу/виводу, а через додаткові високошвидкісні інтерфейси. Ці високошвидкісні інтерфейси підключаються до шини процесора. Плати, що підключаються, будуть мати доступ безпосередньо до процесора через його шину. Це одержало назву LB (Local Bus - локальна шина). Ще в 1992 році з'явився розширений варіант ISA - VLB (VESA Local Bus). VESA(Video Electronic Standard Association) була локальною шиною, що доповнювала існуючі стандарти. При цьому до основних шин додавались нові швидкодіючі локальні слоти. Швидкість передачі даних VLB 128 - 132 Мбайт/сек, а розрядність -32. Тактова частота до 50 Мгц, Основне в новій шині - обмін даними з відеоадаптером.

Шина PCI. Ця 32-розрядна шина використовується починаючи із систем на базі процесорів 486. Знаходиться під керуванням контролера PCI - частини North Bridge чи Memory Controller Hub (MCH) набору мікросхем . На системній платі встановлюються слот , звичайно чотири чи більш, у які можна підключати мережні, SCSI- і відеоадаптери, а також інше устаткування , що підтримує цей інтерфейс. До шини PCI підключається South Bridge, що містить реалізації інтерфейсу IDE і USB. Творці PCI відмовилися від традиційної концепції, увівши ще одну шину між процесором і звичайною шиною вводу - виводу. Замість того щоб підключити її безпосередньо до шини процесора, вони розробили новий комплект мікросхем контролерів для розширення шини. PCI додає до традиційної конфігурації шин ще один рівень . При цьому звичайна шина вводу -виводу не використовується, а створюється фактично ще одна високошвидкісна системна шина з розрядністю, рівній розрядності даних процесора. Одне з основних переваг шини PCI полягає в тому, що вона може функціонувати одночасно із шиною процесора. Це дозволяє процесору обробляти дані зовнішньої кеш -пам'яті одночасно з передачею інформації із шини PCI між іншими компонентами системи. Для підключення адаптерів шини PCI використовується спеціальний слот. Іншою важливою властивістю плати PCI є те, що вона задовольняє специфікації Plug and Play компанії Intel. Це означає, що PCI може за допомогою спеціальної програми настроюватись. Системи з Plug and Play здатні самостійно приєднувати адаптери.

AGP (Accelerated Graphics Port - прискорений графічний порт). Для підвищення ефективності роботи з відео і графікою Intel розробила нову шину - прискорений графічний порт. AGP схожа на PCI, але містить ряд додавань і розширень. Фізично, електрично і логічно вона не залежить від PCI. На відміну від PCI, що є дійсною шиною з декількома слотами, AGP - з'єднання, розроблене спеціально для відео адаптера , причому

у системі для одного відеоадаптера допускається тільки один слот AGP. Оскільки шина AGP незалежна від PCI, при використанні відеоадаптера AGP можна звільнити шину PCI для виконання традиційних функцій вводу-виводу, наприклад для контролерів IDE/ATA, SCSI чи USB, звукових плат і ін. Крім підвищення ефективності роботи відеоадаптера , AGP дозволяє одержувати швидкий доступ безпосередньо до оперативної пам'яті. Завдяки цьому відеоадаптер AGP може використовувати оперативну пам'ять, що зменшує потребу у відеопам'яті. На материнській платі цей порт один. Перший специфікації відповідала тактова частота 66,66 Мгц. Швидкість передачі даних - 533 Мбайт/сек (2х) і 1066 Мбайт/сек (4х)і 8х= 2034,4 Мб/сек. Основний режим AGP називається 1х. У цьому режимі відбувається одиночна передача даних за кожен цикл. У режимі 2х передача відбувається два рази за цикл, 4х - чотири рази за кожен цикл. Ширина AGP 1.0 - 32 біта.

PCMCIA (Personal Computer Memory Card International Association- асоціація виробників плат пам'яті для персональних комп'ютерів) - зовнішня шина комп'ютерів класу NoteBook, розрядність 16/26 (адресний простір - 64 Мб), підтримує автоконфігурацию, можливі підключення і відключення пристроїв у процесі роботи комп'ютера. Конструктив - мініатюрний 68-контактний роз’єм. Контакти живлення зроблені більш довгими, що дозволяє вставляти і виймати карту при включеному живленні комп'ютера.

Ultra-Ata - режим передачі даних з появи стандарту ATA/ATAPI-4 для підключення пристроїв, читання/запису інформації, такі як HDD, CD-ROM, CD-RW, DVD-ROM, DVD-RW. Звичайно маркірується Ultra-Ata/xxx, де xxx швидкість передачі даних за сек.

· Ultra-Ata/33 = 33 Мб/сек
· Ultra-Ata/66 = 66 Мб/сек
· Ultra-Ata/100 = 100Мб/сек
· Ultra-Ata/133 = 133 Мб/сек

Також деякі виробники маркірують Ultra DMA xxx де ті ж маркірування, Ultra DMA 66 і так далі.

SCSI - Small Computer Systems Interface чи просто говорячи "skuzzy"

- прямий конкурент
- стандарт ATA із усіма його режимами Ultra-Ata.

Почав своє існування в 1979 році, на даний момент є багато стандартів SCSI, але 3 покоління, нумерація в порядку еволюції :)

SCSI-1-е покоління:

· SCSI-1 8-ми бітна шина 5Мгц (1986), максимум 5 Мб/сек SCSI-2-е покоління;
· SCSI Fast 8-ми бітна шина 10Мгц (1990), максимум 10 Мб/сек. Той же варіант, але SCSI Fast Wide - 16-ти бітна шина 10Мгц шина, але вже 20 Мб/сек;
· SCSI Fast-20 чи SCSI Ultra - 8-ми бітна шина 20Мгц (1994), максимум 20Мб/сек, той же варіант але SCSI Ultra Wide 16-ти бітна шина 40Мгц і 40Мб/сек;
· SCSI Fast-40 чи SCSI Ultra2 - 8-ми бітна шина 40Мгц, максимум 40Мб/сек той же варіант, але SCSI Ultra2 Wide 16-ти бітна шина 80Мгц і 80Мб/сек SCSI-3-е покоління;
· SCSI Fast-80 чи SCSI Ultra3 чи SCSI Ultra 160 - 16-ти бітна шина 40Мгц, максимум 160Мб/сек;
· SCSI Fast-160 чи SCSI Ultra 320 - 16-ти бітна шина 80Мгц, максимум 320Мб/сек

Serial ATA - еволюційне продовження Ultra-Ata, швидкість = 150 Мб/сек до 600 Мб/сек.

Parallel Port Interface - LPT порт, споконвічно створений тільки для принтерів, але надалі став більш універсальним стандартом. Підтримує наступні режими передачі даних:

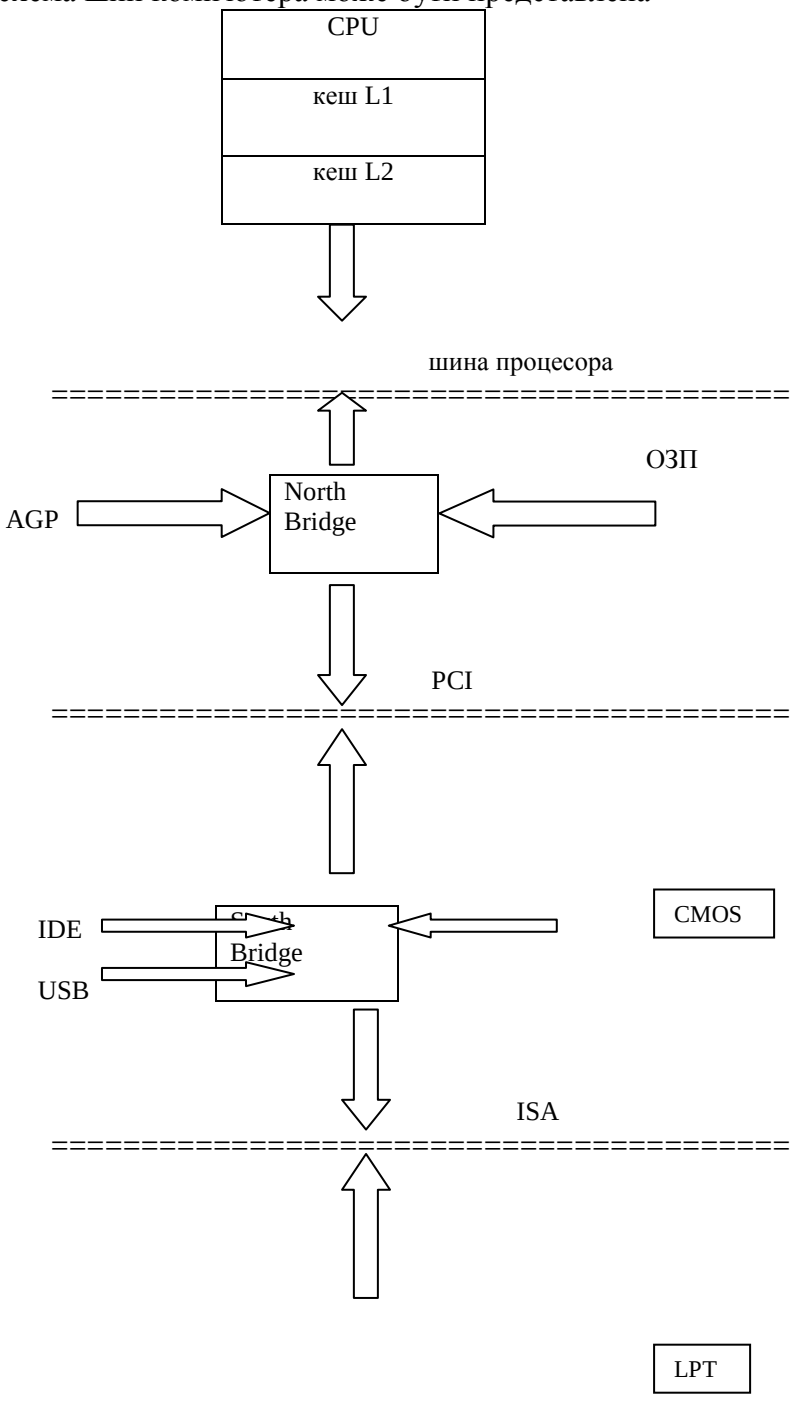
· SPP (Simple Parallel Port) = максимум 0,15 Мб/сек
· Bi-Directional = максимум 0,30 Мб/сек
· ECP чи EPP = максимум від 1 до 3 МБ/сек

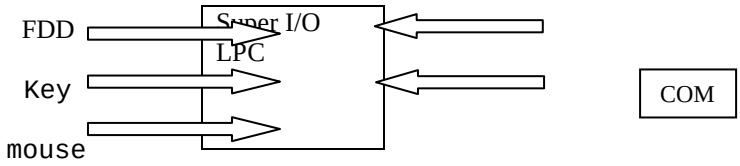
Serial Port - серійний порт, звичайно Com1 і далі. Універсальний стандарт, для підключення різного роду пристроїв максимум 0,11 Мб/сек. Через обмеження у швидкості часто застосуємий для модемів, Isdn пристроїв і старих мишок.

Universal Serial Bus - універсальна послідовна магістраль, засіб для підключення різних зовнішніх пристроїв. Передбачає підключення до 255 зовнішніх пристроїв до одного USB-порту (за принципом загальної шини), була розроблена на початку 1990-их, має два стандарти:

· USB 1.1 = максимум 1,5 Мб/сек
· USB 2.0 = максимум 60 Мб/сек

IEEE-1394 - чи FireWire чи у варіації Sony - i.Link вийшов у світ у 1995, частина розробок SCSI 3 іноді називається Serial SCSI3 (аналог USB), максимальна швидкість = 40 Мб/сек.
Графічно схема шин комп'ютера може бути представлена





У сучасних PC використовується hub-архітектура Intel, замість традиційної North/South Bridge. У цій конструкції основне з'єднання між компонентами набору мікросхем було перенесено у виділений hub-інтерфейс, що дозволило пристроям PCI використовувати повну, без обліку компонента South Bridge, пропускну здатність шини PCI. Крім того, мікросхема Flash ROM BIOS, тепер названа Firmware Hub, з'єднується із системою через шину LPC. У більшості систем для з'єднання мікросхеми Super I/O замість шини ISA тепер використовується шина LPC.

При цьому hub-архітектура дозволяє відмовитися від використання Super I/O. У такій системі пристрої, що використовують стандартні порти, повинні бути приєднані до комп'ютера за допомогою USB. У цих системах звичайно використовується два контролери і до чотирьох загальних портів (додаткові порти можуть бути підключені до вузлів USB).

Визначення типу шин передачі даних виробляється через порти вводу-виводу CONFADD - адреса 0CF8h, CONFDATA - адреса 0CFCh

Для визначення типу шин передачі даних, потрібно:

1. Послати в CONFADD, через регістр EAX, послідовність байтів: Ознака=80h, Номер шини, Номерпристрою*8, Функція=0.
2. Прочитати з CONFDATA значення кодів фірми-виготовлювача і пристрою - це два слова підряд у регістрі EAX.
3. Послати в CONFADD, через регістр EAX, послідовність байтів: Ознака=80h, Номер шини, Номерпристрою*8, Функція=8.
4. Прочитати з CONFDATA значення кодів класу і підкласу для пристрою - це два байти підряд у регістрі AX.

У програмі потрібно організувати перебір номерів шини - від 0 до 255 і номерів пристрою - від 0 до 32.

У циклі виробляється опитування портів CONFADD і CONFDATA і визначаються коди виробника і пристрою. Якщо отриманий код FFFFh, те значить заданого типу шини немає в комп'ютері.

За значеннями КОДІВ ФІРМИ - ВИРОБНИКА і ПРИСТРОЮ, які знайдені з описаного раніше алгоритму, можна визначити повний текст назви виробника і пристрою - для цього є спеціальний файл-довідник.

Фрагмент довідника - відповідності кодів і назв фірм і пристроїв:

[1023h] = Trident
0194h=82C194 PCI to PCMCIA Controller
[104Ch] = Texas Instruments
0508h=TMS380C2X Compressor Interface
[11F6h] = Compex
0112h=ReadyLink ENET100-VG4 Ethernet Adapter
[5333h] = S3
5631h=325 ViRGE GUI Accelerator
[8086h] = Intel
0122h=82437FX Triton FX Chipset, Pentium to PCI Bridge

По кодах класу визначаються типи пристрою:

1 - SCSI Mass Storage Controller;
2 - Network Card;
3 - VGA Display Adapter;
6 - це Міст;

Розбір типів Мостів по коду підкласу:

0 - Host Processor Bridge;

1 - PCI to ISA Bridge;

4 - PCI to PCI Bridge.

2. Визначення типу "миші"

"Миша" - це особливого роду маніпулятор, що дозволяє оптимізувати роботу з великою категорією комп'ютерних програм за рахунок виключення частих повторних натискань деяких клавіш (наприклад, клавіш переміщення курсору). "Миша" може приєднуватися або до послідовного порту (контролера вводу/виводу), або мати свій власний інтерфейс і контролер, установлюваний безпосередньо на шину. Останній варіант одержав поширення в комп'ютерах сімейства PS/2. Найбільше розповсюдженими видами стандартів програмного інтерфейсу "миші" є:

- миша Microsoft Mouse має дві кнопки керування, програмно встановлювані у відповідності натисканню клавіш клавіатури <Esc> і <Enter>;

- миша Mouse System має три кнопки керування (третя кнопка дублює першу).

Як правило, моделі "миші" інших виробників підтримують той чи інший стандарт, а іноді

- обидва ці стандарти.

Для того, щоб використовувати "мишу", насамперед необхідно інсталювати відповідний драйвер. Для "миші" фірми Microsoft у файл CONFIG.SYS повинна бути додана директива:

DEVICE=MOUSE.SYS

Для інсталяції драйвера "миші" фірми IBM повинна бути запущена програма MOUSE.COM. З цією метою у файл AUTOEXEC.BAT може бути доданий рядок виду:

MOUSE

Після того, як драйвер розміщений у системі, будь-які дії по переміщенню "миші" чи натисканню її клавіші будуть викликати генерацію переривання 33h.

Функція 00h переривання 33h забезпечує рестарт драйвера миші. Після виходу з переривання в регістрі AX повідомляється стан "миші" і її драйвера:

0000h - миша чи драйвер не встановлені;

FFFFh - рестарт здійснений, драйвер і "миша" установлені; а в регістрі BX приводиться кількість кнопок у миші:

0000h - відмінне від двох;

0002h - дві кнопки;

0003h - три кнопки.

