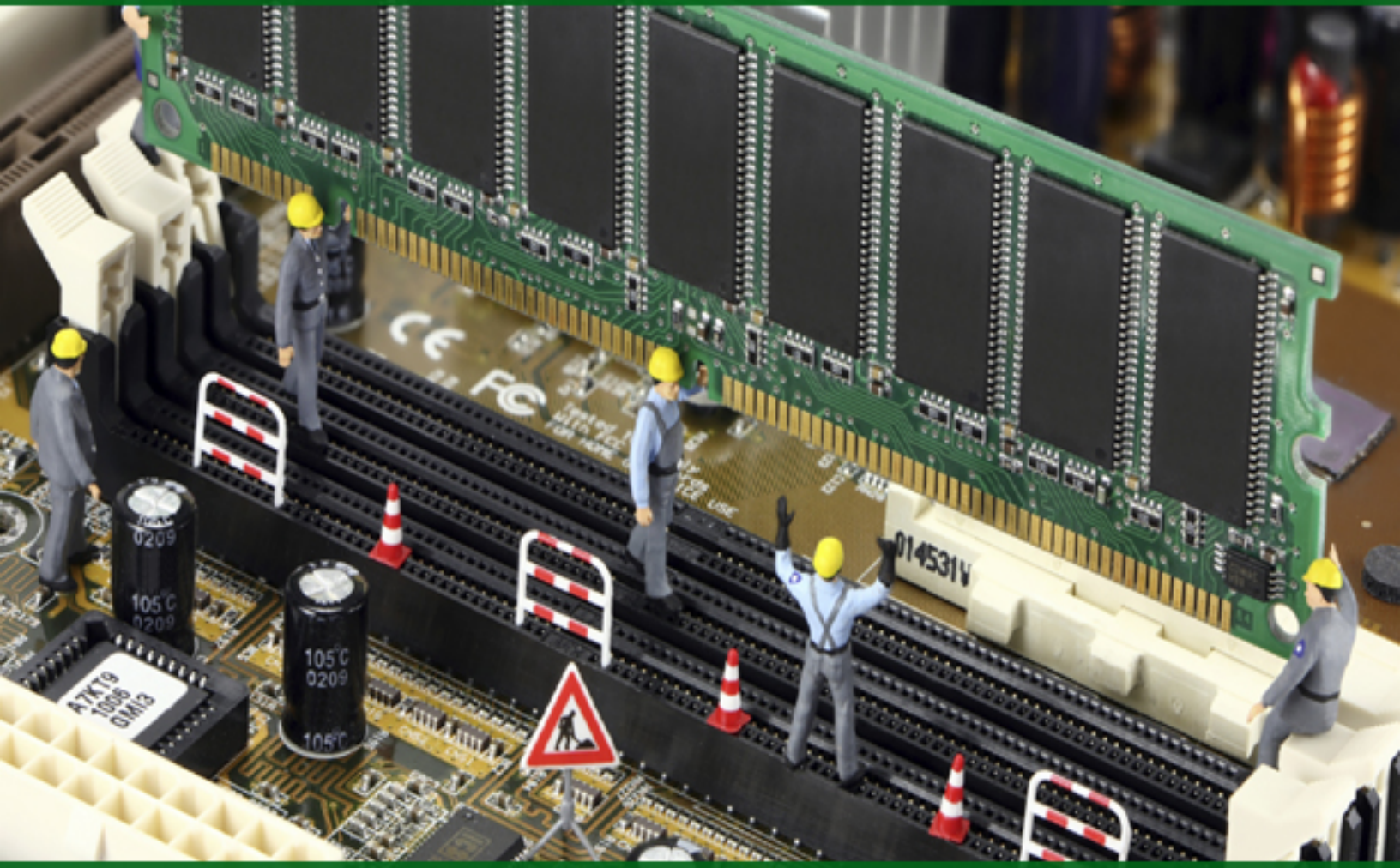
Computer Memory and its Management



Josephina Duong

Anton Harlow

First Edition, 2012

ISBN 978-81-323-1269-7

Published by:
College Publishing House
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

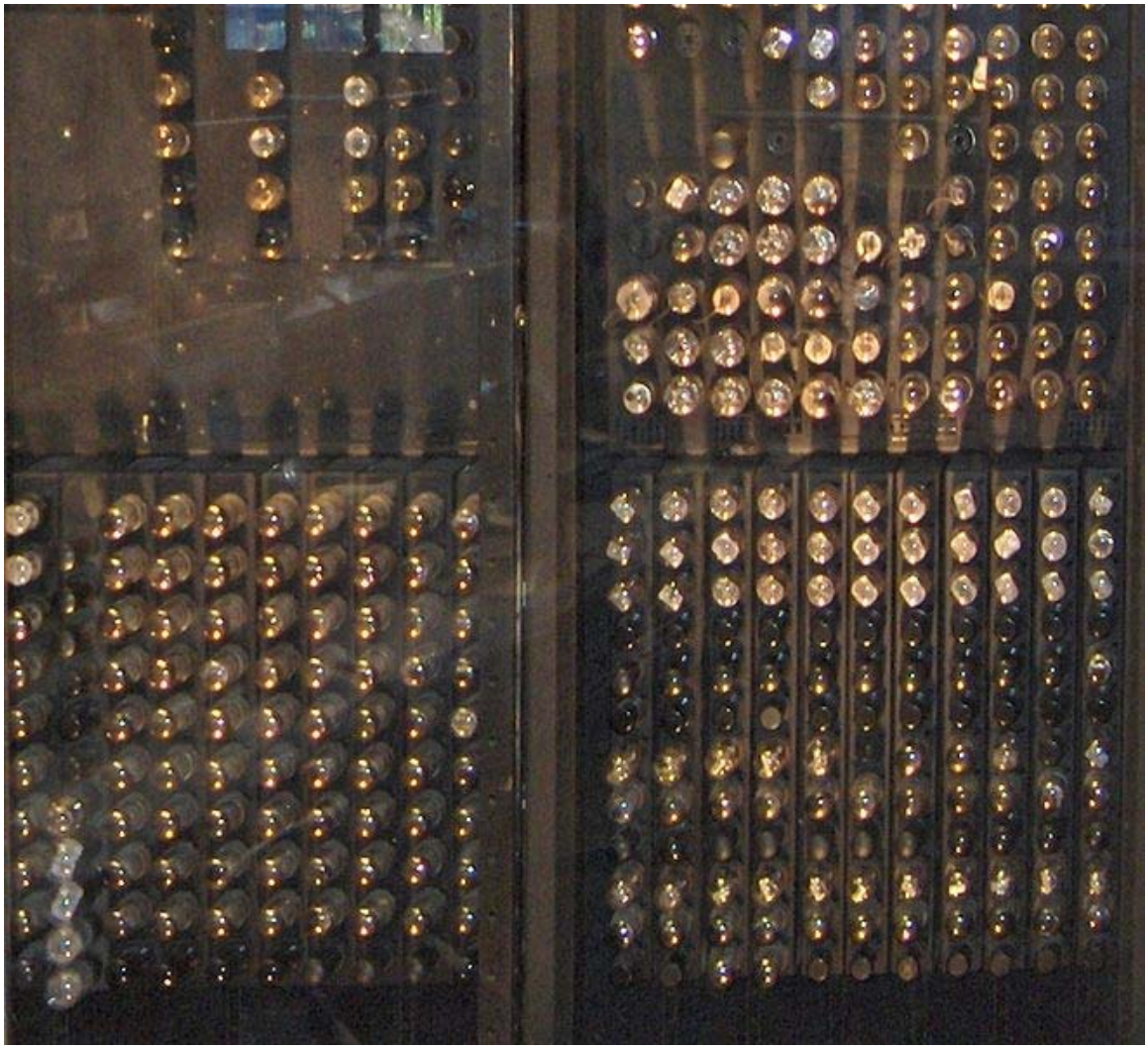
Computer Memory

In computing, **memory** refers to the state information of a computing system, as it is kept active in some physical structure. The term "memory" is used for the information in physical systems which are fast (i.e. RAM), as a distinction from physical systems which are slow to access (i.e. data storage). By design, the term "memory" refers to temporary state devices, whereas the term "storage" is reserved for permanent data. Advances in storage technology have blurred the distinction a bit —memory kept on what is conventionally a storage system is called "virtual memory".

Colloquially, **computer memory** refers to the physical devices used to store data or programs (sequences of instructions) on a temporary or permanent basis for use in an electronic digital computer. Computers represent information in binary code, written as sequences of 0s and 1s. Each binary digit (or "bit") may be stored by any physical system that can be in either of two stable states, to represent 0 and 1. Such a system is called bistable. This could be an on-off switch, an electrical capacitor that can store or lose a charge, a magnet with its polarity up or down, or a surface that can have a pit or not. Today, capacitors and transistors, functioning as tiny electrical switches, are used for temporary storage, and either disks or tape with a magnetic coating, or plastic discs with patterns of pits are used for long-term storage.

Computer memory is usually meant to refer to the semiconductor technology that is used to store information in electronic devices. Current primary computer memory makes use of integrated circuits consisting of silicon-based transistors. There are two main types of memory: volatile and non-volatile.

History



Detail of the back of a section of ENIAC, showing vacuum tubes

In the early 1940s, memory technology mostly permitted a capacity of a few bytes. The first programmable digital computer, the ENIAC, using thousands of octal-base radio vacuum tubes, could perform simple calculations involving 20 numbers of ten decimal digits which were held in the vacuum tube accumulators.

The next significant advance in computer memory was with acoustic delay line memory developed by J. Presper Eckert in the early 1940s. Through the construction of a glass tube filled with mercury and plugged at each end with a quartz crystal, delay lines could store bits of information within the quartz and transfer it through sound waves propagating through mercury. Delay line memory would be limited to a capacity of up to a few hundred thousand bits to remain efficient.

Two alternatives to the delay line, the Williams tube and Selectron tube, were developed in 1946, both using electron beams in glass tubes as means of storage. Using cathode ray tubes, Fred Williams would invent the Williams tube, which would be the first random access computer memory. The Williams tube would prove to be advantageous to the Selectron tube because of its greater capacity (the Selectron was limited to 256 bits, while the Williams tube could store thousands) and being less expensive. The Williams tube would nevertheless prove to be frustratingly sensitive to environmental disturbances.

Efforts began in the late 1940s to find non-volatile memory. Jay Forrester, Jan A. Rajchman and An Wang would be credited with the development of magnetic core memory, which would allow for recall of memory after power loss. Magnetic core memory would become the dominant form of memory until the development of transistor based memory in the late 1960s.

Volatile memory

Volatile memory is computer memory that requires power to maintain the stored information. Current semiconductor volatile memory technology is usually either static RAM or dynamic RAM. Static RAM exhibits data remanence, but is still volatile, since all data is lost when memory is not powered. Whereas, dynamic RAM allows data to be leaked and disappear automatically without a refreshing. This difference accounts for the substantial price difference between the two types of memory, and consequently renders SRAM an impractical choice for system memory. Upcoming volatile memory technologies that hope to replace or compete with SRAM and DRAM include Z-RAM, TTRAM, A-RAM and ETA RAM.

Non-volatile memory

Non-volatile memory is computer memory that can retain the stored information even when not powered. Examples of non-volatile memory include read-only memory, flash memory, most types of magnetic computer storage devices (e.g. hard disks, floppy discs and magnetic tape), optical discs, and early computer storage methods such as paper tape and punched cards. Upcoming non-volatile memory technologies include FeRAM, CBRAM, PRAM, SONOS, RRAM, Racetrack memory, NRAM and Millipede.is called a ROM.

Management of memory

Proper management of memory is vital for a computer system to operate properly. Modern operating systems have complex systems to properly manage memory. Failure to do so can lead to bugs, slow performance, and at worst case, takeover by viruses and malicious software.

Nearly everything a computer programmer does requires him or her to consider how to manage memory. Even storing a number in memory requires the programmer to specify how the memory should store it.

Memory management bugs

Improper management of memory is a common cause of bugs.

- In arithmetic overflow, a calculation results in a number larger than the allocated memory permits. For example, an 8-bit integer allows the numbers −127 to +127. If its value is 127 and it is instructed to add one, the computer can not store the number 128 in that space. Such a case will result in undesired operation, such as changing the number's value to −128 instead of +128.

- A memory leak occurs when a program requests memory from the operating system and never returns the memory when it's done with it. A program with this bug will gradually require more and more memory until the program fails as it runs out.

- A segmentation fault results when a program tries to access memory that it has no permission to access. Generally a program doing so will be terminated by the operating system.

- Buffer overflow means that a program writes data to the end of its allocated space and then continues to write data to memory that belongs to other programs. This may result in erratic program behavior, including memory access errors, incorrect results, a crash, or a breach of system security. They are thus the basis of many software vulnerabilities and can be maliciously exploited.

Early computer systems

In early computer systems, programs typically specified the location to write memory and what data to put there. This location was a physical location on the actual memory hardware. The slow processing of such computers didn't allow for the complex memory management systems used today. Also, as most such systems were single-task, sophisticated systems weren't required as much.

This approach has its pitfalls. If the location specified is incorrect, this will cause the computer to write the data to some other part of the program. The results of an error like this are unpredictable. In some cases, the incorrect data might overwrite memory used by the operating system. Computer crackers can take advantage of this to create viruses and malware.

Virtual memory

Virtual memory is a system where all physical memory is controlled by the operating system. When a program needs memory, it requests it from the operating system. The operating system then decides what physical location to place the memory in.

This offers several advantages. Computer programmers no longer need to worry about where the memory is physically stored or whether the user's computer will have enough memory. It also allows multiple types of memory to be used. For example, some memory can be stored in physical RAM chips while other memory is stored on a hard drive. This drastically increases the amount of memory available to programs. The operating system will place actively used memory in physical RAM, which is much faster than hard disks. When the amount of RAM is not sufficient to run all the current programs, it can result in a situation where the computer spends more time moving memory from RAM to disk and back than it does accomplishing tasks; this is known as thrashing.

Virtual memory systems usually include protected memory, but this is not always the case.

Protected memory

Protected memory is a system where each program is given an area of memory to use and is not permitted to go outside that range. Use of protected memory greatly enhances both the reliability and security of a computer system.

Without protected memory, it is possible that a bug in one program will alter the memory used by another program. This will cause that other program to run off of corrupted memory with unpredictable results. If the operating system's memory is corrupted, the entire computer system may crash and need to be rebooted. At times programs intentionally alter the memory used by other programs. This is done by viruses and malware to take over computers.

Protected memory assigns programs their own areas of memory. If the operating system detects that a program has tried to alter memory that doesn't belong to it, the program is terminated. This way, only the offending program crashes, and other programs are not affected by the error.

Chapter 2

Memory Protection

Memory protection is a way to control memory access rights on a computer, and is a part of most modern operating systems. The main purpose of memory protection is to prevent a process from accessing memory that has not been allocated to it. This prevents a bug within a process from affecting other processes, or the operating system itself. Memory protection is a behavior that is distinct from ASLR and the NX bit.

Methods

Segmentation

Segmentation refers to dividing a computer's memory into segments.

The x86 architecture has multiple segmentation features, which are helpful for using protected memory on this architecture. On the x86 processor architecture, the Global Descriptor Table and Local Descriptor Tables can be used to reference segments in the computer's memory. Pointers to memory segments on x86 processors can also be stored in the processor's segment registers. Initially x86 processors had 4 segment registers, CS (code segment), SS (stack segment), DS (data segment) and ES (extra segment); later another two segment registers were added – FS and GS.

Paged virtual memory

In paging, the memory address space is divided into equal, small pieces, called pages. Using a virtual memory mechanism, each page can be made to reside in any location of the physical memory, or be flagged as being protected. Virtual memory makes it possible to have a linear virtual memory address space and to use it to access blocks fragmented over physical memory address space.

Most computer architectures based on pages, most notably x86 architecture, also use pages for memory protection.

A page table is used for mapping virtual memory to physical memory. The page table is usually invisible to the process. Page tables make it easier to allocate new memory, as each new page can be allocated from anywhere in physical memory.

By such design, it is impossible for an application to access a page that has not been explicitly allocated to it, simply because *any* memory address, even a completely random one, that application may decide to use, either points to an allocated page, or generates a page fault (PF). Unallocated pages simply do not have any addresses from the application point of view.

As a side note, a PF may not be a fatal occurrence. Page faults are used not only for memory protection, but also in another interesting way: the OS may intercept the PF, and may load a page that has been previously swapped out to disk, and resume execution of the application which had caused the page fault. This way, the application receives the memory page as needed. This scheme, known as swapped virtual memory, allows in-memory data not currently in use to be moved to disk storage and back in a way which is transparent to applications, to increase overall memory capacity.

Protection keys

A protection key mechanism divides physical memory up into blocks of a particular size (e.g., 2 kiB), each of which has an associated numerical value called a protection key. Each process also has a protection key value associated with it. On a memory access the hardware checks that the current process's protection key matches the value associated with the memory block being accessed; if not, an exception occurs. This mechanism was used in the System/360 architecture.

The System/360 protection keys described above are associated with physical addresses. This is different from the protection key mechanism used by processors such as the Intel Itanium and the Hewlett-Packard Precision Architecture (HP/PA, also known as PA-RISC), which are associated with virtual addresses, and which allow multiple keys per process.

In the Itanium and PA processor architectures, translations (TLB entries) have *keys* (Itanium) or *access ids* (PA) associated with them. A running process has several protection key registers (16 for Itanium, 4 for HP PA). A translation selected by the virtual address has its key compared to each of the protection key registers. If any of them match (plus other possible checks), the access is permitted. If none match, a fault or exception is generated. The software fault handler can, if desired, check the missing key against a larger list of keys maintained by software; thus, the protection key registers inside the processor may be treated as a software managed cache of a larger list of keys associated with a process.

PA has 15–18 bits of key; Itanium mandates at least 18. Keys are usually associated with *protection domains*, such as libraries, modules, etc.

Simulated segmentation

Simulation is use of a monitoring program to interpret the machine code instructions of some computer. Such an Instruction Set Simulator can provide memory protection by

using a segmentation-like scheme and validating the target address and length of each instruction in real time before actually executing them. The simulator must calculate the target address and length and compare this against a list of valid address ranges that it holds concerning the thread's environment, such as any dynamic memory blocks acquired since the thread's inception plus any valid shared static memory slots. The meaning of "valid" may change throughout the thread's life depending upon context: it may sometimes be allowed to alter a static block of storage, and sometimes not, depending upon the current mode of execution which may or may not depend on a storage key or supervisor state.

It is generally not advisable to use this method of memory protection where adequate facilities exist on a CPU, as this takes valuable processing power from the computer. However it is generally used for debugging and testing purposes to provide an extra fine level of granularity to otherwise generic storage violations and can indicate precisely which instruction is attempting to overwrite the particular section of storage which may have the same storage key as unprotected storage. Early IBM teleprocessing systems, such as CICS, multi-threaded commercial transactions in shared and unprotected storage for around 20 years.

Capability-based addressing

Capability-based addressing is a method of memory protection that is unused in modern commercial computers. In this, pointers are replaced by protected objects (called *capabilities*) that can only be created via using privileged instructions which may only be executed by the kernel, or some other process authorized to do so. This effectively lets the kernel control which processes may access which objects in memory, with no need to use separate address spaces or context switches. Capabilities have never gained mainstream adoption in commercial hardware, but they are widely used in research systems such as KeyKOS and its successors, and are used conceptually as the basis for some virtual machines, most notably Smalltalk and Java.

Measures

A useful estimation of the protection level of a particular implementation, is to measure how closely it adheres to the principle of minimum privilege.

Memory protection in different operating systems

Different Operating Systems use different forms of memory protection or separation. True memory separation was not used in home computer operating systems until Windows XP and Mac OS X, which were released in 2001. It is possible for processes to access System Memory in the Windows 9x family of Operating Systems.

Some operating systems that do implement memory protection include

- Microsoft Windows family from Windows NT 3.1

- most Unix-like systems, including
    - Solaris
    - Linux
    - BSD
    - Mac OS X
    - GNU Hurd

Chapter 3

Dynamic Random-Access Memory

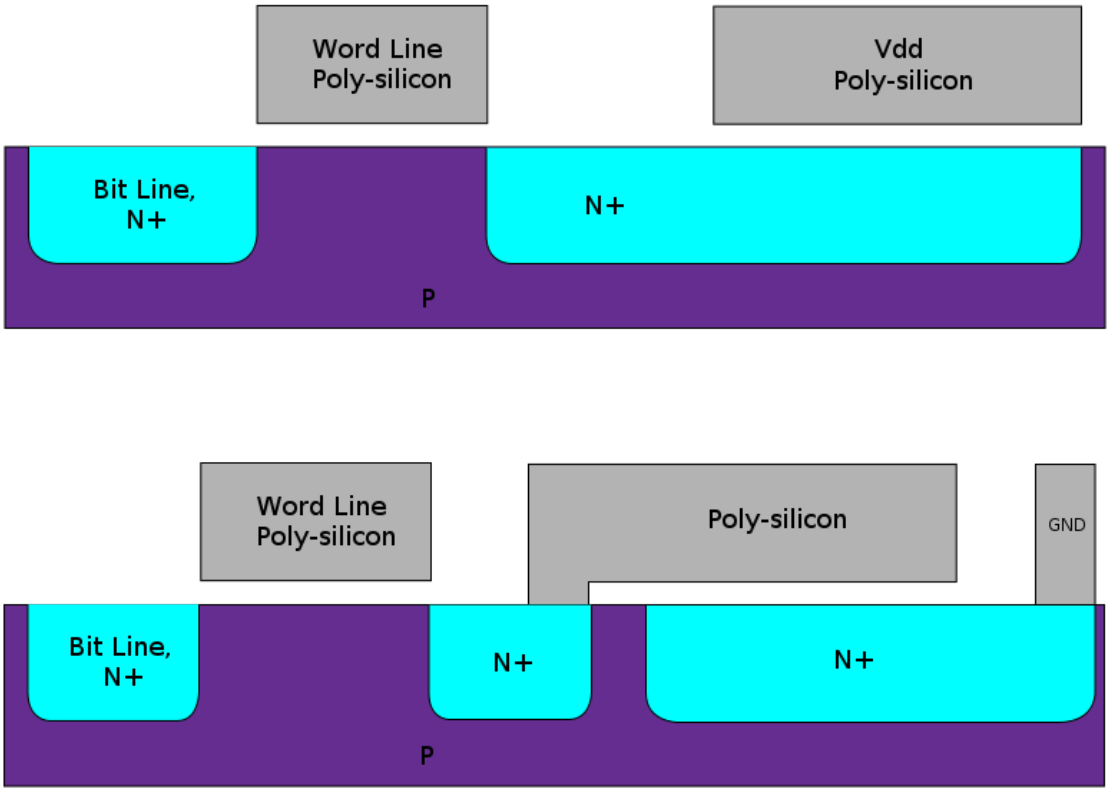
Dynamic random-access memory (**DRAM**) is a type of random-access memory that stores each bit of data in a separate capacitor within an integrated circuit. The capacitor can be either charged or discharged; these two states are taken to represent the two values of a bit, conventionally called 0 and 1. Since capacitors leak charge, the information eventually fades unless the capacitor charge is refreshed periodically. Because of this refresh requirement, it is a *dynamic* memory as opposed to SRAM and other *static* memory.

The main memory (the "RAM") in personal computers is Dynamic RAM (DRAM), as is the "RAM" of home game consoles (PlayStation, Xbox 360 and Wii), laptop, notebook and workstation computers.

The advantage of DRAM is its structural simplicity: only one transistor and a capacitor are required per bit, compared to six transistors in SRAM. This allows DRAM to reach very high densities. Unlike flash memory, DRAM is volatile memory (cf. non-volatile memory), since it loses its data quickly when power is removed. The transistors and capacitors used are extremely small—millions can fit on a single memory chip.

History

The cryptanalytic machine code-named *"Aquarius"* used at Bletchley Park during World War II incorporated a hard-wired dynamic memory. Paper tape was read and the characters on it "were remembered in a dynamic store. ... The store used a large bank of capacitors, which were either charged or not, a charged capacitor representing cross (1) and an uncharged capacitor dot (0). Since the charge gradually leaked away, a periodic pulse was applied to top up those still charged (hence the term 'dynamic')".



Schematic drawing of original designs of DRAM patented in 1968

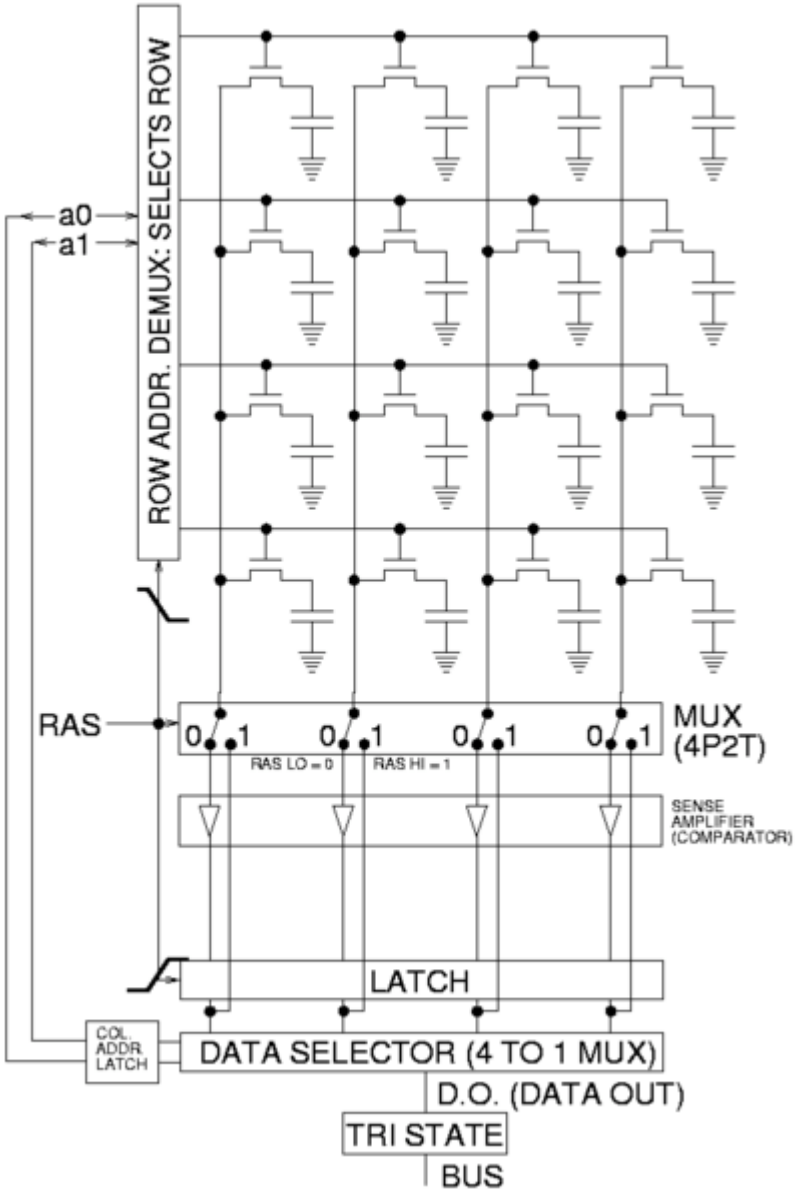
In 1964, Arnold Farber and Eugene Schlig, working for IBM, created a memory cell that was hard wired. It used a transistor gate and tunnel diode latch. They replaced the latch with two transistors and two resistors. This configuration became known as the Farber-Schlig cell. In 1965, Benjamin Agusta and his team at IBM managed to create a 16-bit silicon memory chip based on the Farber-Schlig cell, which consisted of 80 transistors, 64 resistors, and four diodes. In 1966, DRAM was invented by Dr. Robert Dennard at the IBM Thomas J. Watson Research Center. He was awarded U.S. patent number 3,387,286 in 1968. Capacitors had been used for earlier memory schemes such as the drum of the Atanasoff–Berry Computer, the Williams tube and the Selectron tube.

The Toshiba *"Toscal" BC-1411* electronic calculator, which went into production in November 1965, used a form of dynamic RAM built from discrete components.

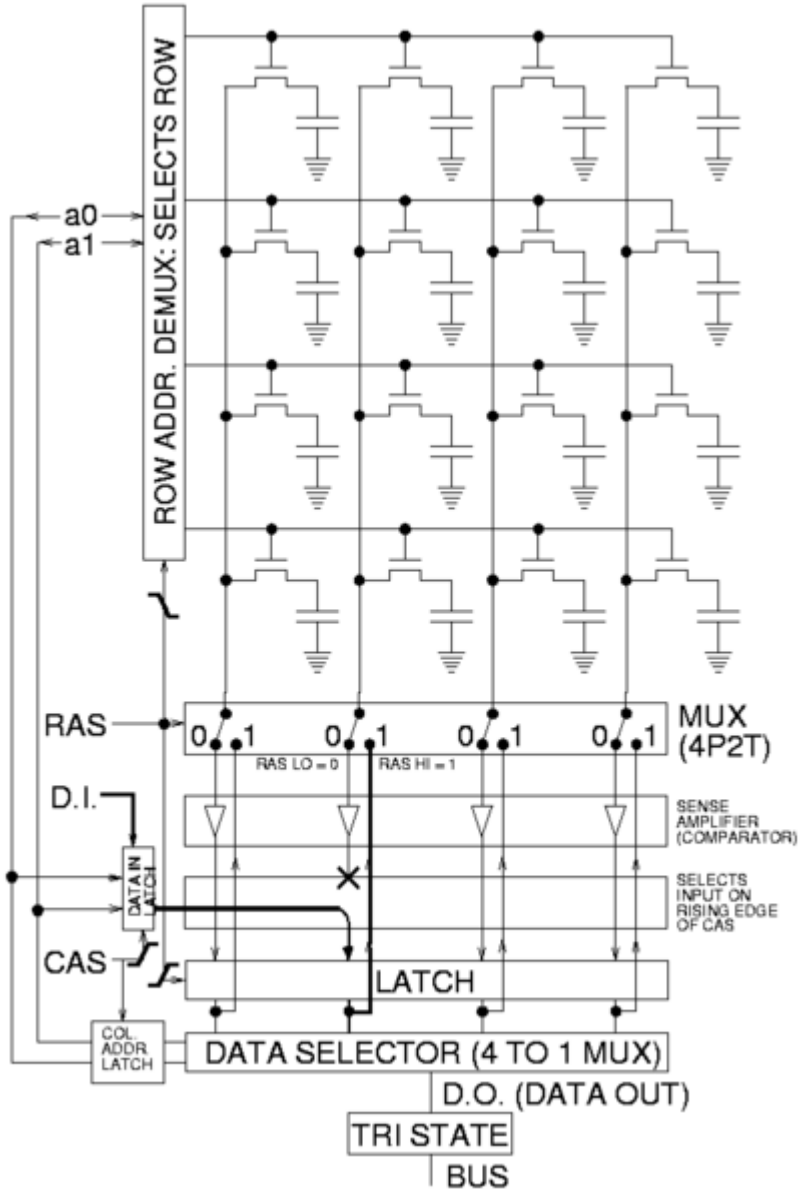
In 1969, Honeywell asked Intel to make a DRAM using a 3-transistor cell that they had developed. This became the Intel 1102 (1024x1) in early 1970. However the 1102 had many problems, prompting Intel to begin work on their own improved design (in secrecy to avoid conflict with Honeywell). This became the first commercially-available DRAM memory, the Intel 1103 (1024x1) in October 1970 (despite initial problems with low yield until the fifth revision of the masks). The 1103 was designed by Joel Karp and laid out by Barbara Maness.

The first DRAM with multiplexed row and column address lines was the Mostek MK4096 (4096x1) designed by Robert Proebsting and introduced in 1973. This addressing scheme, a radical advance, enabled it to fit into packages with fewer pins, a cost advantage that would grow with every jump in memory size. The MK4096 proved to be a very robust design for customer applications. At the 16K density, the cost advantage increased, and the Mostek MK4116 16K DRAM, introduced in 1976, achieved greater than 75% worldwide DRAM market share. However, as density increased to 64K in the early 80s, Mostek was overtaken by Japanese DRAM manufacturers selling higher quality DRAMs using the same multiplexing scheme at below-cost prices.

Operation principle



Principle of operation of DRAM read, for simple 4 by 4 array



Principle of operation of DRAM write, for simple 4 by 4 array

DRAM is usually arranged in a square array of one capacitor and transistor per data bit storage cell. The illustrations to the right show a simple example with only 4 by 4 cells (Modern DRAM matrix are many thousands of cells in height and width).

The long horizontal lines connecting each row are known as Word Lines. Each column of cells is actually composed of two bit lines, each one connected to every other storage cell in the column. (The illustration to the right does not include this important detail.) They are generally known as the + and − bit lines. A sense amplifier is essentially a pair of cross-connected inverters between the bit lines. That is, the first inverter is connected from the + bit line to the − bit line, and the second is connected from the − bit line to the

+ bit line. This is an example of positive feedback, and the arrangement is only stable after one bit line is high and one bit line is low.

To read a bit from a column, the following operations take place:

1. The sense amplifier is disconnected, then the bit lines are precharged to exactly equal voltages that are in-between high and low logic levels. The bit lines are physically symmetrical to keep the capacitance as equal and therefore the voltages as equal as possible.
2. The precharge circuit is switched off. Because the bit lines are relatively long, they have enough capacitance to maintain the pre-charged voltage for a brief time. This is an example of dynamic logic.
3. The desired row's word line is then driven high to connect a cell's storage capacitor to its bit line. This causes the transistor to conduct, transferring charge between the storage cell and the connected bit line. If the storage cell capacitor is discharged, it will greatly decrease the voltage on the bit-line as the precharge is transferred to the storage capacitor. If the storage cell is charged, the bit-line voltage decreases only slightly; this because every effort is made to keep the capacitance of the storage cells high and the capacitance of the bit lines low.
4. The sense amplifier is switched on. The positive feedback takes over and amplifies the small voltage difference between bit-lines until one bit line is fully at the lowest voltage and the other is at the maximum high voltage. Once this has happened, the row is "open" (the desired cell data is available).
5. All columns are sensed in simultaneously and the result sampled into the data latch. A provided Column address then selects which latch bit to connect to the external circuit. Many reads can be performed quickly without delay sense for the open row, all data has already been sensed and latched.
6. While reading all columns proceeds (the normal and desirable method because it most quickly provides data), current is flowing back up the bit lines from the sense amplifiers to the storage cells. This reinforces (i.e. "refreshes") the charge in the storage cell by increasing the voltage in the storage capacitor if it was charged to begin with, or by keeping it discharged if it was empty. Note that due to the length of the bit lines creating a fairly long propagation delay for the charge to be transferred, this takes significant time beyond the end of sense amplification, and thus overlaps with one or more column reads.
7. When done with the reading all the columns in the current row, the word line is switched off to disconnect the cell storage capacitors (the row is "closed"), the sense amplifier is switched off, and the bit lines are precharged again.

To write to memory, the row is opened and a given column's sense amplifier is temporarily forced to the desired high or low voltage state, thus it drives the bit line to charge or discharge the cell storage capacitor to the desired value. Due to positive feedback, the amplifier will then hold it stable even after the forcing is removed. During a write to a particular cell, all the columns in a row are sensed simultaneously just as in reading, a single column's cell storage capacitor charge is changed, and then the entire row is written back in, as illustrated in the figure to the right.

Typically, manufacturers specify that each row must be have its storage cell capacitors refreshed every 64 ms or less, as defined by the JEDEC (Foundation for developing Semiconductor Standards) standard. Refresh logic is provided in a DRAM controller which automates the periodic refresh, that is no software or other hardware has to perform it. This makes the controller's logic circuit more complicated, but this drawback is outweighed by the fact that DRAM is much cheaper per storage cell and because each storage cell is very simple, DRAM has much greater capacity per geographic area than SRAM.

Some systems refresh every row in a burst of activity involving all rows every 64 ms. Other systems refresh one row at a time staggered through out the 64 ms interval. For example, a system with $2^{13} = 8192$ rows would require a staggered refresh rate of one row every 7.8 µs which is 64 ms divided by 8192 rows. A few real-time systems refresh a portion of memory at a time determined by an external timer function that governs the operation of the rest of a system, such as the vertical blanking interval that occurs every 10–20 ms in video equipment. All methods require some sort of counter to keep track of which row is the next to be refreshed. Most DRAM chips include that counter. Older types require external refresh logic to hold the counter. (Under some conditions, most of the data in DRAM can be recovered even if the DRAM has not been refreshed for several minutes.

Memory timing

There are many numbers required to describe the timing of DRAM operation. Here are some examples for two timing grades of asynchronous DRAM, from a data sheet published in 1998:

	"50 ns"	"60 ns"	Description
t_{RC}	84 ns	104 ns	Random read or write cycle time (from one full /RAS cycle to another)
t_{RAC}	50 ns	60 ns	Access time: /RAS low to valid data out
t_{RCD}	11 ns	14 ns	/RAS low to /CAS low time
t_{RAS}	50 ns	60 ns	/RAS pulse width (minimum /RAS low time)
t_{RP}	30 ns	40 ns	/RAS precharge time (minimum /RAS high time)
t_{PC}	20 ns	25 ns	Page-mode read or write cycle time (/CAS to /CAS)
t_{AA}	25 ns	30 ns	Access time: Column address valid to valid data out (includes address setup time before /CAS low)
t_{CAC}	13 ns	15 ns	Access time: /CAS low to valid data out
t_{CAS}	8 ns	10 ns	/CAS low pulse width minimum

Thus, the generally quoted number is the /RAS access time. This is the time to read a random bit from a precharged DRAM array. The time to read additional bits from an open page is much less.

When such a RAM is accessed by clocked logic, the times are generally rounded up to the nearest clock cycle. For example, when accessed by a 100 MHz state machine (i.e. a 10 ns clock), the 50 ns DRAM can perform the first read in five clock cycles, and additional reads within the same page every two clock cycles. This was generally described as "5 - 2 - 2 - 2" timing, as bursts of four reads within a page were common.

When describing synchronous memory, timing is described by clock cycle counts separated by hyphens. These numbers represent t_{CL} - t_{RCD} - t_{RP} - t_{RAS} in multiples of the DRAM clock cycle time. Note that this is half of the data transfer rate when double data rate signaling is used. JEDEC standard PC3200 timing is 3 - 4 - 4 - 8 with a 200 MHz clock, while premium-priced high performance PC3200 DDR DRAM DIMM might be operated at 2 - 2 - 2 - 5 timing.

	PC-3200 (DDR-400)				PC2-6400 (DDR2-800)				PC3-12800 (DDR3-1600)				Description
	Typical		Fast		Typical		Fast		Typical		Fast		
	cycles	time	cycles	time	cycles	time	cycles	time	cycles	time	cycles	time	
t_{CL}	3	15 ns	2	10 ns	5	12.5 ns	4	10 ns	9	11.25 ns	8	10 ns	/CAS low to valid data out (equivalent to t_{CAC})
t_{RCD}	4	20 ns	2	10 ns	5	12.5 ns	4	10 ns	9	11.25 ns	8	10 ns	/RAS low to /CAS low time
t_{RP}	4	20 ns	2	10 ns	5	12.5 ns	4	10 ns	9	11.25 ns	8	10 ns	/RAS precharge time (minimum precharge to active time)
t_{RAS}	8	40 ns	5	25 ns	16	40 ns	12	30 ns	27	33.75 ns	24	30 ns	Row active time (minimum active to precharge time)

It is worth noting that the improvement over 11 years is not that significant. Minimum random access time has improved from t_{RAC} = 50 ns to t_{RCD} + t_{CL} = 22.5 ns, and even the premium 20 ns variety is only 2.5 times better. CAS latency has improved even less, from t_{CAC} = 13 ns to 10 ns. However, the DDR3 memory does achieve 32 times higher bandwidth; due to internal pipelining and wide data paths, it can output two words every 1.25 ns (1600 Mword/s), while the EDO DRAM can output one word per t_{PC} = 20 ns (50 Mword/s).

Timing abbreviations

- t_{CL} – CAS latency
- t_{CR} – Command rate

- t_{PTP} – precharge to precharge delay
- t_{RAS} – RAS active time
- t_{RCD} – RAS to CAS delay
- t_{REF} – Refresh period
- t_{RFC} – Row refresh cycle time
- t_{RP} – RAS precharge
- t_{RRD} – RAS to RAS delay
- t_{RTP} – Read to precharge delay
- t_{RTR} – Read to read delay
- t_{RTW} – Read to write delay
- t_{WR} – Write recovery time
- t_{WTP} – Write to precharge delay
- t_{WTR} – Write to read delay
- t_{WTW} – Write to write delay

Errors and error correction

Electrical or magnetic interference inside a computer system can cause a single bit of DRAM to spontaneously flip to the opposite state. It was initially thought that this was mainly due to alpha particles emitted by contaminants in chip packaging material, but research has shown that the majority of one-off ("soft") errors in DRAM chips occur as a result of background radiation, chiefly neutrons from cosmic ray secondaries, which may change the contents of one or more memory cells or interfere with the circuitry used to read/write them. There was some concern that as DRAM density increases further, and thus the components on DRAM chips get smaller, while at the same time operating voltages continue to fall, DRAM chips will be affected by such radiation more frequently—since lower energy particles will be able to change a memory cell's state. On the other hand, smaller cells make smaller targets, and moves to technologies such as SOI may make individual cells less susceptible and so counteract, or even reverse this trend. Recent studies show that single event upsets due to cosmic radiation have been dropping dramatically with process geometry and previous concerns over increasing bit cell error rates are unfounded.

This problem can be mitigated by using DRAM modules that include extra memory bits and memory controllers that exploit these bits. These extra bits are used to record parity or to use an error-correcting code (ECC). Parity allows the detection of all single-bit errors (actually, any odd number of wrong bits). The most common error correcting code, a SECDED Hamming code, allows a single-bit error to be corrected and (in the usual configuration, with an extra parity bit) double-bit errors to be detected.

Seymour Cray famously said "parity is for farmers" when asked why he left this out of the CDC 6600. He included parity in the CDC 7600. The original IBM PC and all PCs until the early 1990s used parity checking. Later ones mostly did not. Wider memory buses make parity and especially ECC more affordable. Many current microprocessor memory controllers, including almost all AMD 64-bit offerings, support ECC, but many motherboards and in particular those using low-end chipsets do not.

An ECC-capable memory controller as used in many modern PCs can typically detect and correct errors of a single bit per 64-bit "word" (the unit of bus transfer), and detect (but not correct) errors of two bits per 64-bit word. Some systems also 'scrub' the errors, by writing the corrected version back to memory. The BIOS in some computers, and operating systems such as Linux, allow counting of detected and corrected memory errors, in part to help identify failing memory modules before the problem becomes catastrophic.

Typically, only machines intended for server use support ECC. Mainboards intended for desktop (rather than server) machines typically do not support ECC, and even those that do support ECC will be shipped with it disabled to allow use of non-ECC memory. Most modern PCs do not support ECC at all as can be seen by examining computer and motherboard specifications; those that do are often supplied with memory modules that do not support parity or ECC. It may be that most users opt for non-ECC systems and memory even when ECC is available. The most important reasons for this are:

- the higher cost of ECC memory (each bank is 9 memory chips compared to 8 for non-ECC memory, and more importantly there is more volume for non-ECC. In some cases the price ratio reduces to 9/8, as an example, on 2008/11/30, an ECC CL=5 unbuffered 2GB DDR2-667 DIMM cost $30 while the corresponding non-ECC part cost $28, a difference of 1/15, however some ECC modules cost twice as much as their non-ECC equivalents [Crucial CT12872Z40B and CT12864Z40B, Jan 2009]);
- the higher cost of a motherboard that supports ECC functionality in RAM;
- the additional time needed for ECC memory controllers to perform the error checking and possibly the correction steps, which may lead to an all-around performance hit of around 0.5–2 percent, depending on application; and
- simple ignorance of the issue.

Error detection and correction depends on an expectation of the kinds of errors that occur. Implicitly, we have assumed that the failure of each bit in a word of memory is independent and hence that two simultaneous errors are improbable. This used to be the case when memory chips were one bit wide (typical in the first half of the 1980s). Now many bits are in the same chip. This weakness does not seem to be widely addressed; one exception is Chipkill.

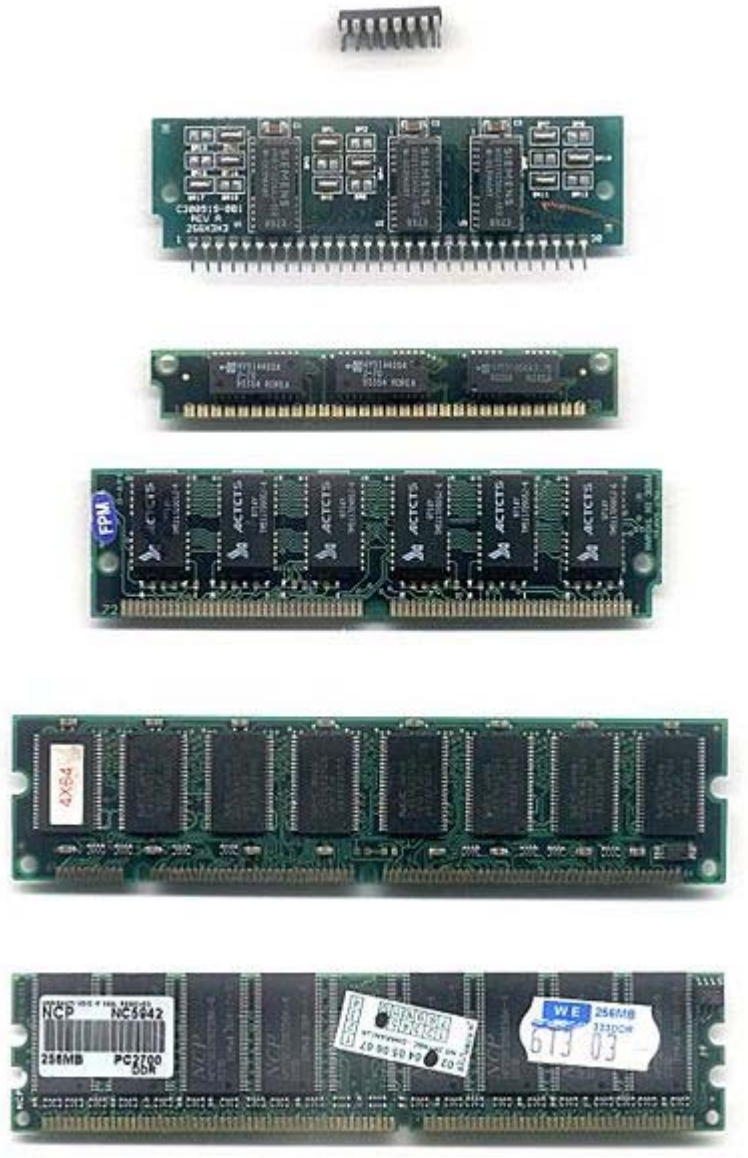
Recent tests give widely varying error rates with over 7 orders of magnitude difference, ranging from 10^{-10}–10^{-17} error/bit·h, roughly one bit error, per hour, per gigabyte of memory to one bit error, per century, per gigabyte of memory.

In most computers used for serious scientific or financial computing and as servers, ECC is the rule rather than the exception, as can be seen by examining manufacturers' specifications.

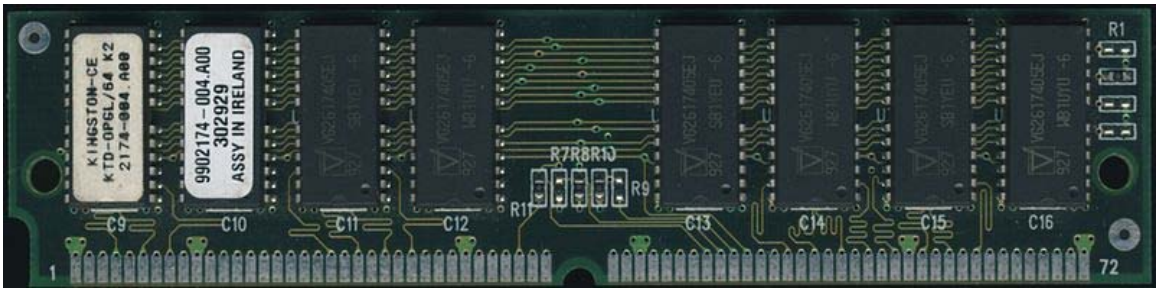
DRAM packaging

For economic reasons, the large (main) memories found in personal computers, workstations, and non-handheld game-consoles (such as PlayStation and Xbox) normally consists of dynamic RAM (DRAM). Other parts of the computer, such as cache memories and data buffers in hard disks, normally use static RAM (SRAM).

General DRAM packaging formats



Common DRAM packages. From top to bottom: DIP, SIPP, SIMM (30-pin), SIMM (72-pin), DIMM (168-pin), DDR DIMM (184-pin).



EDO DRAM memory module

Dynamic random access memory is produced as integrated circuits (ICs) bonded and mounted into plastic packages with metal pins for connection to control signals and buses. Today, these DRAM packages are in turn often assembled into plug-in modules for easier handling. Some standard module types are:

- **DRAM chip (Integrated Circuit or IC)**
  - Dual in-line Package (DIP)
- **DRAM (memory) modules**
  - Single In-line Pin Package (SIPP)
  - Single In-line Memory Module (SIMM)
  - Dual In-line Memory Module (DIMM)
  - Rambus In-line Memory Module (RIMM), technically DIMMs but called RIMMs due to their proprietary slot.
  - Small outline DIMM (SO-DIMM), about half the size of regular DIMMs, are mostly used in notebooks, small footprint PCs (such as Mini-ITX motherboards), upgradable office printers and networking hardware like routers. Comes in versions with:
    - 72-pin (32-bit)
    - 144-pin (64-bit) used for PC100/PC133 SDRAM
    - 200-pin (72-bit) used for DDR and DDR2
    - 240-pin (72-bit) used for DDR3
  - Small outline RIMM (SO-RIMM). Smaller version of the RIMM, used in laptops. Technically SO-DIMMs but called SO-RIMMs due to their proprietary slot.
- **Stacked vs. non-stacked RAM modules**
  - Stacked RAM modules contain two or more RAM chips stacked on top of each other. This allows large modules (like 512 MB or 1 GB SO-DIMM) to be manufactured using cheaper low density wafers. Stacked chip modules draw more power.

Common DRAM modules

Common DRAM packages as illustrated to the right, from top to bottom:

1. DIP 16-pin (DRAM chip, usually pre-FPRAM)
2. SIPP (usually FPRAM)

3. SIMM 30-pin (usually FPRAM)
4. SIMM 72-pin (often EDO RAM but FPM is not uncommon)
5. DIMM 168-pin (SDRAM)
6. DIMM 184-pin (DDR SDRAM)
7. RIMM 184-pin (RDRAM)
8. DIMM 240-pin (DDR2 SDRAM/DDR3 SDRAM)

Variations

While the fundamental DRAM cell and array has maintained the same basic structure (and performance) for many years, there have been many different interfaces for speaking with DRAM chips. When one speaks about "DRAM types", one is generally referring to the interface that is used.

Asynchronous DRAM

This is the basic form, from which all others derive. An asynchronous DRAM chip has power connections, some number of address inputs (typically 12), and a few (typically one or four) bidirectional data lines. There are four active low control signals:

- **/RAS**, the Row Address Strobe. The address inputs are captured on the falling edge of /RAS, and select a row to open. The row is held open as long as /RAS is low.
- **/CAS**, the Column Address Strobe. The address inputs are captured on the falling edge of /CAS, and select a column from the currently open row to read or write.
- **/WE**, Write Enable. This signal determines whether a given falling edge of /CAS is a read (if high) or write (if low). If low, the data inputs are also captured on the falling edge of /CAS.
- **/OE**, Output Enable. This is an additional signal that controls output to the data I/O pins. The data pins are driven by the DRAM chip if /RAS and /CAS are low, /WE is high, and /OE is low. In many applications, /OE can be permanently connected low (output always enabled), but it can be useful when connecting multiple memory chips in parallel.

This interface provides direct control of internal timing. When /RAS is driven low, a /CAS cycle must not be attempted until the sense amplifiers have sensed the memory state, and /RAS must not be returned high until the storage cells have been refreshed. When /RAS is driven high, it must be held high long enough for precharging to complete.

Although the RAM is asynchronous, the signals are typically generated by a clocked memory controller, which limits their timing to multiples of the controller's clock cycle.

Video DRAM (VRAM)

VRAM is a dual-ported variant of DRAM that was once commonly used to store the frame-buffer in some graphics adaptors.

Window DRAM (WRAM)

WRAM is a variant VRAM that was once used in graphics adaptors such as the Matrox Millenium and ATI 3D Rage Pro. WRAM was designed to perform better and cost less than VRAM. WRAM offered up to 25% greater bandwidth than VRAM and accelerated commonly used graphical operations such as text drawing and block fills.

Fast page mode (FPM) DRAM or FPRAM

A 256 k x 4 bit DRAM on an early PC memory card. k = 1024 in this case

Fast page mode DRAM is also called FPM DRAM, Page mode DRAM, Fast page mode memory, or Page mode memory.

In page mode, a row of the DRAM can be kept "open" by holding /RAS low while performing multiple reads or writes with separate pulses of /CAS so that successive reads or writes within the row do not suffer the delay of precharge and accessing the row. This increases the performance of the system when reading or writing bursts of data.

Static column is a variant of page mode in which the column address does not need to be strobed in, but rather, the address inputs may be changed with /CAS held low, and the data output will be updated accordingly a few nanoseconds later.

Nibble mode is another variant in which four sequential locations within the row can be accessed with four consecutive pulses of /CAS. The difference from normal page mode is that the address inputs are not used for the second through fourth /CAS edges; they are generated internally starting with the address supplied for the first /CAS edge.

CAS before RAS refresh

Classic asynchronous DRAM is refreshed by opening each row in turn. This can be done by supplying a row address and pulsing /RAS low; it is not necessary to perform any /CAS cycles. An external counter is needed to iterate over the row addresses in turn.

For convenience, the counter was quickly incorporated into RAM chips themselves. If the /CAS line is driven low before /RAS (normally an illegal operation), then the DRAM

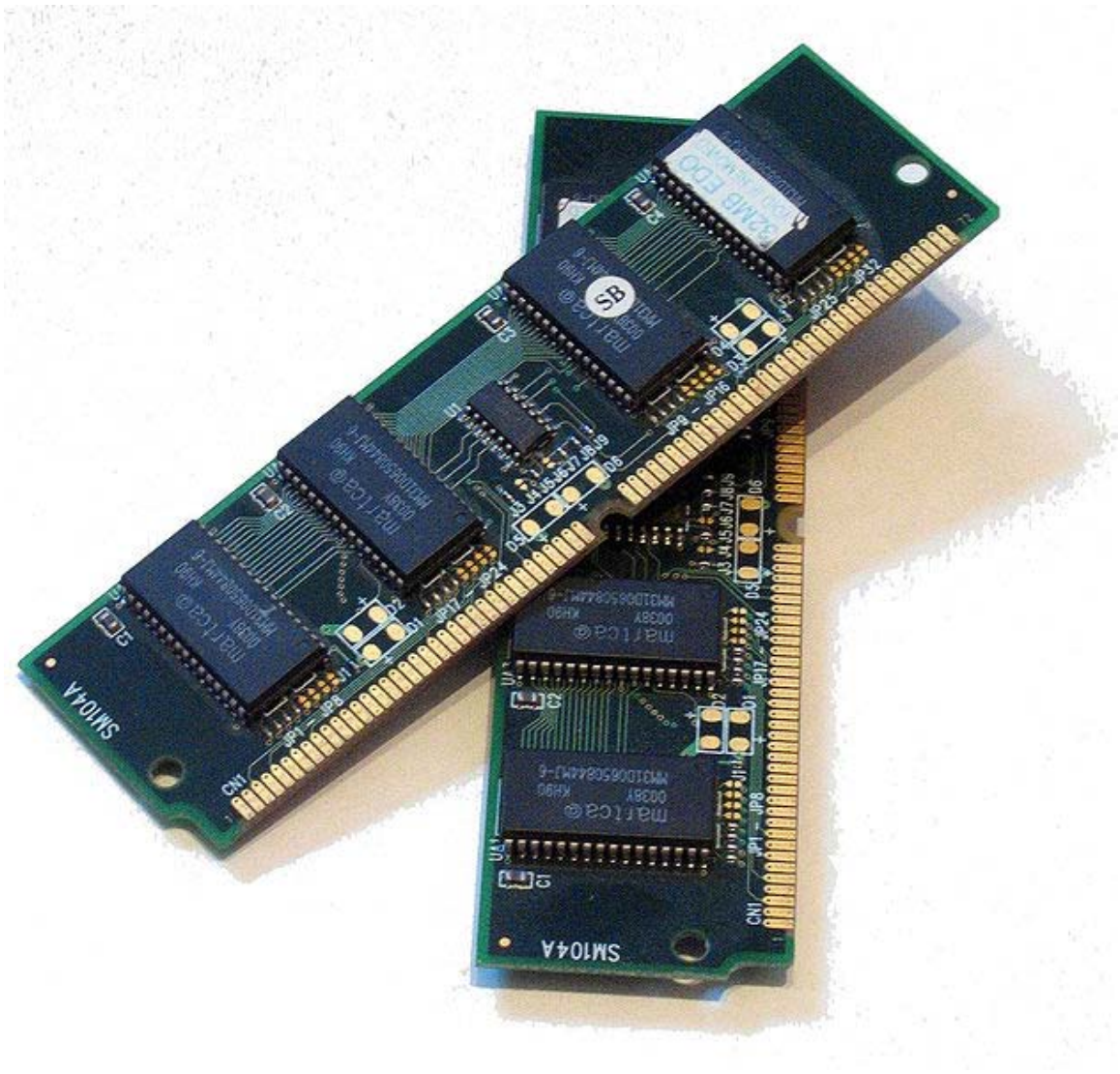
ignores the address inputs and uses an internal counter to select the row to open. This is known as /CAS-before-/RAS (CBR) refresh.

This became the standard form of refresh for asynchronous DRAM, and is the only form generally used with SDRAM.

Hidden refresh

Given support of CAS-before-RAS refresh, it is possible to deassert /RAS while holding /CAS low to maintain data output. If /RAS is then asserted again, this performs a CBR refresh cycle while the DRAM outputs remain valid. Because data output is not interrupted, this is known as "hidden refresh".

Extended data out (EDO) DRAM



A pair of 32 MB EDO DRAM modules

EDO DRAM, sometimes referred to as Hyper Page Mode enabled DRAM, is similar to Fast Page Mode DRAM with the additional feature that a new access cycle can be started while keeping the data output of the previous cycle active. This allows a certain amount of overlap in operation (pipelining), allowing somewhat improved performance. It was 5% faster than Fast Page Mode DRAM, which it began to replace in 1995, when Intel introduced the 430FX chipset that supported EDO DRAM.

To be precise, EDO DRAM begins data output on the falling edge of /CAS, but does not stop the output when /CAS rises again. It holds the output valid (thus extending the data output time) until either /RAS is deasserted, or a new /CAS falling edge selects a different column address.

Single-cycle EDO has the ability to carry out a complete memory transaction in one clock cycle. Otherwise, each sequential RAM access within the same page takes two clock cycles instead of three, once the page has been selected. EDO's performance and capabilities allowed it to somewhat replace the then-slow L2 caches of PCs. It created an opportunity to reduce the immense performance loss associated with a lack of L2 cache, while making systems cheaper to build. This was also good for notebooks due to difficulties with their limited form factor, and battery life limitations. An EDO system with L2 cache was tangibly faster than the older FPM/L2 combination.

Single-cycle EDO DRAM became very popular on video cards towards the end of the 1990s. It was very low cost, yet nearly as efficient for performance as the far more costly VRAM.

Much equipment taking 72-pin SIMMs could use either FPM or EDO. Problems were possible, particularly when mixing FPM and EDO. Early Hewlett-Packard printers had FPM RAM built in; some, but not all, models worked if additional EDO SIMMs were added.

Burst EDO (BEDO) DRAM

An evolution of EDO DRAM, **Burst EDO DRAM**, could process four memory addresses in one burst, for a maximum of 5‑1‑1‑1, saving an additional three clocks over optimally designed EDO memory. It was done by adding an address counter on the chip to keep track of the next address. BEDO also added a pipelined stage allowing page-access cycle to be divided into two components. During a memory-read operation, the first component accessed the data from the memory array to the output stage (second latch). The second component drove the data bus from this latch at the appropriate logic level. Since the data is already in the output buffer, quicker access time is achieved (up to 50% for large blocks of data) than with traditional EDO.

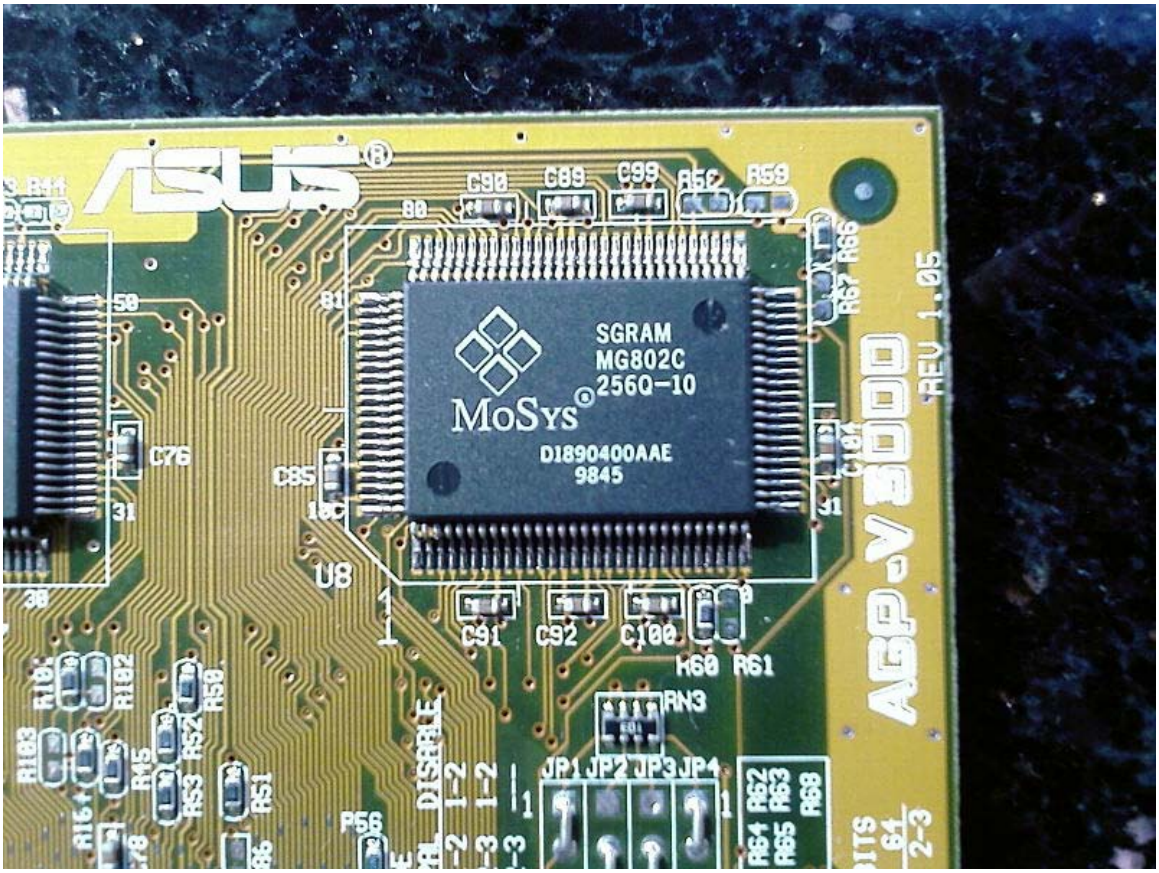
Although BEDO DRAM showed additional optimization over EDO, by the time it was available the market had made a significant investment towards synchronous DRAM, or SDRAM . Even though BEDO RAM was superior to SDRAM in some ways, the latter technology quickly displaced BEDO.

Multibank DRAM (MDRAM)

Multibank RAM applies the interleaving technique for main memory to second level cache memory to provide a cheaper and faster alternative to SRAM. The chip splits its memory capacity into small blocks of 256 kB and allows operations to two different banks in a single clock cycle.

This memory was primarily used in graphic cards with Tseng Labs ET6x00 chipsets, and was made by MoSys. Boards based upon this chipset often used the unusual RAM size configuration of 2.25 MB, owing to MDRAM's ability to be implemented in various sizes more easily. This size of 2.25 MB allowed 24-bit color at a resolution of 1024×768, a very popular display setting in the card's time.

Synchronous graphics RAM (SGRAM)



MoSYS SGRAM

SGRAM is a specialized form of SDRAM for graphics adaptors. It adds functions such as bit masking (writing to a specified bit plane without affecting the others) and block write (filling a block of memory with a single colour). Unlike VRAM and WRAM, SGRAM is single-ported. However, it can open two memory pages at once, which simulates the dual-port nature of other video RAM technologies.

Synchronous dynamic RAM (SDRAM)

Single data rate (SDR)

Single data rate SDRAM (sometimes known as **SDR**) is a synchronous form of DRAM.

Double data rate (DDR)

Double data rate SDRAM (**DDR**) was a later development of SDRAM, used in PC memory beginning in 2000. Subsequent versions are numbered sequentially (**DDR2**, **DDR3**, etc.).

Direct Rambus DRAM (DRDRAM)

Pseudostatic RAM (PSRAM)

PSRAM or **PSDRAM** is dynamic RAM with built-in refresh and address-control circuitry to make it behave similarly to static RAM (SRAM). It combines the high density of DRAM with the ease of use of true SRAM. PSRAM (made by Numonyx) is used in the Apple iPhone and other embedded systems.

Some DRAM components have a "self-refresh mode". While this involves much of the same logic that is needed for pseudo-static operation, this mode is often equivalent to a standby mode. It is provided primarily to allow a system to suspend operation of its DRAM controller to save power without losing data stored in DRAM, not to allow operation without a separate DRAM controller as is the case with PSRAM.

An embedded variant of pseudostatic RAM is sold by MoSys under the name 1T-SRAM. It is technically DRAM, but behaves much like SRAM. It is used in Nintendo Gamecube and Wii consoles.

1T DRAM

Unlike all of the other variants described here, **1T DRAM** is actually a different way of constructing the basic DRAM bit cell. 1T DRAM is a "capacitorless" bit cell design that stores data in the parasitic body capacitor that is an inherent part of Silicon on Insulator transistors. Considered a nuisance in logic design, this floating body effect can be used for data storage. Although refresh is still required, reads are non-destructive; the stored charge causes a detectable shift in the threshold voltage of the transistor.

There are several types of 1T DRAM memories: The commercialized Z-RAM from Innovative Silicon, the TTRAM from Renesas and the A-RAM from the UGR/CNRS consortium.

Note that classic one-transistor/one-capacitor (1T/1C) DRAM cell is also sometimes referred to as "1T DRAM".

RLDRAM

Reduced Latency DRAM is a high performance double data rate (DDR) SDRAM that combines fast, random access with high bandwidth. RLDRAM is mainly designed for networking and caching applications.

Security

Although dynamic memory is only *guaranteed* to retain its contents when supplied with power and refreshed every 64 ms, the memory cell capacitors will often retain their values for significantly longer, particularly at low temperatures.

Under some conditions, most of the data in DRAM can be recovered even if the DRAM has not been refreshed for several minutes.

This property can be used to recover "secure" data kept in memory by quickly rebooting the computer and dumping the contents of the RAM or by cooling the chips and transferring them to a different computer. Such an attack was demonstrated to circumvent popular disk encryption systems, like the open source TrueCrypt, Microsoft's BitLocker Drive Encryption, as well as Apple's FileVault.