Функція 24h переривання 33h виконує читання інформації про версії програмного забезпечення, типи "миші" і номера лінії запиту на переривання IRQ, що використовує контролер "миші". Зміст регістрів після виходу з переривання:

AX - якщо FFFFh, то відбулася помилка читання інформації про "миші";

BH - старші розряди номера версії;

BL - молодші розряди номера версії;

CH - тип інтерфейсу "миші":

01h - "миша", що підключається до шини;

02h - "миша", що підключається до послідовного порту;

03h - Microsoft InPort;

04h - порт координатно-вказівного пристрою комп'ютерів серії IBM PS/2;

05h -"миша" компанії Hewlett-Packard;

CL - номер лінії запиту на переривання IRQ:

00h - координатно-вказівний пристрій комп'ютерів серії PS/2;

01h - не визначено;
02h - IRQ2;
03h - IRQ3;
.....
07h - IRQ7.

Наступний приклад ілюструє застосування функцій 00h і 24h переривання 33h:

```
/* Одержання інформації про мишу  */
#include<stdio.h>
#include<dos.h>
#include<process.h>

int main(void)
 {
   union REGS reg;
   reg.x.ax = 0;  /* Ініціалізація миші */
   int86( 0x33, &reg, &reg);
   if(reg.x.ax == 0)
    { printf("Миша чи її драйвер не встановлені\n");
      return 0;
    }
   printf("Миша з %u кнопками\n", reg.x.bx);
   reg.x.ax = 0x24;
   int86(0x33, &reg, &reg); /*  Читання інформації про мишу */
   if(reg.x.ax == 0xFFFF)
    { printf("Помилка читання інформації про мишу\n");
      return 0;
    }
   printf("ВЕРСІЯ МИШІ: %2u. %2u\n", reg.h.bh , reg.h.bl);
   printf("ІНТЕРФЕЙС: ");
   switch(reg.h.ch)
     { case 1: printf("Миша підключена до шини\n"); break;
       case 2: printf("Миша підключена до послідовного
порту\n");
               break;
       case 3: printf("Microsoft InPort\n"); break;
       case 4: printf("Порт координатно-вказівного пристрою"
                      " IBM PS/2\n"); break;
       case 5: printf("Миша компанії Hewlett-Packard\n"); break;
       default: printf("Невідомий тип\n");
     }
    printf("НОМЕР ЛІНІЇ ЗАПИТУ НА ПЕРЕРИВАННЯ: ");
    switch(reg.h.cl)
     { case 0: printf("координатно-вказівне в PS/2\n");
               break;
       case 2: case 3: case 4: case 5: case 6:
       case 7: printf("IRQ%d\n",reg.h.cl); break;
       default: printf("не визначений\n");
      }
    return 0;
  }
```

1.2. Програмування миші.

Для текстового режиму можна запропонувати спеціальний інструментарій роботи з мишею. Для цього використовуються функції переривання ЗЗН, що викликаються за допомогою функції int86(). Ця функція описана в файлі - заголовку dos.h. У цьому ж файлі знаходиться опис об'єднання структур із змістом регістрів мікропроцесора в - REGS.

```
#include <dos.h>
union REGS rg;
```

Ініціалізація миші проводиться з визначенням кількості використовуваних кнопок:

```
int ms_init(int *nbottoms)
  { rg.x.ax = 0;
    int86(0x33,&rg,&rg);
    *nbottoms = rg.x.bx;
    return rg.x.ax;
}
```

Для упорядкування контролю за маніпулюванням мишею уводиться спеціальна структура опису стану миші:

```
typedef struct _MOUSE_STATE_
  {  unsigned int bottoms;
     unsigned int x;
     unsigned int y;
  } MOUSE_STATE;
```

Скидання миші реалізується окремо:

```
void  ms_esc(void)
{   rg.x.ax = 0x0021;
      int86(0x33,&rg,&rg);
}
```

Для того щоб показати курсор миші на екрані або увести його з екрана також уводяться спеціальні функції:

```
void ms_on(void)
  {  rg.x.ax = 1;
     int86(0x33,&rg,&rg);
  }
void ms_off(void)
  { rg.x.ax = 2;
    int86(0x33,&rg,&rg);
  }
```

При натисканні на кожну з кнопок миші проводиться визначення стану миші - обчислюються координати курсору миші і номер активної кнопки:

```
int ms_querp(MOUSE_STATE *state, int bottom)
 {  rg.x.ax = 0x0003;
    rg.x.bx = bottom;
    int86(0x33,&rg,&rg);
   state->bottoms = rg.x.bx&0x0003;
   state->x       = rg.x.cx/8;
   state->y       = rg.x.dx/8;
  return(state->bottoms);
 }
```

Для текстового режиму дисплея можна злегка видозмінити курсор миші за допомогою стандартної функції переривання 33H.

```
void ms_tform(void)
{ rg.x.ax = 0x;
  rg.x.bx = 0;  /* bx=1; - миготливий курсор */
  rg.x.cx = 0xf000;
  rg.x.dx = 0x00db;
      int86(0x33,&rg,&rg);
}
```

Курсор миші можна установити в задану крапку екрана:

```
void ms_setcr(int x, int y)
  { rg.x.ax = 4;
    rg.x.cx = x*8;
    rg.x.dx = y*8;
      int86(0x33,&rg,&rg);
  }
```

У такий спосіб можна запропонувати фрагмент програми, що забезпечує контроль за положенням курсору миші:

```
  ...
 int botm, x, y;
  if(!ms_init(&botm))
      { printf("Mouse driver not exist"); return; }
  ms_esc();
  ms_setcr(2,2);
  ms_on();
  for(;;)
   { if (ms_querp(st,botm)&2)
       { clrscr();
         ms_setcr(0,0);
       }   /* Button -2 */
     if (ms_querp(st,botm)&1)
       { x=st->x;  y=st->y;
         if(x==0 && y==0) break;  /* Exit */
         gotoxy(x,y);
         cprintf("\n  Position- %u,%u,%u",x,y,st->bottoms);
       }
   }
  ms_off();
  ...
```

У графічному режимі робота з мишею організується подібним способом - з попередньою активізацією графічного режиму дисплея:

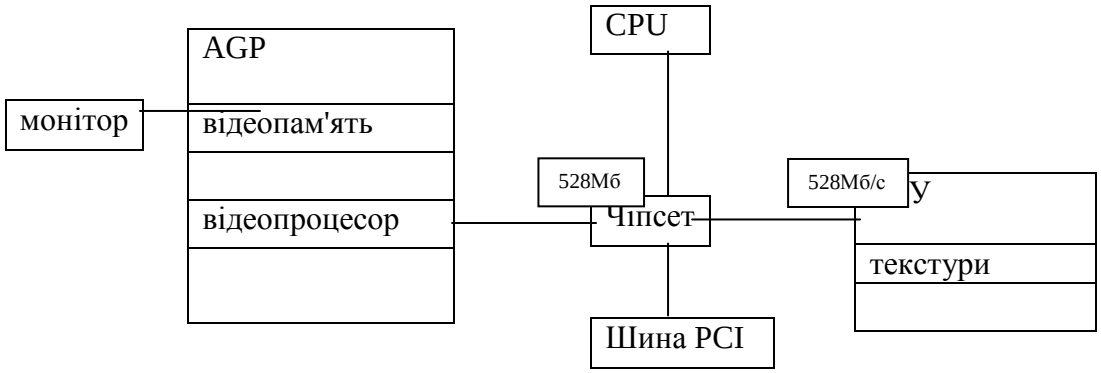
```
  ...
int *gr = DETECT, *gh;
char cc[20];
   initgraph(&gr,&gh,"");
   setbkcolor(BLACK);
   setcolor(GREEN);
 if(!ms_init(&botm))
      ...
         if(x==0 && y==0) break;  /* Exit */
         sprintf(cc,"Pos- %u,%u,%u",x*8,y*8,st->bottoms);
```

```
        outtextxy(x*8,y*8,cc);
} } ms_off(); }
 ...
```

3. ПРОГРАМУВАННЯ ВІДЕОАДАПТЕРА

3.1. Типи відеосистем комп'ютерів IBM PC

Відеосистема персонального комп'ютера, призначена для формування зображень, містить дві складові частини: відеоадаптер і монітор. Основу відеоадаптера складають спеціалізовані схеми для генерування електричних сигналів, керуючих монітором. Усі відеосистеми містять електронні компоненти, що формують сигнали синхронізації, кольоровості і керуючі генеруванням текстових символів. Крім того, у всіх відеосистемах мається відеобуфер. Він являє собою область оперативної пам'яті, що призначена тільки для збереження тексту чи графічної інформації, виведеної на екран монітора.



Найбільш вивченими типами відеоадаптерів є - адаптер VGA і адаптер SuperVGA.

3.2. Визначення типу активного відеоадаптера

Для ідентифікації типу відеоадаптера і типу дисплея служить підфункція 00h функції 1Ah переривання 10h. Якщо після повернення з переривання в регістрі AL записаний код 1Ah, то зазначена функція підтримується на даному комп'ютері. У цьому випадку в регістрі BL зазначено код активного дисплея, а в регістрі BH -код альтернативного дисплея. Використовується наступне кодування:

00h - дисплей відсутній;
01h - монохромний адаптер / монохромний дисплей;
02h - CGA-адаптер / кольоровий дисплей;
03h - зарезервовано;
04h - EGA-адаптер / кольоровий дисплей;
05h - EGA-адаптер / монохромний дисплей;
06h - PGC-адаптер / кольоровий дисплей;
07h - VGA-адаптер / монохромний аналоговий дисплей;
08h - VGA-адаптер / кольоровий аналоговий дисплей;
09h - зарезервовано;
0Ah - зарезервовано;
0Bh - MCGA-адаптер / монохромний аналоговий дисплей;

0Ch - MCGA-адаптер / кольоровий аналоговий дисплей;
0Dh - Feh - зарезервовані;
FFh - не визначений тип адаптера і дисплея.

Інший спосіб визначення моделі відеоадаптера полягає в використанні убудованих тестів графічної бібліотеки C.

Функція

void far detectgraph(int far *graph_adapter, int far *gmode)

при нульовому аргументі *graph_adapter визначає тип графічного адаптера і повертає його код цьому ж параметру *graph_adapter. При цьому використовується наступне кодування:

0 - "Потрібне визначення",
1 - "CGA",
2 - "MCGA",
3 - "EGA",
4 - "EGA з 64Кбайт пам'яті",
5 - "монохромний EGA",
6 - "IBM 8514",
7 - "Hercules монохромний",
8 - "AT&T 6300 PC",
9 - "VGA",
10 - "IBM 3270" .

Якщо графічний адаптер не виявлений, *graph_adapter встановлюється в -2. При визначенні типу графічного адаптера у перемінної *gmode повертається код найбільшого дозволу, припустимого для даного адаптера. Кодування дозволу записане в файлі graphics.h, де знаходиться прототип функції detectgraph.

Скористаємося двома приведеними способами для визначення типу відеоадаптера і дисплея на даному комп'ютері - EGA чи VGA (SVGA):

```
/*  Ідентифікація відеоадаптера  */
#include<dos.h>
#include<bios.h>
#include<stdio.h>
#include<graphics.h>
char *gname[] = { "Потрібно визначення",
                  " "," ",
                  "EGA",
                  " ",
                  "монохромний EGA",
                  " "," "," ",
                  "VGA чи SVGA",
                  " " };
void metod2(void)
    {
           int graph_adapter = 0, gmode;
           detectgraph(&graph_adapter, &gmode);
           if(graph_adapter != -2)
             printf("Відеоадаптер %s\n", gname[graph_adapter]);
            else
             {
              union
              {
               int status;
```

```
                struct
                 {
                  unsigned          : 4;
                  unsigned video    : 2;
                  unsigned          :10;
                 }fields;
               }ob;
               ob.status = biosequip();
               if(ob.fields.video == 3)
                 printf("Монохромний відеоадаптер\n");
                else printf("Невідомий тип відеоадаптера\n");
              }
    }
int main(void)
 {
  union REGS r;
  r.h.ah = 0x1A;
  r.h.al = 0;
  int86(0x10, &r, &r);
  if(r.h.al == 0x1A)
   switch(r.h.bl)
    {
     case 0x00: printf("Дисплей відсутній\n"); break;
     case 0x01: printf("Монохромний адаптер - монохромний "
                       "дисплей\n"); break;
     case 0x04: printf("EGA-адаптер - кольоровий дисплей\n");
break;
     case 0x05: printf("EGA-адаптер - монохромний дисплей\n");
                break;
     case 0x07: printf("VGA-адаптер - монохромний аналоговий "
                      "дисплей\n"); break;
     case 0x08: printf("VGA- чи SVGA- адаптер - кольоровий
аналоговий "
                      "дисплей \n"); break;
     default:   metod2();
    }
   else  /* Функція 1Ah переривання 10h не підтримується */
     metod2();
  return 0;
 }
```

Примітка :

Дана програма визначає тільки найпоширеніші типи адаптерів EGA і VGA(SVGA).

3.3. Метод визначення типу відеоадаптера для VESA - інтерфейсу

Задача ідентифікації типу VESA-відеоадаптера реалізується на основі аналізу результатів роботи функцій переривання 10h. Ефективний алгоритм визначення характеристик відеоадаптера заснований на заповненні полів спеціальної структури:

```
  struct  vesa
  { char sign[4];           //Сигнатура "VESA"
    unsigned char VBEminor,
                  VBEmajor; //Версія Vesa Bios Extend - VBE
```

```
    void far *OEMname;      //Покажчик на рядок з виробником
    int  cf1:1,  cf2:1,  cf3:1, reserved0:5;
    char  reserved1[3];
    void far *vmode;  //Доступні відеорежими - кінець 0xFFFF
                      //потім - додатк.відеорежими, кінець -
0xFFFF
    unsigned int  Vmem_amount;  //відеопам'ять у блоках по 64K
    unsigned char Sminor,
                  Smajor; //Додатк. N Версії  VBE - ст.мол
    void far *vendor;     //Покажчик на рядок з фірмою-
розроблювач
    void far *product;    // Покажчик на рядок з назв.
                             відеоадаптера
    void far *revision;   // Покажчик на додатк. N версії
                             відеоадаптера
    char reserved2[478];
  }  V;
```

Одержання інформації про VBE засновано на запису початкового адресу структури vesa у регістри ES- сегмент адреси; DI - зсув адреси. Викликається функція 4F00h переривання 10h - у результаті структура vesa заповнюється даними. Доступні відеорежими витягаються зі списку - у циклі, до появи коду FFFFh.

Одержання інформації про Present Modes засновано на виклику підфункції 0, функції 4F10h переривання 10h - у результаті регістр BX повертає коди підтримуваних режимів:. 0100h- Standby; 0200h- Suspend; 0400h- Off; 0800h- Reduced.

```
printf("* Тип відеоадаптера :\n");
getadaptertype();
printf("  - VESA розширення :\n");
  struct  vesa
  { char sign[4];           //Сигнатура "VESA"
    unsigned char VBEminor,
            VBEmajor; //Версія Vesa Bios Extend - VBE
    void far *OEMname;      //Указат.на рядок з виробником
    int  cf1:1,  cf2:1,  cf3:1, reserved0:5;
    char  reserved1[3];
    void far *vmode;  //Доступні відеорежими - кінець 0xFFFF
               //потім - додатк.відеорежими, кінець - 0xFFFF
    unsigned int  Vmem_amount;  //відеопам'ять у блоках по 64K
    unsigned char Sminor,
            Smajor; //Додатк. N Версії  VBE - ст.мол
    void far *vendor;     //Покажчик на рядок з фірмою-
розроблювачем
    void far *product;    // Покажчик на рядок з назв.
                             відеоадаптера
    void far *revision;   // Покажчик на додатк. N версії
                             відеоадаптера
    char reserved2[478];
  }  V;

strcpy(V.sign, "VBE2");
sregs.es=FP_SEG(&V);
regs.x.di=FP_OFF(&V);
regs.x.ax=0x4f00;
```

```
int86x(0x10,&regs,&regs,&sregs);
if (regs.h.al!=0x4f) printf (" не підтримується.");
else
 {
  if (regs.h.ah==1) printf (" Помилка при запиті інформації!");
  printf ("Сигнатура : ");
  for (i=0;i<4;i++)
   printf ("%c",V.sign[i]);
  printf ("\nверсія Vesa Bios Extend - VBE : %d. %d
\n",V.VBEmajor,V.VBEminor);
  printf ("Рядок виробника : ");
  if (V.OEMname)printf ("%s\n",V.OEMname);
     else  printf ("відсутній.\n");
  printf ("Кількість відеопам'яті : %uкб\n",V.Vmem_amount*64);
  printf ("Додатк. N Версії  VBE  : %u.
%u\n",V.Smajor,V.Sminor);
  printf ("Рядок з фірми-розроблювача : ");
  if (V.vendor) printf ("%s\n",V.vendor);
     else  printf ("відсутній.\n");
  printf ("Рядок з назв. відеоадаптера : ");
  if (V.product) printf ("%s\n",V.product);
     else  printf ("відсутній.\n");
  printf ("Додатк. N версії відеоадаптера : ");
  if (V.revision) printf ("%s\n",V.revision);
     else  printf ("відсутній.\n");
 }
/*
```

3.4. Установка режиму роботи дисплея

Усі відеосистеми персональних комп'ютерів, за винятком адаптера MDA, можуть працювати в двох основних режимах - текстовому і графічному. У текстовому режимі екран розділяється на окремі символьні позиції, у кожній з яких виводиться один символ. У відеобуфері кожної символьної позиції відповідає два байти.

Байт із парною адресою містить код символу, а сусідній байт із більшою непарною адресою містить атрибути, що визначають, як виводиться символ. Символи зберігаються у відеобуфері у виді лінійної послідовності, що відображаються на прямокутні координати (стовпець, рядок) екрана.

У двох найбільш розповсюджених форматах текстового режиму на екрані формується 25 текстових рядків, що містять по 40 чи 80 символів. Таким чином, ємність відеобуфера, необхідна для заповнення екрана, складає 2000 байт (25 рядків * 40 символів * 2 байти) чи 4000 байт (25 рядків * 80 символів * 2 байти). Цю область називають сторінкою. Якщо ємність відеобуфера перевищує розмір однієї сторінки, у ньому можна організувати кілька сторінок.

Адаптер MDA містить відеобуфер ємністю 4 Кбайт із початковою адресою B000:0000. Цієї пам'яті вистачає тільки для збереження однієї 80-символьної сторінки тексту.

Стартова адреса відеобуфера для текстових режимів адаптерів EGA, MCGA, VGA, SuperVGA і з графічним співпроцесором однакова - B800:0000.

У табл. 9.1 приведений перелік стандартних текстових режимів, підтримуваних BIOS IBM PC [13].

Таблиця 9.1

Режим	Адаптер	Число	Матриця	Формат	Число

Номер	Тип		пікселів	символу	тексту	кольорів
0	Текст	CGA	320x200	8x8	40x25	16 (сірий)
		EGA	320x350	8x14	40x25	16 (сірий)
		MCGA	320x400	8x16	40x25	16
		VGA	360x400	9x16	40x25	16
1	Текст	CGA	320x200	8x8	40x25	16
		EGA	320x350	8x14	40x25	16
		MCGA	320x400	8x16	40x25	16
		VGA	360x400	9x16	40x25	16
2	Текст	CGA	640x200	8x8	80x25	16 (сірий)
		EGA	640x350	8x14	80x25	16 (сірий)
		MCGA	640x400	8x16	80x25	16
		VGA	720x400	9x16	80x25	16
3	Текст	CGA	640x200	8x8	80x25	16
		EGA	640x350	8x14	80x25	16
		MCGA	640x400	8x16	80x25	16
		VGA	720x400	9x16	80x25	16
7	Текст	MDA/EGA/ Hercules/	720x350	9x14	80x25	Моно
		VGA/	720x400	9x16	80x25	Моно
		рідкокрист. дисплей	640x200	8x8	80x25	Моно

BIOS зберігає однобайтну перемінну за адресою 0040:0049, у який записаний номер поточного режиму. Двохбайтне слово за адресою 0040:004A дає число символів у рядку, а слово з адресою 0040:0084 - зменшене на одиницю число текстових рядків. Слово з адресою 0040:004C повідомляє розмір поточної сторінки в байтах, а байт 0040:0062 - номер поточної активної сторінки. Функція 0 переривання 10h установлює режим дисплея (номер режиму повинний задаватися в регістрі AL). У мові С для установки потрібного текстового режиму застосовується функція

```
#include<conio.h>
void textmode(int newmode)
```

При звертанні до функції на місці параметра newmode повинна стояти одна з наступних констант:

BW40 (режим 0)
C40 (режим 1)
BW80 (режим 2)
C80 (режим 3)
MONO (режим 7)
LASTMODE (попередній текстовий режим)

Крім стандартних текстових режимів у відеоадаптері EGA можна установити режим виводу 43 рядків тексту по 80 символів у кожній за рахунок зменшення матриці символу до розміру 8x8. Аналогічним образом створюється текстовий формат у 50 рядків по 80 символів для адаптера VGA. Тому в компіляторах С для функції textmode додано константу C4350, що призначена для установки режиму 43 рядків на EGA чи 50 рядків на VGA.

Графіка.

Для роботи в графічному режимі дисплея, необхідно на початку програми ініціалізувати цей режим, а при виконанні такої програми забезпечити наявність у поточному каталозі спеціального графічного BGI- драйвера, наприклад - EgaVga.bgi.

Функція ініціалізації графічної системи й установки графічного драйвера в режимі перевірки апаратури:

```
#include <graphics.h>
void far initgraph(int far *graphdriver, int far
*graphmode,
                   char far *pathtodriver);
void far detectgraph(int far *grphdriver,
                     int far *grphmode);
```

Для завершення роботи з графікою використовується функція:

```
void far closegraph(void);
```

3.5. Установка кольору й атрибутів символів

У текстовому режимі для кожної позиції із символом на екрані приділяється два байти пам'яті. Молодший байт містить код символу, а старший - атрибути символу. Спосіб інтерпретації байта атрибутів залежить від конкретної відеосистеми. Взагалі формат атрибутного байта складається з двох тетрад. Молодша тетрада визначає атрибути переднього плану, тобто колір і інтенсивність самого символу, а старша тетрада впливає на атрибути коліру, але біт 7 може керувати і мерехтінням символу. Зміст тетрад атрибутного байта перетворюються в сигнали керування монітором.

В операційній системі MS DOS для кодування 16 базових кольорів використовуються числа від 0 (чорний) до 15 (яскраво білий). У мові С у файлі conio.h визначені 16 констант із символічними іменами, що відповідають цим кодам (табл.9.2).

Таблиця 9.2

MS DOS	C	Колір
0	BLACK	чорний
1	BLUE	синій
2	GREEN	зелений
3	CYAN	бірюзовий
4	RED	червоний
5	MAGENTA	вишневий
6	BROWN	коричневий
7	LIGHTGRAY	білий
8	DARKGRAY	сірий
9	LIGHTBLUE	яскраво-синій
10	LIGHTGREEN	яскраво-зелений
11	LIGHTCYAN	яскраво-бірюзовий
12	LIGHTRED	яскраво-червоний
13	LIGHTMAGENTA	яскраво-вишневий
14	YELLOW	жовтий
15	WHITE	яскраво-білий

На С колір фону встановлюється функцією:

```
#include<conio.h>
void textbackground(int backcolor);
```

де для звичайного режиму роботи CGA як аргумент backcolor можна використовувати одну з перших восьми констант (від BLACK до LIGHTGRAY), наприклад:

```
textbackground(LIGHTGRAY);
```

а при 0 біті 5 регістра керування режимом - одну з 16 перерахованих констант.

Колір символу (тобто колір переднього плану) установлюється за допомогою функції

```
#include<conio.h>
void textcolor(int forcolor);
```

у якій як аргумент forcolor можна використовувати кожну з перерахованих вище 16 констант, наприклад:

```
textcolor(YELLOW);
```

У файлі conio.h визначена константа BLINK, що має значення 128, що у звичайному режимі CGA дозволяє включити миготіння символу, наприклад:

```
textcolor(YELLOW + BLINK);
```

Додавання 128 до коду кольору встановлює в одиницю старший біт байта атрибутів, керуючий миготінням (мал. 9.3).

Для установки значення атрибутного байта:

```
#include<conio.h>
void textattr(int newattr);
```

наприклад: textattr(0x0F);

Графіка.

Для програмування в графічному режимі існує сімейство функцій, що забезпечує роботу з колірною палітрою:

```
#include <graphics.h>
```

Установки кольору поточного тла, і поточної лінії (з палітри):

```
void far setbkcolor(int color);
void far setcolor(int color);
```

Зміна одного кольору палітри:

void far setpalette(int colornum, int color);

Повертає колір поточного кольору і колір поточної лінії:

```
int far getbkcolor(void);
int far getcolor(void);
```

Крім того маються функції установки заповнювача pattern і кольору color для поточного контуру, а також заповнення області з крапкою (x,y) кольором border (колір заповнення повинний збігатися з кольором лінії поточного контуру):

```
void far setfillstyle(int pattern, int color);
void far floodfill(int x, int y, int border);
```

3.6. Очищення всього екрана чи його частини

Очищення екрана складається з запису коду пробілу в кожну з позицій на екрані. Цю операцію в С виконує функція

```
#include<conio.h>
void clrscr(void);
```

Мається можливість очистити прямокутне вікно на екрані. Для цього потрібно спочатку задати границі вікна за допомогою функції

```
#include<conio.h>
#void window(int left, int top, int right, int bottom);
```

Аргументи left і top визначають координати лівого верхнього кута вікна. Два інші аргументи right і bottom задають координати правого нижнього кута прямокутного вікна. Для текстового режиму 40*25 параметри left і right можуть приймати значення від 1 (ліва границя всього екрана) до 40 (права границя усього екрана). Для текстового режиму 80*25 діапазон зміни цих параметрів складає від 1 до 80. Друга координата (top чи bottom) може мінятися від 1 (верхня границя всього екрана) до 25 (нижня границя всього екрана). У такий спосіб верхній лівий кут всього екрана має абсолютні координати (1,1), а нижній правий - (40,25) чи (80,25) у залежності від режиму (мал. 9.8).

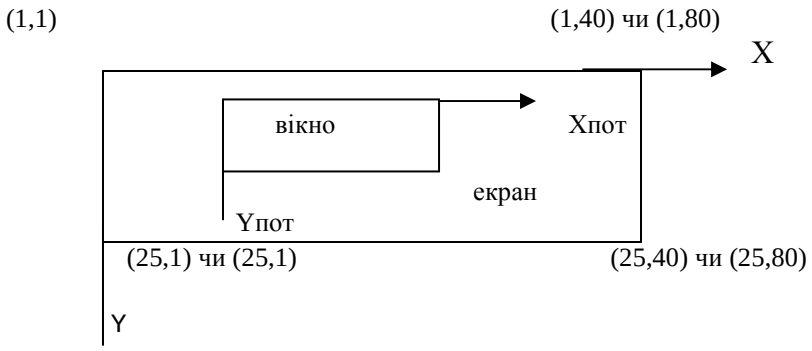


Рис. 9.8. Координатні системи для функцій window і gotoxy.

Установка вікна є істотним для функції clrscr(). Ця функція виконує очищення тільки в межах активного вікна, установленого при останньому виклику функції window. Наприклад, пари операторів

```
window(1, 25, 80, 25);
clrscr();
```

виконує запис пробілів у нижній рядок екрана дисплея, працюючого в режимі 80*25. При цьому в байти атрибутів буде заноситися значення, обумовлене функцією textbackground.

Таким чином, для очищення вікна й установки в ньому необхідного кольору тла потрібно використовувати оператори:

```
window(1, 25, 80, 25);
textbackground(LIGHTGRAY);
clrscr();
```

У графічному режимі очищення екрана проводиться функцією:

```
#include <graphics.h>
void far cleardevice(void);
```

3.7. Керування курсором

Курсор служить двом цілям. По-перше, він є покажчиком позиції на екрані, починаючи з якого оператори програми виводять дані. По-друге, він забезпечує видиму крапку відліку на екрані для користувача. Тільки для другого застосування курсор повинний бути видимим. Коли курсор небачемий (виключений), те він усе рівно указує на позицію екрана. Це важливо, оскільки будь-який вивід, підтримуваний операційною системою, починається з поточної позиції курсору.

У мові С функція window() не тільки визначає поточне текстове вікно, але і позиціонує курсор у його верхній лівий кут.

Для переміщення курсору в межах активного вікна можна використовувати функцію

```
#include<conio.h>
void gotoxy(int x, int y);
```

Координати (x, y) для цієї функції задаються не в абсолютній координатній системі екрана (X, Y), а в системі координат активного вікна (Xтек, Yтек), встановленого функцією window (мал. 9.8). Так, для верхнього лівого кута вікна: x = 1, y = 1, а для правого нижнього кута:

x = right - left + 1

y = bottom - top + 1

Наприклад, для позиціювання курсору в початок третього рядка активного вікна в програмі варто написати оператор:

gotoxy(1,3);

Іноді виникає необхідність виключити курсор. Для цього можна просто позиціювати курсор за межі екрана. Нажаль, функція gotoxy() не дозволяє вивести курсор за межі поточного вікна, у тому числі за межі вікна, що має границі усього екрана. Тому програма повинна безпосередньо звертатися до функції 2 переривання 10h, що здійснює позиціювання курсору в абсолютну позицію екрана. У регістр AH мікропроцесора потрібно записати номер функції, у регістр BH - номер активної відеосторінки (відеопам'ять може мати кілька сторінок, але на екран виводиться одна з них, активна в даний момент), у регістр DH - номер рядка, у регістр DL - номер стовпця позиції курсору. Причому, у MS DOS номера рядків екрана змінюються від 0 (верхня) до 24 (нижня), а позиції в рядку - від 0 (ліва) до 39 чи 79 (права). Нижче приводиться текст функції, призначеної для виводу курсору за межі екрана:

```
/*  Функція для вимикання курсору   */
   void out_cursor(void)
    {
     _AH=2;
     _BH=0;
     _DH=25;
     _DL=0;
     geninterrupt(0x10);
    }
```

У С псевдоперемінні _AH, _BH, _DH, _DL відповідають регістрам мікропроцесора AH, BH, DH, DL. Виклик даної функції здійснюється оператором програми

```
out_cursor();
```

Виключити курсор можна також за допомогою функції 1 переривання 10h, що встановлює розмір курсору. У регістрі CH задається номер початкового (верхнього) рядка в матриці символу, а в регістрі CL - номер кінцевого (нижнього) рядка для курсору. При цьому, якщо в CH занести значення 20h, те курсор стане невидимим.

У компіляторах C,C++ мається бібліотечна функція

```
#include<conio.h>
void _setcursortype(int cur_type);
```

яка установлює вид курсору в залежності від значення аргументу cur_type:

_NOCURSOR - курсор виключений;
_SOLIDCURSOR - курсор у виді прямокутника;
_NORMALCURSOR - звичайний курсор у виді знака підкреслення.

Графіка.

У графічному режимі мається сімейство функцій, що забезпечує малювання ліній різної форми.

```
#include <graphics.h>
```

Малювання прямих - між двома крапками, з поточної позиції на задану відстань і з поточної позиції в крапку (x,y):

```
void far line(int x1, int y1, int x, int y);
void far linerel(int x, int y);
void far lineto(int x, int y);
```

Малювання окружності, повного еліпса і дуги еліпса (з заданими кутами в градусах) з центра (x,y):

```
void far circle(int x, int y, int radius);
void far fillellipse(int x, int y,
                     int xradius, int yradius);
void far ellipse(int x, int y, int stangle,int
endangle,
                          int xradius,int yradius);
```

Т.ч. початок програми з графікою має вид:

```
int *gr = DETECT, *gh;
initgraph(&gr,&gh,"");
```

3.8. Вивід символів на екран

Для виводу одного символу ch на екран у позицію, займану курсором, використовується функція

```
#include<conio.h>
int putch(int ch);
```

наприклад:

```
putch('Q');
```

Вивід здійснюється або прямо у відеопам'ять, або за допомогою функцій BIOS, у залежності від значення глобальної перемінної C directvideo, що оголошена у файлі conio.h. Якщо directvideo має значення 1, то виконується прямий запис у відеопам'ять. Якщо ж directvideo = 0, то вивід здійснюється за допомогою функцій BIOS (помітно повільніше). За замовчуванням directvideo дорівнює 1. Функція putch() при успішному завершенні повертає виведений символ, а при помилці повертається EOF.