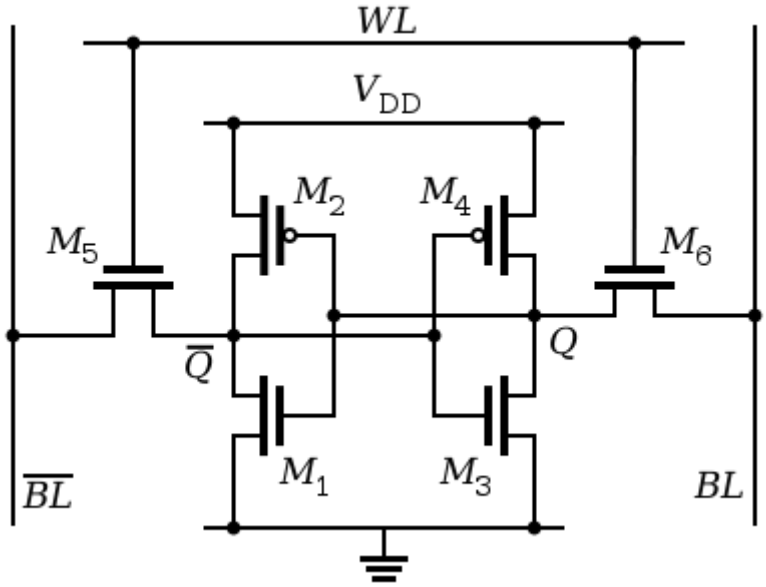
Chapter 4

Static Random-Access Memory

A static ram chip from *NES* clone 2K X 8 bit

Static random-access memory (SRAM) is a type of semiconductor memory where the word *static* indicates that, unlike *dynamic* RAM (DRAM), it does not need to be periodically refreshed, as SRAM uses bistable latching circuitry to store each bit. SRAM exhibits data remanence, but is still *volatile* in the conventional sense that data is eventually lost when the memory is not powered.

Design



A six-transistor CMOS SRAM cell

Each bit in an SRAM is stored on four transistors that **form** two cross-coupled inverters. This storage cell has two stable states which are used to denote **0** and **1**. Two additional *access* transistors serve to control the access to a storage cell during read and write operations. A typical SRAM uses six MOSFETs to store each memory bit. In addition to such 6T SRAM, other kinds of SRAM chips use 8T, 10T, or more transistors per bit. This is sometimes used to implement more than one (read and/or write) port, which may be useful in certain types of video memory and register files implemented with multi ported SRAM circuitry.

Generally, the fewer transistors needed per cell, the smaller each cell can be. Since the cost of processing a silicon wafer is relatively fixed, using smaller cells and so packing more bits on one wafer reduces the cost per bit of memory.

Memory cells that use fewer than 6 transistors are possible — but such 3T or 1T cells are DRAM, not SRAM (even the so-called 1T-SRAM).

Access to the cell is enabled by the word line (WL in figure) which controls the two *access* transistors M_5 and M_6 which, in turn, control whether the cell should be connected to the bit lines: BL and BL. They are used to transfer data for both read and write operations. Although it is not strictly necessary to have two bit lines, both the signal and its inverse are typically provided in order to improve noise margins.

During read accesses, the bit lines are actively driven high and low by the inverters in the SRAM cell. This improves SRAM bandwidth compared to DRAMs—in a DRAM, the bit line is connected to storage capacitors and charge sharing causes the bitline to swing upwards or downwards. The symmetric structure of SRAMs also allows for differential

signaling, which makes small voltage swings more easily detectable. Another difference with DRAM that contributes to making SRAM faster is that commercial chips accept all address bits at a time. By comparison, commodity DRAMs have the address multiplexed in two halves, i.e. higher bits followed by lower bits, over the same package pins in order to keep their size and cost down.

The size of an SRAM with *m* address lines and *n* data lines is 2^m words, or $2^m \times n$ bits.

At present appeared the advanced scheme with disconnected by signal record by feedback, which does not require the transistor of the load and is accordingly saved from high consumption of the energy when writing.

SRAM operation

An SRAM cell has three different states it can be in: *standby* where the circuit is idle, *reading* when the data has been requested and *writing* when updating the contents. The SRAM to operate in read mode and write mode should have "readeability" and "write stability" respectively. The three different states work as follows:

Standby

If the word line is not asserted, the *access* transistors M_5 and M_6 disconnect the cell from the bit lines. The two cross coupled inverters formed by $M_1 – M_4$ will continue to reinforce each other as long as they are connected to the supply.

Reading

Assume that the content of the memory is a **1**, stored at Q. The read cycle is started by precharging both the bit lines to a logical **1**, then asserting the word line WL, enabling both the *access* transistors. The second step occurs when the values stored in Q and Q are transferred to the bit lines by leaving BL at its precharged value and discharging BL through M_1 and M_5 to a logical **0**. On the BL side, the transistors M_4 and M_6 pull the bit line toward V_{DD}, a logical 1. If the content of the memory was a **0**, the opposite would happen and BL would be pulled toward **1** and BL toward **0**. Then these BL and BL-bar will have a small difference of delta between them and then these lines reach a sense amplifier, which will sense which line has higher voltage and thus will tell whether there was 1 stored or 0. The higher the sensitivity of sense amplifier, the faster the speed of read operation is.

Writing

The start of a write cycle begins by applying the value to be written to the bit lines. If we wish to write a **0**, we would apply a **0** to the bit lines, i.e. setting BL to **1** and BL to **0**. This is similar to applying a reset pulse to a SR-latch, which causes the flip flop to change state. A **1** is written by inverting the values of the bit lines. WL is then asserted and the value that is to be stored is latched in. Note that the reason this works is that the

bit line input-drivers are designed to be much stronger than the relatively weak transistors in the cell itself, so that they can easily override the previous state of the cross-coupled inverters. Careful sizing of the transistors in an SRAM cell is needed to ensure proper operation.

Bus behavior

A RAM memory with an access time of 70 ns will output valid data within 70 ns from the time that the address lines are valid. But the data will remain for a hold time as well (5-10 ns). Rise and fall times also influence valid timeslots with approximately ~5 ns. By reading the lower part of an address range bits in sequence (page cycle) one can read with significantly shorter access time (30 ns).

Applications and uses

Characteristics

SRAM is more expensive, but faster and significantly less power hungry (especially idle) than DRAM. It is therefore used where either bandwidth or low power, or both, are principal considerations. SRAM is also easier to control (interface to) and generally more truly *random access* than modern types of DRAM. Due to a more complex internal structure, SRAM is less dense than DRAM and is therefore not used for high-capacity, low-cost applications such as the main memory in personal computers.

Clock rate and power

The power consumption of SRAM varies widely depending on how frequently it is accessed; it **can** be as power-hungry as dynamic RAM, when used at high frequencies, and some ICs can consume many watts at full bandwidth. On the other hand, static RAM used at a somewhat slower pace, such as in applications with moderately clocked microprocessors, draw very little power and can have a nearly negligible power consumption when sitting idle — in the region of a few micro-watts.

Static RAM exists primarily as:

- general purpose products
    - with *asynchronous* interface, such as the 28 pin 32Kx8 chips (usually named XXC256), and similar products up to 16 Mbit per chip
    - with *synchronous* interface, usually used for caches and other applications requiring burst transfers, up to 18 Mbit (256Kx72) per chip
- integrated on chip
    - as RAM or cache memory in micro-controllers (usually from around 32 bytes up to 128 kilobytes)
    - as the primary caches in powerful microprocessors, such as the x86 family, and many others (from 8 kB, up to several megabytes)

- to store the registers and parts of the state-machines used in some microprocessors
- on application specific ICs, or ASICs (usually in the order of kilobytes)
- in FPGAs and CPLDs

Embedded use

Many categories of industrial and scientific subsystems, automotive electronics, and similar, contain static RAM. Some amount (kilobytes or less) is also embedded in practically all modern appliances, toys, etc. that implement an electronic user interface. Several megabytes may be used in complex products such as digital cameras, cell phones, synthesizers, etc.

SRAM in its dual-ported form is sometimes used for realtime digital signal processing circuits.

In computers

SRAM is also used in personal computers, workstations, routers and peripheral equipment: internal CPU caches and external burst mode SRAM caches, hard disk buffers, router buffers, etc. LCD screens and printers also normally employ static RAM to hold the image displayed (or to be printed). Small SRAM buffers are also found in CDROM and CDRW drives; usually 256 kB or more are used to buffer track data, which is transferred in blocks instead of as single values. The same applies to cable modems and similar equipment connected to computers.

Hobbyists

Hobbyists often prefer SRAM due to the ease of interfacing. It is much easier to work with than DRAM as there are no refresh cycles and the address and data buses are directly accessible rather than multiplexed. In addition to buses and power connections, SRAM usually require only three controls: Chip Enable (CE), Write Enable (WE) and Output Enable (OE). In synchronous SRAM, Clock (CLK) is also included.

Types of SRAM

Non-volatile SRAM

Non-volatile SRAMs have standard SRAM functionality, but they save the data when the power supply is lost, ensuring preservation of critical information. nvSRAMs are used in a wide range of situations—networking, aerospace, and medical, among many others—where the preservation of data is critical and where batteries are impractical.

Asynchronous SRAM

Asynchronous SRAM are available from 4 Kb to 32 Mb. The fast access time of SRAM makes asynchronous SRAM appropriate as main memory for small cache-less embedded processors used in everything from industrial electronics and measurement systems to hard disks and networking equipment, among many other applications. They are used in various applications like switches and routers, IP-Phones, IC-Testers, DSLAM Cards, to Automotive Electronics.

By transistor type

- Bipolar junction transistor (used in TTL and ECL) — very fast but consumes a lot of power
- MOSFET (used in CMOS) — low power and very common today

By function

- Asynchronous — independent of clock frequency; data in and data out are controlled by address transition
- Synchronous — all timings are initiated by the clock edge(s). Address, data in and other control signals are associated with the clock signals

By feature

- ZBT (ZBT stands for zero bus turnaround) — the turnaround is the number of clock cycles it takes to change access to the SRAM from **write** to **read** and vice versa. The turnaround for ZBT SRAMs or the latency between read and write cycle is zero.
- syncBurst (syncBurst SRAM or synchronous-burst SRAM) — features synchronous burst write access to the SRAM to increase write operation to the SRAM.
- DDR SRAM — Synchronous, single read/write port, double data rate IO
- Quad Data Rate SRAM — Synchronous, separate read & write ports, quadruple data rate IO

Chapter 5

T-RAM, Z-RAM and Twin Transistor RAM

T-RAM

Thyristor RAM (**T-RAM**) is a new type of DRAM computer memory invented and developed by T-RAM Semiconductor, which departs from the usual designs of memory cells, combining the strengths of the DRAM and SRAM: high speed and high volume. This technology, which exploits the electrical property known as negative differential resistance and is called thin capacitively-coupled thyristor, is used to create memory cells capable of very high packing densities. Due to this, this memory is highly scalable, and already has a storage density that is several times higher than found in conventional six-transistor SRAM memory. It is expected that the next generation of T-RAM memory will have the same density as DRAM.

It is assumed that this type of memory will be used in the next-generation processors by AMD, produced in 32nm and 22nm, replacing the previously licensed but unused Z-RAM technology.

Z-RAM

Zero-capacitor (registered trademark, **Z-RAM**) is a novel DRAM computer memory technology developed by Innovative Silicon based on the floating body effect of silicon on insulator (SOI) process technology. Z-RAM has been licensed by Advanced Micro Devices for possible use in future microprocessors. Innovative Silicon claims the technology offers memory access speeds similar to the standard six-transistor SRAM cell used in cache memory but uses only a single transistor, therefore affording much higher packing densities.

Z-RAM relies on the floating body effect, an artifact of the SOI process technology which places transistors in isolated tubs (the transistor body voltages "float" with respect

to the wafer substrate underneath the tubs). The floating body effect causes a variable capacitance to appear between the bottom of the tub and the underlying substrate, and was a problem that originally bedeviled circuit designs. The same effect, however, allows a DRAM-like cell to be built without adding a separate capacitor, the floating body effect taking the place of the conventional capacitor. Because the capacitor is located under the transistor (instead of adjacent to, or above the transistor as in conventional DRAMs), another connotation of the name "Z-RAM" is that it extends in the negative z-direction.

The reduced cell size leads, in a roundabout way, to Z-RAM being faster than even SRAM if used in large enough blocks. While individual SRAM cells are sensed faster than Z-RAM cells, the significantly smaller cell reduces the size of Z-RAM memory blocks and thus reduces the physical distance that data must transit to exit the memory block. As these metal traces have a fixed delay per unit length independent of memory technology, the shorter lengths of the Z-RAM signal traces can offset the faster SRAM cell access times. For a large cache memory (as typically found in a high performance microprocessor), Z-RAM offers equivalent speed as SRAM but requiring much less space (and thus cost). Response times as low as 3ns have been claimed.

SOI technology is targeted at very high performance computing markets and is relatively expensive compared with more common CMOS technology. Z-RAM offers the hope of cheaper on-chip cache memory, with little or no performance degradation, a compelling proposition if the memory cell can be proven to work in production volumes.

In March 2010, Innovative Silicon announced it was jointly developing a non-SOI version of Z-RAM that could be manufactured on lower cost bulk CMOS technology.

AMD has licensed the second generation Z-RAM to research it for potential use in their future processors, but is not planning to start using it.

DRAM producer Hynix has also licensed Z-RAM for use in DRAM chips.

Innovative Silicon was closed on June 29, 2010

Twin Transistor RAM

Twin Transistor RAM (**TTRAM**) is a new type of computer memory in development by Renesas.

TTRAM is similar to conventional one-transistor, one-capacitor DRAM in concept, but eliminates the capacitor by relying on the floating body effect inherent in a silicon on insulator (SOI) manufacturing process. This effect causes capacitance to build up between the transistors and the underlying substrate, originally considered a nuisance, but

here used to replace a part outright. Since a transistor created using the SOI process is somewhat smaller than a capacitor, TTRAM offers somewhat higher densities than conventional DRAM. Since prices are strongly related to density, TTRAM is theoretically less expensive. However the requirement to be built on SOI fab lines, which are currently the “leading edge”, makes the cost somewhat unpredictable at this point.

In the TTRAM memory cell, two transistors are serially connected on an SOI substrate. One is an access transistor, while the other is used as a storage transistor and fulfils the same function as the capacitor in a conventional DRAM cell. Data reads and writes are performed according to the conduction state of the access transistor and the floating-body potential state of the storage transistor. The fact that TTRAM memory cell operations don't require a step-up voltage or negative voltage, as DRAM cells do, makes the new cell design suitable for use with future finer processes and lower operating voltages.

With the Renesas TTRAM, a read signal from a memory cell appears as a difference in the transistor on-current. A current-mirror type sense amplifier detects this difference at high speed, using a reference memory cell that allows reliable identification of the 0 and 1 data levels. This reading method significantly decreases power consumption by eliminating the charging and discharging of bit lines, operations required for reading DRAM memory cells.

A similar technology is Z-RAM, which uses only a single transistor and is thus even higher density than TTRAM. Like TTRAM, Z-RAM relies on the floating body effect of SOI, and presumably has a similar manufacturing process. Z-RAM also claims to be faster, as fast as SRAM used in cache, which makes it particularly interesting for CPU designs which are being built on SOI lines anyway.

TTRAM should not be confused with TT-RAM, Atari's name for a special bank of DRAM in Atari TT030 personal computers.

Chapter 6

Non-Volatile Memory

Non-volatile memory, **nonvolatile memory**, **NVM** or **non-volatile storage**, in the most basic sense, is computer memory that can retain the stored information even when not powered. Examples of non-volatile memory include read-only memory, flash memory, most types of magnetic computer storage devices (e.g. hard disks, floppy disks, and magnetic tape), optical discs, and early computer storage methods such as paper tape and punched cards.

Non-volatile memory is typically used for the task of secondary storage, or long-term persistent storage. The most widely used form of primary storage today is a volatile form of random access memory (RAM), meaning that when the computer is shut down, anything contained in RAM is lost. Unfortunately, most forms of non-volatile memory have limitations that make them unsuitable for use as primary storage. Typically, non-volatile memory either costs more or performs worse than volatile random access memory.

Several companies are working on developing non-volatile memory systems comparable in speed and capacity to volatile RAM. For instance, IBM is currently developing MRAM (Magnetoresistive RAM). Not only would such technology save energy, but it would allow for computers that could be turned on and off almost instantly, bypassing the slow start-up and shutdown sequence.

Non-volatile data storage can be categorized in electrically addressed systems (read-only memory) and mechanically addressed systems (hard disks, optical disc, magnetic tape, holographic memory, and such). Electrically addressed systems are expensive, but fast, whereas mechanically addressed systems have a low price per bit, but are slow. Non-volatile memory may one day eliminate the need for comparatively slow forms of secondary storage systems, which include hard disks.

Electrically addressed

Electrically addressed non-volatile memories based on charge storage can be categorized according to their write mechanism:

Mask-programmed ROM

One of the earliest forms of non-volatile read-only memory, the mask-programmed ROM was prewired at the design stage to contain specific data; once the mask was used to manufacture the integrated circuits, the data was cast in stone (silicon, actually) and could not be changed.

The mask ROM was therefore useful only for large-volume production, such as for read-only memories containing the start up code in early microcomputers. This program was often referred to as the "bootstrap", as in pulling oneself up by one's own bootstraps.

Due to the very high initial cost and inability to make revisions, the mask ROM is rarely, if ever, used in new designs.

Programmable ROM

The next approach was to create a chip which was initially blank; the programmable ROM originally contained silicon or metal fuses, which would be selectively "blown" or destroyed by a device programmer or PROM programmer in order to change 0s to 1s. Once the bits were changed, there was no way to restore them to their original condition. Non-volatile but still somewhat inflexible.

Early PAL programmable array logic chips used a similar programming approach to that used in the fuse-based PROMs.

Newer Antifuse-based PROMs (which are also referred to as one-time-programmable (OTP) memory) are widely used in consumer and automotive electronics, radio-frequency identification devices (RFID), implantable medical devices, and high-definition multimedia interfaces(HDMI) due to their small footprint, reliability, fast read speed, and long data retention rates.

Erasable PROMs

There are two classes of non-volatile memory chips based on EPROM technology.

UV-erase EPROM

The original erasable non-volatile memories were EPROM's; these could be readily identified by the distinctive quartz window in the center of the chip package. These operated by trapping an electrical charge on the gate of a field-effect transistor in order to change a 1 to a 0 in memory. To remove the charge, one would place the chip under an intense short-wavelength fluorescent ultraviolet lamp for 20–30 minutes, returning the entire chip to its original blank (all ones) state.

OTP (one-time programmable) EPROM

An OTP is electrically an EPROM, but with the quartz window physically missing. Like the fuse, PROM it can be written once, but cannot be erased.

Electrically erasable PROM

Electrically erasable PROMs have the advantage of being able to selectively erase any part of the chip without the need to erase the entire chip and without the need to remove the chip from the circuit. While an erase and rewrite of a location appears nearly instantaneous to the user, the write process is slightly slower than the read process; the chip can be read at full system speeds.

The limited number of times a single location can be rewritten is usually in the 10000-100000 range; the capacity of an EEPROM also tends to be smaller than that of other non-volatile memories. Nonetheless, EEPROMs are useful for storing settings or configuration for devices ranging from dial-up modems to satellite receivers.

Flash memory

The flash memory chip is a close relative to the EEPROM; it differs in that it can only be erased one block or "page" at a time. It is a solid-state chip that maintains stored data without any external power source. Capacity is substantially larger than that of an EEPROM, making these chips a popular choice for digital cameras and desktop PC BIOS chips.

Flash memory devices use two different logical technologies—NOR and NAND—to map data. NOR flash provides high-speed random access, reading and writing data in specific memory locations; it can retrieve as little as a single byte. NAND flash reads and writes sequentially at high speed, handling data in small blocks called pages, however it’s slower on read when compared to NOR. NAND flash reads faster than it writes, quickly transferring whole pages of data. Less expensive than NOR flash at high densities, NAND technology offers higher capacity for the same-size silicon.

List of NOR Flash providers:

- Amtel
- Intel
- Macronix
- Micron Technology (formerly Numonyx)
- Silicon Storage Technology (SST)
- Spansion
- STMicroelectronics

List of NAND Flash providers:

- Hynix
- Intel
- Micron Technology
- Qimonda
- Renesas Electronics Corporation
- Samsung
- Spansion
- STMicroelectronics
- Toshiba

Magnetoresistive RAM (MRAM)

Magnetoresistive RAM is one of the newest approaches to non-volatile memory and stores data in magnetic storage elements called magnetic tunnel junctions (MTJ's). MRAM has an especially promising future as it seeks to encompass all the desirable features of the other popular types of memory (non-volatility, infinite endurance, high-speed reading/writing, low cost).

The 1st generation of MRAM, such as Everspin Technologies' 4 Mbit, utilized field induced writing. The 2nd generation is being developed mainly through two approaches: Thermal Assisted Switching (TAS) which is being developed by Crocus Technology, and Spin Torque Transfer (STT) which Crocus, Hynix, IBM, and several other companies are developing.

Mechanically addressed systems

Mechanically addressed systems utilize a contact structure ('head') to read and write on a designated storage medium. Since circuitry layout is not a key factor for data density, the amount of storage is typically much larger than for electrically addressed systems.

Organic

There are polymer printed ferroelectric memory.

Thin Film Electronics ("Thinfilm") produces rewriteable non-volatile organic memory based on ferroelectric polymers. Thinfilm successfully demonstrated roll-to-roll printed memories in 2009.

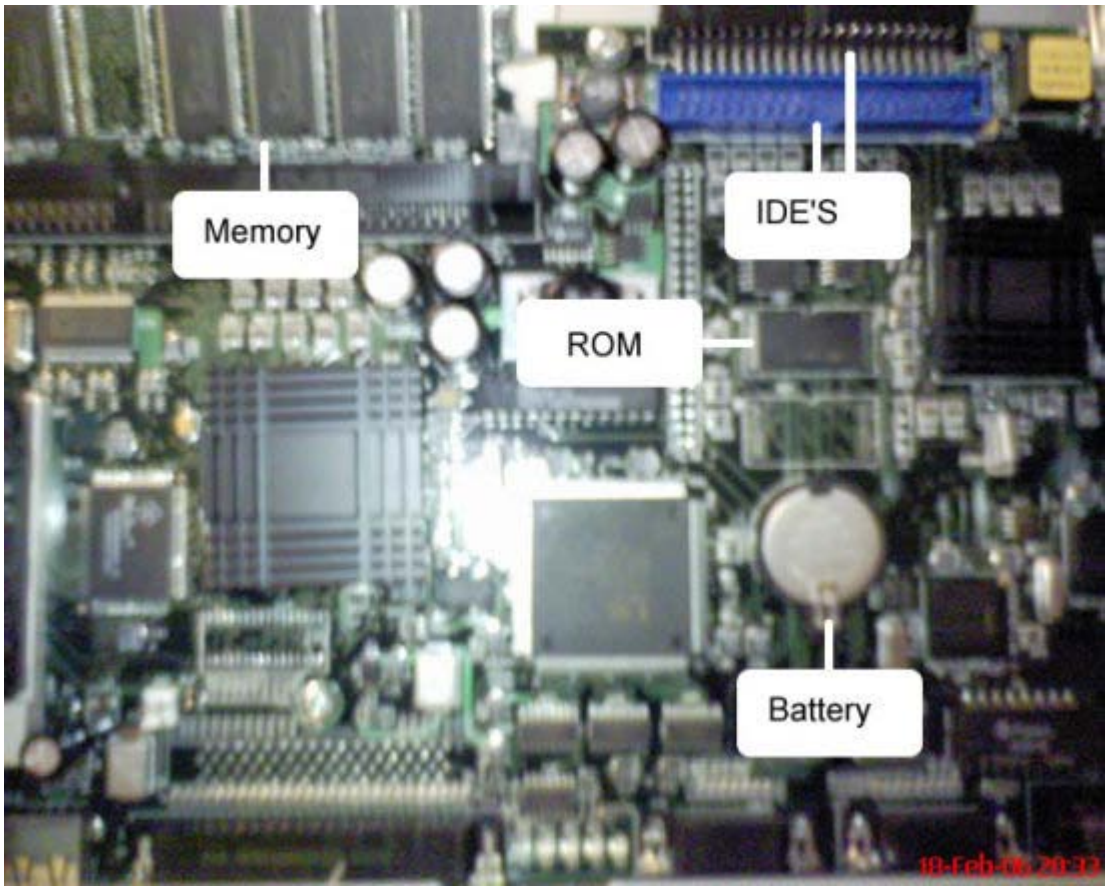
In Thinfilm's organic memory the ferroelectric polymer is sandwiched between two sets of electrodes in a passive matrix. Each crossing of metal lines is a ferroelectric capacitor and defines a memory cell. This gives a non-volatile memory comparable to ferroelectric RAM technologies and offer the same functionality as flash memory.

Specifications

Specification March 2007	2.5" HDD	1" Microdrive	Flash Memory	Optical Disk	Tape	MRAM
Device Model:	Hitachi Travelstar 5k160	Hitachi Microdrive 3k8	Hynix HY27UH08AG5M	Blu Ray	HP Ultrium 960	Everspin MR2A16A
Density (GBit/cm^2)	20.3	18.4	6.7	3.8	0.047	0.0021
Capacity (GByte)	160	8	2	50	400	0.004
Price per bit (Eur/GByte)	1.5	9.0	6.0	1.25	0.075	35000
Price per unit (Eur)	110	87	14	635	2340	17.4
Price per medium (Eur)	nd	nd	nd	40	30	nd
Data rate (Mbit/s)	540	80	23	144	640	436
Access time (ms)	11	12	0.025	180	72000	0.000035
Power consumption (W)	1.8	0.6	0.1	25	20	0.08
Form factor h x w x d (cm)	0.95x7x10	0.5x3x4	0.1x1.2x2	4x15x19	2x10x10	0.1x1x1.8

Chapter 7

Read-Only Memory



ROM - Read Only Memory

Read-only memory (**ROM**) is a class of storage media used in computers and other electronic devices. Data stored in ROM cannot be modified, or can be modified only slowly or with difficulty, so it is mainly used to distribute firmware (software that is very closely tied to specific hardware, and unlikely to need frequent updates).

In its strictest sense, ROM refers only to mask ROM (the oldest type of solid state ROM), which is fabricated with the desired data permanently stored in it, and thus can never be modified. Despite the simplicity of mask ROM, economies of scale and field-

programmability often make reprogrammable technologies more flexible and inexpensive, so mask ROM is rarely used in new products as of 2007.

Other types of non-volatile memory such as erasable programmable read only memory (EPROM) and electrically erasable programmable read-only memory (EEPROM or Flash ROM) are sometimes referred to, in an abbreviated way, as "read-only memory" (ROM), but this is actually a misnomer because these types of memory can be erased and re-programmed multiple times. When used in this less precise way, "ROM" indicates a *non-volatile* memory which serves functions typically provided by mask ROM, such as storage of program code and nonvolatile data.

History

The simplest type of solid state ROM is as old as semiconductor technology itself. Combinational logic gates can be joined manually to map *n*-bit **address** input onto arbitrary values of *m*-bit **data** output (a look-up table). With the invention of the integrated circuit came mask ROM. Mask ROM consists of a grid of word lines (the address input) and bit lines (the data output), selectively joined together with transistor switches, and can represent an arbitrary look-up table with a regular physical layout and predictable propagation delay.

In mask ROM, the data is physically encoded in the circuit, so it can only be programmed during fabrication. This leads to a number of serious disadvantages:

1. It is only economical to buy mask ROM in large quantities, since users must contract with a foundry to produce a custom design.
2. The turnaround time between completing the design for a mask ROM and receiving the finished product is long, for the same reason.
3. Mask ROM is impractical for R&D work since designers frequently need to modify the contents of memory as they refine a design.
4. If a product is shipped with faulty mask ROM, the only way to fix it is to recall the product and physically replace the ROM.

Subsequent developments have addressed these shortcomings. PROM, invented in 1956, allowed users to program its contents exactly once by physically altering its structure with the application of high-voltage pulses. This addressed problems 1 and 2 above, since a company can simply order a large batch of fresh PROM chips and program them with the desired contents at its designers' convenience. The 1971 invention of EPROM essentially solved problem 3, since EPROM (unlike PROM) can be repeatedly reset to its unprogrammed state by exposure to strong ultraviolet light. EEPROM, invented in 1983, went a long way to solving problem 4, since an EEPROM can be programmed in-place if the containing device provides a means to receive the program contents from an external source (e.g. a personal computer via a serial cable). Flash memory, invented at Toshiba in the mid-1980s, and commercialized in the early 1990s, is a form of EEPROM that makes very efficient use of chip area and can be erased and reprogrammed thousands of times without damage.

All of these technologies improved the flexibility of ROM, but at a significant cost-per-chip, so that in large quantities mask ROM would remain an economical choice for many years. (Decreasing cost of reprogrammable devices had almost eliminated the market for mask ROM by the year 2000.) Furthermore, despite the fact that newer technologies were increasingly less "read-only," most were envisioned only as replacements for the traditional use of mask ROM.

The most recent development is NAND flash, also invented by Toshiba. Its designers explicitly broke from past practice, stating plainly that "the aim of NAND Flash is to replace hard disks," rather than the traditional use of ROM as a form of non-volatile primary storage. As of 2007, NAND has partially achieved this goal by offering throughput comparable to hard disks, higher tolerance of physical shock, extreme miniaturization (in the form of USB flash drives and tiny microSD memory cards, for example), and much lower power consumption.

Use for storing programs

Every stored-program computer needs some form of non-volatile, or erasable, storage to store the initial program that runs when the computer is powered on or otherwise begins execution (a process known as bootstrapping, often abbreviated to "booting" or "booting up"). Likewise, every non-trivial computer needs some form of mutable memory to record changes in its state as it executes.

Forms of read-only memory were employed as non-volatile storage for programs in most early stored-program computers, such as ENIAC after 1948 (until then it was not a stored-program computer as every program had to be manually wired into the machine, which could take days to weeks). Read-only memory was simpler to implement since it needed only a mechanism to read stored values, and not to change them in-place, and thus could be implemented with very crude electromechanical devices. With the advent of integrated circuits in the 1960s, both ROM and its mutable counterpart static RAM were implemented as arrays of transistors in silicon chips; however, a ROM memory cell could be implemented using fewer transistors than an SRAM memory cell, since the latter needs a latch (comprising 5-20 transistors) to retain its contents, while a ROM cell might consist of the absence (logical 0) or presence (logical 1) of one transistor connecting a bit line to a word line. Consequently, ROM could be implemented at a lower cost-per-bit than RAM for many years.

Most home computers of the 1980s stored a BASIC interpreter or operating system in ROM as other forms of non-volatile storage such as magnetic disk drives were too costly. For example, the Commodore 64 included 64 KB of RAM and 20 KB of ROM contained a BASIC interpreter and the "KERNAL" of its operating system. Later home or office computers such as the IBM PC XT often included magnetic disk drives, and larger amounts of RAM, allowing them to load their operating systems from disk into RAM, with only a minimal hardware initialization core and bootloader remaining in ROM (known as the BIOS in IBM-compatible computers). This arrangement allowed for a more complex and easily upgradeable operating system.

In modern PCs, "ROM" (or Flash) is used to store the basic bootstrapping firmware for the main processor, as well as the various firmware needed to internally control self contained devices such as graphic cards, hard disks, DVD drives, TFT screens, etc, in the system. Today, many of these "read-only" memories – especially the BIOS – are often replaced with Flash memory (see below), to permit in-place reprogramming should the need for a firmware upgrade arise. However, simple and mature sub-systems (such as the keyboard or some communication controllers in the ICs on the main board, for example) may employ mask ROM or OTP (one time programmable).

ROM and successor technologies such as Flash are prevalent in embedded systems. These are in everything from industrial robots to home appliances and consumer electronics (MP3 players, set-top boxes, etc) all of which are designed for specific functions, but are based on general-purpose microprocessors in most cases. With software usually tightly coupled to hardware, program changes are rarely needed in such devices (which typically lack devices such as hard disks for reasons of cost, size, and/or power consumption). As of 2008, most products use Flash rather than mask ROM, and many provide some means for connecting to a PC for firmware updates; for example, a digital audio player might be updated to support a new file format. Some hobbyists have taken advantage of this flexibility to reprogram consumer products for new purposes; for example, the iPodLinux and OpenWRT projects have enabled users to run full-featured Linux distributions on their MP3 players and wireless routers, respectively.

ROM is also useful for binary storage of cryptographic data, as it makes them difficult to replace, which may be desirable in order to enhance information security.

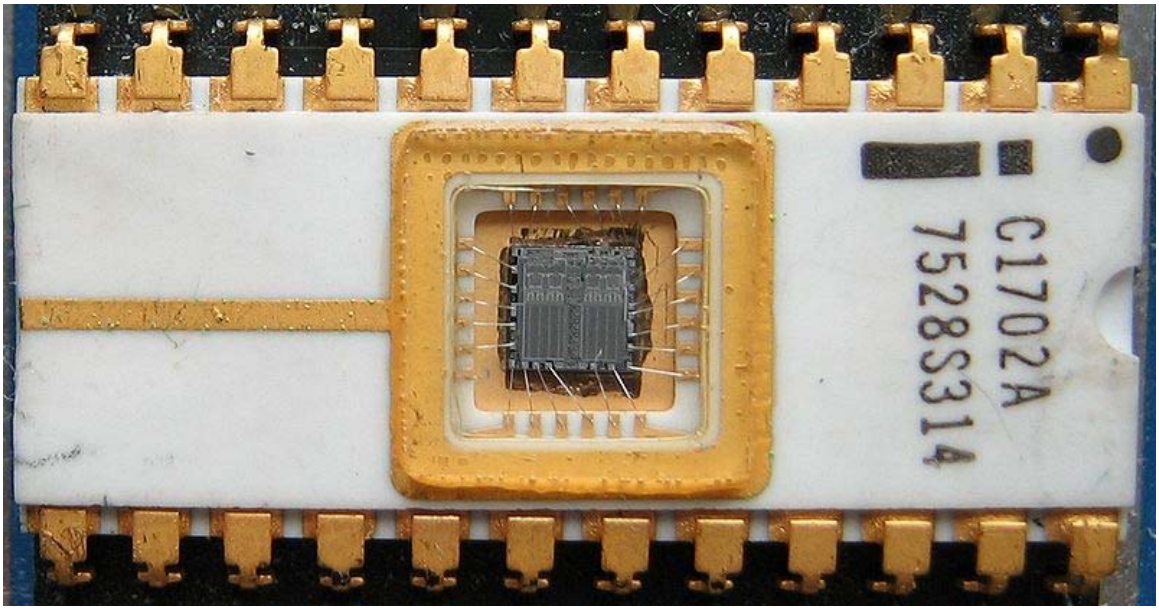
Use for storing data

Since ROM (at least in hard-wired mask form) cannot be modified, it is really only suitable for storing data which is not expected to need modification for the life of the device. To that end, ROM has been used in many computers to store look-up tables for the evaluation of mathematical and logical functions (for example, a floating-point unit might tabulate the sine function in order to facilitate faster computation). This was especially effective when CPUs were slow and ROM was cheap compared to RAM.

Notably, the display adapters of early personal computers stored tables of bitmapped font characters in ROM. This usually meant that the text display font could not be changed interactively. This was the case for both the CGA and MDA adapters available with the IBM PC XT.