Для виводу рядка символів str застосовується функція

```
#include<stdio.h>
int cputs(const char *str);
```

наприклад:

```
cputs("Для виходу натисніть ESC");
```

До виведеного рядка не приєднується символ переходу на новий рядок. Рядок виводиться прямо у відеопам'ять, якщо directvideo=1, чи за допомогою виклику BIOS, якщо directvideo=0. Причому, на відміну від функції puts, функція cputs не перетворить символ переходу рядка \n у послідовність повернення каретки/переходу рядка \r\n. Функція cputs повертає останній виведений символ.

Для форматного виводу значень змінних, констант і функцій різних типів використовується функція

```
#include<conio.h>
int cprintf(const char *format [, argument, ...]);
```

де

const char *format - рядок формату,

[, argument, ...] - список аргументів,

наприклад:

```
cprintf("%s","F2 - тип IBM PC");
```

Функція cprintf одержує набір аргументів, застосовує до кожного аргументу специфікацію формату, що міститься в рядку формату з покажчиком format, і виводить відформатовані дані на екран у активне текстове вікно. Число аргументів і специфікацій повинне бути однаковим. Якщо directvideo=1, то рядок пишеться прямо в відеопам'ять, у противному випадку використовується вивід через BIOS. Функція cprintf не перетворить символ переходу рядка \n у послідовність повернення каретки/переходу рядка \r\n. Функція повертає кількість виведених символів.

У графічному режимі є функція видачі тексту в зазначену позицію екрана (x,y):

```
#include <graphics.h>
void far outtextxy(int x, int y, char far *textstring);
```

3.9. Пересилання фрагмента тексту з екрана в буфер і назад

Функція C

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

```
int gettext(int left, int top, int right, int bottom,
            void *buffer);
```

копіює зміст прямокутної області екрана з верхнім лівим кутом (left, top) і правим нижнім кутом (right, bottom) у програмний буфер, заданий покажчиком buffer. Усі координати повинні бути задані в абсолютній координатній системі екрана (X, Y) (мал. 9.8). Функція gettext зчитує зміст прямокутної області послідовно з ліва на право і зверху вниз. Оскільки кожна позиція на екрані вимагає 2 байти, то для запам'ятовування прямокутної області шириною W і висотою H потрібно W*H*2 байтів. У випадку успішного завершення функція gettext повертає

1. У випадку помилки функція повертає 0.

Функція C

```
#include<conio.h>
int puttext(int left, int top, int right, int bottom,
            void *buffer);
```

копіює зміст буфера, адреса якого визначається аргументом buffer, у прямокутну область екрана з верхнім лівим кутом (left, top) і правим нижнім кутом (right, bottom). Усі координати повинні бути задані в абсолютній координатній системі екрана (X, Y) (мал. 9.8). Функція puttext розміщає на екрані зміст буфера послідовно з ліва на право і зверху вниз. При цьому виконується пряме відображення у відеопам'ять. У випадку успішного завершення функція puttext повертає ненульове значення. У випадку помилки (наприклад, якщо задані координати, що виходять за межі екрана) функція повертає 0.

Розглянемо як приклад використання даних функцій програму, що виділяє на екрані прямокутну область з координатами лівого верхнього кута (left, top) і координатами правого нижнього кута (right, bottom), пересилає у свій буфер з адресою buffer зміст цієї області екрана, створює й очищає вікно в межах виділеної області, виводить у кожен рядок вікна заданий текст, виключає курсор, а потім очікує натискання клавіші ESC. По натисканню цієї клавіші, програма пересилає назад збережений у буфері текст, відновлюючи в такий спосіб зображення на екрані, що там було до виклику даної програми. Програма оформлена у виді універсальної функції:

```
/*  Функція для виводу послідовності рядків у вікно і виходу з закриттям вікна по
натисканню ESC      */
#define ESC 27
int  little_window(int left, int top, int right, int bottom,
                   char *p[], int ground, int color)
 {
  int size, n, i;
  char *buffer;
  n = bottom - top + 1;
  size = 2*(right - left + 1)*n;
  buffer = malloc(size);
  if (buffer == NULL) return -1; /* Вихід з кодом повернення -1
при
                    неможливості виділити size байтів пам'яті
*/
  gettext(left, top, right, bottom, buffer);
  textbackground(ground);
  window(left, top, right, bottom);
  clrscr();
  textcolor(color);
  for(i=1; i<=n; i++) {
     gotoxy(1, i);
     cprintf("%s", p[i-1]);
```

```
    }
   out_cursor();
   while(getch() != 27);
   puttext(left, top, right, bottom, buffer);
   free(buffer);
   return 0;
}
```

У приведеній програмі перемінна n визначає кількість рядків у виділюваній прямокутній області, а перемінна size - необхідний розмір буфера в байтах для збереження змісту цієї області. Функція malloc(size), прототип якої знаходиться у файлі alloc.h, динамічно виділяє необхідну пам'ять і повертає її початкову адресу покажчику buffer. При неможливості виділити size байтів пам'яті функція malloc повертає NULL. Передача виведених у вікно рядків тексту здійснюється через масив покажчиків

p[0], p[1], ..., p[i], ..., p[n-1].

При цьому тло усього вікна визначається параметром ground, а колір символів - параметром color. Перед завершенням роботи буфер звільняється за допомогою виклику free(buffer).

Для виклику даної функції можна використовувати, наприклад, таку послідовність операторів:

```
char *ptr[3];
    ptr[0] = malloc(62);
    ptr[1] = malloc(62);
    ptr[2] = malloc(62);
    strcpy (ptr[0], "");
    strcpy (ptr[1], "Це - IBM PC");
    strcpy (ptr[2], "Для виходу натисніть ESC");
    little_window(10, 5, 70, 7, ptr, RED, BLUE);
    free(ptr[0]);
    free(ptr[1]);
    free(ptr[2]);
```

Функція С

```
#include<string.h>
    char *strcpy(char *dest, const char *scr);
```

забезпечує копіювання змісту рядка, заданого іншим параметром str, у рядок, обумовлений першим параметром dest. Функція повертає dest.

Якщо виникне необхідність вивести у вікно не рядок символів, а, наприклад, ціле число, те його потрібно попередньо перетворити в рядок символів. Таке перетворення в мові С виконує функція

```
#include<stdlib.h>
char *itoa(int value, char *string, int radix);
```

Ця функція перетворює ціле значення value у рядок символів і записує результат у параметр string. Аргумент radix визначає основу системи числення, що буде використана при перетворенні значення value. Функція повертає покажчик на рядок string, причому завершення помилково неможливо.

Крім того, можна зчепити два рядки, використовуючи функцію

```
#include<string.h>
char *strcat(char *dest, char *src);
```

Дана функція приєднує в кінець рядка, заданої параметром dest, копію рядка, обумовлену параметром src. Функція strcat повертає покажчик на рядок dst.

Нижче приводиться фрагмент програми, що визначає версію MS DOS і забезпечує вивід результатів за допомогою функції

little_window() :

```
extern unsigned char _osmajor, _osminor;
```

```
char vh[6], vl[6];
char *ptr[3];
ptr[0] = malloc(62);
ptr[1] = malloc(62);
ptr[2] = malloc(62);
strcpy(ptr[0] , "        Версія MS-DOS");
itoa(_osmajor, vh, 10);
strcat(vh,".");
itoa(_osminor, vl, 10);
strcat(vh, vl);
strcpy(ptr[1],"               ");
strcat(ptr[1], vh);
strcpy(ptr[2] , "      Для виходу натисніть ESC");
little_window(10, 5, 70, 7, ptr, RED, BLUE);
free(ptr[0]);
free(ptr[1]);
free(ptr[2]);
```

Графіка.

У графічному режимі мається сімейство функцій, що забезпечує перенос графічних образів екрана в пам'ять і назад.

Функції - визначення кількості байт, необхідних для збереження частини образа; збереження частини образа в спеціальній області пам'яті; виводу частини образа на екран (можливо з накладенням - op= COPY_PUT):

```
#include <graphics.h>
unsigned far imagesize(int left, int top,int right,int
bottom);
void far getimage(int left, int top, int right,int
bottom, void far *bitmap);
void far putimage(int left, int top,void far *bitmap,
int op);
```

Визначення пристроїв на шині PCI на прикладі відеоконтролера.

Перше, що треба врахувати, простір портів (переривань) шини PCI відображається на простір портів шини ISA. У шині PCI будь-яка передача сигналів відбувається пакетним образом, де кожен пакет розбитий на фази. Починається пакет з фази адреси, за якою, як правило, випливає один чи кілька фаз даних. Кількість фаз даних у пакеті може бути невизначена, але обмежена таймером, що визначає максимальний час, у протязі якого пристрій може використовуватися шиною. Такий таймер має кожен підключений пристрій, а його значення може бути задане при конфігуруванні. Відповідно до концепції PCI передачею пакета даних керує не CPU, а міст, включений між ним і шиною PCI (Host Bridge Cashe/DRAM Controller).

PCI пристосована для розпізнавання апаратних засобів і аналізу конфігурації системи у відповідності зі стандартом P&P, розробленим корпорацією Intel.

Специфікація шини PCI визначає три типи ресурсів: два звичайних (діапазони пам'яті і діапазон вводу/виводу, як їх називає компанія Microsoft) і configuration space - конфігураційний простір.

31	16 15	0
00	Device ID	Vendor ID	Незалежно
04	Status	Code	
08	Class Code	Revision ID	від типу
0C	BIST Header Type /Latency Timer/Cashe Line Size	пристрою
10	Base Address Registers	Визначається
24	- - - - - - -	значенням
28	Cardbuus CIS Pointer	Header
2C	Subsustem IO/ Subsystem Vendor ID	Type
30	Expansion ROM Base Address	
34	Reserved	
38	Reserved	
3C	Interrupt Line	
40	Device	Визначається
	---------- Specific ----------	
FF	Register	користувачем

Воно складається з трьох регіонів:

- заголовка, незалежного від пристрою (device-independent header region)
- регіону, обумовленого типом пристрою (header-type region)
- регіону, обумовленого користувачем (user-defined region)

У заголовку міститься інформація про виробника і тип пристрою - поле Class Code (мережний адаптер, контролер диска, мультимедіа і так далі) і інша службова інформація. Наступний регіон містить регістри діапазонів пам'яті і вводу/виводу, які дозволяють динамічно виділяти пристрою область системної пам'яті й адресного простору. У залежності від реалізації системи конфігурація пристроїв виробляється або BIOS (при виконанні POST - Power On-Self Test), або програмно. Базовий регістр expansion ROM аналогічно дозволяє відображати ROM пристрої в системну пам'ять. Поле CIS (Card Information Structure) pointer використовується картами cardbus (PCMCIA). Останні 4 байти регіону використовуються для визначення переривання і часу запиту/володіння.

В.Г.Кулаковим розроблена програма, фрагмент якої демонструє визначення коду виготовлювача, використовуваного адресного простору і номера переривання IRQ

```
IDEAL
P386
LOCALS
MODEL MEDIUM


SEGMENT sseg para stack 'STACK'
DB 400h DUP(?)
ENDS

DATASEG
; Параметри пристрою PCI
```

```
VendorID      DW ? ;ідентифікатор виготовлювача
DeviceID      DW ? ;ідентифікатор пристрою
ClassCode     DD ? ;тип пристрою
BaseAddress0  DD ? ;базова адреса 0
BaseAddress1  DD ? ;базова адреса 1
BaseAddress2  DD ? ;базова адреса 2
BaseAddress3  DD ? ;базова адреса 3
BaseAddress4  DD ? ;базова адреса 4
BaseAddress5  DD ? ;базова адреса 5
InterruptLine DB ? ;номер використовуваного переривання IRQ
; Координати пристрою PCI
BusNumber     DB ? ;номер шини
DeviceNumber  DB ? ;номер пристрою і номер функції
; Лічильник операцій натискання/відпускання клавіш
PressCounter  DW ?
; Текстові повідомлення
PCI   DB 0,25,"ТЕСТУВАННЯ ФУНКЦІЙ PCI BIOS",0
      DB 4,26,"ПАРАМЕТРИ ВІДЕОКОНТРОЛЕРА",0
      DB 6,28,"Номер шини:",0
      DB 7,22,"Номер пристрою:",0
      DB 8,25,"Номер функції:",0
      DB 9,12,"Ідентифікатор виготовлювача:",0
      DB 10,14,"Ідентифікатор пристрою:",0
      DB 11,24,"Тип пристрою:",0
      DB 12,23,"Базова адреса 0:",0
      DB 13,23,"Базова адреса 1:",0
      DB 14,23,"Базова адреса 2:",0
      DB 15,23,"Базова адреса 3:",0
      DB 16,23,"Базова адреса 4:",0
      DB 17,23,"Базова адреса 5:",0
      DB 18,8,"Номер використовуваного переривання:",0
NoIRQ DB 18,40,"не використовується",0
Any  DB YELLOW,24,29,"Натисніть будь-яку клавішу",0
; Повідомлення про помилки
NoPCI  DB 12,18,"Система не підтримує PCI BIOS",0
NoSVGA DB 12,23,"Відеоконтролер SVGA PCI не знайден",0
BadReg DB 12,28,"Невірний номер регістра",0
ENDS

CODESEG
;*****************************
;* Основний модуль програми *
;*****************************
PROC PCITest
        mov     AX,DGROUP
        mov     DS,AX
        mov     [CS:MainDataSeg],AX

; Опитування PCI-пристроїв
; Перевірити наявність PCI BIOS
        mov     AX,0B101h
        int     1Ah
        jc      @@PCIBIOSNotFound
```

```
        cmp     EDX,20494350h
        jne     @@PCIBIOSNotFound
; Знайти відеоконтролер (перший пристрій
; типу 30000h)
        mov     AX,0B103h
        mov     ECX,030000h
        mov     SI,0
        int     1Ah
        jc      @@DeviceNotFound
        mov     [BusNumber],BH
        mov     [DeviceNumber],BL
; Одержати ідентифікатор виготовлювача
        mov     AX,0B109h ;читати слово
        mov     DI,0      ;зсув слова
        int     1Ah
        jc      @@BadRegisterNumber
        mov     [VendorID],CX
; Одержати ідентифікатор пристрою
        mov     AX,0B109h ;читати слово
        mov     DI,2      ;зсув слова
        int     1Ah
        jc      @@BadRegisterNumber
        mov     [DeviceID],CX
; Одержати тип пристрою (самоперевірка)
        mov     AX,0B10Ah ;читати подвійне слово
        mov     DI,8      ;зсув слова
        int     1Ah
        jc      @@BadRegisterNumber
        shr     ECX,8
        mov     [ClassCode],ECX
; Одержати базову адресу 0
        mov     AX,0B10Ah ;читати подвійне слово
        mov     DI,10h    ;зсув слова
        int     1Ah
        jc      @@BadRegisterNumber
        mov     [BaseAddress0],ECX
; Одержати базову адресу 1
        mov     AX,0B10Ah ;читати подвійне слово
        mov     DI,14h    ;зсув слова
        int     1Ah
        jc      @@BadRegisterNumber
        mov     [BaseAddress1],ECX
; Одержати базову адресу 2
        mov     AX,0B10Ah ;читати подвійне слово
        mov     DI,18h    ;зсув слова
        int     1Ah
        jc      @@BadRegisterNumber
        mov     [BaseAddress2],ECX
; Одержати базову адресу 3
        mov     AX,0B10Ah ;читати подвійне слово
        mov     DI,1Ch    ;зсув слова
        int     1Ah
        jc      @@BadRegisterNumber
```

```
        mov     [BaseAddress3],ECX
; Одержати базову адресу 4
        mov     AX,0B10Ah ;читати подвійне слово
        mov     DI,20h    ;зсув слова
        int     1Ah
        jc      @@BadRegisterNumber
        mov     [BaseAddress4],ECX
; Одержати базову адресу 5
        mov     AX,0B10Ah ;читати подвійне слово
        mov     DI,24h    ;зсув слова
        int     1Ah
        jc      @@BadRegisterNumber
        mov     [BaseAddress5],ECX

; Одержати номер використовуваного пристроєм
; переривання IRQ
        mov     AX,0B108h ;читати байт
        mov     DI,3Ch    ;зсув байта
        int     1Ah
        jc      @@BadRegisterNumber
        mov     [InterruptLine],CL

; Далі вивести отримані дані на екран у
; шістнадцатеричному коді
```