The use of ROM to store such small amounts of data has disappeared almost completely in modern general-purpose computers. However, Flash ROM has taken over a new role as a medium for mass storage or secondary storage of files.

Types



The first EPROM, an Intel 1702, with the die and wire bonds clearly visible through the erase window.

Semiconductor based

Classic **mask-programmed ROM** chips are integrated circuits that physically encode the data to be stored, and thus it is impossible to change their contents after fabrication. Other types of non-volatile solid-state memory permit some degree of modification:

- **Programmable read-only memory** (PROM), or **one-time programmable ROM** (OTP), can be written to or **programmed** via a special device called a **PROM programmer**. Typically, this device uses high voltages to permanently destroy or create internal links (fuses or antifuses) within the chip. Consequently, a PROM can only be programmed once.
- **Erasable programmable read-only memory** (EPROM) can be erased by exposure to strong ultraviolet light (typically for 10 minutes or longer), then rewritten with a process that again needs higher than usual voltage applied. Repeated exposure to UV light will eventually wear out an EPROM, but the **endurance** of most EPROM chips exceeds 1000 cycles of erasing and reprogramming. EPROM chip packages can often be identified by the prominent quartz "window" which allows UV light to enter. After programming, the window is typically covered with a label to prevent accidental erasure. Some EPROM chips are factory-erased before they are packaged, and include no window; these are effectively PROM.
- **Electrically erasable programmable read-only memory** (EEPROM) is based on a similar semiconductor structure to EPROM, but allows its entire contents (or selected **banks**) to be electrically erased, then rewritten electrically, so that they need not be removed from the computer (or camera, MP3 player, etc.). Writing or

flashing an EEPROM is much slower (milliseconds per bit) than reading from a ROM or writing to a RAM (nanoseconds in both cases).

- **Electrically alterable read-only memory** (EAROM) is a type of EEPROM that can be modified one bit at a time. Writing is a very slow process and again needs higher voltage (usually around 12 V) than is used for read access. EAROMs are intended for applications that require infrequent and only partial rewriting. EAROM may be used as non-volatile storage for critical system setup information; in many applications, EAROM has been supplanted by CMOS RAM supplied by mains power and backed-up with a lithium battery.
- **Flash memory** (or simply **flash**) is a modern type of EEPROM invented in 1984. Flash memory can be erased and rewritten faster than ordinary EEPROM, and newer designs feature very high endurance (exceeding 1,000,000 cycles). Modern NAND flash makes efficient use of silicon chip area, resulting in individual ICs with a capacity as high as 32 GB as of 2007; this feature, along with its endurance and physical durability, has allowed NAND flash to replace magnetic in some applications (such as USB flash drives). Flash memory is sometimes called **flash ROM** or **flash EEPROM** when used as a replacement for older ROM types, but not in applications that take advantage of its ability to be modified quickly and frequently.

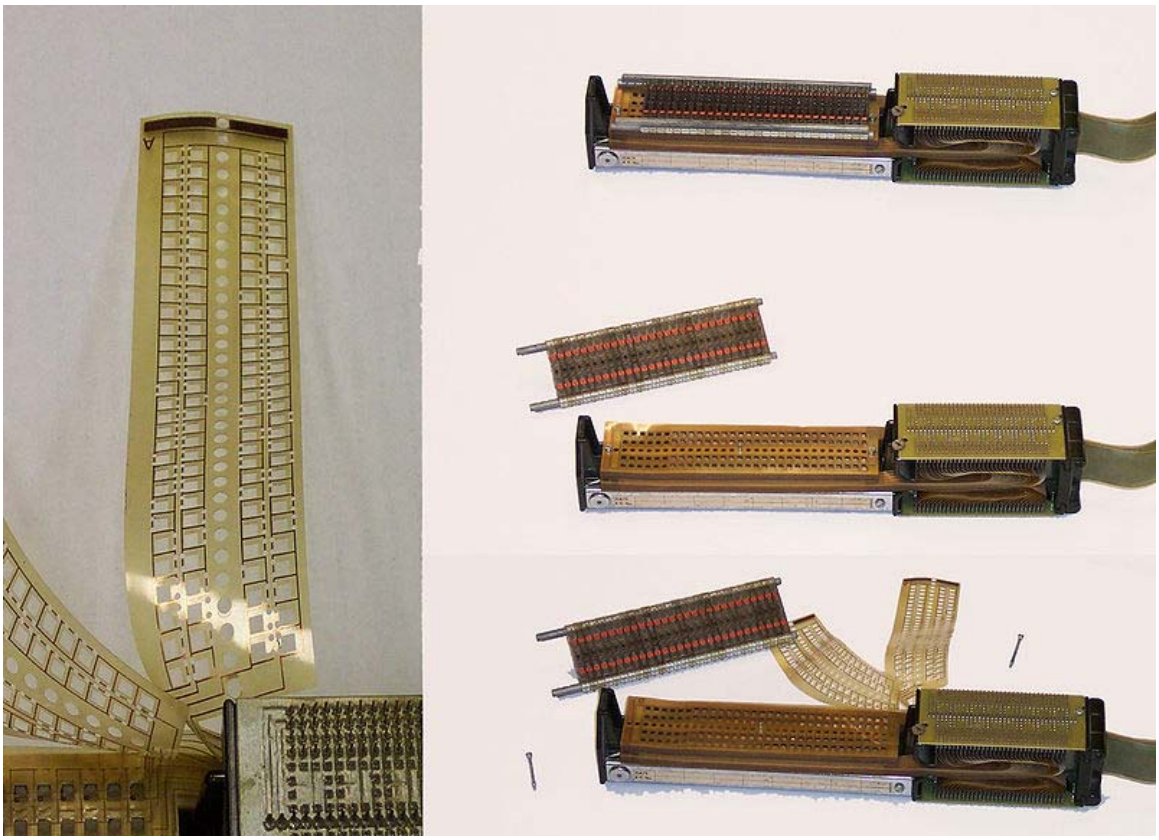
By applying write protection, some types of reprogrammable ROMs may temporarily become read-only memory.

Other technologies

There are other types of non-volatile memory which are not based on solid-state IC technology, including:

- Optical storage media, such CD-ROM which is read-only (analogous to masked ROM). CD-R is Write Once Read Many (analogous to PROM), while CD-RW supports erase-rewrite cycles (analogous to EEPROM); both are designed for backwards-compatibility with CD-ROM.

Historical examples



Transformer matrix ROM (TROS), from the IBM System 360/20

- Diode matrix ROM, used in small amounts in many computers in the 1960s as well as electronic desk calculators and keyboard encoders for terminals. This ROM was programmed by installing discrete semiconductor diodes at selected locations between a matrix of *word line traces* and *bit line traces* on a printed circuit board.
- Resistor, capacitor, or transformer matrix ROM, used in many computers until the 1970s. Like diode matrix ROM, it was programmed by placing components at selected locations between a matrix of *word lines* and *bit lines*. ENIAC's Function Tables were resistor matrix ROM, programmed by manually setting rotary switches. Various models of the IBM System/360 and complex peripheral devices stored their microcode in either capacitor (called *BCROS* for *balanced capacitor read-only storage* on the 360/50 & 360/65, or *CCROS* for *card capacitor read-only Storage* on the 360/30) or transformer (called *TROS* for *transformer read-only storage* on the 360/20, 360/40 and others) matrix ROM.
- Core rope, a form of transformer matrix ROM technology used where size and/or weight were critical. This was used in NASA/MIT's Apollo Spacecraft Computers, DEC's PDP-8 computers, and other places. This type of ROM was programmed by hand by weaving "word line wires" inside or outside of ferrite transformer cores.

- The perforated metal character mask ("stencil") in Charactron cathode ray tubes, which was used as ROM to shape a wide electron beam to form a selected character shape on the screen either for display or a scanned electron beam to form a selected character shape as an overlay on a video signal.

Speed

Reading

Although the relative speed of RAM vs. ROM has varied over time, as of 2007 large RAM chips can be read faster than most ROMs. For this reason (and to allow uniform access), ROM content is sometimes copied to RAM or **shadowed** before its first use, and subsequently read from RAM.

Writing

For those types of ROM that can be electrically modified, writing speed is always much slower than reading speed, and it may need unusually high voltage, the movement of jumper plugs to apply write-enable signals, and special lock/unlock command codes. Modern NAND Flash achieves the highest write speeds of any rewritable ROM technology, with speeds as high as 15 MB/s (or 70 ns/bit), by allowing (needing) large blocks of memory cells to be written simultaneously.

Endurance and data retention

Because they are written by forcing electrons through a layer of electrical insulation onto a floating transistor gate, rewriteable ROMs can withstand only a limited number of write and erase cycles before the insulation is permanently damaged. In the earliest EAROMs, this might occur after as few as 1,000 write cycles, while in modern Flash EEPROM the **endurance** may exceed 1,000,000, but it is by no means infinite. This limited endurance, as well as the higher cost per bit, means that Flash-based storage is unlikely to completely supplant magnetic disk drives in the near future.

The timespan over which a ROM remains accurately readable is not limited by write cycling. The **data retention** of EPROM, EAROM, EEPROM, and Flash *may* be limited by charge leaking from the floating gates of the memory cell transistors. Leakage is accelerated by high temperatures or radiation. Masked ROM and fuse/antifuse PROM do not suffer from this effect, as their data retention depends on physical rather than electrical permanence of the integrated circuit (although *fuse re-growth* was once a problem in some systems).

Content images

The contents of ROM chips in video game console cartridges can be extracted with special software or hardware devices. The resultant memory dump files are known as **ROM images**, and can be used to produce duplicate cartridges, or in console emulators.

The term originated when most console games were distributed on cartridges containing ROM chips, but achieved such widespread usage that it is still applied to images of newer games distributed on CD-ROMs or other optical media.

ROM images of commercial games usually contain copyrighted software. The unauthorized copying and distribution of copyrighted software is usually a violation of copyright laws (in some jurisdictions, duplication of ROM cartridges for backup purposes may be considered fair use). Nevertheless, there is a thriving community engaged in the illegal distribution and trading of such software. In such circles, the term "ROM images" is sometimes shortened simply to "ROMs" or sometimes changed to "romz" to highlight the connection with "warez".

Chapter 8

Programmable Read-Only Memory and EPROM

Programmable read-only memory

D23128C PROM on the board of ZX Spectrum

A **programmable read-only memory** (**PROM**) or **field programmable read-only memory** (**FPROM**) or **one-time programmable non-volatile memory** (**OTP NVM**) is a form of digital memory where the setting of each bit is locked by a fuse or antifuse. Such PROMs are used to store programs permanently. The key difference from a strict ROM is that the programming is applied after the device is constructed.

PROMs are manufactured blank and, depending on the technology, can be programmed at wafer, final test, or in system. The availability of this technology allows companies to keep a supply of blank PROMs in stock, and program them at the last minute to avoid large volume commitment. These types of memories are frequently seen in video game consoles, mobile phones, radio-frequency identification (RFID) tags, implantable medical devices, high-definition multimedia interfaces (HDMI) and in many other consumer and automotive electronics products.

History

The PROM was invented in 1956 by Wen Tsing Chow, working for the Arma Division of the American Bosch Arma Corporation in Garden City, New York. The invention was

conceived at the request of the United States Air Force to come up with a more flexible and secure way of storing the targeting constants in the Atlas E/F ICBM's airborne digital computer. The patent and associated technology was held under secrecy order for several years while the Atlas E/F was the main operational missile of the United States ICBM force. The term "burn," referring to the process of programming a PROM, is also in the original patent, as one of the original implementations was to literally burn the internal whiskers of diodes with a current overload to produce a circuit discontinuity. The first PROM programming machines were also developed by Arma engineers under Mr. Chow's direction and were located in Arma's Garden City lab and Air Force Strategic Air Command (SAC) headquarters.

Commercially available semiconductor antifuse-based OTP memory arrays have been around at least since 1969, with initial antifuse bit cells dependent on blowing a capacitor between crossing conductive lines. Texas Instruments developed a MOS gate-oxide breakdown antifuse in 1979. A dual-gate-oxide two-transistor (2T) MOS antifuse was introduced in 1982. Early oxide breakdown technologies exhibited a variety of scaling, programming, size and manufacturing problems that prevented volume production of memory devices based on these technologies.

Although antifuse OTP has been available for decades, it wasn’t available in standard CMOS until 2001 when Kilopass Technology Inc. patented 1T, 2T, and 3.5T antifuse bit cell technologies using a standard CMOS process, enabling integration of PROM into logic CMOS chips. The first process node antifuse can be implemented in standard CMOS is 0.18 um. Since the Gox breakdown is less than the junction breakdown, special diffusion steps were not required to create the antifuse programming element. In 2005, a split channel antifuse device was introduced by Sidense. This Split Channel bit cell combines the thick (IO) and thin (gate) oxide devices into one transistor (1T) with a common polysilicon gate.

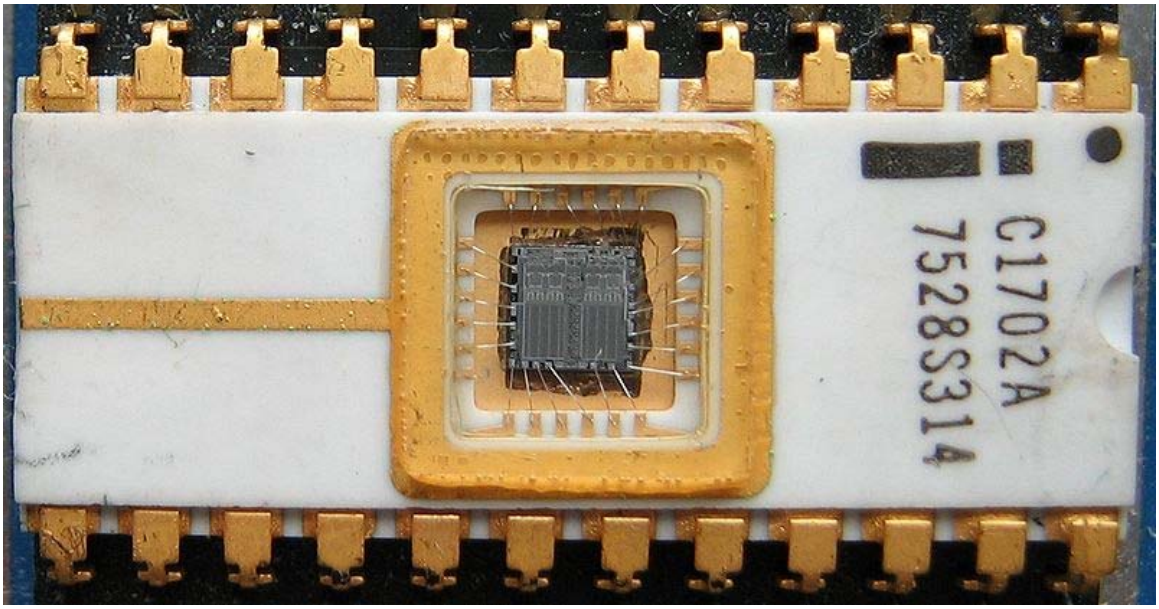
Programming

A typical PROM comes with all bits reading as "1". Burning a fuse bit during programming causes the bit to read as "0". The memory can be programmed just once after manufacturing by "blowing" the fuses, which is an irreversible process. Blowing a fuse opens a connection while programming an antifuse closes a connection (hence the name). While it is impossible to "unblow" the fuses, it is often possible to change the contents of the memory after initial programming by blowing additional fuses, changing some remaining "1" bits in the memory to "0"s. (Once all of the bits are "0", no further programming change is possible.)

The bit cell is programmed by applying a high-voltage pulse not encountered during normal operation across the gate and substrate of the thin oxide transistor (around 6V for a 2 nm thick oxide, or 30MV/cm) to break down the oxide between gate and substrate. The positive voltage on the transistor’s gate forms an inversion channel in the substrate below the gate, causing a tunneling current to flow through the oxide. The current produces additional traps in the oxide, increasing the current through the oxide and

ultimately melting the oxide and forming a conductive channel from gate to substrate. The current required to form the conductive channel is around 100µA/100nm2 and the breakdown occurs in approximately 100µs or less.

EPROM

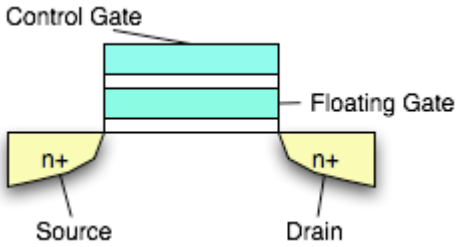


An EPROM. The small quartz window admits UV light for erasure.

An **EPROM** (rarely EROM), or **erasable programmable read only memory**, is a type of memory chip that retains its data when its power supply is switched off. In other words, it is non-volatile. It is an array of floating-gate transistors individually programmed by an electronic device that supplies higher voltages than those normally used in digital circuits. Once programmed, an EPROM can be erased by exposing it to strong ultraviolet light from a mercury-vapor light source. EPROMs are easily recognizable by the transparent fused quartz window in the top of the package, through which the silicon chip is visible, and which permits exposure to UV light during erasing.

Operation

Development of the EPROM memory cell started with investigation of faulty integrated circuits where the gate connections of transistors had broken. Stored charge on these isolated gates changed their properties. The EPROM was invented by Dov Frohman of Intel in 1971, who was awarded US patent 3660189 in 1972.



A cross-section of a floating-gate transistor

Each storage location of an EPROM consists of a single field-effect transistor. Each field-effect transistor consists of a channel in the semiconductor body of the device. Source and drain contacts are made to regions at the end of the channel. An insulating layer of oxide is grown over the channel, then a conductive (silicon or aluminum) gate electrode is deposited, and a further thick layer of oxide is deposited over the gate electrode. The floating gate electrode has no connections to other parts of the integrated circuit and is completely insulated by the surrounding layers of oxide. A control gate electrode is deposited and further oxide covers it.

To retrieve data from the EPROM, the address represented by the values at the address pins of the EPROM is decoded and used to connect one word (usually an 8-bit byte) of storage to the output buffer amplifiers. Each bit of the word is a 1 or 0, depending on the storage transistor being switched on or off, conducting or non-conducting.

The switching state of the field-effect transistor is controlled by the voltage on the control gate of the transistor. Presence of a voltage on this gate creates a conductive channel in the transistor, switching it on. In effect, the stored charge on the floating gate allows the threshold voltage of the transistor to be programmed.

Storing data in the memory requires selecting a given address and applying a higher voltage to the transistors. This creates an avalanche discharge of electrons, which have enough energy to pass through the insulating oxide layer and accumulate on the gate electrode. When the high voltage is removed, the electrons are trapped on the electrode. Because of the high insulation value of the silicon oxide surrounding the gate, the stored charge cannot readily leak away and the data can be retained for decades.

Unlike EEPROMs, the programming process is not electrically reversible. To erase the data stored in the array of transistors, ultraviolet light is directed onto the die. Photons of the UV light create ionization within the silicon oxide, which allow the stored charge on the floating gate to dissipate. Since the whole memory array is exposed, all the memory is erased at the same time. The process takes several minutes for UV lamps of convenient sizes; sunlight would erase a chip in weeks, and indoor fluorescent lighting over several years. Generally the EPROMs must be removed from equipment to be erased, since it's not usually practical to build in a UV lamp to erase parts in-circuit.

Details

As the quartz window is expensive to make, OTP (one-time programmable) chips were introduced; here, the die is mounted in an opaque package so it cannot be erased after programming - this also eliminates the need to test the erase function, further reducing cost. OTP versions of both EPROMs and EPROM-based microcontrollers are manufactured. However, OTP EPROM (whether separate or part of a larger chip) is being increasingly replaced by EEPROM for small amounts where the cell cost isn't too important and flash for larger amounts.

A programmed EPROM retains its data for about ten to twenty years and can be read an unlimited number of times. The erasing window must be kept covered with an opaque label to prevent accidental erasure by sunlight. Old PC BIOS chips were often EPROMs, and the erasing window was often covered with a label containing the BIOS publisher's name, the BIOS revision, and a copyright notice. The practice of covering the BIOS chip with a label is still commonly seen as of today, even though current BIOS chips are actually EEPROMs or NOR flashes, with no erase windows.

Erasure of the EPROM begins to occur with wavelengths shorter than 400 nm. Exposure time for sunlight of 1 week or 3 years for room fluorescent lighting may cause erasure. The recommended erasure procedure is exposure to UV light at 253.7 nm of at least 15 W-sec/cm2 for 20 to 30 minutes, with the lamp at a distance of about 1 inch.

Erasure can also be accomplished with X-rays:

"Erasure, however, has to be accomplished by non-electrical methods, since the gate electrode is not accessible electrically. Shining ultraviolet light on any part of an unpackaged device causes a photocurrent to flow from the floating gate back to the silicon substrate, thereby discharging the gate to its initial, uncharged condition. This method of erasure allows complete testing and correction of a complex memory array before the package is finally sealed. Once the package is sealed, information can still be erased by exposing it to X radiation in excess of $5*10^4$ rads, a dose which is easily attained with commercial X-ray generators." ($5*10^4$ rad = 500 J/kg)

"In other words, to erase your EPROM, you would first have to X-ray it and then put it in an oven at about 600 degrees Celsius (to anneal semiconductor alterations caused by the x-rays). The effects of this process on the reliability of the part would have required extensive testing so they decided on the window instead." (any temperature between 450 - 1410 °C should work).

EPROMs had a limited but large number of erase cycles; the silicon dioxide around the gates would accumulate damage from each cycle, making the chip unreliable after several thousand cycles. EPROM programming is slow compared to other forms of memory. Because higher-density parts have little exposed oxide between the layers of interconnects and gate, ultraviolet erasing becomes less practical for very large memories. Even dust inside the package can prevent some cells from being erased.

Application

For large volumes of parts (thousands of pieces or more), mask-programmed ROMs are the lowest cost devices to produce. However, these require many weeks lead time to make, since the artwork for an IC mask layer must be altered to store data on the ROMs. Initially, it was thought that the EPROM would be too expensive for mass production use and that it would be confined to development only. It was soon found that small-volume production was economical with EPROM parts, particularly when the advantage of rapid upgrades of firmware was considered.

Some microcontrollers, from before the era of EEPROMs and flash memory, use an on-chip EPROM to store their program. Such microcontrollers include some versions of the Intel 8048, the Freescale 68HC11, and the "C" versions of the PIC microcontroller. Like EPROM chips, such microcontrollers came in windowed (expensive) versions that were useful for debugging and program development. The same chip came in (somewhat cheaper) opaque OTP packages for production. Leaving the die of such a chip exposed to light can also change behavior in unexpected ways when moving from a windowed part used for development to a non-windowed part for production.

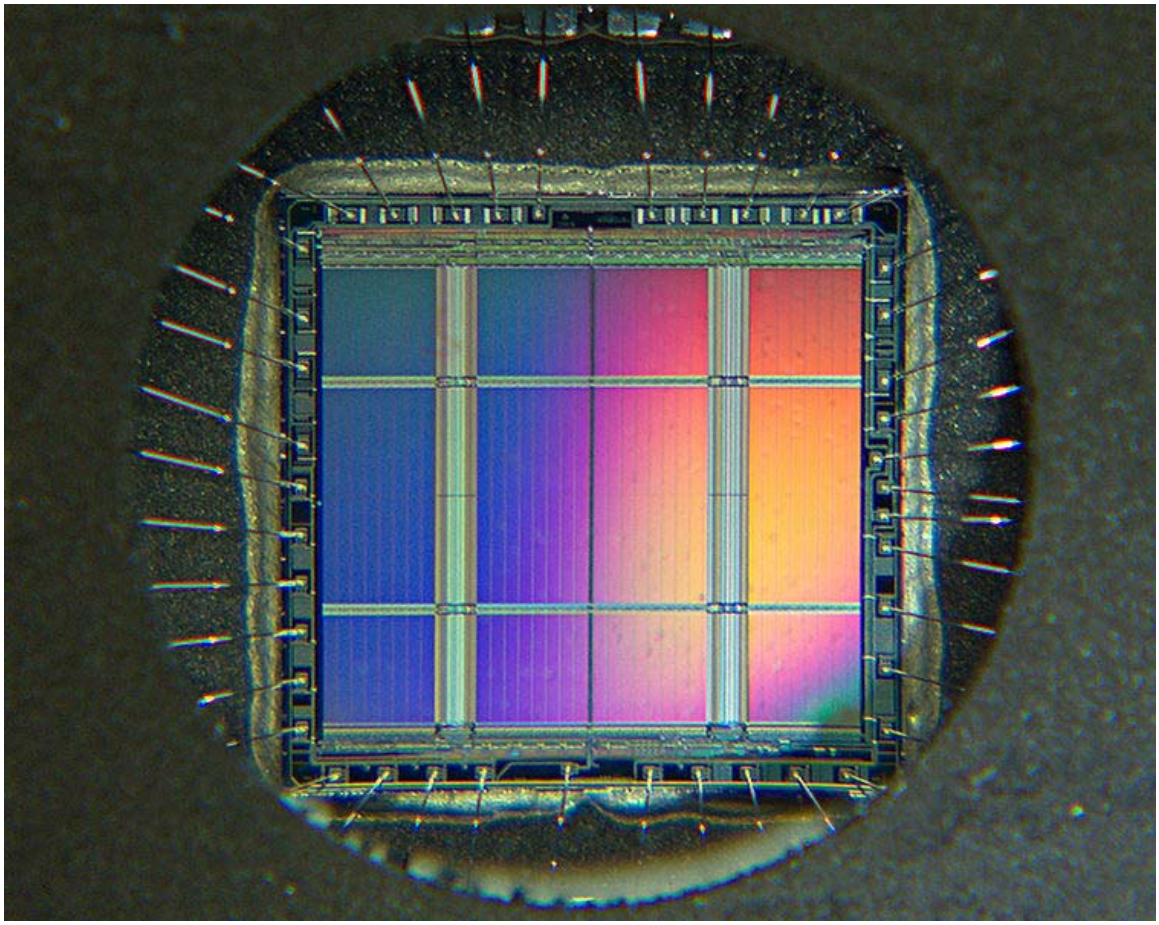
EPROM sizes and types

EPROMs come in several sizes both in physical packaging as well and storage capacity. While parts of the same type number from different manufacturers are compatible as long as they're only being read, there are subtle differences in the programming process.

Most EPROMS could be identified by the programmer through "signature mode" by forcing 12V on pin A9 and reading out two bytes of data. However, as this was not universal, programmer software also would allow manual setting of the manufacturer and device type of the chip to ensure proper programming.

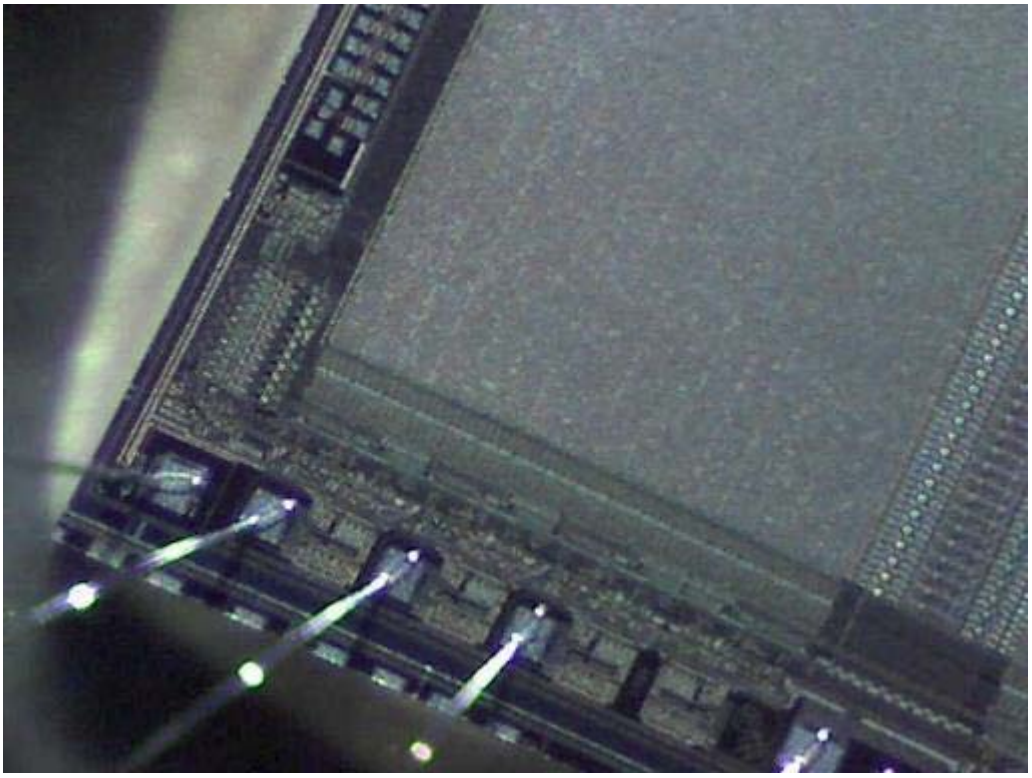
EPROM Type	Size — bits	Size — bytes	Length (hex)	Last address (hex)
1702, 1702A	2 Kbit	256	100	FF
2704	4 Kbit	512	200	1FF
2708	8 Kbit	1 KB	400	3FF
2716, 27C16	16 Kbit	2 KB	800	7FF
2732, 27C32	32 Kbit	4 KB	1000	FFF
2764, 27C64	64 Kbit	8 KB	2000	1FFF
27128, 27C128	128 Kbit	16 KB	4000	3FFF
27256, 27C256	256 Kbit	32 KB	8000	7FFF
27512, 27C512	512 Kbit	64 KB	10000	FFFF
27C010, 27C100	1 Mbit	128 KB	20000	1FFFF
27C020	2 Mbit	256 KB	40000	3FFFF
27C040, 27C400	4 Mbit	512 KB	80000	7FFFF

27C080	8 Mbit	1 MB	100000	FFFFF
27C160	16 Mbit	2 MB	200000	1FFFFF
27C320	32 Mbit	4 MB	400000	3FFFFF

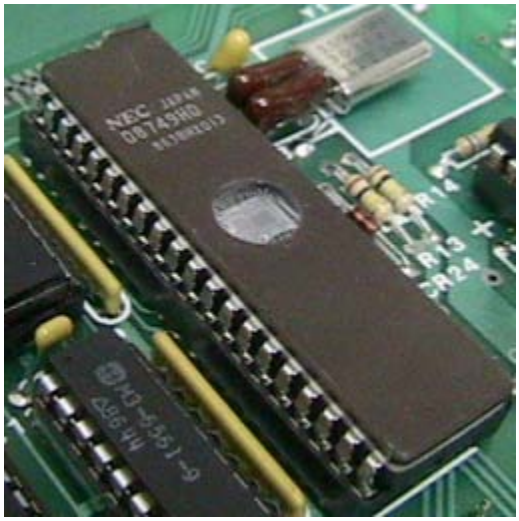


Close up of a EPROM die

A 32KB (256Kbit) EPROM



EPROM. 60x closeup



This 8749 Microcontroller stores its program in internal EPROM

Chapter 9

EEPROM

EEPROM stands for **E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory and is a type of non-volatile memory used in computers and other electronic devices to store small amounts of data that must be saved when power is removed, e.g., calibration tables or device configuration.

When larger amounts of static data are to be stored (such as in USB flash drives) a specific type of EEPROM such as flash memory is more economical than traditional EEPROM devices. EEPROMs are realized as arrays of floating-gate transistors.

EEPROM is user-modifiable read-only memory (ROM) that can be erased and reprogrammed (written to) repeatedly through the application of higher than normal electrical voltage generated externally or internally in the case of modern EEPROMs. EPROM usually must be removed from the device for erasing and programming, whereas EEPROMs can be programmed and erased in circuit. Originally, EEPROMs were limited to single byte operations which made them slower, but modern EEPROMs allow multi-byte page operations. It also has a limited life - that is, the number of times it could be reprogrammed was limited to tens or hundreds of thousands of times. That limitation has been extended to a million write operations in modern EEPROMs. In an EEPROM that is frequently reprogrammed while the computer is in use, the life of the EEPROM can be an important design consideration. It is for this reason that EEPROMs were used for configuration information, rather than random access memory.

History

In 1978, George Perlegos at Intel developed the Intel 2816, which was built on earlier EPROM technology, but used a thin gate oxide layer so that the chip could erase its own bits without requiring a UV source. Perlegos and others later left Intel to form Seeq Technology, which used on-device charge pumps to supply the high voltages necessary for programming EEPROMs.

Functions of EEPROM

There are different types of electrical interfaces to EEPROM devices. Main categories of these interface types are:

- Serial bus
- Parallel bus

How the device is operated depends on the electrical interface.

Serial bus devices

Most common serial interface types are SPI, I²C, Microwire, UNI/O, and 1-Wire. These interfaces require between 1 and 4 control signals for operation, resulting in a memory device in an 8 pin (or less) package.

The serial EEPROM (or **SEEPROM**) typically operates in three phases: OP-Code Phase, Address Phase and Data Phase. The OP-Code is usually the first 8-bits input to the serial input pin of the EEPROM device (or with most I²C devices, is implicit); followed by 8 to 24 bits of addressing depending on the depth of the device, then data to be read or written.

Each EEPROM device typically has its own set of OP-Code instructions to map to different functions. Some of the common operations on SPI EEPROM devices are:

- Write Enable (WREN)
- Write Disable (WRDI)
- Read Status Register (RDSR)
- Write Status Register (WRSR)
- Read Data (READ)
- Write Data (WRITE)

Other operations supported by some EEPROM devices are:

- Program
- Sector Erase
- Chip Erase commands

Parallel bus devices

Parallel EEPROM devices typically have an 8-bit data bus and an address bus wide enough to cover the complete memory. Most devices have chip select and write protect pins. Some microcontrollers also have integrated parallel EEPROM.

Operation of a parallel EEPROM is simple and fast when compared to serial EEPROM, but these devices are larger due to the higher pin count (28 pins or more) and have been decreasing in popularity in favor of serial EEPROM or Flash.

Other devices

EEPROM memory is used to enable features in other types of products that are not strictly memory products. Products such as real-time clocks, digital potentiometers, digital temperature sensors, among others, may have small amounts of EEPROM to store calibration information or other data that needs to be available in the event of power loss.

Failure modes

There are two limitations of stored information; endurance, and data retention.