Для зручності приводимо опис використаних переривань.

```
-------------------------------------------------
INT 1A B101 - PCI BIOS v2.0c+ - INSTALLATION CHECK
-------------------------------------------------
Inp.:
     AX = B101h
     EDI = 00000000h
Return: AH = 00h if installed
         CF clear
         EDX = 20494350h (' ICP')
         EDI = physical address of protected-mode entry point
         AL = PCI hardware characteristics
         BH = PCI interface level major version (BCD)
         BL = PCI interface level minor version (BCD)
         CL = number of last PCI bus in system
     EAX, EBX, ECX, and EDX may be modified
     all other flags (except IF) may be modified
```

Notes: this function may require up to 1024 byte of stack; it will not enable interrupts if they were disabled before making the call some BIOSes do not change EDI, so applications looking for the
protected-mode entry point should set EDI to 00000000h before calling this function

```
-----------------------------------------------
INT 1A B102 - PCI BIOS v2.0c+ - FIND PCI DEVICE
-----------------------------------------------
Inp.:
     AX = B102h
```

```
     CX = device ID
     DX = vendor ID
     SI = device index (0-n)
Return: CF clear if successful
     CF set on error
     AH = status (00h,83h,86h)
          00h successful
           BH = bus number
           BL = device/function number (bits 7-3 device,
           bits 2-0 func)
     EAX, EBX, ECX, and EDX may be modified
all other flags (except IF) may be modified
```

Notes: this function may require up to 1024 byte of stack; it will not enable interrupts if they were disabled before making the call device ID FFFFh may be reserved as a wildcard in future implementations the meanings of BL and BH on return were exchanged between the initial drafts of the specification and final implementation all devices sharing a single vendor ID and device ID may be enumerated by incrementing SI from 0 until error 86h is returned

```
Table 00735
-----------
Values for ATI PCI device code:
 4158h    68800AX (Mach32)
 4354h    215CT222
 4358h    210888CX
 4758h    210888GX (Mach64)
 5654h    215VT222 Video Expression

Table 00878 Format of PCI Configuration Data:
------------------------------------------------
Offset    Size Description
 00h WORD vendor ID (read-only)
          FFFFh returned if requested device non-existent
 02h WORD device ID (read-only)
 04h WORD command register
 06h WORD status register
 08h BYTE revision ID
 09h  3 BYTEs  class code
          bits 7-0: programming interface
          bits 15-8: sub-class
          bits 23-16: class code
 0Ch BYTE cache line size
 0Dh BYTE latency timer
 0Eh BYTE header type
          bits 6-0: header format
               00h other
               01h PCI-to-PCI bridge
               02h PCI-to-CardBus bridge
          bit 7: multi-function device
 0Fh BYTE Built-In Self-Test result
---header type 00h---
 10h DWORD     base address 0
          (OpenHCI) base address of host controller registers
```

```
 14h DWORD     base address 1
 18h DWORD     base address 2
 1Ch DWORD     base address 3
 20h DWORD     base address 4
 24h DWORD     base address 5
 28h DWORD     CardBus CIS pointer (read-only)
 2Ch WORD subsystem vendor ID or 0000h
 2Eh WORD subsystem ID or 0000h
 30h DWORD     expansion ROM base address
 34h BYTE offset of capabilities list within configuration space
(R/O)(only valid if status register bit 4 set)
 35h  3 BYTEs  reserved
 38h DWORD     reserved
 3Ch BYTE interrupt line
00h = none, 01h = IRQ1 to 0Fh = IRQ15
 3Dh BYTE interrupt pin (read-only)
          (00h = none, else indicates INTA# to INTD#)
 3Eh BYTE minimum time bus master needs PCI bus ownership, in
          250ns units(read-only)
 3Fh BYTE maximum latency, in 250ns units (bus masters only)
          (read-only)
 40h 48 DWORDs varies by device
---header type 01h---
 10h DWORD     base address 0
 14h DWORD     base address 1
 18h BYTE primary bus number (for bus closer to host processor)
 19h BYTE secondary bus number (for bus further from host
processor)
 1Ah BYTE subordinate bus number
 1Bh BYTE secondary latency timer
 1Ch BYTE I/O base
 1Dh BYTE I/O limit
 1Eh WORD secondary status
 20h WORD memory base
 22h WORD memory limit
 24h WORD prefetchable memory base
 26h WORD prefetchable memory limit
 28h DWORD     prefetchable base, upper 32 bits
 2Ch DWORD     prefetchable limit, upper 32 bits
 30h WORD I/O base, upper 16 bits
 32h WORD I/O limit, upper 16 bits
 34h DWORD     reserved
 38h DWORD     expansion ROM base address
 3Ch BYTE interrupt line
 3Dh BYTE interrupt pin (read-only)
 3Eh WORD bridge control
 40h 48 DWORDs varies by device
---header type 02h---
 10h DWORD     CardBus Socket/ExCa base address
          bits 31-12: start address of socket interface register
          block  in 4K blocks
          bits 11-0: reserved (0)
 14h BYTE offset of capabilities list within configuration
```

```
          space (R/O)(only valid if status register bit 4 set)
 15h BYTE reserved
 16h WORD secondary status
 18h BYTE PCI bus number
 19h BYTE CardBus bus number
 1Ah BYTE subordinate bus number
 1Bh BYTE CardBus latency timer
 1Ch DWORD     memory base address 0
 20h DWORD     memory limit 0
 24h DWORD     memory base address 1
 28h DWORD     memory limit 1
 2Ch WORD I/O base address 0
 2Eh WORD I/O base address 0 high word (optional)
 30h WORD I/O limit 0
 32h WORD I/O limit 0 high word (optional)
 34h WORD I/O base address 1
 36h WORD I/O base address 1 high word (optional)
 38h WORD I/O limit 1
 3Ah WORD I/O limit 1 high word (optional)
 3Ch BYTE interrupt line
 3Dh BYTE interrupt pin (read-only) (no interrupt used if 00h)
 3Eh WORD bridge control
 40h WORD subsystem vendor ID
 42h WORD subsystem device ID
 44h DWORD     16-bit PC Card legacy mode base address (for
accessing ExCa registers)
 48h 14 DWORDs reserved
 80h 32 DWORDs varies by device

Table 00732 Values for PCI vendor ID:
---------------------------------------
 0E11h    Compaq
 1002h    ATI
 1008h    Epson
 100Ah    Phoenix Technologies
 100Bh    National Semiconductor
 1011h    DEC
 1013h    Cirrus Logic
 1014h    IBM
 101Ch    Western Digital
 101Eh    AMI
 1020h    Hitachi Computer Electronics
 1023h    Trident Microsystems
 1024h    Zenith Data Systems
 1025h    Acer
 1028h    Dell Computer Corporation
 102Eh    Olivetti Advanced Technology
 102Fh    Toshiba America
 1032h    Compaq
 1033h    NEC Corporation
 1037h    Hitachi Micro Systems
 103Ch    Hewlett-Packard
 1043h    Asustek
```

```
 1045h    OPTi
 104Ch    Texas Instruments
 104Dh    Sony Corporation
 1054h    Hitachi, Ltd.
 1057h    Motorola
 1067h    Mitsubishi Electronics
 106Bh    Apple Computer
 1073h    Yamaha Corporation
 1078h    Cyrix Corporation
 1088h    Microcomputer Systems (M) Son
 1089h    Data General Corporation
 108Eh    Sun Microsystems
 1099h    Samsung Electronics Co. Ltd.
 10A2h    Quantum Corporation
 10AFh    Microcomputer Systems
 10C4h    Award Software International Inc.
 10C5h    Xerox Corporation
 10CAh    Fujitsu
 10DEh    NVIDIA Corporation
 10ECh    Realtek Semiconductor
 10FDh    Soyo Technology Corp. Ltd.
 1106h    VIA Technologies
 1131h    Philips Semiconductors
 1137h    Cisco Systems Inc
 11DAh    Novell
 A200h    NEC Corp.
 A259h    Hewlett Packard
 A304h    Sony
 C0DEh    Motorola
. . . . . .
і далі на http://www.pobox.com/~ralf/files.html)
-----------------------------------
INT 1A B103 - PCI BIOS v2.0c+ - FIND PCI CLASS CODE
-----------------------------------
Inp.:
     AX = B103h
     ECX = class code
         bits 31-24 unused
         bits 23-16 class
         bits 15-8  subclass
         bits 7-0   programming interface
     SI = device index (0-n)
Return: CF clear if successful
     CF set on error
     AH = status (00h,86h)
         00h successful
          BH = bus number
          BL = device/function number (bits 7-3 device,
          bits 2-0 func)
         86h device not found
     EAX, EBX, ECX, and EDX may be modified all other flags
(except IF) may be modified
```

Notes: this function may require up to 1024 byte of stack; it will not enable interrupts if they were disabled before making the call the meanings of BL and BH on return were exchanged between the initial drafts of the specification and final implementation all devices sharing the same Class Code may be enumerated by incrementing SI from 0 until error 86h is returned

```
---------------------------------------------------------
INT 1A B108 - PCI BIOS v2.0c+ - READ CONFIGURATION BYTE
---------------------------------------------------------
Inp.:
     AX = B108h
     BH = bus number
     BL = device/function number (bits 7-3 device, bits 2-0
function)
     DI = register number (0000h-00FFh)
Return: CF clear if successful
         CL = byte read
     CF set on error
     AH = status (00h,87h)
     EAX, EBX, ECX, and EDX may be modified
     all other flags (except IF) may be modified
```

Notes: this function may require up to 1024 byte of stack; it will not enable interrupts if they were disabled before making the call the meanings of BL and BH on entry were exchanged between the initial drafts of the specification and final implementation BUG: the Award BIOS 4.51PG (dated 05/24/96) incorrectly returns FFh for register 00h if the PCI function number is nonzero

```
---------------------------------------------------------
INT 1A B109 - PCI BIOS v2.0c+ - READ CONFIGURATION WORD
---------------------------------------------------------
Inp.:
     AX = B109h
     BH = bus number
     BL = device/function number (bits 7-3 device, bits 2-0
function)
     DI = register number (0000h-00FFh, must be multiple of 2)
Return: CF clear if successful
         CX = word read
     CF set on error
     AH = status (00h,87h)
     EAX, EBX, ECX, and EDX may be modified
     all other flags (except IF) may be modified
---------------------------------------------------------
INT 1A B10A - PCI BIOS v2.0c+ - READ CONFIGURATION DWORD
---------------------------------------------------------
Inp.:
     AX = B10Ah
     BH = bus number
     BL = device/function number (bits 7-3 device, bits 2-0
function)
     DI = register number (0000h-00FFh, must be multiple of 4)
Return: CF clear if successful
         ECX = dword read
     CF set on error
```

AH = status (00h,87h)
EAX, EBX, ECX, and EDX may be modified
all other flags (except IF) may be modified

Тема № 8. КЕРУВАННЯ КЛАВІАТУРОЮ

1. Параметри клавіатури

Клавіатура виконана у вигляді окремого пристрою, що підключається до комп'ютера за допомогою одного з роз'ємів, частіше всього PS/2 або USB. Існують два мікроконтролери, що забезпечують процес обробки вводу від клавіатури: один — на материнській платі ПК, другий, — в самій клавіатурі. Клавіатура побудована на основі мікроконтролера 8042, який постійно сканує натиснення клавіш на клавіатурі.

За кожною клавішею клавіатури закріплений номер, однозначно пов'язаний з распайкою клавіатурної матриці і не залежний безпосередньо від позначень, нанесених на поверхню клавіш. Цей номер називається скан-кодом (назву підкреслює той факт, що комп'ютер сканує клавіатуру для пошуку натиснутої клавіші). Скан-код не відповідає ASCII-коду клавіші, одній і тій же клавіші можуть відповідати декілька значень ASCII-коду.

Насправді клавіатура генерує два скан-коди для кожної клавіші — коли користувач натискає клавішу і коли відпускає. Наявність двох скан-кодів важлива, оскільки деякі клавіші мають сенс тільки тоді, коли вони натиснуті (наприклад, Shift, Control, Alt).

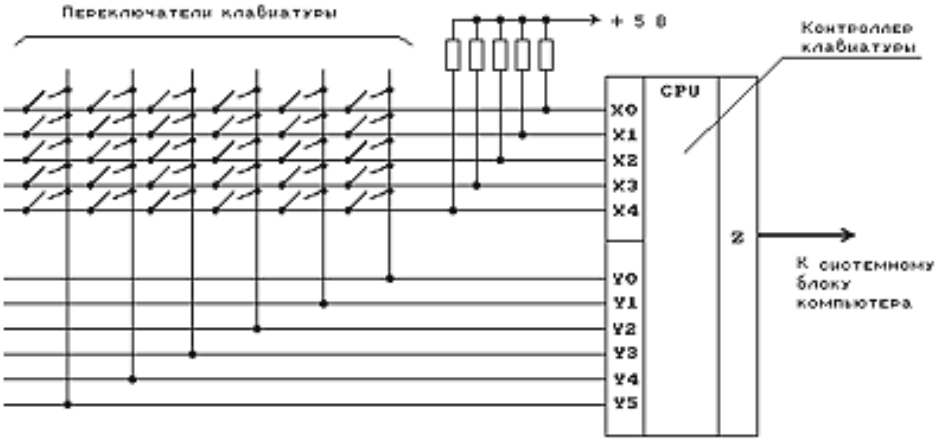


Рис. 1. Спрощена схема клавіатури

Коли користувач натискає клавішу на клавіатурі, він замикає електричний контакт. В результаті при наступному скануванні мікроконтролер фіксує натиснення певної клавіші і посилає в центральний комп'ютер скан-код натиснутої клавіші і запит на переривання. Аналогічні дії виконуються і тоді, коли оператор відпускає натиснуту раніше клавішу. Другий мікроконтролер отримує скан-код, проводить перетворення скан-коду, робить його доступним на порту вводу-виводу 60h і потім генерує апаратне переривання центрального процесора. Після цього процедура обробки переривання може отримати скан-код з вказаного порту введення-виводу.

Розширена 101-клавішна клавіатура має програмний інтерфейс, унаслідок чого можна встановлювати її деякі характеристики. При завантаженні MS DOS встановлює затримку початку автоповторення генерації коду натиснутої (і не відпущеної) клавіші у 0,5 секунди від моменту натискання і частоту автоповторення в 10 повторень у секунду. Зовнішня команда MS DOS MODE (що міститься у файлі MODE.COM), дозволяє змінювати ці параметри у версіях MS DOS 4.0 і вище. Формат команди MODE для настроювання клавіатури:

```
MODE CON: RATE=частота DELAY=затримка
```

де частота - частота автоповторення кодів натиснутої клавіші. Діапазон значень від 1 до 32;

затримка - затримка початку автоповторення від моменту натискання клавіші. Допускаються значення 1, 2, 3, 4, що відповідає затримкам 0,25 , 0,5 , 0,75 і 1 с.

П р и к л а д.

```
MODE CON: DELAY=1 RATE=20
```

2. Перетворення скан-кода в код символу

Клавіатура містить 8-розрядний мікропроцесор, що сприймає кожне натискання на клавішу і видає скан-код у порт А мікросхеми інтерфейсу з периферією, розташованої на системної платі ПЕОМ. Скан-код - це однобайтне число, яке представляє собою ідентифікаційний номер клавіші.

Клавіатура працює серіями - при натисканні видається два підряд скан-кода клавіші, а при відпусканні - код F0h і скан-код.

Таблиця 8.1

16й код	10й код	Кл.	16й код	10й код	Клавіша	16й код	10й код	Клавіша	16й код	10й код	Клавіша
01	1	Esc	16	22	U	2b	43	\ |	40	64	F6
02	2	1 !	17	23	I	2c	44	Z	41	65	F7
03	3	2 @	18	24	O	2d	45	X	42	66	F8
04	4	3 #	19	25	P	2e	46	C	43	67	F9
05	5	4 $	1a	26	[{	2f	47	V	44	68	F10
06	6	5 %	1b	27	] }	30	48	B	45	69	NumLock
07	7	6 ^	1c	28	Enter	31	49	N	46	70	ScrollLock
08	8	7 &	1d	29	Ctrl	32	50	M	47	71	Home [7]
09	9	8 *	1e	30	A	33	51	, <	48	72	↑ [8]
0a	10	9 (	1f	31	S	34	52	. >	49	73	PgUp [9]
0b	11	0)	20	32	D	35	53	/ ?	4a	74	K -
0c	12	- _	21	33	F	36	54	Shft(Rt)	4b	75	← [4]
0d	13	+ =	22	34	G	37	55	* PrtSc	4c	76	[5]
0e	14	bksp	23	35	H	38	56	Alt	4d	77	^# [6]
0f	15	Tab	24	36	J	39	57	spacebar	4e	78	K +
10	16	Q	25	37	K	3a	58	CapsLock	4f	79	End [1]
11	17	W	26	38	L	3b	59	F1	50	80	↓ [2]
12	18	E	27	39	; :	3c	60	F2	51	81	PgDn [3]
13	19	R	28	40	" '	3d	61	F3	52	82	Ins [0]
14	20	T	29	41	` ~	3e	62	F4	53	83	Del [.]
15	21	Y	2a	42	Shft(L)	3f	63	F5			

Розширена клавіатура для 83 основних клавіш генерує ті ж скан-коди, перераховані в табл. 8.1. У табл. 8.2 приведені додаткові скан-коди унікальних клавіш розширеної 101-клавішної клавіатури.