During rewrites, the gate oxide in the floating-gate transistors gradually accumulates trapped electrons. The electric field of the trapped electrons adds to the electrons in the floating gate, lowering the window between threshold voltages for zeros vs ones. After sufficient number of rewrite cycles, the difference becomes too small to be recognizable, the cell is stuck in programmed state, and endurance failure occurs. The manufacturers usually specify the maximum number of rewrites being 10^6 or more.

During storage, the electrons injected into the floating gate may drift through the insulator, especially at increased temperature, and cause charge loss, reverting the cell into erased state. The manufacturers usually guarantee data retention of 10 years or more.

Related types

Flash memory is a later form of EEPROM. In the industry, there is a convention to reserve the term EEPROM to byte-wise erasable memories compared to block-wise erasable flash memories. EEPROM takes more die area than flash memory for the same capacity because each cell usually needs both a read, write and erase transistor, while in flash memory the erase circuits are shared by large blocks of cells (often 512×8).

Newer non-volatile memory technologies such as FeRAM and MRAM are slowly replacing EEPROMs in some applications, but are expected to remain a small fraction of the EEPROM market for the foreseeable future.

Comparison with EPROM and EEPROM/Flash

The difference between EPROM and EEPROM lies in the way that the memory programs and erases. EEPROM can be programmed and erased electrically using field electron emission (more commonly known in the industry as "Fowler–Nordheim tunneling").

EPROMs can't be erased electrically, and are programmed via hot carrier injection onto the floating gate. Erase is via an ultraviolet light source, although in practice many EPROMs are encapsulated in plastic that is opaque to UV light, and are "one-time programmable".

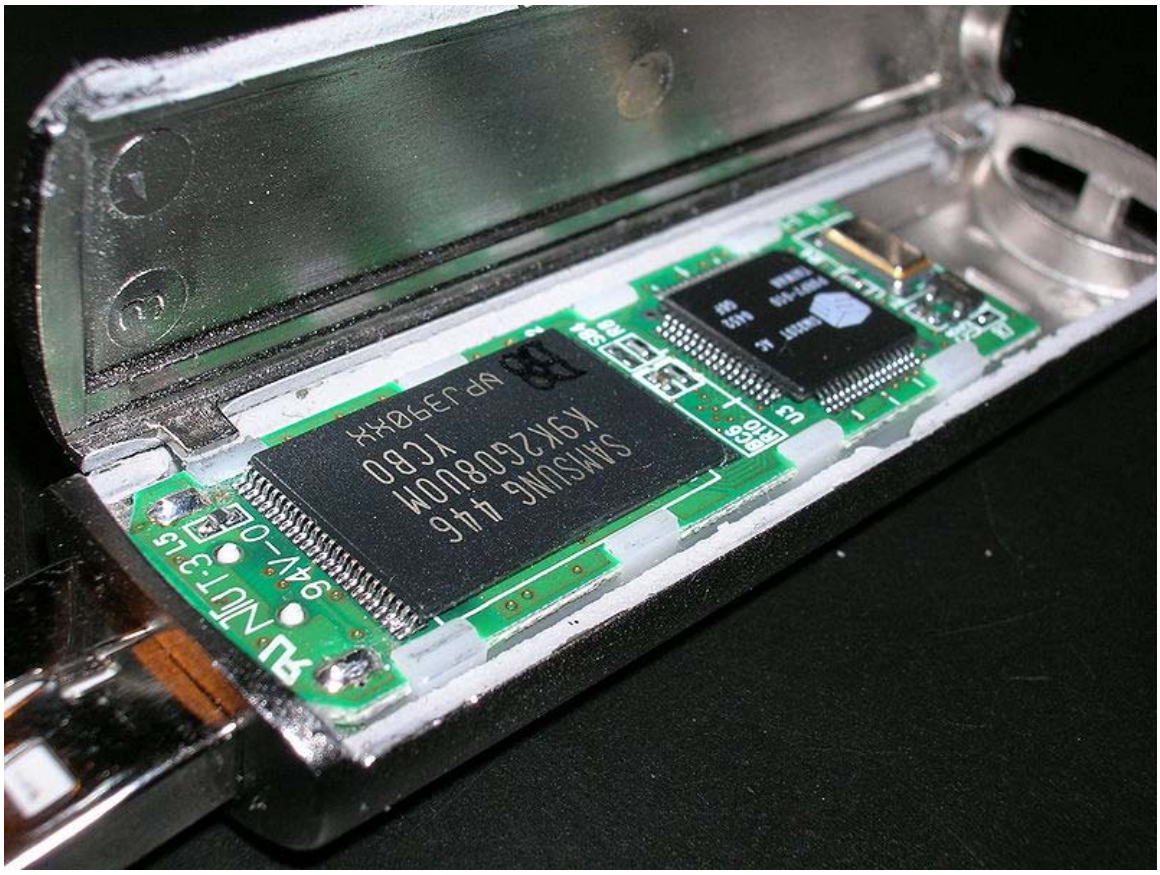
Most NOR Flash memory is a hybrid style—programming is through hot carrier injection and erase is through Fowler–Nordheim tunneling.

EEPROM manufacturers

- Atmel
- Hitachi
- Infineon
- Linear Technology
- Macronix Int'l Co.
- Maxwell Technologies
- Microchip Technology
- Mitsubishi
- NXP Semiconductors
- ON Semiconductor
- Renesas Technology
- ROHM
- Samsung Electronics
- STMicroelectronics
- Seiko Instruments
- Winbond

Chapter 10

Flash Memory



A USB flash drive. The chip on the left is the flash memory. The controller is on the right.

Flash memory is a non-volatile computer storage chip that can be electrically erased and reprogrammed. It is primarily used in memory cards, USB flash drives, MP3 players and solid-state drives for general storage and transfer of data between computers and other digital products. It is a specific type of EEPROM (electrically erasable programmable read-only memory) that is erased and programmed in large blocks; in early flash the entire chip had to be erased at once. Flash memory costs far less than byte-programmable EEPROM and therefore has become the dominant technology wherever a significant amount of non-volatile, solid state storage is needed. Example applications include PDAs

(personal digital assistants), laptop computers, digital audio players, digital cameras and mobile phones. It has also gained popularity in console video game hardware, where it is often used instead of EEPROMs or battery-powered static RAM (SRAM) for game save data. Flash memory is non-volatile, meaning no power is needed to maintain the information stored in the chip. In addition, flash memory offers fast read access times (although not as fast as volatile DRAM memory used for main memory in PCs) and better kinetic shock resistance than hard disks. These characteristics explain the popularity of flash memory in portable devices. Another feature of flash memory is that when packaged in a "memory card," it is extremely durable, being able to withstand intense pressure, extremes of temperature, and even immersion in water.

Although technically a type of EEPROM, the term "EEPROM" is generally used to refer specifically to non-flash EEPROM which is erasable in small blocks, typically bytes. Because erase cycles are slow, the large block sizes used in flash memory erasing give it a significant speed advantage over old-style EEPROM when writing large amounts of data.

History

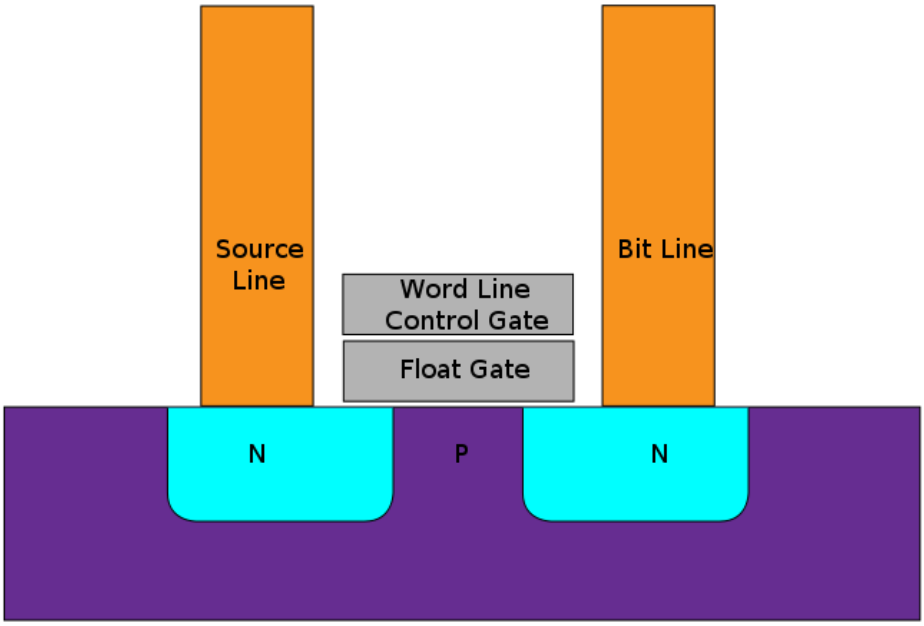
Flash memory (both NOR and NAND types) was invented by Dr. Fujio Masuoka while working for Toshiba circa 1980. According to Toshiba, the name "flash" was suggested by Dr. Masuoka's colleague, Mr. Shoji Ariizumi, because the erasure process of the memory contents reminded him of the flash of a camera. Dr. Masuoka presented the invention at the *IEEE 1984 International Electron Devices Meeting* (IEDM) held in San Francisco, California.

Intel Corporation saw the massive potential of the invention and introduced the first commercial NOR type flash chip in 1988. NOR-based flash has long erase and write times, but provides full address and data buses, allowing random access to any memory location. This makes it a suitable replacement for older read-only memory (ROM) chips, which are used to store program code that rarely needs to be updated, such as a computer's BIOS or the firmware of set-top boxes. Its endurance is 10,000 to 1,000,000 erase cycles. NOR-based flash was the basis of early flash-based removable media; CompactFlash was originally based on it, though later cards moved to less expensive NAND flash.

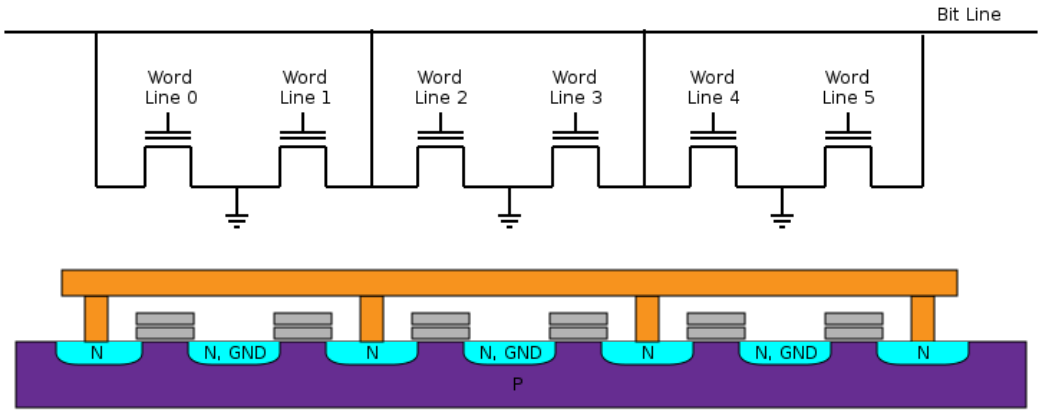
Toshiba announced NAND flash at the 1987 *International Electron Devices Meeting*. It has reduced erase and write times, and requires less chip area per cell, thus allowing greater storage density and lower cost per bit than NOR flash; it also has up to ten times the endurance of NOR flash. However, the I/O interface of NAND flash does not provide a random-access external address bus. Rather, data must be read on a block-wise basis, with typical block sizes of hundreds to thousands of bits. This made NAND flash unsuitable as a drop-in replacement for program ROM since most microprocessors and microcontrollers required byte-level random access. In this regard NAND flash is similar to other secondary storage devices such as hard disks and optical media, and is thus very suitable for use in mass-storage devices such as memory cards. The first NAND-based

removable media format was SmartMedia in 1995, and many others have followed, including MultiMediaCard, Secure Digital, Memory Stick and xD-Picture Card. A new generation of memory card formats, including RS-MMC, miniSD and microSD, and Intelligent Stick, feature extremely small form factors. For example, the microSD card has an area of just over 1.5 cm^2, with a thickness of less than 1 mm. microSD capacities range from 64 MB to 32 GB, as of March 2010.

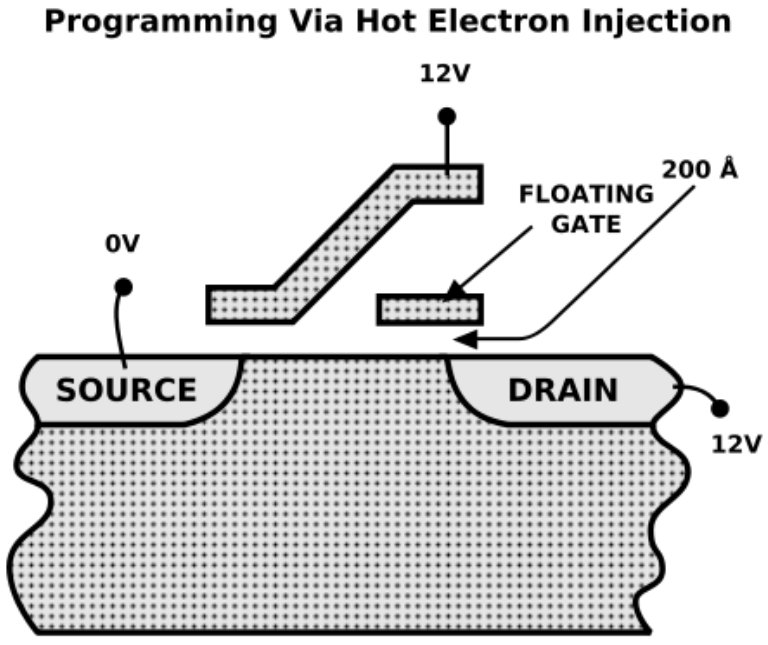
Principles of operation



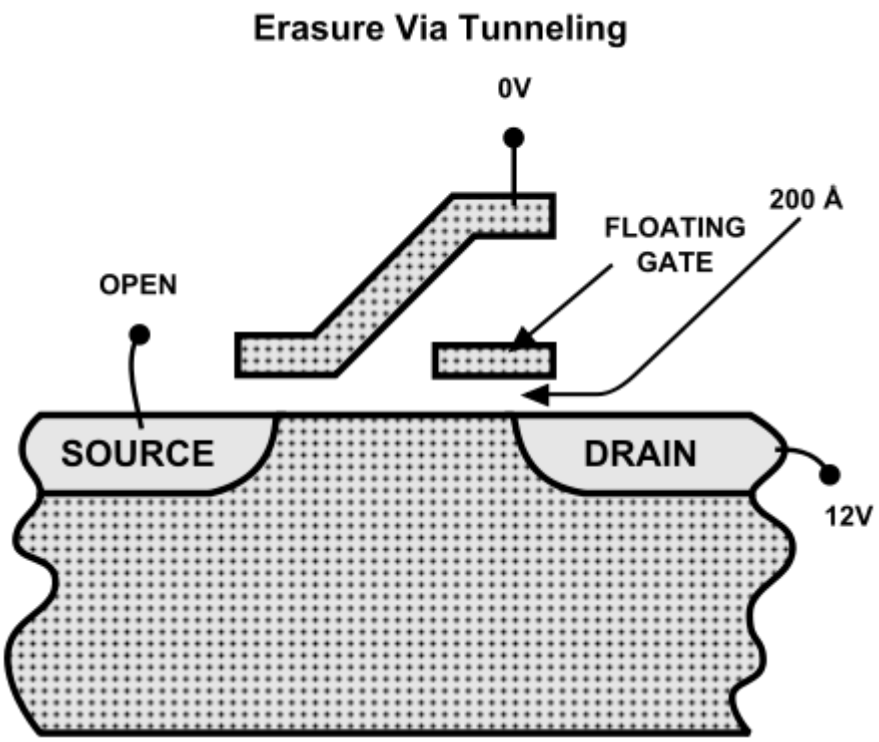
A flash memory cell



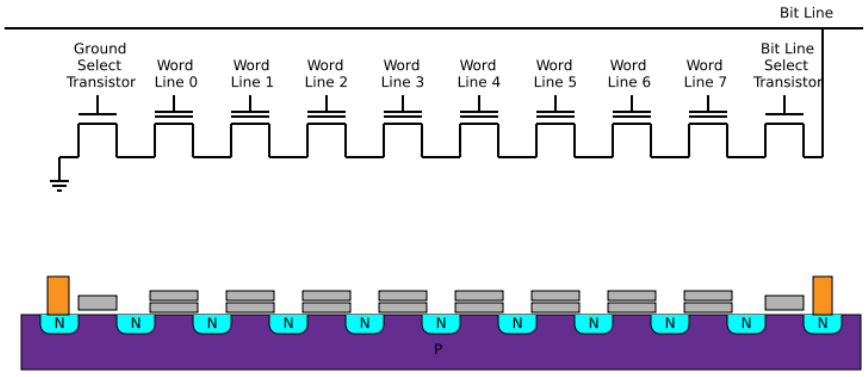
NOR flash memory wiring and structure on silicon



Programming a NOR memory cell (setting it to logical 0), via hot-electron injection.



Erasing a NOR memory cell (setting it to logical 1), via quantum tunneling



NAND flash memory wiring and structure on silicon

Flash memory stores information in an array of memory cells made from floating-gate transistors. In traditional single-level cell (SLC) devices, each cell stores only one bit of information. Some newer flash memory, known as multi-level cell (MLC) devices, can store more than one bit per cell by choosing between multiple levels of electrical charge to apply to the floating gates of its cells.

The floating gate may be conductive (typically polysilicon in most kinds of flash memory) or non-conductive (as in SONOS flash memory).

Floating-gate transistor

In flash memory, each memory cell resembles a standard MOSFET, except the transistor has two gates instead of one. On top is the control gate (CG), as in other MOS transistors, but below this there is a floating gate (FG) insulated all around by an oxide layer. The FG is interposed between the CG and the MOSFET channel. Because the FG is electrically isolated by its insulating layer, any electrons placed on it are trapped there and, under normal conditions, will not discharge for many years. When the FG holds a charge, it screens (partially cancels) the electric field from the CG, which modifies the threshold voltage (V_T) of the cell. During read-out, a voltage intermediate between the possible threshold voltages is applied to the CG, and the MOSFET channel will become conducting or remain insulating, depending on the V_T of the cell, which is in turn controlled by charge on the FG. The current flow through the MOSFET channel is sensed and forms a binary code, reproducing the stored data. In a multi-level cell device, which stores more than one bit per cell, the amount of current flow is sensed (rather than simply its presence or absence), in order to determine more precisely the level of charge on the FG.

NOR flash

In NOR gate flash, each cell has one end connected directly to ground, and the other end connected directly to a bit line.

This arrangement is called "NOR flash" because it acts like a NOR gate: when one of the word lines is brought high, the corresponding storage transistor acts to pull the output bit line low.

Programming

A single-level NOR flash cell in its default state is logically equivalent to a binary "1" value, because current will flow through the channel under application of an appropriate voltage to the control gate. A NOR flash cell can be programmed, or set to a binary "0" value, by the following procedure:

- an elevated on-voltage (typically >5 V) is applied to the CG
- the channel is now turned on, so electrons can flow from the source to the drain (assuming an NMOS transistor)

- the source-drain current is sufficiently high to cause some high energy electrons to jump through the insulating layer onto the FG, via a process called hot-electron injection

Erasing

To erase a NOR flash cell (resetting it to the "1" state), a large voltage *of the opposite polarity* is applied between the CG and source, pulling the electrons off the FG through quantum tunneling. Modern NOR flash memory chips are divided into erase segments (often called blocks or sectors). The erase operation can only be performed on a block-wise basis; all the cells in an erase segment must be erased together. Programming of NOR cells, however, can generally be performed one byte or word at a time.

Internal charge pumps

Despite the need for high programming and erasing voltages, virtually all flash chips today require only a single supply voltage, and produce the high voltages via on-chip charge pumps.

NAND flash

NAND flash also uses floating-gate transistors, but they are connected in a way that resembles a NAND gate: several transistors are connected in series, and only if all word lines are pulled high (above the transistors' V_T) is the bit line pulled low. These groups are then connected via some additional transistors to a NOR-style bit line array.

To read, most of the word lines are pulled up above the V_T of a programmed bit, while one of them is pulled up to just over the V_T of an erased bit. The series group will conduct (and pull the bit line low) if the selected bit has not been programmed.

Despite the additional transistors, the reduction in ground wires and bit lines allows a denser layout and greater storage capacity per chip. In addition, NAND flash is typically permitted to contain a certain number of faults (NOR flash, as is used for a BIOS ROM, is expected to be fault-free). Manufacturers try to maximize the amount of usable storage by shrinking the size of the transistor below the size where they can be made reliably, to the size where further reductions would increase the number of faults faster than it would increase the total storage available.

NAND flash uses tunnel injection for writing and tunnel release for erasing. NAND flash memory forms the core of the removable USB storage devices known as USB flash drives, as well as most memory card formats and solid-state drives available today.

Limitations

Block erasure

One limitation of flash memory is that although it can be read or programmed a byte or a word at a time in a random access fashion, it can only be erased a "block" at a time. This generally sets all bits in the block to 1. Starting with a freshly erased block, any location within that block can be programmed. However, once a bit has been set to 0, only by erasing the entire block can it be changed back to 1. In other words, flash memory (specifically NOR flash) offers random-access read and programming operations, but cannot offer arbitrary random-access rewrite or erase operations. A location can, however, be rewritten as long as the new value's 0 bits are a superset of the over-written value's. For example, a nibble value may be erased to 1111, then written as 1110. Successive writes to that nibble can change it to 1010, then 0010, and finally 0000. Essentially, erasure sets (all) bits, and programming can only clear bits. Filesystems designed for flash devices can make use of this capability to represent sector metadata.

Although data structures in flash memory cannot be updated in completely general ways, this allows members to be "removed" by marking them as invalid. This technique may need to be modified for Multi-level Cell devices, where one memory cell holds more than one bit.

Common flash devices such as USB keys and memory cards provide only a block-level interface, or flash translation layer (FTL), which writes to a different cell each time to wear-level the device. This prevents incremental writing within a block, however it does help the device from being prematurely worn out by abusive and/or poorly designed hardware/software. For example, nearly all consumer devices ship formatted with MS-FAT file system, which pre-dates flash memory, having been designed for DOS, and disk media.

Memory wear

Another limitation is that flash memory has a finite number of program-erase cycles (typically written as P/E cycles). Most commercially available flash products are guaranteed to withstand around 100,000 P/E cycles, before the wear begins to deteriorate the integrity of the storage. Micron Technology and Sun Microsystems announced an SLC flash memory chip rated for 1,000,000 P/E cycles on December 17, 2008.

The guaranteed cycle count may apply only to block zero (as is the case with TSOP NAND parts), or to all blocks (as in NOR). This effect is partially offset in some chip firmware or file system drivers by counting the writes and dynamically remapping blocks in order to spread write operations between sectors; this technique is called wear leveling. Another approach is to perform write verification and remapping to spare sectors in case of write failure, a technique called Bad Block Management (BBM). For portable consumer devices, these wearout management techniques typically extend the life of the flash memory beyond the life of the device itself, and some data loss may be acceptable

in these applications. For high reliability data storage, however, it is not advisable to use flash memory that would have to go through a large number of programming cycles. This limitation is meaningless for 'read-only' applications such as thin clients and routers, which are only programmed once or at most a few times during their lifetimes.

Low-level access

The low-level interface to flash memory chips differs from those of other memory types such as DRAM, ROM, and EEPROM, which support bit-alterability (both zero to one and one to zero) and random-access via externally accessible address buses.

While NOR memory provides an external address bus for read and program operations (and thus supports random-access); unlocking and erasing NOR memory must proceed on a block-by-block basis. With NAND flash memory, read and programming operations must be performed page-at-a-time while unlocking and erasing must happen in block-wise fashion.

NOR memories

Reading from NOR flash is similar to reading from random-access memory, provided the address and data bus are mapped correctly. Because of this, most microprocessors can use NOR flash memory as execute in place (XIP) memory, meaning that programs stored in NOR flash can be executed directly from the NOR flash without needing to be copied into RAM first. NOR flash may be programmed in a random-access manner similar to reading. Programming changes bits from a logical one to a zero. Bits that are already zero are left unchanged. Erasure must happen a block at a time, and resets all the bits in the erased block back to one. Typical block sizes are 64, 128, or 256 KB.

Bad block management is a relatively new feature in NOR chips. In older NOR devices not supporting bad block management, the software or device driver controlling the memory chip must correct for blocks that wear out, or the device will cease to work reliably.

The specific commands used to lock, unlock, program, or erase NOR memories differ for each manufacturer. To avoid needing unique driver software for every device made, a special set of Common Flash Memory Interface (CFI) commands allow the device to identify itself and its critical operating parameters.

Apart from being used as random-access ROM, NOR memories can also be used as storage devices by taking advantage of random-access programming. Some devices offer read-while-write functionality so that code continues to execute even while a program or erase operation is occurring in the background. For sequential data writes, NOR flash chips typically have slow write speeds compared with NAND flash.

NAND memories

NAND flash architecture was introduced by Toshiba in 1989. These memories are accessed much like block devices such as hard disks or memory cards. Each block consists of a number of pages. The pages are typically 512 or 2,048 or 4,096 bytes in size. Associated with each page are a few bytes (typically 1/32 of the data size) that can be used for storage of an error correcting code (ECC) checksum.

Typical block sizes include:

- 32 pages of 512+16 bytes each for a block size of 16 KB
- 64 pages of 2,048+64 bytes each for a block size of 128 KB
- 64 pages of 4,096+128 bytes each for a block size of 256 KB
- 128 pages of 4,096+128 bytes each for a block size of 512 KB.

While reading and programming is performed on a page basis, erasure can only be performed on a block basis. Number of Operations (NOPs) is the number of times the pages can be programmed. So far this number for MLC flash is always one whereas for SLC flash it is four.

NAND devices also require bad block management by the device driver software, or by a separate controller chip. SD cards, for example, include controller circuitry to perform bad block management and wear leveling. When a logical block is accessed by high-level software, it is mapped to a physical block by the device driver or controller. A number of blocks on the flash chip may be set aside for storing mapping tables to deal with bad blocks, or the system may simply check each block at power-up to create a bad block map in RAM. The overall memory capacity gradually shrinks as more blocks are marked as bad.

NAND relies on ECC to compensate for bits that may spontaneously fail during normal device operation. A typical ECC will correct a one bit error in each 2048 bits (256 bytes) using 22 bits of ECC code, or a one bit error in each 4096 bits (512 bytes) using 24 bits of ECC code. If ECC cannot correct the error during read, it may still detect the error. When doing erase or program operations, the device can detect blocks that fail to program or erase and mark them bad. The data is then written to a different, good block, and the bad block map is updated.

Most NAND devices are shipped from the factory with some bad blocks which are typically identified and marked according to a specified bad block marking strategy. By allowing some bad blocks, the manufacturers achieve far higher yields than would be possible if all blocks had to be verified good. This significantly reduces NAND flash costs and only slightly decreases the storage capacity of the parts.

When executing software from NAND memories, virtual memory strategies are often used: memory contents must first be paged or copied into memory-mapped RAM and executed there (leading to the common combination of NAND + RAM). A memory

management unit (MMU) in the system is helpful, but this can also be accomplished with overlays. For this reason, some systems will use a combination of NOR and NAND memories, where a smaller NOR memory is used as software ROM and a larger NAND memory is partitioned with a file system for use as a non-volatile data storage area.

NAND is best suited to systems requiring high capacity data storage. This type of flash architecture offers higher densities and larger capacities at lower cost with faster erase, sequential write, and sequential read speeds, sacrificing the random-access and execute in place advantage of the NOR architecture.

Standardization

A group called the Open NAND Flash Interface Working Group (ONFI) has developed a standardized low-level interface for NAND flash chips. This allows interoperability between conforming NAND devices from different vendors. The ONFI specification version 1.0 was released on December 28, 2006. It specifies:

- a standard physical interface (pinout) for NAND flash in TSOP-48, WSOP-48, LGA-52, and BGA-63 packages
- a standard command set for reading, writing, and erasing NAND flash chips
- a mechanism for self-identification (comparable to the Serial Presence Detection feature of SDRAM memory modules)

The ONFI group is supported by major NAND flash manufacturers, including Hynix, Intel, Micron Technology, and Numonyx, as well as by major manufacturers of devices incorporating NAND flash chips.

A group of vendors, including Intel, Dell, and Microsoft formed a Non-Volatile Memory Host Controller Interface (NVMHCI) Working Group. The goal of the group is to provide standard software and hardware programming interfaces for nonvolatile memory subsystems, including the "flash cache" device connected to the PCI Express bus.

Distinction between NOR and NAND flash

NOR and NAND flash differ in two important ways:

- the connections of the individual memory cells are different
- the interface provided for reading and writing the memory is different (NOR allows random-access for reading, NAND allows only page access)

These two are linked by the design choices made in the development of NAND flash. A goal of NAND flash development was to reduce the chip area required to implement a given capacity of flash memory, and thereby to reduce cost per bit and increase maximum chip capacity so that flash memory could compete with magnetic storage devices like hard disks.

NOR and NAND flash get their names from the structure of the interconnections between memory cells. In NOR flash, cells are connected in parallel to the bit lines, allowing cells to be read and programmed individually. The parallel connection of cells resembles the parallel connection of transistors in a CMOS NOR gate. In NAND flash, cells are connected in series, resembling a NAND gate. The series connections consume less space than parallel ones, reducing the cost of NAND flash. It does not, by itself, prevent NAND cells from being read and programmed individually.

When NOR flash was developed, it was envisioned as a more economical and conveniently rewritable ROM than contemporary EPROM and EEPROM memories. Thus random-access reading circuitry was necessary. However, it was expected that NOR flash ROM would be read much more often than written, so the write circuitry included was fairly slow and could only erase in a block-wise fashion. On the other hand, applications that use flash as a replacement for disk drives do not require word-level write address, which would only add to the complexity and cost unnecessarily.

Because of the series connection and removal of wordline contacts, a large grid of NAND flash memory cells will occupy perhaps only 60% of the area of equivalent NOR cells (assuming the same CMOS process resolution, e.g. 130nm, 90 nm, 65 nm). NAND flash's designers realized that the area of a NAND chip, and thus the cost, could be further reduced by removing the external address and data bus circuitry. Instead, external devices could communicate with NAND flash via sequential-accessed command and data registers, which would internally retrieve and output the necessary data. This design choice made random-access of NAND flash memory impossible, but the goal of NAND flash was to replace hard disks, not to replace ROMs.

Write endurance

The write endurance of SLC floating-gate NOR flash is typically equal or greater than that of NAND flash, while MLC NOR and NAND flash have similar endurance capabilities. Example Endurance cycle ratings listed in datasheets for NAND and NOR flash are provided.

- SLC NAND flash is typically rated at about 100k cycles (Samsung OneNAND KFW4G16Q2M)
- MLC NAND flash is typically rated at about 5–10k cycles (Samsung K9G8G08U0M)
- SLC floating-gate NOR flash has typical endurance rating of 100k to 1M cycles (Numonyx M58BW 100k; Spansion S29CD016J 1,000k)
- MLC floating-gate NOR flash has typical endurance rating of 100k cycles (Numonyx J3 flash)

However, by applying certain algorithms and design paradigms such as wear leveling and memory over-provisioning, the endurance of a storage system can be tuned to serve specific requirements.

Flash file systems

Because of the particular characteristics of flash memory, it is best used with either a controller to perform wear leveling and error correction or specifically designed flash file systems, which spread writes over the media and deal with the long erase times of NOR flash blocks. The basic concept behind flash file systems is: When the flash store is to be updated, the file system will write a new copy of the changed data to a fresh block, remap the file pointers, then erase the old block later when it has time.

In practice, flash file systems are only used for memory technology devices (MTDs), which are embedded flash memories that do not have a controller. Removable flash memory cards and USB flash drives have built-in controllers to perform wear leveling and error correction so use of a specific flash file system does not add any benefit.

Capacity

Multiple chips are often arrayed to achieve higher capacities for use in consumer electronic devices such as multimedia players or GPS. The capacity of flash chips generally follows Moore's Law because they are manufactured with many of the same integrated circuits techniques and equipment.

Consumer flash drives typically have sizes measured in powers of two (e.g., 512 MB, 8 GB). This includes SSDs as hard drive replacements, even though traditional hard drives tend to use decimal units. Thus, an SSD marked as "64 GB" is actually $64 \times 1{,}024^3$ bytes (64 GB). In reality, most users will have slightly less capacity than this available, due to the space taken by file system metadata.

In 2005, Toshiba and SanDisk developed a NAND flash chip capable of storing 1 GB of data using multi-level cell (MLC) technology, capable of storing two bits of data per cell. In September 2005, Samsung Electronics announced that it had developed the world's first 2 GB chip.

In March 2006, Samsung announced flash hard drives with a capacity of 4 GB, essentially the same order of magnitude as smaller laptop hard drives, and in September 2006, Samsung announced an 8 GB chip produced using a 40-nm manufacturing process. In January 2008, Sandisk announced availability of their 16 GB MicroSDHC and 32 GB SDHC Plus cards.

In 2009, Kingston announced a 256 GB flash drive available only in the UK and other parts of Europe. As of 2010, however, it is available in the USA.

There are still flash-chips manufactured with capacities under or around 1 MB, e.g., for BIOS-ROMs and embedded applications.

Transfer rates

NAND flash memory cards are much faster at reading than writing so it is the maximum read speed that is commonly advertised.

As a chip wears out, its erase/program operations slow down considerably, requiring more retries and bad block remapping. Transferring multiple small files, each smaller than the chip-specific block size, could lead to much a lower rate. Access latency also influences performance, but less so than with their hard drive counterpart.

The speed is sometimes quoted in MB/s (megabytes per second), or as a multiple of that of a legacy single speed CD-ROM, such as 60×, 100× or 150×. Here 1× is equivalent to 150 kB/s. For example, a 100× memory card gives 150 kB/s × 100 = 15,000 kB/s = 14.65 MB/s.

Performance also depends on the quality of memory controllers. Even when the only change to manufacturing is die-shrink, the absence of an appropriate controller can result in degraded speeds.

Applications

Serial flash

Serial flash is a small, low-power flash memory that uses a serial interface, typically SPI, for sequential data access. When incorporated into an embedded system, serial flash requires fewer wires on the PCB than parallel flash memories, since it transmits and receives data one bit at a time. This may permit a reduction in board space, power consumption, and total system cost.