Таблиця 8.2

Клавіша	16-я последовательн	Схема обробки кодів
F11	57	1. Натискання клавіші
F12	58	2. 8р-Процесор клавіатури
Right-Alt	e0 38	3. Інтерфейс з периферією. Порт А
Right-Ctrl	e0 1d	4. Буфер клав. Системна пам'ять
PrintScreen	e0 2a e0 37	5. Перерив. виводу на екран
Shft-PrintScreen (SysReq)	e0 37	6. Відеобуфер
Ctrl-PrintScreen (SysReq)	e0 37	7. Відображення на екрані
Alt-PrintScreen	54	8. Вплив програми користувача на
Pause	e1 1d 45 e1 9d c5	п.5
Ctrl-Pause (Break)	e0 46 e0 c6	
Insert	e0 53	

```
Shft-Insert ..... e0 aa e0 52
Delete ..... e0 53
Shft-Delete ..... e0 aa e0 53
← ..... e0 4b
Shft-← ..... e0 aa e0 4b
Home .....e0 47
Shft-Home .....e0 aa e0 47
End .....e0 4f
Shft-End .....e0 aa e0 4f
↑ .....e0 48
Shft-↑ .....e0 aa e0 48
├ .....e0 50
Shft-├ .....e0 aa e0 50
PageUp .....e0 49
Shft-PageUp .....e0 aa e0 49
PageDown .....e0 51
Shft-PageDown .....e0 aa e0 51
^#& .....e0 4d
Shft-^#& .....e0 aa e0 4d
K Enter .....e0 1c
K / .....e0 35
Shft-K / .....e0 aa e0 35
```

Коли скан-код видається в порт А мікросхеми інтерфейсу з периферією, то викликається апаратне переривання клавіатури 9h.Процесор моментально припиняє свою роботу і виконує процедуру, що аналізує скан-код. Коли надходить код від клавіші зрушення чи перемикача, то в спеціальній змінній у пам'яті ПЕОМ фіксується зміна статусу (наприклад, переключення з верхнього регістра на нижній). В усіх інших випадках скан-код перетворюється в код символу ASCII за умови, що він подається при натисканні клавіші (у противному випадку скан-код відкидається). Звичайно, процедура спочатку визначає установку клавіш зрушення і перемикачів, щоб правильно одержати код, що вводиться (це "а" чи "А"). Після цього введений код міститься в буфер клавіатури, що являє собою область пам'яті, здатну запам'ятати до 15 символів, що вводяться, поки програма занадто зайнята, щоб обробити їх. Переривання 9h є апаратно-залежним, тому його оброблювач для 83-клавіатури відрізняється від оброблювача для розширеної 101-клавішний клавіатури.

Скан-код, що зчитується підпрограмою обробки переривання 9h, у більшості випадків є числом, що представляє собою відносне положення клавіші на клавіатурі. Так, скан-код 01h відноситься до клавіші <Esc>, 02h - до клавіші !/1 , 03h - до клавіші @/2 і т.д. ліворуч праворуч по верхньому ряді клавіатури. На відміну від більшості схем нумерації відсутня клавіша, зв'язана з кодом 0. Оброблювач переривання 9h використовує 0 як псевдо-скан-код для відображення комбінації Ctrl-Break.

Для дешифрації скан-кодів оброблювач переривання 9h містить спеціальну таблицю пошуку. Для перебування відповідності в таблиці використовується ряд алгоритмів. Як тільки скан-код розшифрований, програма обробки переривання 9h зберігає скан-код і ASCII-символ у буфері клавіатури і повертає керування перерваній програмі. Однак до передачі керування оброблювач переривання 9h перевіряє, чи не є скан-код кодом FFh, 4-символьного апаратного буфера, що вказує на переповнення скан-кодів клавіатури (не слід плутати цей буфер з 15-символьним буфером, використовуваним оброблювачем переривання 9h). При виявленні коду FFh переривання 9h повідомляє про переповнення сигналом динаміка. У протилежному випадку, думаючи, що переповнення відсутнє, оброблювач переривання 9h переглядає таблицю перекладу в логічному порядку. Спочатку перевіряються клавіші зміни функцій: клавіші зрушення - лівий і правий Shift, Ctrl, Alt і чотири фіксуємих клавіші Insert, CapsLock, NumLock, ScrollLock. Якщо скан-код належить однієї з чотирьох клавіш зрушення, це відображається установкою відповідного

біта в інформаційному байті прапора 1 регістрів клавіатури, розташованому за адресою 0040:0017 в області даних BIOS (див. табл. 5.2). При виявленні скан-коду відпускання клавіші зрушення відповідний біт скидається в 0. Такий метод дозволяє програмі обробки переривання 9h завжди знати, яка з цих клавіш натиснута.

Мається два типи кодів символів: коди ASCII і розширені коди. Коди ASCII - це однобайтні числа, що кодують латинські і російські літери, цифри, спеціальні і керуючі символи в відповідності з таблицею ASCII. Розширені коди мають довжину 2 байта, причому перший байт завжди представляє код ASCII 0. Розширені коди привласнені клавішам чи комбінаціям клавіш, які не мають їхнього символу, що представляє, ASCII. Наприклад, комбінація клавіш Alt-A формує розширений код 0:30. Початковий нуль дозволяє програмі визначити, чи належить даний код набору ASCII чи розширеному набору.

Призначення перших 32-х однобайтних кодів ASCII, що є керуючими, показане в табл. 8.3. Інші коди і що зображують їхні символи приведені в табл. 8.4 [24].

Таблиця 8.3

10-й код	16-й код	Ctrl	Ім'я	Призначення
0	00	^@	NUL	порожньо (кінець рядка)
1	01	^A	SOH	початок заголовка
2	02	^B	STX	початок тексту
3	03	^C	ETX	кінець тексту
4	04	^D	EOT	кінець передачі
5	05	^E	ENQ	запит
6	06	^F	ACK	підтвердження
7	07	^G	BEL	дзвоник
8	08	^H	BS	крок назад
9	09	^I	HT	TAB гориз. табуляція
10	0a	^J	LF	перенесення рядка
11	0b	^K	VT	вертик. табуляція
12	0c	^L	FF	подача форми
13	0d	^M	CR	повернення каретки
14	0e	^N	SO	shift out
15	0f	^O	SI	shift in
16	10	^P	DLE	data line escape
17	11	^Q	DC1	device ctrl 1 (X-ON)
18	12	^R	DC2	device ctrl 2
19	13	^S	DC3	device ctrl 3 (X-OFF)
20	14	^T	DC4	device ctrl 4
21	15	^U	NAK	від'ємне підтвердження
22	16	^V	SYN	синхронізація
23	17	^W	ETB	кінець блоку передачі
24	18	^X	CAN	зняти
25	19	^Y	EM	кінець носія
26	1a	^Z	SUB	підстановка
27	1b	^[	ESC	escape
28	1c	^\	FS	родільник файлів
29	1d	^]	GS	роздільник груп
30	1e	^^	RS	роздільник записів
31	1f	^_	US	роздільник полів

Таблиця 8.4

Ст. Тетр.	Молодша тетрада коду															
	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f

20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
80	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
90	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
a0	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
b0	░	▒	▓	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐
c0	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
d0	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	▉	▄	▌	▐	▀
e0	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
f0	Ё	ё	Є	є	Ї	ї	Ў	ў	°	∙	·	√	№	¤	■	

Варто звернути увагу на те, що натискання деяких різних клавіш може привести до появи того самого коду ASCII. Наприклад, якщо натиснути білу клавішу "-", розташовану в верхньому ряді між клавішами "0" і "=", і сіру клавішу "-", розташовану на цифровій клавіатурі, то в обох випадках з'явиться код 2Dh. Однак скан-коди для цих клавіш будуть різними - 0Ch і 4Ah відповідно (табл. 8.1). Ці специфічні для кожної клавіші скан-коди можна використовувати в прикладній програмі для точного визначення, яка з них натиснута.

У табл. 8.5 і 8.6 приведене значення другого байта розширених кодів 83-клавішної клавіатури, що діють і для 101-клавішної клавіатури. У табл. 8.7 показане значення другого байта розширених кодів, унікальних для 101-клавішної клавіатури.

Таблиця 8.5

Кл.	16й код	10й код	Клавіша	16й код	10й код	Клавіша	16й код	10й код	Клавіша	16й код	10й код
F1	3b	59	Shift-F1	54	84	Ctrl-F1	5e	94	Alt-F1	68	104
F2	3c	60	Shift-F2	55	85	Ctrl-F2	5f	95	Alt-F2	69	105
F3	3d	61	Shift-F3	56	86	Ctrl-F3	60	96	Alt-F3	6a	106
F4	3e	62	Shift-F4	57	87	Ctrl-F4	61	97	Alt-F4	6b	107
F5	3f	63	Shift-F5	58	88	Ctrl-F5	62	98	Alt-F5	6c	108
F6	40	64	Shift-F6	59	89	Ctrl-F6	63	99	Alt-F6	6d	109
F7	41	65	Shift-F7	5a	90	Ctrl-F7	64	100	Alt-F7	6e	110
F8	42	66	Shift-F8	5b	91	Ctrl-F8	65	101	Alt-F8	6f	111
F9	43	67	Shift-F9	5c	92	Ctrl-F9	66	102	Alt-F9	70	112
F10	44	68	Shift-F10	5d	93	Ctrl-F10	67	103	Alt-F10	71	113

Таблиця 8.6

Клав.	16й код	10й код	Клав.	16й код	10й код	Клавіша	16й код	10й код	Клавіша	16й код	10й код

Alt-A	1e	30	Alt-P	19	25	Alt-3	7a	122	down ├	50	80
Alt-B	30	48	Alt-Q	10	16	Alt-4	7b	123	left ←	4b	75
Alt-C	2e	46	Alt-R	13	19	Alt-5	7c	124	right ^#&	4d	77
Alt-D	20	32	Alt-S	1f	31	Alt-6	7d	125	up ↑	48	72
Alt-E	12	18	Alt-T	14	20	Alt-7	7e	126	End	4f	79
Alt-F	21	33	Alt-U	16	22	Alt-8	7f	127	Home	47	71
Alt-G	22	34	Alt-V	2f	47	Alt-9	80	128	PgDn	51	81
Alt-H	23	35	Alt-W	11	17	Alt--	82	130	PgUp	49	73
Alt-I	17	23	Alt-X	2d	45	Alt-=	83	131			
Alt-J	24	36	Alt-Y	15	21				^left	73	115
Alt-K	25	37	Alt-Z	2c	44	NUL	03	3	^right	74	116
Alt-L	26	38				Shift-Tab	0f	15	^End	75	117
Alt-M	32	50	Alt-0	81	129	Ins	52	82	^Home	77	119
Alt-N	31	49	Alt-1	78	120	Del	53	83	^PgDn	76	118
Alt-O	18	24	Alt-2	79	121	^PrtSc	72	114	^PgUp	84	132

Таблиця 8.7

Клавіша	16й код	10й код	Клавіша	16й код	10й код	Клавіша	16й код	10й код
F11	85	133	Alt-Bksp	0e	14	Alt- K /	a4	164
F12	86	134	Alt-Enter	1c	28	Alt- K *	37	55
Shft-F11	87	135	Alt-Esc	01	1	Alt- K -	4a	74
Shft-F12	88	136	Alt-Tab	a5	165	Alt- K +	4e	78
Ctrl-F11	89	137	Ctrl-Tab	94	148	Alt- K Enter	a6	166
Ctrl-F12	8a	138						
Alt-F11	8b	139	Alt-up ↑	98	152	Ctrl- K /	95	149
Alt-F12	8c	140	Alt-down ├	a0	160	Ctrl- K *	96	150
Alt-[	1a	26	Alt-left ←	9b	155	Ctrl- K -	8e	142
Alt-]	1b	27	Alt-right	9d	157	Ctrl- K +	90	144
Alt-;	27	39						
Alt-'	28	40	Alt-Delete	a3	163	Ctrl- K ↑ [8]	8d	141
Alt-`	29	41	Alt-End	9f	159	Ctrl- K 5 [5]	8f	143
Alt-\	2b	43	Alt-Home	97	151	Ctrl- K ├ [2]	91	145
Alt-,	33	51	Alt-Insert	a2	162	Ctrl- K Ins[0]	92	146
Alt-.	34	52	Alt-PageUp	99	153	Ctrl- K Del[.]	93	147
Alt-/	35	53	Alt-PageDown	a1	161			

Мається кілька комбінацій клавіш, що виконують спеціальні функції і не генерують скан-коди. Дані комбінації включають <Ctrl><Break>, <Ctrl><Alt><Del>, <Shift><PrtSc> чи <PrtSc>, а також <SysReq> для IBM PC AT. Ці виключення приводять до заздалегідь визначеним результатам. Всі інші натискання клавіш повинні інтерпретуватися програмою користувача.

8.3. Очищення буфера клавіатури

Програма повинна очистити буфер клавіатури перед тим, як видати запит на введення, вилучая тим самим сторонні натискання клавіш, що можуть на той час нагромадитися у буфері. Буфер може накопичувати до 15 натискань на клавішу незалежно від того, чи є вони однобайтними кодами ASCII чи двохбайтними розширеними кодами. Таким чином, буфер повинний відвести два байти пам'яті для кожного натискання на клавішу. Для однобайтових кодів перший байт містить код ASCII, а другий скан-код клавіші. Для розширених кодів перший байт містить ASCII 0, а другий - значення розширеного коду. Це значення звичайне збігається з скан-кодом клавіші, але не завжди, оскільки деякі клавіші можна комбінувати з клавішами зрушення для генерації різних кодів.

Буфер улаштований як циклічна черга, що називають також буфером FIFO (перший увійшов - перший пішов). Він займає безупинну область адреси пам'яті. Однак визначеної

комірки пам'яті, що зберігає "початок рядка" у буфері, немає. Замість цього два покажчики зберігають позиції голови і хвоста рядка символів, що знаходиться в буфері в сучасний момент. Нові натискання клавіш зберігаються в позиціях, що випливають за хвостом (у більш старших адресах пам'яті), і відповідно обновляється покажчик хвоста буфера. Після того як витрачено весь буферний простір, нові символи продовжують вставлятися із самого початку буферної області; тому можливі ситуації, коли голова рядка в буфері має більшу адресу, чим хвіст. Коли буфер заповнений, нові символи, що вводяться, ігноруються. На мал. 8.1 показані деякі можливі конфігурації даних у буфері [15].

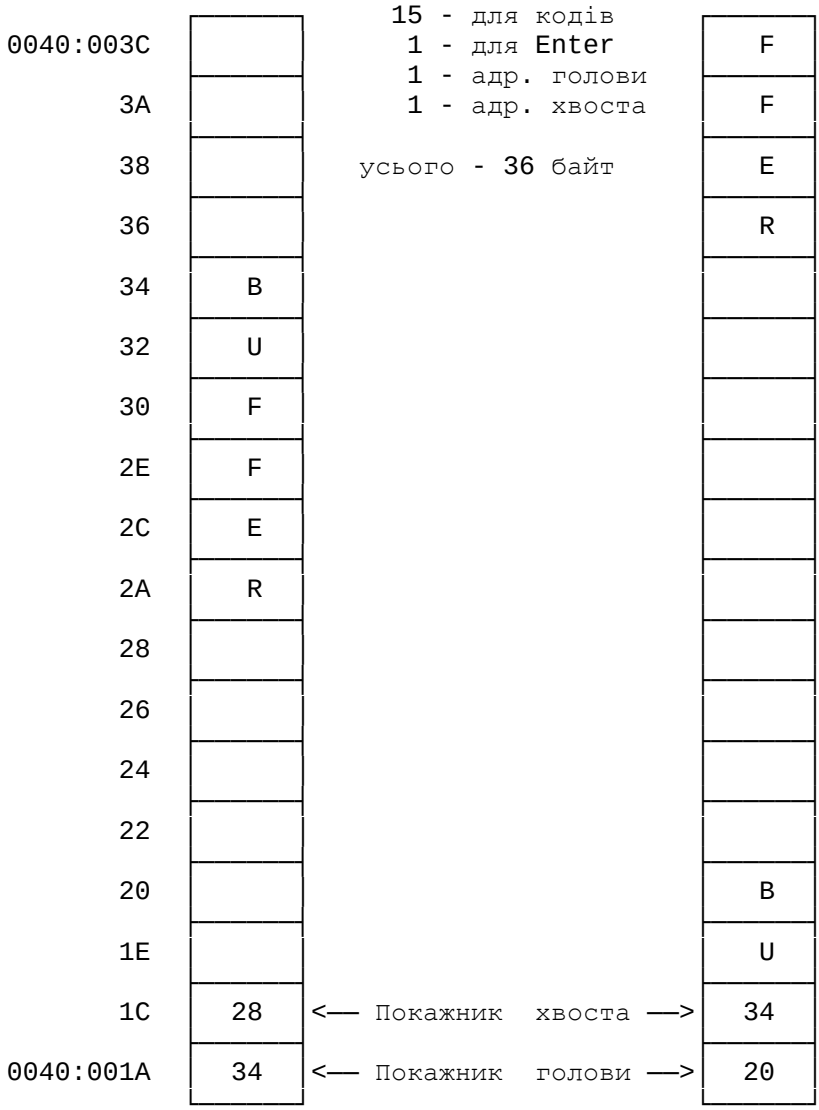


Рис. 8.1. Конфігурація буфера клавіатури

У той час як покажчик на голову встановлений на перший уведений символ, покажчик на хвіст установлений на позицію за останнім уведеним символом. Коли обидва покажчики рівні, буфер порожній. Щоб дозволити введення 15 символів, потрібно 16-а порожня позиція, два байти якої завжди містять код повернення каретки (ASCII 13) і скан-код клавіші <Enter>, рівний 28. Ця порожня позиція безпосередньо передує голові рядка символів. 32 байта буфера починаються з адреси 0040:001E. Кінець буфера розташований за адресою 0040:003E, причому слово з адресою 0040:003E буферу не належить. Покажчики на голову і хвіст, розташовані по адресам 0040:001A і 0040:001C

відповідно. Хоча під покажчики відведено 2 байти, використовується тільки молодший байт. Значення покажчиків міняються від 001C до 003C, що відповідає зсувам в області даних BIOS, сегментна адреса якої 0040. Крім того, в області даних BIOS містяться двохбайтні покажчики на зсуви фізичного початку і фізичного кінця буфера по адресам 0040:0080 і 0040:0082 відповідно. Звичайно ці зсуви рівні 001E і 003E, але їх можна змінювати для зміни положення і розміру буфера.

Для очищення буфера клавіатури за допомогою вирівнювання значень покажчиків можна застосувати наступну програму:

```
#include<stdio.h>
#include<dos.h>
int main(void)
 {
  poke(0x40, 0x1C, peek(0x40, 0x1A));
  return 0;
 }
```

Однак цей метод не найкращий. Деякі програми можуть створювати свій буфер де-небудь в іншім місці пам'яті, а крім того, завжди існує можливість переривання клавіатури в середині рядка, що змінить покажчик хвоста. По цим причинам краще залишити покажчики буфера в спокої. Потрібно читати з буфера доти, поки не буде повернутий символ ASCII 0, який показує, що буфер порожній.

8.4. Переривання 16h і функція C,C++ bioskey()

8.4.1. Чекання натискання клавіші і введення коду символу

Якщо єдиний обов'язок апаратного переривання 9h - це прийом даних із клавіатури, їхня інтерпретація і занесення в буфер, те за допомогою програмного переривання 16h прикладна програма може в міру необхідності одержувати дані про натиснуті клавіші. У даного переривання мається ряд функцій. Перша функція з номером 0h передає прикладній програмі інформацію про натиснуту клавішу (її скан- і ASCII-коди) з буфера клавіатури. При використанні цієї функції треба враховувати одну її особливість. Справа в тім, що буфер клавіатури велику частину часу порожній, а функція 0h переривання 16h зобов'язано повернути інформацію про натиснуту клавішу. Тому функція 0h змушує центральний процесор виконувати цикл чекання до тих пір, поки в буфері не з'явиться який-небудь код - тобто поки не буде натиснуто яку-небудь клавішу. При натисканні якої-небудь клавіші цикл переривається, і викликається оброблювач апаратного переривання 9h, у результаті чого код клавіші заноситься в буфер. По закінченні переривання 9h функція 0h переривання 16h передає код натиснутої клавіші прикладній програмі.

Функція 1h переривання 16h дозволяє прикладній програмі визначити, є в буфері дані чи ні і потім негайно повертає керування прикладній програмі. Усе, що залишається програмі - це періодично звертатися до функції 1h переривання 16h і перевіряти заповнювання буфера. Якщо буде отримана відповідь "так", те програма зможе негайно прочитати символ за допомогою функції 0h переривання 16h.

У мовах C, C++ інтерфейс із перериванням 16h реалізований через функцію bioskey(), що має наступний синтаксис:

```
#include<bios.h>
int  bioskey(int cmd);
```

де cmd - виконувана операція:

1) Якщо аргумент cmd дорівнює 0, те функція bioskey() повертає двохбайтне число, що відповідає натиснутої на клавіатурі клавіші. При цьому, якщо молодші 8 біт числа відмінні від нуля, то в них передається символ у коді ASCII. У противному випадку

передається розширений код: у молодших восьми бітах - нуль, а в старших восьми бітах - значення другого байта розширеного коду (наприклад, 59 для клавіші F1);

2) Якщо аргумент cmd дорівнює 1, то виробляється перевірка наявності символу для читання. Якщо при перевірці повертається значення 0, те це означає, що клавіша не була натиснута. У противному випадку повертається значення коду натиснутої клавіші. Сам же код зберігається в буфері клавіатури для зчитування при наступному виклику функції bioskey() зі значенням параметра cmd, рівним 0.

Наступний приклад ілюструє використання даної функції. Програма, написана на С, спочатку очищає екран, потім виводить у його верхній частині наступний текст:

Призначення функціональних клавіш
F1 - генерація тону 500 Гц
F2 - генерація тону 1000 Гц
ESC - вихід

При натисканні на функціональну клавішу F1 (розширений двохбайтний код 0:59) формується звуковий сигнал частотою 500 Гц. При натисканні на функціональну клавішу F2 (розширений двохбайтний код 0:60) формується звуковий сигнал частотою 1000 Гц. Нарешті, при натисканні на клавішу ESC (однобайтний код ASCII 27) відбувається завершення програми:

```
#include<stdio.h>
#include<dos.h>
#include<bios.h>
#include<stdlib.h>
#define ESC 27
#define F1  59
#define F2  60
#define CMD_TEST 1
#define CMD_READ 0
int main(void)
 {
  union
    {
      int word;
      struct
        {
          unsigned low_byte :8; /* Молодший байт коду клавіші */
          unsigned high_byte:8; /* Старший байт коду клавіші */
        }st;
     }key;
   system("cls");
   printf("\nпризначення функціональних клавіш\n");
   printf("   F1  - генерація тону 500 Гц\n");
   printf("   F2  - генерація тону 1000 Гц\n");
   printf("   ESC - вихід\n");
   for(;;)
    {
      if(bioskey(CMD_TEST))
        {
         key.word = bioskey(CMD_READ);
         if(key.st.low_byte == ESC) return 0;
          else if(key.st.low_byte == 0 && key.st.high_byte ==
F1)
                {
                 sound(500);
```

```
                    sleep(1); /* Затримка на 1 сек  */
                    nosound();
                   }
          else if(key.st.low_byte == 0 && key.st.high_byte ==
F2)
                   {
                    sound(1000);
                    sleep(1); /* Затримка на 1 сек  */
                    nosound();
           }
        }
    }
 }
```

8.4.2. Перевірка/установка статусу клавіш перемикачів

Два байти, розташовані в комірках пам'яті 0040:0017 і 0040:0018, містять біти, що відбивають статус клавіш зрушення й інших клавіш-перемикачів (табл. 8.8).

Таблиця 8.8

Адреса	Біт	Клавіша	Значення, коли біт дорівнює 1
0040:0017	7	Insert	режим вставки включений
	6	CapsLock	режим CapsLock включений
	5	NumLock	режим NumLock включений
	4	ScrollLock	режим ScrollLock включений
	3	Alt	клав. натисн.(з будь-якої стор.)
	2	Ctrl	клав. натисн.(з будь-якої стор.)
	1	лівий Shift	клавіша натиснута
	0	правий Shift	клавіша натиснута
0040:0018	7	Insert	клавіша натиснута
	6	CapsLock	клавіша натиснута
	5	NumLock	клавіша натиснута
	4	ScrollLock	клавіша натиснута
	3	Ctrl-NumLock	режим паузи включений
тільки для 101-клав. клавіатури	2	SysReq	клавіша SysReq натиснута
	1	Alt	лівий Alt натиснут
	0	Ctrl	лівий Ctrl натиснут

Апаратне переривання клавіатури 9h негайно обновляє ці біти статусу, як тільки буде натиснута одна з клавіш - перемикачів, навіть якщо не було зчитано жодного символу з буфера клавіатури. Це вірно і для клавіші <Ins>, що єдина з цих клавіш поміщає код у буфер (установка статусу Ins міняється, навіть якщо в буфері немає місця для символу). Слід зазначити, що біт 3 за адресою 0040:0018 встановлюється в 1, коли діє режим затримки Ctrl-NumLock; оскільки в цьому стані програму припинено, цей біт несуттєвий.

Функція 2 переривання BIOS 16h надає доступ до одного з байтів статусу, що має адресу 0040:0017. Це переривання використовується функцією мови C bioskey(), зі значенням аргументу cmd, рівним 2. Функція повертає вміст байта за адресою 0040:0017. Програма користувача може проаналізувати уміст цього байта. При необхідності вона також може змінити визначені біти і записати нове значення за допомогою функції pokeb(), що має синтаксис:

```
#include<dos.h>
 void pokeb(unsigned segment, unsigned offset,
  char value);
```

де segment - сегментна частина адреси байта,
offset - зсув в адресі байта,
value - значення, що заноситься в байт.

Розглянемо наступний приклад. Необхідно написати програму, виконуючу команду MS DOS DIR із ключем /w при натисканні клавіші "стрілка нагору" і команду MS DOS VER при натисканні клавіші "стрілка вниз". Клавіші "стрілки нагору", "стрілка вліво", "стрілка вправо", "стрілки вниз", розташовані на цифровій клавіатурі праворуч, видають розширені коди 0:72, 0:75, 0:77 і 0:80 відповідно тільки в тому випадку, якщо виключений режим NumLock (біт 5 байта за адресою 0040:0017 встановлений у нуль). При включеному режимі NumLock дані клавіші видають однобайтні коди ASCII, що відповідають цифрам 2, 4, 6, 8.

Ці коди збігаються з кодами, що генеруються при натисканні цифрових клавіш у верхньому ряді основної клавіатури.

Програма, що нижче приводиться, спочатку перевіряє, чи встановлений режим NumLock, і, якщо він установлений, то скасовує цей режим шляхом запису нуля в 5-й біт байта за адресою 0040:0017. Цим самим дозволяється робота клавіш зі стрілками додаткової цифрової клавіатури в режимі генерації двохбайтних розширених кодів:

```
/*  Тестувати після виходу з компілятора */
#include<stdio.h>
#include<dos.h>
#include<bios.h>
#include<stdlib.h>

#define CMD_TEST 1
#define CMD_READ 0
#define ESC     27
#define UP      72
#define DOWN    80
#define MASK_NumLock  0x20  /* Виділення 5-го розряду */
int main(void)
 {
  union
    {
      int word;
      struct
        {
          unsigned low_byte :8;  /* Молодший байт коду */
          unsigned high_byte:8;  /* Старший байт коду */
        }st;
     }key;
   if( (key.word = bioskey(2)) & MASK_NumLock)
      pokeb(0x40, 0x17, key.st.low_byte & ~MASK_NumLock);
   system("cls");
   printf("\nпризначення функціональних клавіш\n");
   printf("   Стрілка нагору  - команда MS DOS  DIR /w\n");
   printf("   Стрілка вниз   - команда MS DOS  VER\n");
   printf("   ESC - вихід\n");
   for(;;)
    {
    if(bioskey(1))
     {
      key.word = bioskey(0);
      if(key.st.low_byte == ESC) return 0;
```

```
        else if(key.st.low_byte == 0 && key.st.high_byte == UP)
                system("DIR /w");
        else if(key.st.low_byte == 0 && key.st.high_byte == DOWN)
                system("VER");
      }
    }
  }
```

Для деяких розширених клавіатур дана програма не буде виключати світлодіодний індикатор "NumLock", у результаті чого порушиться синхронізація між станом NumLock і світінням індикатора, однак результуючий стан 5-го біта байта 0040:0017 буде вірним.

Прочитати вміст байта 0040:0018 можна за допомогою виклику функції peekb(), що має синтаксис:

```
#include<dos.h>
char peekb(unsigned segment, unsigned offset);
```

де segment - сегментна частина адреси байта,
offset - зсув в адресі байта,

П р и к л а д.

```
.................................
char byte_18;
byte_18 = peekb(0x0040, 0x0018);
.................................
```

8.4.3. Функції переривання 16h для 101-клавішної клавіатури

З появою 101-клавішної клавіатури були введені дві нові функціональні клавіші F11 і F12. Для того, щоб зробити ці клавіші доступними для нових прикладних програм, а також для забезпечення сумісності з більш ранніми програмами, написаними до появи цих клавіш, у BIOS 101-клавішної клавіатури були уведено три додаткові функції, що називаються розширеними. Ці три функції 10h, 11h і 12h мають те ж призначення, що і старі функції 0h, 1h і 2h, за винятком того, що вони повертають інформацію про нові клавіші 101-клавішної
клавіатури, у тому числі і про F11 і F12. Новий BIOS застосовує спеціальні правила для функцій зчитування даних клавіатури, які використовувалися до введення 101-клавішної клавіатури і використовуються багатьма програмами в даний час.

Функція 10h переривання 16h 101-клавішної клавіатури (дата видання BIOS 15 листопада 1895 р. чи пізніше) повертає двохбайтне число, що відповідає натиснутої на клавіатурі клавіші. При цьому в регістрі AL передається код ACSII (якщо він не дорівнює нулю), а у регістрі AH - скан-код. Якщо регістр AL містить 0, то передається розширений код: у молодших восьми бітах - нуль, а в старших восьми бітах - значення другого байта розширеного коду (наприклад, 59 для клавіші F1).

Функція 11h переривання 16h 101-клавішної клавіатури дозволяє прикладній програмі визначити, є в буфері дані чи немає. При відсутності в буфері символу для зчитування функція установлює прапор нульового результату ZF регістра прапорів (мал. 16) у 1, а при наявності символу - у 0. В останньому випадку через пари регістрів AL і AH передається код, так само як і при виклику функції 10h, однак сам символ не витягається з буфера. Програма може прочитати символ за допомогою функції 10h переривання 16h.

Наступний приклад демонструє застосування функцій 10h і 11h переривання 16h для відстеження натискання клавіш F11 і F12:

```
/*  Тестувати після виходу з компілятора  */
#include<stdio.h>
#include<dos.h>
```

```
#include<bios.h>
#include<stdlib.h>

#define CMD_TEST 0x11
#define CMD_READ 0x10
#define ESC     27
#define F12    134
#define F11    133
#define MASK_REG_FLAG 0x40   /* Виділення прапора ZF (6-й
розряд)
                                у регістрі прапорів
*/
int main(void)
 {
   union REGS inr, outr;
   system("cls");
   printf("\nпризначення функціональних клавіш\n");
   printf("   F12  - команда MS DOS  DIR /w\n");
   printf("   F11  - команда MS DOS  VER\n");
   printf("   ESC  - вихід\n");
    for(;;)
     {
      inr.h.ah = CMD_TEST;
      int86(0x16, &inr, &outr);
      if(!(outr.x.flags & MASK_REG_FLAG )) return 0;
     {
      inr.h.ah = CMD_READ;
      int86(0x16, &inr, &outr);
      if(outr.h.al == ESC) return 0;
       else if(outr.h.al == 0 && outr.h.ah == F12)
               system("DIR /w");
       else if(outr.h.al == 0 && outr.h.ah == F11)
               system("VER");
     }
   }
 }
```

Функція 12h переривання 16h 101-клавішної клавіатури повертає в регістрі AL уміст байта 0040:0017, що відбиває статус частини клавіш-перемикачів (табл. 8.8), однак у регістрі AH повертається не вміст байта 0040:0018, а восьмирозрядна структура прапорів, показана на мал. 8.2.