There are several reasons why a serial device, with fewer external pins than a parallel device, can significantly reduce overall cost:

- Many ASICs are pad-limited, meaning that the size of the die is constrained by the number of wire bond pads, rather than the complexity and number of gates used for the device logic. Eliminating bond pads thus permits a more compact integrated circuit, on a smaller die; this increases the number of dies that may be fabricated on a wafer, and thus reduces the cost per die.
- Reducing the number of external pins also reduces assembly and packaging costs. A serial device may be packaged in a smaller and simpler package than a parallel device.
- Smaller and lower pin-count packages occupy less PCB area.
- Lower pin-count devices simplify PCB routing.

There are two major SPI flash types. The Atmel AT45 *DataFlash*™ was the first type and is characterized by small pages and one or more internal SRAM page buffers allowing a complete page to be read to the buffer, partially modified, and then written

back. The second type is called *SPI flash* and typically has larger sectors. The smallest sectors typically found in an SPI flash are 4 kB, but they can be as large as 64 kB. Since the SPI flash lacks an internal SRAM buffer, the complete page must be read out and modified before written back, making it slow to manage. *SPI flash* is cheaper than *DataFlash* and is therefore a good choice when the application is code shadowing.

The two types are not easily exchangeable, since they do not have the same pinout, and the command sets are incompatible.

Firmware storage

With the increasing speed of modern CPUs, parallel flash devices are often much slower than the memory bus of the computer they are connected to. Conversely, modern SRAM offers access times below 10 ns, while DDR2 SDRAM offers access times below 20 ns. Because of this, it is often desirable to shadow code stored in flash into RAM; that is, the code is copied from flash into RAM before execution, so that the CPU may access it at full speed. Device firmware may be stored in a serial flash device, and then copied into SDRAM or SRAM when the device is powered-up. Using an external serial flash device rather than on-chip flash removes the need for significant process compromise (a process that is good for high speed logic is generally not good for flash and vice-versa). Once it is decided to read the firmware in as one big block it is common to add compression to allow a smaller flash chip to be used. Typical applications for serial flash include storing firmware for hard drives, Ethernet controllers, DSL modems, wireless network devices, etc.

Flash memory as a replacement for hard drives

One more recent application for flash memory is as a replacement for hard disks. Flash memory does not have the mechanical limitations and latencies of hard drives, so a solid-state drive, or SSD, is attractive when considering speed, noise, power consumption, and reliability. Flash drives are gaining traction as mobile device secondary storage devices; they are also used as hard drives in high-performance desktop computers and some servers with RAID and SAN architectures.

There remain some aspects of flash-based SSDs that make them unattractive. Most important, the cost per gigabyte of flash memory remains significantly higher than that of platter-based hard drives. Although this ratio is decreasing rapidly for flash memory, it is not yet clear that flash memory will catch up to the capacities and affordability offered by platter-based storage. Still, research and development is sufficiently vigorous that it is not clear that it will not happen, either.

There is also some concern that the finite number of P/E cycles of flash memory would render flash memory unable to support an operating system. This seems to be a decreasing issue as warranties on flash-based SSDs are approaching those of current hard drives.

In June 2006, Samsung Electronics released the first flash-memory based PCs, the Q1-SSD and Q30-SSD, both of which used 32 GB SSDs, and were at least initially available only in South Korea. Dell Computer introduced a 32GB SSD option on its Latitude D420 and D620 ATG laptops in April 2007—at $549 more than a hard-drive equipped version.

At the Las Vegas CES 2007 Summit Taiwanese memory company A-DATA showcased SSD hard disk drives based on flash technology in capacities of 32 GB, 64 GB and 128 GB. Sandisk announced an OEM 32 GB 1.8" SSD drive at CES 2007. The XO-1, developed by the One Laptop Per Child (OLPC) association, uses flash memory rather than a hard drive. As of March 2009, a Salt Lake City company called Fusion-io claims the fastest SSD with sequential read/write speeds of 1500 MB/1400 MB's per second.

Rather than entirely replacing the hard drive, hybrid techniques such as hybrid drive and ReadyBoost attempt to combine the advantages of both technologies, using flash as a high-speed cache for files on the disk that are often referenced, but rarely modified, such as application and operating system executable files. Also, Addonics has a PCI adapter for four CF cards, creating a RAID-able array of solid-state storage that is much cheaper than the hardwired-chips PCI card kind.

Early versions of the ASUS Eee PC used a flash-based SSD of 2 GB to 20 GB, depending on model, although later versions of the machine use conventional hard disks. The Apple Inc. Macbook Air has the option to upgrade the standard hard drive to a 128 GB Solid State hard drive. The Lenovo ThinkPad X300 also features a built-in 64 GB Solid State Drive. The Apple iPad has flash-based SSD's of 16, 32, and 64 GB.

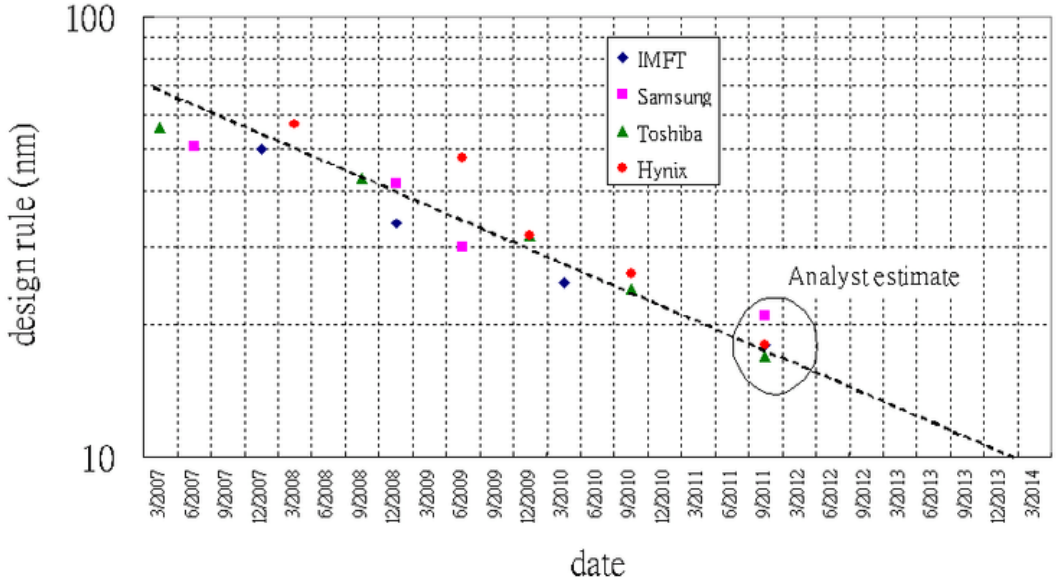
Sharkoon has developed a device that uses six SDHC cards in RAID-0 configuration as an SSD alternative; users may use more affordable High-Speed 8 GB SDHC cards to get similar or better results than can be obtained from traditional SSDs at a lower cost.

In mid-2010, Apple Inc. introduced MacBook Air notebooks that use flash as the primary device of storage instead of hard drives. However, Apple has already seen success with this hardware model in its most successful iPhones and iPads.

Industry

One source states that, in 2008, the flash memory industry includes about US$9.1 billion in production and sales. Apple Inc. is the third largest purchaser of flash memory, consuming about 13% of production by itself. Other sources put the flash memory market at a size of more than US$20 billion in 2006, accounting for more than eight percent of the overall semiconductor market and more than 34 percent of the total semiconductor memory market.

Flash scalability



The aggressive trend of process design rule shrinks in NAND flash memory technology effectively accelerates Moore's Law.

Due to its relatively simple structure and high demand for higher capacity, NAND flash memory is the most aggressively scaled technology among electronic devices. The heavy competition among the top few manufacturers only adds to the aggressiveness. Current projections show the technology to reach approximately 20 nm by around late 2011. While the expected shrink timeline is a factor of two every three years per original version of Moore's law, this has recently been accelerated in the case of NAND flash to a factor of two every two years.

As the feature size of flash memory cells reach the minimum limit (currently estimated ~20 nm), further flash density increases will be driven by greater levels of MLC, possibly 3-D stacking of transistors, and improvements to the manufacturing process. The decrease in endurance and increase in uncorrectable bit error rates that accompany feature size shrinking can be compensated by improved error correction mechanisms. Even with these advances, it may be impossible to economically scale flash to smaller and smaller dimensions. Many promising new technologies (such as FeRAM, MRAM, PMC, PCM, and others) are under investigation and development as possible more scalable replacements for flash.

Chapter 11

Shared Memory

In computing, **shared memory** is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies. Depending on context, programs may run on a single processor or on multiple separate processors. Using memory for communication inside a single program, for example among its multiple threads, is generally not referred to as *shared memory*.

In hardware

In computer hardware, **shared memory** refers to a (typically) large block of random access memory that can be accessed by several different central processing units (CPUs) in a multiple-processor computer system.

A shared memory system is relatively easy to program since all processors share a single view of data and the communication between processors can be as fast as memory accesses to a same location.

The issue with shared memory systems is that many CPUs need fast access to memory and will likely cache memory, which has two complications:

- CPU-to-memory connection becomes a bottleneck. Shared memory computers cannot scale very well. Most of them have ten or fewer processors.
- Cache coherence: Whenever one cache is updated with information that may be used by other processors, the change needs to be reflected to the other processors, otherwise the different processors will be working with incoherent data. Such coherence protocols can, when they work well, provide extremely high-performance access to shared information between multiple processors. On the other hand they can sometimes become overloaded and become a bottleneck to performance

The alternatives to shared memory are distributed memory and distributed shared memory, each having a similar set of issues.

In software

In computer software, *shared memory* is either

- a method of inter-process communication (IPC), i.e. a way of exchanging data between programs running at the same time. One process will create an area in RAM which other processes can access, *or*
- a method of conserving memory space by directing accesses to what would ordinarily be copies of a piece of data to a single instance instead, by using virtual memory mappings or with explicit support of the program in question. This is most often used for shared libraries and for XIP.

Since both processes can access the shared memory area like regular working memory, this is a very fast way of communication (as opposed to other mechanisms of IPC such as named pipes, Unix domain sockets or CORBA). On the other hand, it is less powerful, as for example the communicating processes must be running on the same machine (whereas other IPC methods can use a computer network), and care must be taken to avoid issues if processes sharing memory are running on separate CPUs and the underlying architecture is not cache coherent.

IPC by shared memory is used for example to transfer images between the application and the X server on Unix systems, or inside the IStream object returned by CoMarshalInterThreadInterfaceInStream in the COM libraries under Windows.

Dynamic libraries are generally held in memory once and mapped to multiple processes, and only pages that had to be customized for the individual process (because a symbol resolved differently there) are duplicated, usually with a mechanism that transparently copies the page when a write is attempted, and then lets the write succeed on the private copy.

POSIX provides a standardized API for using shared memory, *POSIX Shared Memory*. This uses the function shm_open from sys/mman.h.

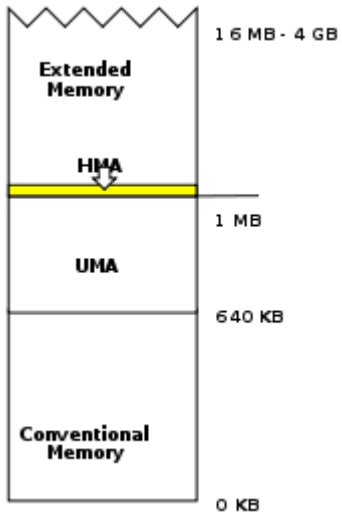
Unix System 5 provides an API for shared memory as well. This uses shmget from sys/shm.h.

BSD systems provide "anonymous mapped memory" which can be used by several processes.

Recent 2.6 Linux kernel builds have started to offer /dev/shm as shared memory in the form of a RAM disk, more specifically as a world-writable directory that is stored in memory. /dev/shm support is completely optional within the kernel configuration file. It is included by default in both Fedora and Ubuntu distributions.

Chapter 12

DOS Memory Management



Physical memory areas of the IBM PC family

In IBM PC compatible computing, **DOS memory management** refers to software and techniques employed to give applications access to more than 640K of "conventional memory". Memory management on the IBM family was made complex by the need to maintain backward compatibility to the original PC design and real-mode PC DOS (MS DOS), while allowing computer users to take advantage of large amounts of low-cost memory and new generations of processors. Since MS DOS has given way to Windows and other 32-bit operating systems, managing the memory of a personal computer no longer requires the user to manually manipulate internal settings and parameters of the system.

Conventional memory

The 8088 processor used in the original IBM PC had 20 address lines and so could directly address 1 megabyte of memory. Different areas of this address space were allocated to different kinds of memory used for different purposes. Starting at the lowest end of the address space, the PC had read/write memory (RAM) installed, which was used by PC DOS and application programs. The first part of this memory was installed on the motherboard of the system (in very early machines, 64 kilobytes, later revised to 256

kilobytes). Additional memory could be added on cards plugged into the expansion slots; each card contained straps or switches to control what part of the address space access memory and devices on that card.

On the IBM PC, all the address space up to 640 kilobytes was available for RAM. This part of the address space is called "conventional memory" since it accessible to all versions of PC DOS (and MS DOS) automatically on start up. Normally expansion memory is set to be contiguous in the address space with the memory on the motherboard. If there was an unallocated gap between motherboard memory and the expansion memory, the memory would not be automatically detected as usable by PC DOS.

Upper memory area

The upper memory area (UMA) refers to the address space between 640 KB and 1024 KB (0xA0000–0xFFFFF). Three 128 kB regions were defined in this area. The 128 kB between 0XA0000 and 0xBFFFF was reserved for video adapter screen memory. The physical address space between 0xC0000 and 0xDFFFF was reserved for device BIOS ROMs, and special RAM usually shared with physical devices (for example, shared memory for a network adapter). The IBM PC reserved the uppermost 384 KB of the address space from 0xE000 to 0XFFFFF for the BIOS and Cassette BASIC read-only memory (ROM).

For example, the monochrome video adapter memory area runs from 704 to 736 KB (0xB0000–B7FFF). If only a monochrome display adapter was used, the address space between 640 kB and 704 kB could be used for RAM, which would be contiguous with the conventional memory.

The system BIOS ROMs must be at the upper end of the address space because the CPU starting address is fixed by the design of the processor. The starting address is loaded into the program counter of the CPU after a hardware reset and must have a defined value that endures after power is interrupted to the system. On reset or power up, the CPU loads the address from the system ROM and then jumps to a defined ROM location to begin executing the system power-on self test, and eventually load an operating system.

Since an expansion card such as a video adapter, hard drive controller, or network adapter could use allocations of memory in many of the upper memory areas, configuration of some combinations of cards required careful reading of documentation, or experimentation, to find card settings and memory mappings that worked. Mapping two devices to use the same physical memory addresses could result in a stalled or unstable system. Not all addresses in the upper memory area were used in a typical system; unused physical addresses would return undefined and system-dependent data if accessed by the processor.

Expanded memory

As memory prices declined, application programs such as spreadsheets and computer-aided drafting were changed to take advantage of more and more physical memory in the system. Virtual memory in the 8088 and 8086 was not supported by the processor hardware, and disk technology of the time would make it too slow and cumbersome to be practical. Expanded memory was a system that allowed application programs to access more RAM than directly visible to the processor's address space. The process was a form of bank switching. When extra RAM was needed, driver software would temporarily make a piece of expanded memory accessible to the processor; when the data in that piece of memory was updated, another part could be swapped into the processor's address space. For the PC and XT, with only 20 address lines, special-purpose expanded memory cards were made containing perhaps a megabyte, or more, of expanded memory, with logic on the board to make that memory accessible to the processor in defined parts of the 8088 address space.

Allocation and use of expanded memory was not transparent to application programs. The application had to keep track of which bank of expanded memory contained a particular piece of data, and when access to that data was required, the application had to request (through a driver program) the expanded memory board to map that part of memory into the processor's address space. Although applications could use expanded memory with relative freedom, many other software components such as drivers and TSRs were still normally constrained to reside within the 640K "conventional memory" area, which soon became a critically scarce resource.

The 80286 and the high memory area

When the IBM AT was introduced, the segmented memory architecture of the Intel family processors had the byproduct of allowing slightly more than 1 megabyte of memory to be addressed in the "real" mode. Since the 80286 had more than 20 address lines, certain combinations of segment and offset could point into memory above the 0x010000 (2^21) location. To maintain compatibility with the PC and XT behavior, the AT included a logic that made the AT address wrap around to low memory as they would have on an 8088 processor. However, this logic could be turned off on the AT, allowing programs to access an additional 64 kB - 16 bytes of memory in real mode.

80386 and subsequent processors

Intel processors of the 386 version and later allowed a virtual 8086 mode, which simplified the hardware required to implement expanded memory for MS DOS applications. Expanded memory managers such as Quarterdeck's QEMM product and Microsoft's EMM386supported the expanded memory standard without requirement for special memory boards.

On 386 and subsequent processors, memory managers like QEMM might move the bulk of the code for a driver or TSR into extended memory and replace it with a small

fingerhold that was capable of accessing the extended-memory-resident code. They might analyze memory usage to detect drivers that required more RAM during startup than they did subsequently, and recover and reuse the memory that was no longer needed after startup. They might even remap areas of memory normally used for memory-mapped I/O. Many of these tricks involved assumptions about the functioning of drivers and other components. In effect, memory managers might reverse-engineer and modify other vendors' code on the fly. As might be expected, such tricks did not always work. Therefore, memory managers also incorporated very elaborate systems of configurable options, and provisions for recovery should a selected option render the PC unbootable (a frequent occurrence).

Installing and configuring a memory manager might involve hours of experimentation with options, repeatedly rebooting the machine, and testing the results. But conventional memory was so valuable that PC owners felt that such time was well-spent if the result was to free up 30K or 40K of conventional memory space.

Extended memory

In the context of IBM PC compatible computers, *extended memory* refers to memory in the address space of the 80286 and subsequent processors, beyond the 1 megabyte limit imposed by the 20 address lines of the 8088 and 8086. Such memory is not directly available to PC DOS (MS DOS) applications running in the so-called "real mode" of the 80286 and subsequent processors. This memory is only accessible in the protected or virtual modes of 80286 and higher processors.

Chapter 13

Reference Counting

In computer science, **reference counting** is a technique of storing the number of references, pointers, or handles to a resource such as an object, block of memory, disk space or other resource. It may also refer, more specifically, to a garbage collection algorithm that uses these reference counts to deallocate objects which are no longer referenced.

Use in garbage collection

As a garbage collection algorithm, reference counting tracks for each object a count of the number of references to it held by other objects. If an object's reference count reaches zero, the object has become inaccessible, and can be destroyed.

When an object is destroyed, any objects referenced by that object also have their reference counts decreased. Because of this, removing a single reference can potentially lead to a large number of objects being freed. A common modification allows reference counting to be made incremental: instead of destroying an object as soon as its reference count becomes zero, it is added to a list of unreferenced objects, and periodically (or as needed) one or more items from this list are destroyed.

Simple reference counts require frequent updates. Whenever a reference is destroyed or overwritten, the reference count of the object it references is decremented, and whenever one is created or copied, the reference count of the object it references is incremented.

Reference counting is also used in disk operating systems and distributed systems, where full non-incremental tracing garbage collection is too time consuming because of the size of the object graph and slow access speed.

Advantages and disadvantages

The main advantage of reference counting over tracing garbage collection is that objects are reclaimed *as soon as* they can no longer be referenced, and in an incremental fashion, without long pauses for collection cycles and with clearly defined lifetime of every object. In real-time applications or systems with limited memory, this is important to maintain responsiveness. Reference counting is also among the simplest forms of garbage collection to implement. It also allows for effective management of non-memory

resources such as operating system objects, which are often much scarcer than memory (tracing GC systems use finalizers for this, but the delayed reclamation may cause problems). Weighted reference counts are a good solution for garbage collecting a distributed system.

Tracing garbage collection cycles are triggered too often if the set of live objects fills most of the available memory; it requires extra space to be efficient. Reference counting performance does not deteriorate as the total amount of free space decreases.

Reference counts are also useful information to use as input to other runtime optimizations. For example, systems that depend heavily on immutable objects such as many functional programming languages can suffer an efficiency penalty due to frequent copies. However, if we know an object has only one reference (as most do in many systems), and that reference is lost at the same time that a similar new object is created (as in the string append statement `str ← str + "a"`), we can replace the operation with a mutation on the original object.

Reference counting in naive form has two main disadvantages over the tracing garbage collection, both of which require additional mechanisms to ameliorate:

- The frequent updates it involves are a source of inefficiency. While tracing garbage collectors can impact efficiency severely via context switching and cache line faults, they collect relatively infrequently, while accessing objects is done continually. Also, less importantly, reference counting requires every memory-managed object to reserve space for a reference count. In tracing garbage collectors, this information is stored implicitly in the references that refer to that object, saving space, although tracing garbage collectors, particularly incremental ones, can require additional space for other purposes.
- The naive algorithm described above can't handle reference cycles, an object which refers directly or indirectly to itself. A mechanism relying purely on reference counts will never consider cyclic chains of objects for deletion, since their reference count is guaranteed to stay nonzero. Methods for dealing with this issue exist but can also increase the overhead and complexity of reference counting — on the other hand, these methods need only be applied to data that might form cycles, often a small subset of all data. One such method is the use of weak references.

Graph interpretation

When dealing with garbage collection schemes, it's often helpful to think of the **reference graph**, which is a directed graph where the vertices are objects and there is an edge from an object A to an object B if A holds a reference to B. We also have a special vertex or vertices representing the local variables and references held by the runtime system, and no edges ever go to these nodes, although edges can go from them to other nodes.

In this context, the simple reference count of an object is the in-degree of its vertex. Deleting a vertex is like collecting an object. It can only be done when the vertex has no incoming edges, so it does not affect the out-degree of any other vertices, but it can affect the in-degree of other vertices, causing their corresponding objects to be collected as well.

The connected component containing the special vertex contains the objects that can't be collected, while other connected components of the graph only contain garbage. By the nature of reference counting, each of these garbage components must contain at least one cycle.

Dealing with inefficiency of updates

Incrementing and decrementing reference counts every time a reference is created or destroyed can significantly impede performance. Not only do the operations take time, but they damage cache performance and can lead to pipeline bubbles. Even read-only operations like calculating the length of a list require a large number of reads and writes for reference updates with naive reference counting.

One simple technique is for the compiler to combine a number of nearby reference updates into one. This is especially effective for references which are created and quickly destroyed. Care must be taken, however, to put the combined update at the right position so that a premature free is avoided.

The Deutsch-Bobrow method of reference counting capitalizes on the fact that most reference count updates are in fact generated by references stored in local variables. It ignores these references, only counting references in data structures, but before an object with reference count zero can be deleted, the system must verify with a scan of the stack and registers that no other reference to it still exists.

Another technique devised by Henry Baker involves **deferred increments**, in which references which are stored in local variables do not immediately increment the corresponding reference count, but instead defer this until it is necessary. If such a reference is destroyed quickly, then there is no need to update the counter. This eliminates a large number of updates associated with short-lived references. However, if such a reference is copied into a data structure, then the deferred increment must be performed at that time. It is also critical to perform the deferred increment before the object's count drops to zero, resulting in a premature free.

A dramatic decrease in the overhead on counter updates was obtained by Levanoni and Petrank,. They introduce the update coalescing method which coalesces many of the redundant reference count updates. Consider a pointer that in a given interval of the execution is updated several times. It first points to an object *O1*, then to an object *O2*, and so forth until at the end of the interval it points to some object *On*. A reference counting algorithm would typically execute *rc(O1)--*, *rc(O2)++*, *rc(O2)--*, *rc(O3)++*, *rc(O3)--*, ..., *rc(On)++*. But most of these updates are redundant. In order to have the

reference count properly evaluated at the end of the interval it is enough to perform *rc(O1)--* and *rc(On)++*. The rest of the updates are redundant. Levanoni and Petrank show how to use such update coalescing in a reference counting collector. It turns out that when using update coalescing with an appropriate treatment of new objects, more than 99% of the counter updates are eliminated for typical Java benchmarks. In addition, the need for atomic operations during pointer updates on parallel processors is eliminated. Finally, they present an enhanced algorithm that may run concurrently with multithreaded applications employing only fine synchronization.

Blackburn and McKinley's ulterior reference counting combines deferred reference counting with a copying nursery, observing that the majority of pointer mutations occur in young objects. This algorithm achieves throughput comparable with the fastest generational copying collectors with the low bounded pause times of reference counting.

More work on improving performance of reference counting collectors can be found in Paz's Ph.D thesis. In particular, he advocates the use of age oriented collectors and prefetching.

Dealing with reference cycles

There are a variety of ways of handling the problem of detecting and collecting reference cycles. One is that a system may explicitly forbid reference cycles. In some systems like filesystems this is a common solution. Cycles are also sometimes ignored in systems with short lives and a small amount of cyclic garbage, particularly when the system was developed using a methodology of avoiding cyclic data structures wherever possible, typically at the expense of efficiency.

Another solution is to periodically use a tracing garbage collector to reclaim cycles. Since cycles typically constitute a relatively small amount of reclaimed space, the collection cycles can be spaced much farther apart than with an ordinary tracing garbage collector.

Bacon describes a cycle-collection algorithm for reference counting systems with some similarities to tracing systems, including the same theoretical time bounds, but that takes advantage of reference count information to run much more quickly and with less cache damage. It's based on the observation that an object cannot appear in a cycle until its reference count is decremented to a nonzero value. All objects which this occurs to are put on a *roots* list, and then periodically the program searches through the objects reachable from the roots for cycles. It knows it has found a cycle when decrementing all the reference counts on a cycle of references brings them all down to zero. An enhanced version of this algorithm by Paz et al. is able to run concurrently with other operations and improve its efficiency by using the update coalescing method of Levanoni and Petrank.

Variants of reference counting

Although it's possible to augment simple reference counts in a variety of ways, often a better solution can be found by performing reference counting in a fundamentally different way. Here we describe some of the variants on reference counting and their benefits and drawbacks.

Weighted reference counting

In weighted reference counting, we assign each reference a *weight*, and each object tracks not the number of references referring to it, but the total weight of the references referring to it. The initial reference to a newly-created object has a large weight, such as 2^{16}. Whenever this reference is copied, half of the weight goes to the new reference, and half of the weight stays with the old reference. Because the total weight does not change, the object's reference count does not need to be updated.

Destroying a reference decrements the total weight by the weight of that reference. When the total weight becomes equal to the partial weight, all references have been destroyed. If an attempt is made to copy a reference with a weight of 1, we have to "get more weight" by adding to the total weight and then adding this new weight to our reference, and then split it.

The property of not needing to access a reference count when a reference is copied is particularly helpful when the object's reference count is expensive to access, for example because it is in another process, on disk, or even across a network. It can also help increase concurrency by avoiding many threads locking a reference count to increase it. Thus, weighted reference counting is most useful in parallel, multiprocess, database, or distributed applications.

The primary problem with simple weighted reference counting is that destroying a reference still requires accessing the reference count, and if many references are destroyed this can cause the same bottlenecks we seek to avoid. Some adaptations of weighted reference counting seek to avoid this by attempting to give weight back from a dying reference to one which is still active.

Weighted reference counting was independently devised by Bevan, in the paper *Distributed garbage collection using reference counting*, and Watson, in the paper *An efficient garbage collection scheme for parallel computer architectures*, both in 1987.

Indirect reference counting

In indirect reference counting, it is necessary to keep track of whom the reference was obtained from. This means that two references are kept to the object: a direct one which is used for invocations; and an indirect one which forms part of a diffusion tree, such as in the Dijkstra-Scholten algorithm, which allows a garbage collector to identify dead objects. This approach prevents an object from being discarded prematurely.

Examples of use

COM

Microsoft's Component Object Model (COM) makes pervasive use of reference counting. In fact, the three methods that all COM objects must provide (in the IUnknown interface) all increment or decrement the reference count. Much of the Windows Shell and many Windows applications (including MS Internet Explorer, MS Office, and countless third-party products) are built on COM, demonstrating the viability of reference counting in large-scale systems.

One primary motivation for reference counting in COM is to enable interoperability across different programming languages and runtime systems. A client need only know how to invoke object methods in order to manage object life cycle; thus, the client is completely abstracted from whatever memory allocator the implementation of the COM object uses. As a typical example, a Visual Basic program using a COM object is agnostic towards whether that object was allocated (and must later be deallocated) by a C++ allocator or another Visual Basic component.

However, this support for heterogeneity has a major cost: it requires correct reference count management by all parties involved. While high-level languages like Visual Basic manage reference counts automatically, C/C++ programmers are entrusted to increment and decrement reference counts at the appropriate time. C++ programs can and should avoid the task of managing reference counts manually by using smart pointers. Bugs caused by incorrect reference counting in COM systems are notoriously hard to resolve, especially because the error may occur in an opaque, third-party component.

Microsoft has abandoned reference counting in favor of tracing garbage collection for the .NET Framework.

Cocoa

Apple's Cocoa framework (and related frameworks, such as Core Foundation) use manual reference counting, much like COM. However, as of Mac OS X v10.5, Cocoa when used with Objective-C 2.0 also has automatic garbage collection.

Delphi

One language that uses reference counting for garbage collection is Delphi. Delphi is not a completely garbage collected language, in that user-defined types must still be manually allocated and deallocated. It does provide automatic collection, however, for a few built-in types, such as strings, dynamic arrays, and interfaces, for ease of use and to simplify the generic database functionality. It is up to the programmer to decide whether to use the built-in types or not; Delphi programmers have complete access to low-level memory management like in C/C++. So all potential cost of Delphi's reference counting can, if desired, be easily circumvented.

Some of the reasons reference counting may have been preferred to other forms of garbage collection in Delphi include:

- The general benefits of reference counting, such as prompt collection.
- Cycles either cannot occur or do not occur in practice because all of the small set of garbage-collected built-in types are not arbitrarily nestable.
- The overhead in code size required for reference counting is very small (typically a single LOCK INC or LOCK DEC instruction, which ensures atomicity in any environment), and no separate thread of control is needed for collection as would be needed for a tracing garbage collector.
- Many instances of the most commonly used garbage-collected type, the string, have a short lifetime, since they are typically intermediate values in string manipulation.
- The reference count of a string is checked before mutating a string. This allows reference count 1 strings to be mutated directly whilst higher reference count strings are copied before mutation. This allows the general behaviour of old style pascal strings to be preserved whilst eliminating the cost of copying the string on every assignment.
- Because garbage-collection is only done on built-in types, reference counting can be efficiently integrated into the library routines used to manipulate each datatype, keeping the overhead needed for updating of reference counts low. Moreover a lot of the runtime library is in handoptimized assembler.

GObject

The GObject object-oriented programming framework implements reference counting on its base types, including weak references. Reference incrementing and decrementing uses atomic operations for thread safety. A significant amount of the work in writing bindings to GObject from high-level languages lies in adapting GObject reference counting to work with the language's own memory management system.

PHP

PHP uses a reference counting mechanism for its internal variable management. Since PHP 5.3, it implements the algorithm from Bacon's above mentioned paper. PHP allows you to turn on and off the cycle collection with user-level functions. It also allows you to manually force the purging mechanism to be run.

Python

Python also uses reference counting and offers cycle detection as well.

Squirrel

Squirrel also uses reference counting and offers cycle detection as well. This tiny language is relatively unknown outside the video game industry; however, it is a concrete

example of how reference counting can be practical and efficient (especially in realtime environments).

Tcl

Tcl 8 uses reference counting for memory management of values (Tcl Obj structs). Since Tcl's values are immutable, reference cycles are impossible to form and no cycle detection scheme is needed. Operations that would replace a value with a modified copy are generally optimized to instead modify the original when its reference count indicates it to be unshared. The references are counted at a data structure level, so the problems with very frequent updates discussed above do not arise.

Disk operating systems

Many disk operating systems maintain a count of the number of references to any particular block or file. When the count falls to zero, the file can be safely deallocated. In addition, while references can still be made from directories, some Unixes allow that the referencing can be solely made by live processes, and there can be files that do not exist in the file system hierarchy.

Chapter 14

Garbage Collection (Computer Science)

In computer science, **garbage collection** (**GC**) is a form of automatic memory management. It is a special case of *resource management*, in which the limited resource being managed is memory. The *garbage collector*, or just *collector*, attempts to reclaim *garbage*, or memory occupied by objects that are no longer in use by the program. Garbage collection was invented by John McCarthy around 1959 to solve problems in Lisp.

Garbage collection is often portrayed as the opposite of manual memory management, which requires the programmer to specify which objects to deallocate and return to the memory system. However, many systems use a combination of the two approaches, and other techniques such as stack allocation and region inference can carve off parts of the problem. There is an ambiguity of terms, as theory often uses the terms *manual garbage collection* and *automatic garbage collection* rather than *manual memory management* and *garbage collection*, and does not restrict garbage collection to memory management, rather considering that any logical or physical resource may be garbage collected.

Garbage collection does not traditionally manage limited resources other than memory that typical programs use, such as network sockets, database handles, user interaction windows, and file and device descriptors. Methods used to manage such resources, particularly destructors, may suffice as well to manage memory, leaving no need for GC. Some GC systems allow such other resources to be associated with a region of memory that, when collected, causes the other resource to be reclaimed; this is called *finalization*. Finalization may introduce complications limiting its usability, such as intolerable latency between disuse and reclaim of especially limited resources, or a lack of control over which thread performs the work of reclaiming.

Principles

The basic principles of garbage collection are:

1. Find data objects in a program that cannot be accessed in the future
2. Reclaim the resources used by those objects

By making manual memory deallocation unnecessary (and often forbidding it), garbage collection frees the programmer from much of the worry about releasing objects that are

no longer needed, which can otherwise consume substantial design effort. It also prevents some kinds of runtime errors.

Many computer languages require garbage collection, either as part of the language specification (e.g., Java, C#, and most scripting languages) or effectively for practical implementation (e.g., formal languages like lambda calculus); these are said to be **garbage collected languages**. Other languages were designed for use with manual memory management, but have garbage collected implementations available (e.g., C, C++). Some languages, like Ada, Modula-3, and C++/CLI allow both garbage collection and manual memory management to co-exist in the same application by using separate heaps for collected and manually managed objects; others, like D, are garbage collected but allow the user to manually delete objects and also entirely disable garbage collection when speed is required. While integrating garbage collection into the language's compiler and runtime system enables a much wider choice of methods, *post hoc* GC systems exist, including some that do not require recompilation. (*Post-hoc* GC is sometimes distinguished as *litter collection*.) The garbage collector will almost always be closely integrated with the memory allocator.

Benefits

Garbage collection frees the programmer from manually dealing with memory deallocation. As a result, certain categories of bugs are eliminated or substantially reduced:

- **Dangling pointer bugs**, which occur when a piece of memory is freed while there are still pointers to it, and one of those pointers is then used. By then the memory may have been re-assigned to another use, with unpredictable results.
- **Double free bugs**, which occur when the program tries to free a region of memory that has already been freed, and perhaps already been allocated again.
- Certain kinds of **memory leaks**, in which a program fails to free memory occupied by objects that will not be used again, leading, over time, to memory exhaustion.

Some of these bugs can have security implications.

Disadvantages

Typically, garbage collection has certain disadvantages:

- Garbage collection consumes computing resources in deciding which memory to free, reconstructing facts that may have been known to the programmer. The penalty for the convenience of not annotating object lifetime manually in the source code is overhead, often leading to decreased or uneven performance. Interaction with memory hierarchy effects can make this overhead intolerable in circumstances that are hard to predict or to detect in routine testing.

- The moment when the garbage is actually collected can be unpredictable, resulting in stalls scattered throughout a session. Unpredictable stalls can be unacceptable in real-time environments such as device drivers, in transaction processing, or in interactive programs. This unpredictability also means that techniques such as RAII, which allows for automatic cleaning up of resources (e.g. closing file handles, deleting temporary files, etc.), cannot be used, since the cleanup might happen too late.
- Memory may leak despite the presence of a garbage collector, if references to unused objects are not themselves manually disposed of. This is described as a *logical* memory leak. For example, recursive algorithms normally delay release of stack objects until after the final call has completed. Caching and memoizing, common optimization techniques, commonly lead to such logical leaks. The belief that garbage collection eliminates all leaks leads many programmers not to guard against creating such leaks.
- In virtual memory environments typical of modern desktop computers, it can be difficult for the garbage collector to notice when collection is needed, resulting in large amounts of accumulated garbage, a long, disruptive collection phase, and other programs' data being swapped out.
- Perhaps the most significant problem is that programs that rely on garbage collectors often exhibit poor locality (interacting badly with cache and virtual memory systems), occupy more address space than the program actually uses at any one time, and touch otherwise idle pages. These may combine in a phenomenon called thrashing, in which a program spends more time copying data between various grades of storage than performing useful work. They may make it impossible for a programmer to reason about the performance effects of design choices, making performance tuning difficult. They can lead garbage-collecting programs to interfere with other programs competing for resources.

Tracing garbage collectors

Tracing garbage collectors are the most common type of garbage collector. They first determine which objects are *reachable* (or potentially reachable), and then discard all remaining objects.

Reachability of an object

Informally, a reachable object can be defined as an object for which there exists some variable in the program environment that leads to it, either directly or through references from other reachable objects. More precisely, objects can be reachable in only two ways:

1. A distinguished set of objects are assumed to be reachable: these are known as the *roots*. Typically, these include all the objects referenced from anywhere in the call stack (that is, all local variables and parameters in the functions currently being invoked), and any global variables.
2. Anything referenced from a reachable object is itself reachable; more formally, reachability is a transitive closure.

The reachability definition of "garbage" is not optimal, insofar as the last time a program uses an object could be long before that object falls out of the environment scope. A distinction is sometimes drawn between *syntactic garbage*, those objects the program cannot possibly reach, and *semantic garbage*, those objects the program will in fact never again use. For example:

```
Object x = new Foo();
Object y = new Bar();
x = new Quux();
/* at this point, we know that the Foo object will
 * never be accessed: it is syntactic garbage
 */

if(x.check_something()) {
 x.do_something(y);
}
System.exit(0);
/* in the above block, y *could* be semantic garbage,
 * but we won't know until x.check_something() returns
 * some value: if it returns at all
 */
```

The problem of precisely identifying semantic garbage can easily be shown to be partially decidable: a program that allocates an object *X*, runs an arbitrary input program *P*, and uses *X* if and only if *P* finishes would require a semantic garbage collector to solve the halting problem. Although conservative heuristic methods for semantic garbage detection remain an active research area, essentially all practical garbage collectors focus on syntactic garbage.

Another complication with this approach is that, in languages with both reference types and unboxed value types, the garbage collector needs to somehow be able to distinguish which variables on the stack or fields in an object are regular values and which are references: in memory, an integer and a reference might look alike. The garbage collector then needs to know whether to treat the element as a reference and follow it, or whether it is a primitive value. One common solution is the use of tagged pointers.

Strong and weak references

The garbage collector can reclaim only objects that have no references. An object that is reachable cannot be garbage collected by the garbage collector. Such a reference is known as a strong reference. An object can also be referred to as a weak reference, also known as the target. An object is eligible for garbage collection if it does not contain any strong references, irrespective of the number of weak references it contains. There are two types of weak references; a long weak reference tracks resurrection while a short weak reference does not. The primary advantage of maintaining weak references to an object is that it allows the garbage collector to collect or reclaim memory of the object if it runs out of memory in the managed heap.

Basic algorithm

Tracing collectors are so called because they trace through the working set of memory. These garbage collectors perform collection in cycles. A cycle is started when the collector decides (or is notified) that it needs to reclaim memory, which happens most often when the system is low on memory. The original method involves a naïve **mark-and-sweep** in which the entire memory set is touched several times.

Naïve mark-and-sweep

In the naïve mark-and-sweep method, each object in memory has a flag (typically a single bit) reserved for garbage collection use only. This flag is always *cleared*, except during the collection cycle. The first stage of collection sweeps the entire 'root set', marking each accessible object as being 'in-use'. All objects transitively accessible from the root set are marked, as well. Finally, each object in memory is again examined; those with the in-use flag still cleared are not reachable by any program or data, and their memory is freed. (For objects which are marked in-use, the in-use flag is cleared again, preparing for the next cycle.)

This method has several disadvantages, the most notable being that the entire system must be suspended during collection; no mutation of the working set can be allowed. This will cause programs to 'freeze' periodically (and generally unpredictably), making real-time and time-critical applications impossible. In addition, the entire working memory must be examined, much of it twice, potentially causing problems in paged memory systems.

Tri-colour marking

Because of these pitfalls, most modern tracing garbage collectors implement some variant of the *tri-colour marking* abstraction, but simple collectors (such as the *mark-and-sweep* collector) often do not make this abstraction explicit. Tri-colour marking works as follows:

1. Create initial white, grey, and black sets; these sets will be used to maintain progress during the cycle.
    - Initially the white set or *condemned set* is the set of objects that are candidates for having their memory recycled.
    - The black set is the set of objects that cheaply can be proven to have no references to objects in the white set; in many implementations the black set starts off empty.
    - The grey set is all the objects that are reachable from root references but the objects referenced by grey objects haven't been scanned yet. Grey objects are known to be reachable from the root, so cannot be garbage collected: grey objects will eventually end up in the black set. The grey state means we still need to check any objects that the object references.

    - The grey set is initialised to objects which are referenced directly at root level; typically all other objects are initially placed in the white set.
    - Objects can move from white to grey to black, never in the other direction.
2. Pick an object from the grey set. *Blacken* this object (move it to the black set), by *greying* all the white objects it references directly. This confirms that this object cannot be garbage collected, and also that any objects it references cannot be garbage collected.
3. Repeat the previous step until the grey set is empty.
4. When there are no more objects in the grey set, then all the objects remaining in the white set have been demonstrated not to be reachable, and the storage occupied by them can be reclaimed.

The 3 sets partition memory; every object in the system, including the root set, is in precisely one set.

The tri-colour marking algorithm preserves an important invariant:

> No black object points directly to a white object.

This ensures that the white objects can be safely destroyed once the grey set is empty. (Some variations on the algorithm do not preserve the tricolour invariant but they use a modified form for which all the important properties hold.)

The tri-colour method has an important advantage: it can be performed 'on-the-fly', without halting the system for significant time periods. This is accomplished by marking objects as they are allocated and during mutation, maintaining the various sets. By monitoring the size of the sets, the system can perform garbage collection periodically, rather than as-needed. Also, the need to touch the entire working set each cycle is avoided.

Implementation strategies

In order to implement the basic tri-colour algorithm, several important design decisions must be made, which can significantly affect the performance characteristics of the garbage collector.

Moving vs. non-moving

Once the unreachable set has been determined, the garbage collector may simply release the unreachable objects and leave everything else as it is, or it may copy some or all of the reachable objects into a new area of memory, updating all references to those objects as needed. These are called "non-moving" and "moving" garbage collectors, respectively.

At first, a moving GC strategy may seem inefficient and costly compared to the non-moving approach, since much more work would appear to be required on each cycle. In

fact, however, the moving GC strategy leads to several performance advantages, both during the garbage collection cycle itself and during actual program execution:

- No additional work is required to reclaim the space freed by dead objects; the entire region of memory from which reachable objects were moved can be considered free space. In contrast, a non-moving GC must visit each unreachable object and somehow record that the memory it alone occupied is available.
- Similarly, new objects can be allocated very quickly. Since large contiguous regions of memory are usually made available by the moving GC strategy, new objects can be allocated by simply incrementing a 'free memory' pointer. A non-moving strategy may, after some time, lead to a heavily fragmented heap, requiring expensive consultation of "free lists" of small available blocks of memory in order to allocate new objects.
- If an appropriate traversal order is used (such as cdr-first for list conses), objects that refer to each other frequently can be moved very close to each other in memory, increasing the likelihood that they will be located in the same cache line or virtual memory page. This can significantly speed up access to these objects through these references.

One disadvantage of a moving garbage collector is that it only allows access through references that are managed by the garbage collected environment, and does not allow pointer arithmetic. This is because any native pointers to objects will be invalidated when the garbage collector moves the object (they become dangling pointers). For interoperability with native code, the garbage collector must copy the object contents to a location outside of the garbage collected region of memory. An alternative approach is to **pin** the object in memory, preventing the garbage collector from moving it and allowing the memory to be directly shared with native pointers (and possibly allowing pointer arithmetic).

Copying vs. mark-and-sweep vs. mark-and-don't-sweep

To further refine the distinction, tracing collectors can also be divided by considering how the three sets of objects (white, grey, and black) are maintained during a collection cycle.

The most straightforward approach is the **semi-space collector**, which dates to 1969. In this moving GC scheme, memory is partitioned into a "from space" and "to space". Initially, objects are allocated into "to space" until they become full and a collection is triggered. At the start of a collection, the "to space" becomes the "from space", and vice versa. The objects reachable from the root set are copied from the "from space" to the "to space". These objects are scanned in turn, and all objects that they point to are copied into "to space", until all reachable objects have been copied into "to space". Once the program continues execution, new objects are once again allocated in the "to space" until it is once again full and the process is repeated. This approach has the advantage of conceptual simplicity (the three object color sets are implicitly constructed during the copying process), but the disadvantage that a (possibly) very large contiguous region of free

memory is necessarily required on every collection cycle. This technique is also known as **stop-and-copy**. Cheney's algorithm is an improvement on the semi-space collector.

A **mark and sweep** garbage collector maintains a bit (or two) with each object to record whether it is white or black; the grey set is either maintained as a separate list (such as the process stack) or using another bit. As the reference tree is traversed during a collection cycle (the "mark" phase), these bits are manipulated by the collector to reflect the current state. A final "sweep" of the memory areas then frees white objects. The mark and sweep strategy has the advantage that, once the unreachable set is determined, either a moving or non-moving collection strategy can be pursued; this choice of strategy can even be made at runtime, as available memory permits. It has the disadvantage of "bloating" objects by a small amount.

A **mark and don't sweep** garbage collector, like the mark-and-sweep, maintains a bit with each object to record whether it is white or black; the gray set is either maintained as a separate list (such as the process stack) or using another bit. There are two key differences here. First, black and white mean different things than they do in the mark and sweep collector. In a "mark and don't sweep" system, all reachable objects are always black. An object is marked black at the time it is allocated, and it will stay black even if it becomes unreachable. A white object is unused memory and may be allocated. Second, the interpretation of the black/white bit can change. Initially, the black/white bit may have the sense of (0=white, 1=black). If an allocation operation ever fails to find any available (white) memory, that means all objects are marked used (black). The sense of the black/white bit is then inverted (for example, 0=black, 1=white). Everything becomes white. This momentarily breaks the invariant that reachable objects are black, but a full marking phase follows immediately, to mark them black again. Once this is done, all unreachable memory is white. No "sweep" phase is necessary.

Generational GC (ephemeral GC)

It has been empirically observed that in many programs, the most recently created objects are also those most likely to become unreachable quickly (known as *infant mortality* or the *generational hypothesis*). A generational GC (also known as ephemeral GC) divides objects into generations and, on most cycles, will place only the objects of a subset of generations into the initial white (condemned) set. Furthermore, the runtime system maintains knowledge of when references cross generations by observing the creation and overwriting of references. When the garbage collector runs, it may be able to use this knowledge to prove that some objects in the initial white set are unreachable without having to traverse the entire reference tree. If the generational hypothesis holds, this results in much faster collection cycles while still reclaiming most unreachable objects.

In order to implement this concept, many generational garbage collectors use separate memory regions for different ages of objects. When a region becomes full, those few objects that are referenced from older memory regions are promoted (copied) up to the next highest region, and the entire region can then be overwritten with fresh objects. This

technique permits very fast incremental garbage collection, since the garbage collection of only one region at a time is all that is typically required.

Generational garbage collection is a heuristic approach, and some unreachable objects may not be reclaimed on each cycle. It may therefore occasionally be necessary to perform a full mark and sweep or copying garbage collection to reclaim all available space. In fact, runtime systems for modern programming languages (such as Java and the .NET Framework) usually use some hybrid of the various strategies that have been described thus far; for example, most collection cycles might look only at a few generations, while occasionally a mark-and-sweep is performed, and even more rarely a full copying is performed to combat fragmentation. The terms "minor cycle" and "major cycle" are sometimes used to describe these different levels of collector aggression.

Stop-the-world vs. incremental vs. concurrent

Simple *stop-the-world* garbage collectors completely halt execution of the program to run a collection cycle, thus guaranteeing that new objects are not allocated and objects do not suddenly become unreachable while the collector is running.

This has the obvious disadvantage that the program can perform no useful work while a collection cycle is running (sometimes called the "embarrassing pause"). Stop-the-world garbage collection is therefore mainly suitable for non-interactive programs. Its advantage is that it is both simpler to implement and faster than incremental garbage collection.

Incremental and *concurrent* garbage collectors are designed to reduce this disruption by interleaving their work with activity from the main program. Incremental garbage collectors perform the garbage collection cycle in discrete phases, with program execution permitted between each phase (and sometimes during some phases). Concurrent garbage collectors do not stop program execution at all, except perhaps briefly when the program's execution stack is scanned. However, the sum of the incremental phases takes longer to complete than one batch garbage collection pass, so these garbage collectors may yield lower total throughput.

Careful design is necessary with these techniques to ensure that the main program does not interfere with the garbage collector and vice versa; for example, when the program needs to allocate a new object, the runtime system may either need to suspend it until the collection cycle is complete, or somehow notify the garbage collector that there exists a new, reachable object.

Precise vs. conservative and internal pointers

Some collectors running in a particular environment can correctly identify all pointers (references) in an object; these are called "precise" (also "exact" or "accurate") collectors, the opposite being a "conservative" or "partly conservative" collector. Conservative collectors have to assume that any bit pattern in memory could be a pointer if (when

interpreted as a pointer) it would point into any allocated object. Thus, conservative collectors may have some false negatives, where storage is not released because of accidental fake pointers, but this is rarely a significant drawback in practice, unless the program has to handle data that is or seems to be random and could easily be a pointer. Naturally 64-bit systems suffer less from this problem than 32-bit systems, since the possibility for a 64-bit pattern to form a valid pointer is smaller than for a 32-bit pattern as the domain of a 64-bit value is much greater than a 32-bit value. Thus it is less probable for a "random" 64-bit value to point to a region on system's memory. Whether a precise collector is practical usually depends on the type safety properties of the programming language in question. An example for which a conservative garbage collector would be needed is the C language, which allows typed (non-void) pointers to be type cast into untyped (void) pointers, and vice versa.

A related issue concerns *internal pointers*, or pointers to fields within an object. If the semantics of a language allow internal pointers, then there may be many different addresses that can refer to the same object, which complicates determining whether an object is garbage or not. An example for this is the C++ language, in which multiple inheritance can cause pointers to base objects to have different addresses. Even in languages like Java, however, internal pointers can exist during the computation, say, of an array element address, and in a tightly-optimized program the corresponding pointer to the object itself may have been overwritten in its register, so such internal pointers need to be scanned.

Performance implications

Tracing garbage collectors require some implicit runtime overhead that may be beyond the control of the programmer, and can sometimes lead to performance problems. For example, commonly used stop-the-world garbage collectors, which pause program execution at arbitrary times, may make garbage collection inappropriate for some embedded systems, high-performance server software, and applications with real-time needs.

Manual heap allocation

- search for best/first-fit block of sufficient size
- free list maintenance

Garbage collection

- locate reachable objects
- copy reachable objects for moving collectors
- read/write barriers for incremental collectors
- search for best/first-fit block and free list maintenance for non-moving collectors

It is difficult to compare the two cases directly, as their behavior depends on the situation. For example, in the best case for a garbage collecting system, allocation just increments a

pointer, but in the best case for manual heap allocation, the allocator maintains freelists of specific sizes and allocation only requires following a pointer. However, this size segregation usually cause a large degree of external fragmentation, which can have an adverse impact on cache behaviour. Memory allocation in a garbage collected language may be implemented using heap allocation behind the scenes (rather than simply incrementing a pointer), so the performance advantages listed above don't necessarily apply in this case. In some situations, most notably embedded systems, it is possible to avoid both garbage collection and heap management overhead by preallocating pools of memory and using a custom, lightweight scheme for allocation/deallocation.

The overhead of write barriers is more likely to be noticeable in an imperative-style program which frequently writes pointers into existing data structures than in a functional-style program which constructs data only once and never changes them.

Some advances in garbage collection can be understood as reactions to performance issues. Early collectors were stop-the-world collectors, but the performance of this approach was distracting in interactive applications. Incremental collection avoided this disruption, but at the cost of decreased efficiency due to the need for barriers. Generational collection techniques are used with both stop-the-world and incremental collectors to increase performance; the trade-off is that some garbage is not detected as such for longer than normal.

Determinism

Tracing garbage collection is not deterministic. An object which becomes eligible for garbage collection will usually be cleaned up eventually, but there is no guarantee when (or even if) that will happen.

This can cause problems:

- Most environments with tracing GC require manual deallocation of limited non-memory resources, as an automatic deallocation during the garbage collection phase (usually using a finalizer) may run too late or in the wrong circumstances.
- The performance impact caused by GC is seemingly random and hard to predict.

Reference counting

Reference counting is a form of automatic memory management where each object has a count of the number of references to it. An object's reference count is incremented when a reference to it is created, and decremented when a reference is destroyed. The object's memory is reclaimed when the count reaches zero.

There are two major disadvantages to reference counting:

- If two or more objects refer to each other, they can create a cycle whereby neither will be collected as their mutual references never let their reference counts

become zero. Some garbage collection systems using reference counting (like the one in CPython) use specific cycle-detecting algorithms to deal with this issue.
- In naive implementations, each assignment of a reference and each reference falling out of scope often require modifications of one or more reference counters. However, optimizations to this are described in the literature. When used in a multithreaded environment, these modifications (increment and decrement) may need to be interlocked. This may be an expensive operation for processors without atomic operations such as compare-and-swap.

One important advantage of reference counting is that it provides deterministic garbage collection (as opposed to tracing GC).

Escape analysis

Escape analysis can be used to convert heap allocations to stack allocations, thus reducing the amount of work needed to be done by the garbage collector.

Availability

Generally speaking, higher-level programming languages are more likely to have garbage collection as a standard feature. In languages that do not have built in garbage collection, it can often be added through a library, as with the Boehm garbage collector for C and C++. This approach is not without drawbacks, such as changing object creation and destruction mechanisms.

Most functional programming languages, such as ML, Haskell, and APL, have garbage collection built in. Lisp, which introduced functional programming, is especially notable for introducing this mechanism.

Other dynamic languages, such as Ruby (but not Perl 5, or PHP, which use reference counting), also tend to use GC. Object-oriented programming languages such as Smalltalk, Java and ECMAScript usually provide integrated garbage collection. Notable exceptions are C++ and Delphi which have destructors. Objective-C has not traditionally had it, but ObjC 2.0 as implemented by Apple for Mac OS X uses a runtime collector developed in-house, while the GNUstep project uses a Boehm collector.

Historically, languages intended for beginners, such as BASIC and Logo, have often used garbage collection for variable-length data types, such as strings and lists, so as not to burden programmers with manual memory management. On early microcomputers, with their limited memory and slow processors, BASIC garbage collection could often cause apparently random, inexplicable pauses in the midst of program operation.

Limited environments

Garbage collection is rarely used on embedded or real-time systems because of the perceived need for very tight control over the use of limited resources. However, garbage

collectors compatible with such limited environments have been developed. The Microsoft .NET Micro Framework and Java Platform, Micro Edition are embedded software platforms that, like their larger cousins, include garbage collection.

Chapter 15

Paging

In computer operating systems, **paging** is one of the memory-management schemes by which a computer can store and retrieve data from secondary storage for use in main memory. In the paging memory-management scheme, the operating system retrieves data from secondary storage in same-size blocks called *pages*. The main advantage of paging is that it allows the physical address space of a process to be noncontiguous. Before the time paging was used, systems had to fit whole programs into storage contiguously, which caused various storage and fragmentation problems.

Paging is an important part of virtual memory implementation in most contemporary general-purpose operating systems, allowing them to use disk storage for data that does not fit into physical random-access memory (RAM).

Overview

The main functions of paging are performed when a program tries to access pages that are not currently mapped to physical memory (RAM). This situation is known as a page fault. The operating system must then take control and handle the page fault, in a manner invisible to the program. Therefore, the operating system must:

1. Determine the location of the data in auxiliary storage.
2. Obtain an empty page frame in RAM to use as a container for the data.
3. Load the requested data into the available page frame.
4. Update the page table to show the new data.
5. Return control to the program, transparently retrying the instruction that caused the page fault.

Because RAM is faster than auxiliary storage, paging is avoided until there is not enough RAM to store all the data needed. When this occurs, a page in RAM is moved to auxiliary storage, freeing up space in RAM for use. Thereafter, whenever the page in secondary storage is needed, a page in RAM is saved to auxiliary storage so that the requested page can then be loaded into the space left behind by the old page. Efficient paging systems must determine the page to swap by choosing one that is least likely to be needed within a short time. There are various page replacement algorithms that try to do this.

Most operating systems use some approximation of the least recently used (LRU) page replacement algorithm (the LRU itself cannot be implemented on the current hardware) or working set based algorithm.

If a page in RAM is modified (i.e. if the page becomes *dirty*) and then chosen to be swapped, it must either be written to auxiliary storage, or simply discarded.

To further increase responsiveness, paging systems may employ various strategies to predict what pages will be needed soon so that it can preemptively load them.

Demand paging

When demand paging is used, no preemptive loading takes place. Paging only occurs at the time of the data request, and not before. In particular, when a demand pager is used, a program usually begins execution with none of its pages pre-loaded in RAM. Pages are copied from the executable file into RAM the first time the executing code references them, usually in response to page faults. As such, much of the executable file might never be loaded into memory if pages of the program are never executed during that run.

Anticipatory paging

This technique preloads a process's non-resident pages that are likely to be referenced in the near future (taking advantage of locality of reference). Such strategies attempt to reduce the number of page faults a process experiences.

Free page queue

The free page queue is a list of page frames that are available for assignment after a page fault. Some operating systems support page reclamation; if a page fault occurs for a page that had been stolen and the page frame was never reassigned, then the operating system avoids the necessity of reading the page back in by assigning the unmodified page frame.

Page stealing

Some operating systems periodically look for pages that have not been recently referenced and add them to the Free page queue, after paging them out if they have been modified.

Swap prefetch

A few operating systems use anticipatory paging, also called swap prefetch. These operating systems periodically attempt to guess which pages will soon be needed, and start loading them into RAM. There are various heuristics in use, such as *"if a program references one virtual address which causes a page fault, perhaps the next few pages' worth of virtual address space will soon be used"* and *"if one big program just finished*

execution, leaving lots of free RAM, perhaps the user will return to using some of the programs that were recently paged out".

Pre-cleaning

Unix operating systems periodically use sync to pre-clean all dirty pages, that is, to save all modified pages to hard disk. Windows operating systems do the same thing via "modified page writer" threads.

Pre-cleaning makes starting a new program or opening a new data file much faster. The hard drive can immediately seek to that file and consecutively read the whole file into pre-cleaned page frames. Without pre-cleaning, the hard drive is forced to seek back and forth between writing a dirty page frame to disk, and then reading the next page of the file into that frame.

Thrashing

Most programs reach a steady state in their demand for memory locality both in terms of instructions fetched and data being accessed. This steady state is usually much less than the total memory required by the program. This steady state is sometimes referred to as the working set: the set of memory pages that are most frequently accessed.

Virtual memory systems work most efficiently when the ratio of the working set to the total number of pages that can be stored in RAM is low enough that the time spent resolving page faults is not a dominant factor in the workload's performance. A program that works with huge data structures will sometimes require a working set that is too large to be efficiently managed by the page system resulting in constant page faults that drastically slow down the system. This condition is referred to as thrashing: pages are swapped out and then accessed causing frequent faults.

An interesting characteristic of thrashing is that as the working set grows, there is very little increase in the number of faults until the critical point (when faults go up dramatically and majority of system's processing power is spent on handling them).

An extreme example of this sort of situation occurred on the IBM System/360 Model 67 and IBM System/370 series mainframe computers, in which a particular instruction could consist of an execute instruction, which crosses a page boundary, that the instruction points to a move instruction, that itself also crosses a page boundary, targeting a move of data from a source that crosses a page boundary, to a target of data that also crosses a page boundary. The total number of pages thus being used by this particular instruction is eight, and all eight pages must be present in memory at the same time. If the operating system will allocate less than eight pages of actual memory in this example, when it attempts to swap out some part of the instruction or data to bring in the remainder, the instruction will again page fault, and it will thrash on every attempt to restart the failing instruction.

To decrease excessive paging, and thus possibly resolve thrashing problem, a user can do any of the following:

- Increase the amount of RAM in the computer (generally the best long-term solution).
- Decrease the number of programs being concurrently run on the computer.

The term *thrashing* is also used in contexts other than virtual memory systems, for example to describe cache issues in computing or silly window syndrome in networking.

Terminology

Historically, *paging* sometimes referred to a memory allocation scheme that used fixed-length pages as opposed to variable-length segments, without implicit suggestion that virtual memory technique were employed at all or that those pages were transferred to disk. Such usage is rare today.

Some modern systems use the term *swapping* along with *paging*. Historically, *swapping* referred to moving from/to secondary storage a whole program at a time, in a scheme known as roll-in/roll-out. In the 1960s, after the concept of virtual memory was introduced—in two variants, either using segments or pages—the term *swapping* was applied to moving, respectively, either segments or pages, between disk and memory. Today with the virtual memory mostly based on pages, not segments, *swapping* became a fairly close synonym of *paging*, although with one difference.

In many popular systems, there is a concept known as page cache, of using the same single mechanism for *both* virtual memory and disk caching. A page may be then transferred to or from *any* ordinary disk file, not necessarily a dedicated space. *Page in* is transferring a page from the disk to RAM. *Page out* is transferring a page from RAM to the disk. *Swap in* and *out* only refer to transferring pages between RAM and dedicated *swap space* or *swap file*, and not any other place on disk.

On Windows NT based systems, dedicated swap space is known as a page file and paging/swapping are often used interchangeably.

Implementations

Ferranti Atlas

The first computer to support paging was the Atlas, jointly developed by Ferranti, the University of Manchester and Plessey. The machine had an associative (content-addressable) memory with one entry for each 512 word page. The Supervisor handled non-equivalence interruptions and managed the transfer of pages between core and drum in order to provide a one-level store to programs.

Windows 3.x and Windows 9x

Virtual memory has been a feature of Microsoft Windows since Windows 3.0 in 1990. Microsoft introduced virtual memory in response to the failures of Windows 1.0 and Windows 2.0, attempting to slash resource requirements for the operating system.

Confusion abounds about Microsoft's decision to refer to the swap file as "virtual memory". Novices unfamiliar with the concept accept this definition without question, and speak of adjusting Windows' virtual memory size. In fact *every* process has a fixed, unchangeable virtual memory size, usually 2 GB. The user has only an option to change disk capacity dedicated to paging.

Windows 3.x creates a hidden file named `386SPART.PAR` or `WIN386.SWP` for use as a swap file. It is generally found in the root directory, but it may appear elsewhere (typically in the WINDOWS directory). Its size depends on how much swap space the system has (a setting selected by the user under Control Panel → Enhanced under "Virtual Memory".) If the user moves or deletes this file, a blue screen will appear the next time Windows is started, with the error message "The permanent swap file is corrupt". The user will be prompted to choose whether or not to delete the file (whether or not it exists).

Windows 95, Windows 98 and Windows Me use a similar file, and the settings for it are located under Control Panel → System → Performance tab → Virtual Memory. Windows automatically sets the size of the page file to start at 1.5× the size of physical memory, and expand up to 3× physical memory if necessary. If a user runs memory-intensive applications on a system with low physical memory, it is preferable to manually set these sizes to a value higher than default.

Windows NT

In NT-based versions of Windows (such as Windows XP, Windows Vista, and Windows 7), the file used for paging is named `pagefile.sys`. The default location of the page file is in the root directory of the partition where Windows is installed. Windows can be configured to use free space on any available drives for pagefiles. It is required, however, for the boot partition (i.e. the drive containing the Windows directory) to have a pagefile on it if the system is configured to write either kernel or full memory dumps after a crash. Windows uses the paging file as temporary storage for the memory dump. When the system is rebooted, Windows copies the memory dump from the pagefile to a separate file and frees the space that was used in the pagefile.

Fragmentation

In Windows's default configuration the pagefile is allowed to expand beyond its initial allocation when necessary. If this happens gradually, it can become heavily fragmented which can potentially cause performance problems. The common advice given to avoid this is to set a single "locked" pagefile size so that Windows will not expand it. However,

the pagefile only expands when it has been filled, which, in its default configuration, is 150% the total amount of physical memory. Thus the total demand for pagefile-backed virtual memory must exceed 250% of the computer's physical memory before the pagefile will expand.

The fragmentation of the pagefile that occurs when it expands is temporary. As soon as the expanded regions are no longer in use (at the next reboot, if not sooner) the additional disk space allocations are freed and the pagefile is back to its original state.

Locking a page file's size can be problematic in the case that a Windows application requests more memory than the total size of physical memory and the page file. In this case, requests to allocate memory fail, which may cause applications and system processes to fail. Supporters of this view will note that the page file is rarely read or written in sequential order, so the performance advantage of having a completely sequential page file is minimal. However, it is generally agreed that a large page file will allow use of memory-heavy applications, and there is no penalty except that more disk space is used.

Defragmenting the page file is also occasionally recommended to improve performance when a Windows system is chronically using much more memory than its total physical memory. This view ignores the fact that, aside from the temporary results of expansion, the pagefile does not become fragmented over time. In general, performance concerns related to pagefile access are much more effectively dealt with by adding more physical memory.

Unix and Unix-like systems

Unix systems, and other Unix-like operating systems, use the term "swap" to describe both the act of moving memory pages between RAM and disk, and the region of a disk the pages are stored on. In some of those systems, it is common to use a separate whole partition of a hard disk for swapping. These partitions are called *swap partitions*. Some of those systems only support swapping to a swap partition; others also support swapping to files.

Linux

From a software point of view with the 2.6 Linux kernel, swap files are just as fast as swap partitions. The kernel keeps a map of where the swap file exists, and accesses the disk directly, bypassing caching and filesystem overhead. Red Hat recommends using a swap partition. With a swap partition one can choose where on the disk it resides and place it where the disk throughput is highest. The administrative flexibility of swap files can outweigh the other advantages of swap partitions.

Linux supports using a virtually unlimited number of swapping devices, each of which can be assigned a priority. When the operating system needs to swap pages out of physical memory, it uses the highest-priority device with free space. If multiple devices

are assigned the same priority, they are used in a fashion similar to level 0 RAID arrangements. This provides improved performance as long as the devices can be accessed efficiently in parallel. Therefore, care should be taken assigning the priorities. For example, swaps located on the same physical disk should not be used in parallel, but in order ranging from the fastest to the slowest (i.e.: the fastest having the highest priority).

Mac OS X

Mac OS X supports both swap partitions and the use of swap files, but the default and recommended configuration is to use multiple swap files.

Solaris

Solaris allows swapping to raw disk slices as well as files. The traditional method is to use slice 1 (ie. the second slice) on the OS disk to house swap. Swap setup is managed by the system boot process if there are entries in the "vfstab" file, but can also be managed manually through the use of the "swap" command. While it is possible to remove, at runtime, all swap from a lightly loaded system, Sun does not recommend it. Recent additions to the ZFS file system allow creation of ZFS devices that can be used as swap partitions. Swapping to normal files on ZFS file systems is not supported.

AmigaOS 4

AmigaOS 4.0 "Final update" revision introduced a new system for allocating RAM and defragmenting physical memory. It still uses flat shared address space that can not be defragmented. It is based on slab allocation method and paging memory that allows swapping. Paging was finally implemented in AmigaOS 4.1. Swap memory could be activated and deactivated any moment allowing the user to choose to use only physical RAM.

Performance

The backing store for a virtual memory operating system is typically many orders of magnitude slower than RAM. Additionally, using mechanical storage devices introduces delay, several milliseconds for a harddisk. Therefore it is desirable to reduce or eliminate swapping, where practical. Some operating systems offer settings to influence the kernel's decisions.

1. Linux offers the `/proc/sys/vm/swappiness` parameter, which changes the balance between swapping out runtime memory, as opposed to dropping pages from the system page cache.
2. Windows 2000, XP, and Vista offer the `DisablePagingExecutive` registry setting, which controls whether kernel-mode code and data can be eligible for paging out.

3. Mainframe computers frequently used head-per-track disk drives or drums for swap storage to eliminate rotational delay.
4. Flash memory has a finite number of erase-write cycles and the smallest amount of data that can be erased at once might be very large (128 KiB for an Intel X25-M SSD), seldom coinciding with pagesize. Therefore, flash memory may wear out quickly if used as swap space under tight memory conditions. On the attractive side, flash memory is practically delayless compared to harddisks, and not volatile as RAM chips. Schemes like ReadyBoost and Intel Turbo Memory are made to exploit these characteristics.