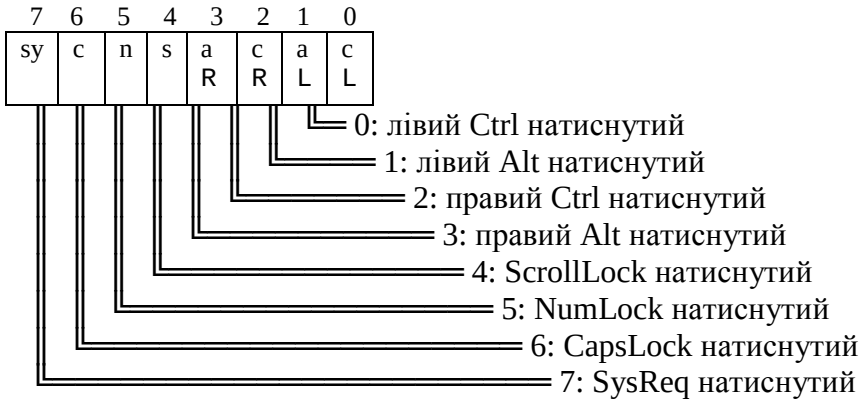


Рис. 8.2. Значення регістра AH після виконання

функції 12h переривання 16h.

8.4.4. Функції C,C++ kbhit(), getch(), getche()

Функція C,C++ kbhit(), що має синтаксис:

```
#include<conio.h>
int kbhit(void);
```

перевіряє, чи була натиснута яка-небудь клавіша і, отже, чи занесений її код у буфер клавіатури. Якщо клавіша була натиснута, функція повертає ненульове значення, у противному випадку вона повертає 0. Код натиснутої клавіші можна визначити за допомогою виклику функції getch():

```
#include<conio.h>
int getch(void);
```

Функція зчитує введений символ без відображення його на екрані. Причому, ця функція при першому виклику повертає молодший байт коду символу (перетворений до типу int). Програма повинна перевірити його на рівність нулю для виявлення розширених кодів і при позитивному результаті перевірки викликати функцію getch() ще раз. При повторному виклику повертається перетворений до типу int старший байт розширеного коду.

Функція

```
#include<conio.h>
int getche(void);
```

виконує ті ж дії, що і getch(), але з відображенням уведеного з клавіатури символу на екрані дисплея. Наступний приклад ілюструє застосування перерахованих функцій:

```
/*  Тестувати після виходу з компілятора */
#include<stdio.h>
#include<dos.h>
#include<conio.h>
#include<stdlib.h>
#define ESC     27
#define F11    133
#define F2      60
int main(void)
 {
   int key;
   system("cls");
   printf("\nпризначення функціональних клавіш\n");
   printf("   F11 - команда MS DOS  DIR  A:/w\n");
   printf("   F2  - команда MS DOS  DIR C:/w\n");
   printf("   ESC - вихід\n");
   for(;;)
   {
    if(kbhit())      /* Є чи символ у буфері ? */
     {
      key = getch(); /* Читання молодшого байта коду символу */
      if(key == ESC) return 0;
       else if(key == 0)
              {
               key = getch();/* Читання старшого байта коду
                                символу */
               if(key == F11)
                  {
                    printf("Уставте дискету\nготові ?(Д/Н)\n");
```

```
                    while((key = getche()) != 'Д' && key != 'д')
                        ;
                    system("DIR A:/w");
                   }
                 else if(key == F2) system("DIR C:/w");
               }
     }
   }
 }
```

8.4.5. Занесення коду символу в буфер клавіатури

Прикладна програма може вставляти символи в буфер клавіатури, завершуючи рядок символом повернення каретки і змінюючи значення покажчиків. Якщо це зроблено перед завершенням програми, те при поверненні керування в MS DOS ці символи будуть лічені і може бути автоматично завантажена інша програма.

Функція 5h переривання 16h (BIOS з датою видання 15 листопада 1985 р. чи пізніше) заносить код символу в буфер клавіатури і змінює потрібним образом покажчики буфера. При цьому в регістрі CL повинний бути заданий ASCII-код символу, а в регістрі CH - скан-код чи 0, якщо значення скан-коду не важливо. При успішному виконанні операції функція повертає 0 у регістрі AL, а при помилці в цьому регістрі повертається значення 1, що говорить про переповнення буфера клавіатури.

Наступний приклад ілюструє застосування функції 5h переривання 16h. Програма перед своїм закінченням заносить у буфер клавіатури рядок символів DIR, що завершується кодом клавіші Enter, у результаті чого відразу після виходу в MS DOS буде виконана команда перегляду поточного каталогу DIR:

```
/*  Тестувати після виходу в MS DOS  */
#include<stdio.h>
#include<conio.h>
#include<dos.h>
int push_ch(unsigned char ascii, unsigned char scan)
 { /* Функція put_ch() заносить код символу в буфер клавіатури
*/
   union REGS inr, outr;
   inr.h.ah = 0x5;
   inr.h.cl = ascii;
   inr.h.ch = scan;
   int86(0x16, &inr, &outr);
   return outr.h.al; /* Функція при успішному завершенні
                        повертає 0, а у випадку помилки  1 */
 }
int main(void)
 {
  if( push_ch('D', 0) || push_ch('I', 0) || push_ch('R', 0)
     || push_ch(0x, 0) ) /* 0x - це ASCII-код клавіші Enter */
   {
    printf("Помилка: буфер повний!\n");
    poke(0x40, 0x1C, peek(0x40, 0x1A));  /* Очищення буфера */
    return 1;
   }
   else return 0;
 }
```

8.4.6. Приклади тестів

Тестування індикаторів клавіатури:

```
#include <stdio.h>
#include <dos.h>
void main (void)
{
#define MASK_CAPS    0x40
#define MASK_NUM     0x20
#define MASK_SCROLL  0x10
int value = 0;
  value = peekb(0x0040,0x0017);
  if(value & MASK_NUM)
           pokeb(0x0040,0x0017,value & ~MASK_NUM);
  else     pokeb(0x0040,0x0017,value | MASK_NUM);
    value = peekb(0x0040,0x0017);
    if(value & MASK_CAPS)
             pokeb(0x0040,0x0017,value & ~MASK_CAPS);
    else     pokeb(0x0040,0x0017,value | MASK_CAPS);
. . .
  і т.д.
}
```

Тестування відеорежимів:

```
#include <stdio.h>
#include <dos.h>
void main (void)
{  char V_Mod;
   int i;
   union REGS rr;
  rr.h.ah = 0x0f;     // Читання відеорежиму
  int86(0x10, &rr, &rr);
  V_Mod = rr.h.al;    // Запам'ятати режим
    rr.h.ah = 0;        // Установити відеорежим
    rr.h.al = 4;
    int86(0x10, &rr, &rr);
    puts("MODE - 04H.\n FORMAT - 320x200.\n CELL - 8x8.\n");
    puts("COLOR - 4.\n ADAPTER - CGA,EGA.\n ADR - B800.\n\n");
    getch();
  rr.h.ah = 0;        // Встановити відеорежим
  rr.h.al = 0x0d;
  int86(0x10, &rr, &rr);
  puts("MODE - 0DH.\n FORMAT - 320x200.\n CELL - 8x8.\n");
  puts("COLOR - 16.\n ADAPTER - EGA,VGA.\n ADR - A000.\n\n");
  getch();
  . . .
  і т.д.

// Відновити вихідний відеорежим
}
```

Тестування відеоатрибутів:

```
#include <stdio.h>
#include <dos.h>
void main (void)
{  int i;
   union REGS rr;
  textmode(BW40);
  gotoxy(1,1);    cprintf("РЕЖИМ - 00H.");
  gotoxy(15,7);   cprintf("НОРМАЛЬНИЙ.");
  textattr(0x87);    gotoxy(16,8);  cprintf("МИГОТЛИВИЙ.");
  textattr(0x0F);    gotoxy(16,9);  cprintf(" ЯСКРАВИЙ.");
  textattr(0x8F);    gotoxy(12,10); cprintf(" ЯСКРАВИЙ
МИГОТЛИВИЙ.");
  textattr(0x70);    gotoxy(14,11); cprintf(" НЕГАТИВНИЙ.");
  textattr(0x0);    gotoxy(10,12); cprintf(" НЕГАТИВНИЙ
МИГОТЛИВИЙ.");
  getch();
    textmode(LASTMODE);
    textmode(C40);   gotoxy(1,1);    cprintf("РЕЖИМ - 01H.");
    gotoxy(15,7);    cprintf("НОРМАЛЬНИЙ.");
    textattr(0x87);  gotoxy(16,8);   cprintf("МИГОТЛИВИЙ.");
    textattr(0x0F);  gotoxy(16,9);   cprintf(" ЯСКРАВИЙ.");
    textattr(0x8F);  gotoxy(12,10);  cprintf(" ЯСКРАВИЙ
МИГОТЛИВИЙ.");
    textattr(0x70);  gotoxy(14,11);  cprintf(" НЕГАТИВНИЙ.");
    textattr(0x0);  gotoxy(10,12);  cprintf(" НЕГАТИВНИЙ
МИГОТЛИВИЙ.");
    getch();
 . . .
  і т.д.

// Відновити вихідні відеоатрибути
}
```

Тестування динаміка

```
#include <stdio.h>
#include <dos.h>
void DinWin( void )
{   sound(1000);
    sleep(1); // Затримка на 1 сек
    nosound();
    sound(500); sleep(1);
    nosound();
 . . .
  і т.д.
}
```

У ОС Windows обробку апаратного переривання клавіатури IRQ 1 виконує драйвер i8042prt.sys. ОС має доступ до клавіатури, використовуючи потік необробленого вводу (Raw Input Thread, RIT, який віддає драйвер клавіатури), яке є частиною системного

процесу csrss.exe. Операційна система при старті створює RIT і системну чергу апаратного введення (system hardware input queue, SHIQ).

Коли користувач натискає або відпускає одну з клавіш, системний контролер клавіатури виробляє апаратне переривання. Його обробник викликає спеціальну процедуру обробки переривання IRQ 1 (interrupt service routine, ISR), зареєстровану в системі драйвером i8042prt. Дана процедура прочитує з внутрішньої черги контроллера клавіатури дані, що з'явилися. Обробка апаратного переривання повинна бути максимально швидкою, тому ISR ставить в чергу виклик відкладеної процедури (Deferred Procedure Call, DPC) I8042KeyboardIsrDpc і завершує свою роботу. Як тільки це стане можливо (IRQL знизиться до DISPATCH_LEVEL), DPC буде викликана системою. У цей момент буде викликана процедура зворотного виклику KeyboardClassServiceCallback, зареєстрована драйвером Kbdclass у драйвера i8042prt. KeyboardClassServiceCallback витягне з своєї черги чекаючий завершення запит IRP, заповнить максимальну кількість структур KEYBOARD_INPUT_DATA, що несуть всю необхідну інформацію про натиснення/відпуски клавіш, і завершить IRP. Потік необробленого введення прокидається, обробляє отриману інформацію і знов посилає IRP типу IRP_MJ_READ драйверу класу, який знову ставиться в чергу до наступного натиснення/відпуску клавіші. Таким чином, у стека клавіатури завжди є принаймні один чекаючий завершення IRP, і знаходиться він в черзі драйвера Kbdclass.

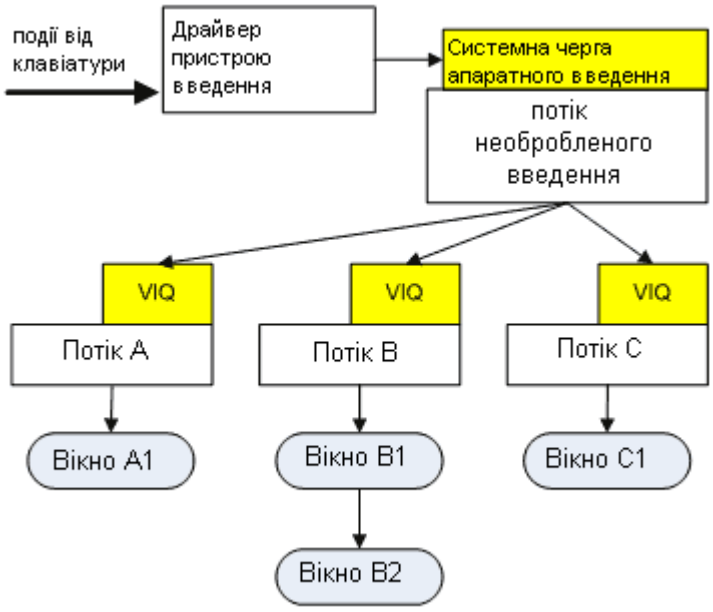


Рис. 2.2. Обробка клавіатурного введення в призначеному для користувача режимі

Всі клавіатурні події, що прийшли, поміщаються в системну чергу апаратного введення, після чого вони послідовно перетворяться в повідомлення Windows (типу WM_KEY*, WM_?BUTTON* або WM_MOUSEMOVE) і ставляться в кінець черги віртуального введення (virtualized input queue, VIQ) активного потоку. У повідомленнях Windows скан-коди клавіш замінюються на коди віртуальних клавіш, відповідні не розташуванню клавіші на клавіатурі, а дії, яка виконує ця клавіша. Механізм перетворення кодів залежить від активної розкладки клавіатури, одночасних натиснень інших клавіш (наприклад, SHIFT) і інших чинників.

Коли користувач входить в систему, процес Windows Explorer породжує потік, який створює панель завдань і робочий стіл (WinSta0_RIT). Цей потік прив'язується до RIT. Якщо користувач запускає MS Word, то його потік, що створив вікно, негайно підключиться до RIT. Після цього потік, належний Explorer, відключається від RIT, оскільки одноразово з RIT може бути зв'язаний тільки один потік. При натисненні клавіші в SHIFT з'явиться відповідний елемент, що приведе до того, що RIT прокинеться,

перетворить подію апаратного введення в повідомлення від клавіатури і помістить його в VIQ потоку додатку MS Word.

Таким чином, алгоритм проходження сигналу від натиснення користувачем клавіш на клавіатурі до появи символів на екрані можна представити таким чином:

1. Операційна система при старті створює в системному процесі csrss.exe потік необробленого введення і системну чергу апаратного введення.
2. Потік необробленого введення в циклі посилає запити читання драйверу класу клавіатури, які залишаються в стані очікування до появи подій від клавіатури.
3. Коли користувач натискає або відпускає клавішу на клавіатурі, мікроконтролер клавіатури фіксує натиснення/відпуск клавіші і посилає в центральний комп'ютер скан-код натиснутої клавіші і запит на переривання.
4. Системний контролер клавіатури отримує скан-код, проводить перетворення скан-коду, робить його доступним на порту введення-виводу 60h і генерує апаратне переривання центрального процесора.
5. Контролер переривань викликає процедуру обробки переривання IRQ 1, — ISR, зареєстровану в системі функціональним драйвером клавіатури i8042prt.
6. Процедура ISR прочитує з внутрішньої черги контроллера клавіатури дані, що з'явилися, переводить скан-коди в коди віртуальних клавіш (незалежні значення, визначені системою) і ставить в чергу виклик відкладеної процедури i8042KeyboardIsrDpc.
7. Як тільки це стає можливим, система викликає DPC, яка у свою чергу викликає процедуру зворотного виклику KeyboardClassServiceCallback, зареєстровану драйвером класу клавіатури Kbdclass.
8. Процедура KeyboardClassServiceCallback витягує зі своєї черги чекаючий завершення запит від потоку необробленого введення і повертає в нім інформацію про натиснуту клавішу.
9. Потік необробленого введення зберігає отриману інформацію в системній черзі апаратного введення і формує на її основі базові клавіатурні повідомлення Windows WM_KEYDOWN, WM_KEYUP, які ставляться в кінець черги віртуального введення VIQ активного потоку.
10. Цикл обробки повідомлень потоку видаляє повідомлення з черги і передає його відповідній віконній процедурі для обробки. При цьому може бути викликана системна функція TranslateMessage, яка на основі базових клавіатурних повідомлень створює додаткові «символьні» повідомлення WM_CHAR, WM_SYSCHAR, WM_DEADCHAR і WM_SYSDEADCHAR.