Many Unix-like operating systems (for example AIX, Linux and Solaris) allow using multiple storage devices for swap space in parallel, to increase performance.

Tuning swap space size

In some older virtual memory operating systems, space in swap backing store is reserved when programs allocate memory for runtime data. OS vendors typically issue guidelines about how much swap space should be allocated.

Reliability

Swapping can decrease system reliability by some amount. If swapped data gets corrupted on the disk (or at any other location during transfer), the memory will also have incorrect contents after the data has later been returned.

Addressing limits on 32 bit hardware

It is not uncommon to find 32 bit computers with 4 GB of RAM, the maximum amount addressable without the use of, e.g., PAE. For some machines, e.g., the IBM S/370 in XA mode, the upper bit was not part of the address and only 2 GB could be addressed.

Paging and swap space can be used beyond this 4 GB limit, due to it being addressed in terms of pages rather than individual bytes.

While 32 bit programs on machines with linear address spaces will continue to be limited to the 4 GB they're capable of addressing, because they each exist in their own virtual address space, a group of programs can together grow beyond this limit and into any available space.

On machines with segment registers, e.g., the access registers on an IBM System/370 in ESA mode, the address space size is limited only by OS constraints, e.g., the need to fit the mapping tables into the available storage.

Chapter 16

Physical Address Extension

In computing, **Physical Address Extension** (**PAE**) is a feature to allow x86 processors to access a physical address space (including random access memory and memory mapped devices) larger than 4 gigabytes.

First implemented in the Intel Pentium Pro in 1995, it was extended by AMD to add a level to the page table hierarchy, to allow it to handle up to 52-bit physical addresses, add NX bit functionality, and make it the mandatory memory paging model in long mode. PAE is provided by Intel Pentium Pro and above CPUs, including all later Pentium-series processors (except the 400 MHz-bus versions of the Pentium M). It is also available on other processors with similar or more advanced versions of the same architecture, such as the AMD Athlon and later AMD processor models.

x86 processor hardware-architecture is augmented with additional address lines used to select the additional memory, so physical address size increases from 32 bits to 36 bits. This, theoretically, increases maximum physical memory size from 4 GB to 64 GB. The 32-bit size of the virtual address is not changed, so regular application software continues to use instructions with 32-bit addresses and (in a flat memory model) is limited to 4 gigabytes of virtual address space. The operating system uses page tables to map this 4-GB address space into the 64 GB of physical memory. The mapping is typically applied differently for each process. In this way, the extra memory is useful even though no single regular application can access it all simultaneously.

To use PAE, operating system support is required. Intel versions of Mac OS X support PAE. The Linux kernel supports PAE as a build option and most major distributions provide a PAE kernel either as the default or as an option. FreeBSD and NetBSD also support PAE as a kernel build option.

Microsoft Windows implements PAE if booted with the appropriate option, but current 32-bit desktop editions enforce the physical address space within 4GB even in PAE mode. According to Geoff Chappell, Microsoft limits 32-bit versions of Windows to 4GB due to a licensing restriction, and Microsoft Technical Fellow Mark Russinovich says that some drivers were found to be unstable when encountering physical addresses above 4GB. Unofficial kernel patches for Windows Vista and Windows 7 32-bit are available that break this enforced limitation, though the stability is not guaranteed.

For application software which needs access to more than 4 GB of RAM, operating systems may provide some special mechanisms in addition to the regular PAE support. On Windows this mechanism is called Address Windowing Extensions, while on Unix-like systems a variety of techniques are used, such as using `mmap()` to map regions of a file into and out of the address space as needed.

Page table structures

In traditional 32-bit protected mode, x86 processors use a two-level page translation scheme, where the control register `CR3` points to a single 4 kiB long *page directory* divided 1024 × 4 byte entries that point to 4 kiB long page tables, similarly consisting of 1024 × 4 byte entries pointing to 4 KiB long pages.

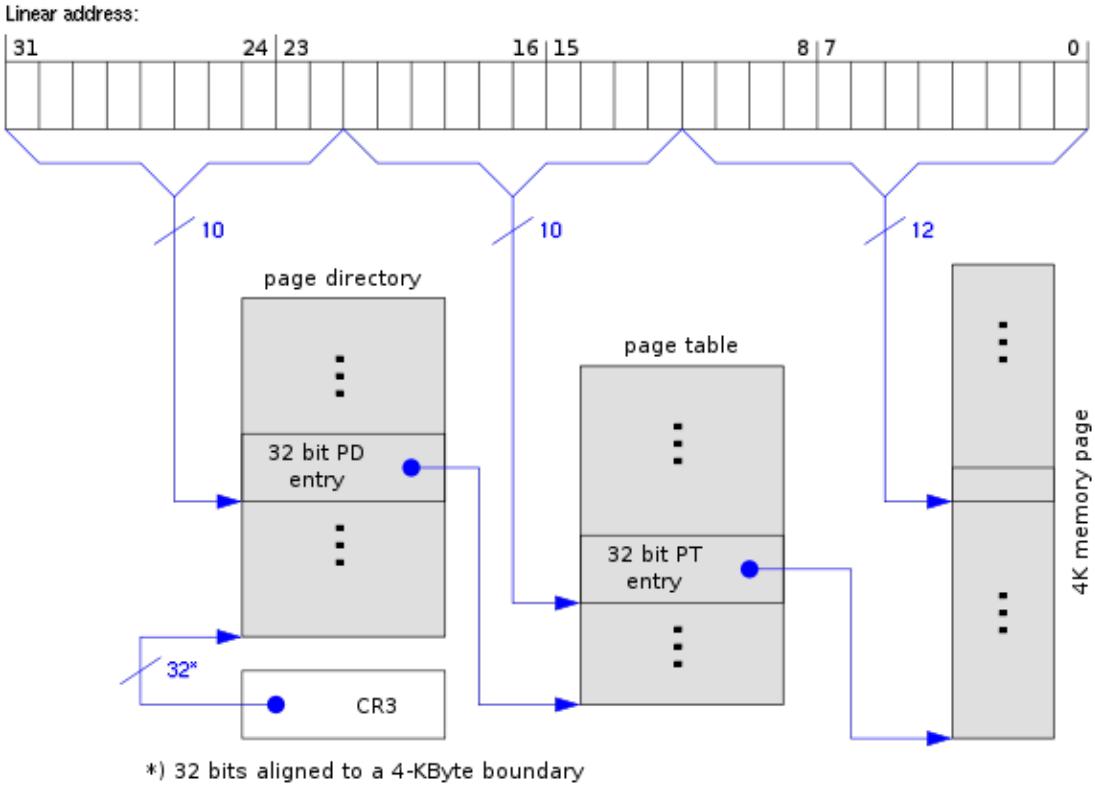
Enabling PAE (by setting bit 5, *PAE*, of the system register `CR4`) causes major changes to this scheme. By default, the size of each page remains as 4 kiB. Each entry in the page table and page directory grows to 64 bits (8 bytes) rather than 32 bits - to allow for additional address bits; however, the size of tables *does not* change, so both table and directory now have only 512 entries. Because this allows only one quarter of the entries of the original scheme, an extra level of hierarchy has been added, so CR3 now points to the *Page Directory Pointer Table*, a short table which contains pointers to 4 page directories.

The entries in the page directory have an additional flag in bit 7, named `PS` (for *page size*). If the system has set this bit to 1, the page directory entry does not point to a page table, but to a single large 2 MiB page. The NX bit is another flag in the page directory, in bit 63, to mark pages as *no execute*. Because the 12 least significant bits of the page table entry's 64 bits are either similar flags or are available for OS-specific data, a maximum of 52 bits can be potentially used in the future to address 2^{52} bytes, or 4 petabytes, of physical memory.

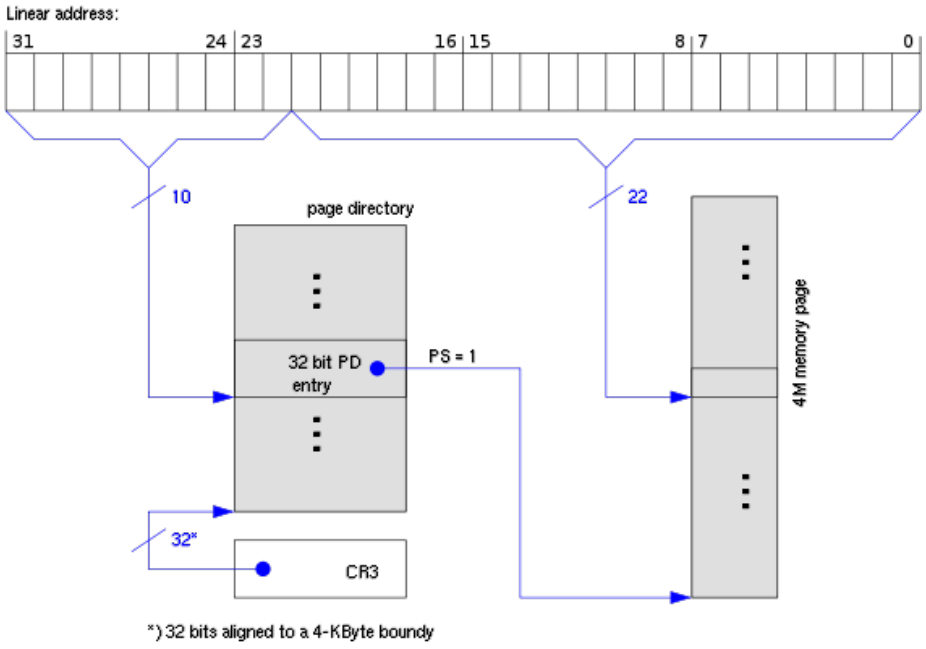
Software can identify via the CPUID flag *PAE* whether a CPU supports PAE mode.

On x86-64 processors in native long mode, the address translation scheme uses PAE but adds a fourth table, the 512-entry *page-map level 4* table, and extends the page directory pointer table to 512 entries instead of the original 4 entries it has in protected mode. 36 bits of virtual page number are translated, giving a virtual address space of up to 256 TB. In the page table entries, in the original specification, 40 bits of physical page number are implemented.

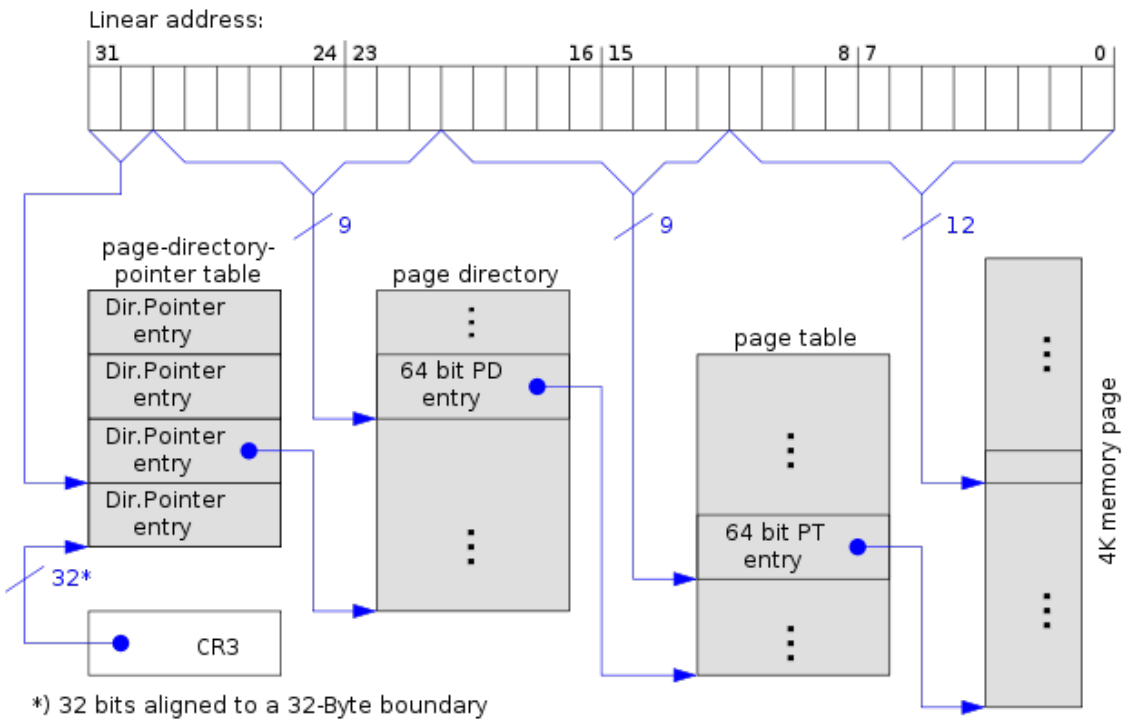
Page table structures



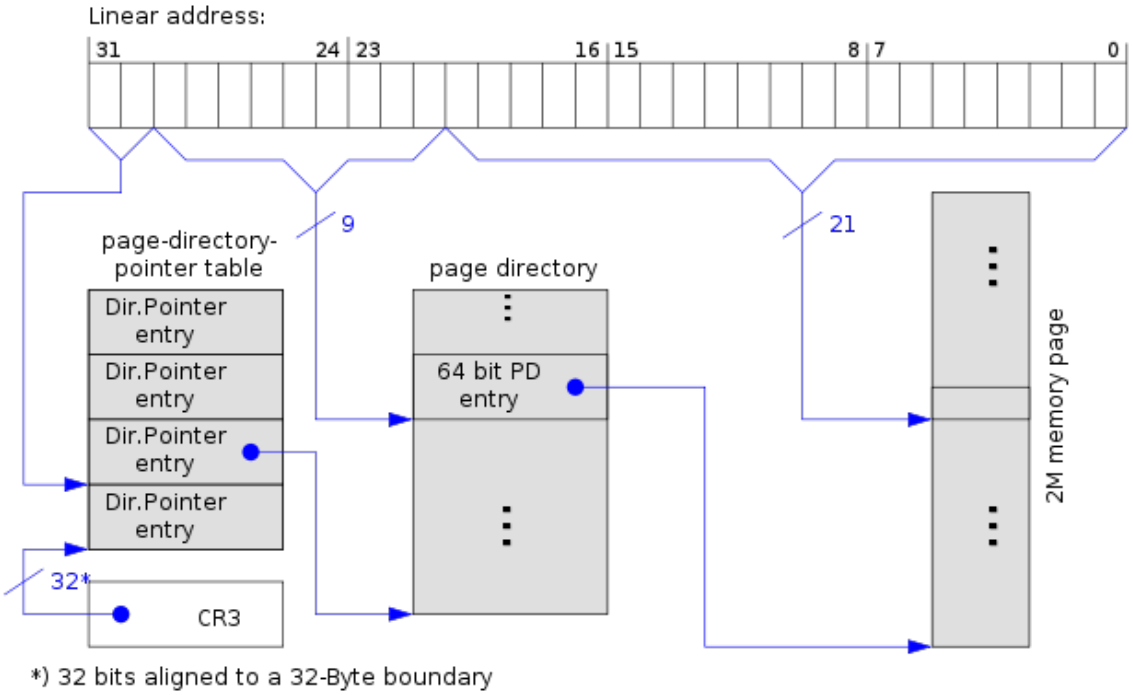
No PAE, 4 kiB pages



No PAE, 4 MiB pages



PAE with 4 kiB pages



PAE with 2 MiB pages

Operating-system support

FreeBSD

FreeBSD supports PAE in the 4.x series starting with 4.9, in the 5.x series starting with 5.1, and in all 6.x and later releases. Support requires the kernel **PAE** configuration-option. Loadable kernel modules can only be loaded into a kernel with PAE enabled if the modules were built with PAE enabled; the binary modules in FreeBSD distributions are not built with PAE enabled, and thus cannot be loaded into PAE kernels. Not all drivers support more than 4 GB of physical memory; those drivers won't work correctly on a system with PAE.

Linux

The Linux kernel includes full PAE mode support starting with version 2.3.23, enabling access of up to 64 GB of memory on 32-bit machines. A PAE-enabled Linux-kernel requires that the CPU also support PAE. As of 2009, some common Linux distributions have started to use a PAE-enabled kernel as the distribution-specific default because it adds the NX bit.

Mac OS X

Versions 10.4.4 through 10.5.8 of Mac OS X will run on both x86 and PowerPC processors. Version 10.6 and the future 10.7 version of OS X will only run on an x86 processor. So far, all x86 Macs have used Intel (not AMD) CPUs. OS X versions that are compatible with x86 fully support PAE and the NX bit on all Intel Macs. Mac Pro and Xserve systems can use up to 32 GB of RAM. The Mac OS X 10.5 (Leopard) kernel remains 32-bit. Mac OS X 10.6 (Snow Leopard) can be booted into a 64-bit version of the kernel on certain systems.

Microsoft Windows

The x86 versions of the following releases of Microsoft Windows support PAE: Itanium versions of these operating systems (if they exist) do not use PAE because the Itanium does not need nor implement PAE. x64 editions of Windows always implement PAE, because it is a mandatory feature of Long mode.

Windows Versions (and their maximum addressable physical memory/RAM (GB))	32-bit Editions	64-bit Editions
Windows 2000 Professional, Server	4	N/A
Windows 2000 Advanced Server	8	N/A
Windows 2000 Datacenter	32	N/A
Windows XP Starter	0.5	N/A
Windows XP Home & Media Center	4	N/A

Windows XP Professional	4	128
Windows Server 2003 Web	2	N/A
Windows Server 2003 Small Business, Home, Storage	4	N/A
Windows Server 2003 Storage Server	4	N/A
Windows Server 2003 R2 Standard Edition	4	32
Windows Server 2003 Standard Edition (SP1)	4	32
Windows Server 2003 Standard Edition (SP2)	4	32
Windows Server 2003 Enterprise Edition (SP1)	16 GB with 4GT	N/A
Windows Server 2003 R2 Enterprise Edition	64	1 TB
Windows Server 2003 R2 Datacenter	64	1 TB
Windows Server 2003 Datacenter Edition (SP1)	16 GB with 4GT	1 TB
Windows Vista Starter	1	N/A
Windows Vista Home Basic	4	8
Windows Vista Home Premium	4	16
Windows Vista Business, Enterprise, Ultimate	4	128
Windows Server 2008 Standard, Web	4	32
Windows Server 2008 Enterprise, Datacenter	64	2 TB
Windows 7 Starter	2	N/A
Windows 7 Home Basic	4	8
Windows 7 Home Premium	4	16
Windows 7 Professional, Enterprise, Ultimate	4	192
Windows Server 2008 R2 Foundation	N/A	8
Windows Server 2008 R2 Standard	N/A	32
Windows Server 2008 R2 Enterprise, Datacenter, or Itanium	N/A	2 TB

Annotations: Values for Windows Server 2008 64-bit also apply to Windows Server 2008 *R2* (which dropped 32-bit support). Values for Windows Server 2003 64-bit depend on:

1. service pack level
2. the release being R2 or not

The highest possible values appear here. The original release of Windows XP used PAE mode to allow RAM to extend beyond the 4GB limit. However, it led to compatibility problems with 3rd party drivers which led Microsoft to remove this capability in Windows XP Service Pack 1. Windows XP SP2 and later, by default, on processors with the no-execute (NX) or execute-disable (XD) feature, runs in PAE mode in order to allow NX. The *no execute* (NX, or XD for *execution disable*) bit resides in bit 63 of the page table entry and, without PAE, page table entries on 32-bit systems have only 32 bits;

therefore PAE mode is required in order to exploit the NX feature. However, "client" versions of 32-bit Windows (Windows XP SP1 and later, Windows Vista, Windows 7) limit physical address space to the first 4 GB for driver compatibility and licensing reasons, even though these versions do run in PAE mode if NX support is enabled.

Solaris

Solaris supports PAE beginning with Solaris version 7. However, third-party drivers used with version 7 which do not specifically include PAE support may operate erratically or fail outright on a system with PAE.

Chapter 17

Malloc

In computing, **`malloc`** is a subroutine for performing dynamic memory allocation in the C and C++ programming languages, though its use in C++ has been largely superseded by operators `new` and `new[]`. `malloc` is part of the standard library for both languages and is declared in the `stdlib.h` header although it is *also* declared within the `std` namespace via the C++'s `cstdlib` header.

Many implementations of `malloc` are available, each of which performs differently depending on the computing hardware and how a program is written. Performance varies in both execution time and required memory. Programs must properly manage dynamic memory allocated through the use of `malloc` to avoid memory leaks and memory corruption.

Rationale

The C programming language manages memory statically, automatically, or dynamically. Static-duration variables are allocated in main (fixed) memory and persist for the lifetime of the program; automatic-duration variables are allocated on the stack and come and go as functions are called and return. For static-duration and, before C99 (which allows variable-length automatic arrays), automatic-duration variables, the size of the allocation is required to be compile-time constant. If the required size is not known until run-time (for example, if data of arbitrary size is being read from the user or from a disk file), then using fixed-size data objects is inadequate.

The lifetime of allocated memory is also a concern. Neither static- nor automatic-duration memory is adequate for all situations. Automatic-allocated data cannot persist across multiple function calls, while static data persists for the life of the program whether it is needed or not. In many situations the programmer requires greater flexibility in managing the lifetime of allocated memory.

These limitations are avoided by using dynamic memory allocation in which memory is more explicitly (but more flexibly) managed, typically, by allocating it from the heap, an area of memory structured for this purpose. In C, the library function `malloc` is used to allocate a block of memory on the heap. The program accesses this block of memory via a pointer that `malloc` returns. When the memory is no longer needed, the pointer is passed to ***`free`*** which deallocates the memory so that it can be used for other purposes.

Some platforms provide library calls which allow run-time dynamic allocation from the C stack rather than the heap (e.g. Unix `alloca()`, Microsoft Windows CRTL's `malloca()`). This memory is automatically freed when the calling function ends. The need for this is lessened by changes in the C99 standard, which added support for variable-length arrays of block scope having sizes determined at runtime.

Dynamic memory allocation in C

The `malloc` function is one of the functions in standard C to allocate memory. Its function prototype is

```
void *malloc(size_t size);
```

which allocates *size* bytes of memory. If the allocation succeeds, a pointer to the block of memory is returned which is guaranteed to be suitable aligned to any type (including struct and such), otherwise a NULL pointer is returned.

Memory allocated via `malloc` is persistent: it will continue to exist until the program terminates or the memory is explicitly deallocated by the programmer (that is, the block is said to be "free"). This is achieved by use of the `free` function. Its prototype is

```
void free(void *pointer);
```

which releases the block of memory pointed to by `pointer`. `pointer` must have been previously returned by `malloc`, `calloc`, or `realloc` and must be passed to `free` only once. It is safe to call free on a NULL pointer, which has no effect.

Usage example

The standard method of creating an array of 10 int objects:

```
int array;
```

However, if one wishes to allocate a similar array dynamically, the following code could be used:

```
/* Allocate space for an array with ten elements of type
   int. */
int *ptr = malloc(10 * sizeof (int));
if (ptr == NULL) {
    /* Memory could not be allocated, the program should
       handle the error here as appropriate. */
} else {
    /* Allocation succeeded.  Do something.  */
    free(ptr);  /* We are done with the int objects, and
                   free the associated pointer. */
    ptr = NULL; /* The pointer must not be used again,
                   unless re-assigned by using malloc
                   again. */
```

```
}
```

`malloc` returns a null pointer to indicate that no memory is available, or that some other error occurred which prevented memory being allocated.

A useful idiom with `malloc` is shown in this example:

```
int *ptr = malloc(10 * sizeof(*ptr));
```

That is, instead of writing a hard-wired type into the argument to malloc, one uses the `sizeof` operator on the content of the pointer to be allocated. This ensures that the types on the left and right of the assignment will never get out of sync when code is revised.

Casting and type safety

`malloc` returns a void pointer (`void *`), which indicates that it is a pointer to a region of unknown data type. The lack of a specific pointer type returned from `malloc` is type-unsafe behaviour: `malloc` allocates based on byte count but not on type. This distinguishes it from the C++ new operator that returns a pointer whose type relies on the operand.

One may "cast" this pointer to a specific type:

```
int *ptr;
ptr = malloc(10 * sizeof (int)); // Without a cast
ptr = (int*)malloc(10 * sizeof (int)); // With a cast
```

There are advantages and disadvantages to performing such a cast.

Advantages to casting

- Including the cast allows for compatibility to C++, which does require the cast to be made.
- If the cast is present and the type of the left-hand-side pointer is subsequently changed, a warning will be generated to help the programmer in correcting behaviour that otherwise could become erroneous.
- The cast allows for older versions of `malloc` that originally returned a `char *`.

Disadvantages to casting

- Under the ANSI C standard, the cast is redundant.
- Adding the cast may mask failure to include the header `stdlib.h`, in which the prototype for `malloc` is found. In the absence of a prototype for `malloc`, the standard requires that the C compiler assume `malloc` returns an `int`. If there is no cast, a warning is issued when this integer is assigned to the pointer; however, with the cast, this warning is not produced, hiding a bug. On certain architectures and data models (such as LP64 on 64-bit systems, where `long` and pointers are

64-bit and `int` is 32-bit), this error can actually result in undefined behaviour, as the implicitly declared `malloc` returns a 32-bit value whereas the actually defined function returns a 64-bit value. Depending on calling conventions and memory layout, this may result in stack smashing.

Related functions

calloc

`malloc` returns a block of memory that is allocated for the programmer to use, but is uninitialized. The memory is usually initialized by hand if necessary—either via the **`memset`** function, or by one or more assignment statements that dereference the pointer. An alternative is to use the **`calloc`** function, which allocates memory and then initializes it. Its prototype is

```
void *calloc(size_t nelements, size_t elementSize);
```

which allocates a region of memory, initialized to 0, of size `nelements` × `elementSize`. This can be useful when allocating an array of characters to hold as string as in the example below:

```
char *word = calloc(200, sizeof(char));
```

realloc

It is often useful to be able to grow or shrink a block of memory. This can be done using `realloc` which returns a pointer to a memory region of the specified size, which contains the same data as the old region pointed to by `pointer` (truncated to the minimum of the old and new sizes). If `realloc` is unable to resize the memory region in place, it allocates new storage, copies the required data, and frees the old pointer. If this allocation fails, `realloc` maintains the original pointer unaltered, and returns the null pointer value. The newly allocated region of memory is uninitialized (its contents are not predictable). The function prototype is

```
void *realloc(void *pointer, size_t size);
```

With size bigger than zero, `realloc` behaves like `malloc` if the first argument is NULL:

```
void *p = malloc(42);
void *p = realloc(NULL, 42); /* equivalent */
```

In both C89 and C99, `realloc` with length 0 is a special case. The C89 standard explicitly states that the pointer given is freed, and that the return is either a null pointer or a pointer to the newly allocated space. The C99 standard says that the behavior is implementation-defined. It's possible that malloc and realloc with size 0 return different (null and non-null) pointers. Other standards, such as the Open Group's UNIX standards, make it implementation defined whether `realloc(p, 0)` frees `p` without allocating new

space, or possibly frees `p` and returns a valid pointer to at least zero bytes of memory. Under all standards, NULL can be returned on memory allocation failure. When using `realloc`, one should always use a temporary variable. For example

```
void *p = malloc(orig_size);
/* and later... */
void *tmp = realloc(p, big_size);
if (tmp != NULL) {
   p = tmp; /* OK, assign new, larger storage to p */
} else {
   /* handle the problem somehow */
}
```

If instead one did

```
void *p = malloc(orig_size);
/* and later... */
p = realloc(p, big_size);
```

and if it is not possible to obtain `big_size` bytes of memory, then p will have value NULL and we no longer have a pointer to the memory previously allocated for p, creating a memory leak.

Common errors

The improper use of `malloc` and related functions can frequently be a source of bugs.

Allocation failure

`malloc` is not guaranteed to succeed—if there is no memory available, or if the program has exceeded the amount of memory it is allowed to reference, `malloc` will return a null pointer, which should always be checked for after allocation. Many programs do not check for `malloc` failure. Such a program would attempt to use the null pointer returned by `malloc` as if it pointed to allocated memory, and the program would crash.

Memory leaks

When a call to `malloc`, `calloc` or `realloc` succeeds, the return value of the call should eventually be passed to the `free` function. This releases the allocated memory, allowing it to be reused to satisfy other memory allocation requests. If this is not done, the allocated memory will not be released until the process exits (and in some environments, not even then)—in other words, a memory leak will occur. Typically, memory leaks are caused by losing track of pointers, for example not using a temporary pointer for the return value of `realloc`, which may lead to the original pointer being overwritten with a null pointer, for example:

```
void *ptr;
size_t size = BUFSIZ;
```

```
ptr = malloc(size);

/* some further execution happens here... */

/* now the buffer size needs to be doubled */
if (size > SIZE_MAX / 2) {
  /* handle overflow error */
  /* . probably appropriate to use free( ptr ) here . */
  return (1);
}
size *= 2;
ptr = realloc(ptr, size);
if (ptr == NULL) {
  /* the realloc failed (it returned a null pointer), but
     the original address in ptr has been lost so the
     memory cannot be freed and a leak has occurred */
  /* ... */
  return 1;
}
/* ... */
```

Use after free

After a pointer has been passed to `free`, it becomes a dangling pointer: it references a region of memory with undefined content, which may not be available for use. The pointer's value cannot be accessed. For example:

```
int *ptr = malloc(sizeof (int));
free(ptr);
*ptr = 7; /* Undefined behavior */
```

Code like this has undefined behavior: its effect may vary.

Commonly, the system may have reused freed memory for other purposes. Therefore, writing through a pointer to a deallocated region of memory may result in overwriting another piece of data somewhere else in the program. Depending on what data is overwritten, this may result in data corruption or cause the program to crash at a later time. A particularly bad example of this problem is if the same pointer is passed to `free` twice, known as a *double free*. 'Double free' bugs can lead to security vulnerabilities. To avoid this, some programmers set pointers to `NULL` after passing them to `free`:

```
free(ptr);
ptr = NULL; /*is safe (throws away the pointer's location).*/
```

However, this will not protect other aliases to the same pointer from being doubly freed.

Best practice is that a pointer passes out of scope immediately after being freed.

Freeing unallocated memory

Another problem is when `free` is passed an address that was not allocated by `malloc`, `realloc` or `calloc`. This can be caused when a pointer to a literal string or the name of a declared array is passed to `free`, for example:

```
char *msg = "Default message";
int tbl;
```

Passing either of the above pointers to `free` will result in undefined behaviour.

Implementations

The implementation of memory management depends greatly upon operating system and architecture. Some operating systems supply an allocator for malloc, while others supply functions to control certain regions of data. The same dynamic memory allocator is often used to implement both `malloc` and operator `new` in C++. Hence, it is referred to below as the *allocator* rather than `malloc`.

Heap-based

Implementation of the allocator on IA-32 architectures is commonly done using the heap, or data segment. The allocator will usually expand and contract the heap to fulfill allocation requests.

The heap method suffers from a few inherent flaws, stemming entirely from fragmentation. Like any method of memory allocation, the heap will become fragmented; that is, there will be sections of used and unused memory in the allocated space on the heap. A good allocator will attempt to find an unused area of already allocated memory to use before resorting to expanding the heap. The major problem with this method is that the heap has only two significant attributes: base, or the beginning of the heap in virtual memory space; and length, or its size. The heap requires enough system memory to fill its entire length, and its base can never change. Thus, any large areas of unused memory are wasted. The heap can get "stuck" in this position if a small used segment exists at the end of the heap, which could waste any magnitude of address space, from a few megabytes to a few hundred.

dlmalloc and its derivatives

Doug Lea is the author of a well known memory allocator called dlmalloc ("Doug Lea's Malloc") whose source code describes itself as:

"This is not the fastest, most space-conserving, most portable, or most tunable malloc ever written. However it is among the fastest while also being among the most space-conserving, portable and tunable. Consistent balance across these factors results in a good general-purpose allocator for malloc-intensive programs."

The first implementation of dlmalloc was created in 1987. It is written in C and is highly portable, being known to work well on all major operating systems and processor architectures and on systems ranging from medium/small embedded right up to supercomputers. Due to its longevity and open source nature, dlmalloc is widely used for teaching purposes and as a foundation for other allocators, the best known of which is ptmalloc2/ptmalloc3. Since the v2.3 release, the GNU C library (glibc) uses a **modified** ptmalloc2, which itself is based on dlmalloc v2.7.0.

Another, lesser known dlmalloc derivative is nedmalloc which is based on dlmalloc v2.8.4 and is essentially dlmalloc wrapped by a per-thread lookaside cache to improve execution concurrency.

Memory on the heap is allocated as "chunks", an 8-byte aligned data structure which contains a header and usable memory. Allocated memory contains a 8 or 16 byte overhead for the size of the chunk and usage flags. Unallocated chunks also store pointers to other free chunks in the usable space area, making the minimum chunk size 24-bytes.

Unallocated memory is grouped into "bins" of similar sizes, implemented by using a double-linked list of chunks (with pointers stored in the unallocated space inside the chunk).

For requests below 256 bytes (a "smallbin" request), a simple two power best fit allocator is used. A CPU special op (on GCC this is __builtin_clz()) is used to very quickly find the first set bit and a block is returned for the bin corresponding to that top bit position. If there are no free blocks in that bin, a block from the next highest bin is split in two.

For requests of 256 bytes or above but below the mmap threshold, i.e. what dlmalloc calls a "largebin" request, recent versions of dlmalloc use an in-place *bitwise trie* algorithm. This traverses a binary tree on the basis of each bit state after the first set bit which modern out-of-order CPUs can do very efficiently and which is mostly invariant to the number of allocated blocks. A big advantage of this algorithm is that if it fails to find the size requested, it returns the block with the next biggest size which makes it very easy to split that block into the size required.

For requests above the mmap threshold, the memory is always allocated using the mmap system call. The threshold is 256 KB (1 MB on ptmalloc2) by default, but can be changed by calling the mallopt function. The mmap method averts problems with huge buffers trapping a small allocation at the end after their expiration, but always allocates an entire page of memory, which on many architectures are 4096 bytes in size.

If additional memory below the threshold but less than the available free space needs to be allocated, dlmalloc may use the `brk()` call to the Linux kernel to increase the size of the heap. Increasing the size of the heap increases the size of the top-most chunk (wilderness chunk), which is always unallocated, and is treated specially by malloc.

dlmalloc has a fairly weak free space segment coalescer algorithm, mainly because free space coalescing tends to be extremely slow due to causing TLB cache exhaustion. It is called every (by default) 4096 free() operations and it works by iterating each of the segments previously requested from the system which were not contiguously returned by the system. It tries to identify large ranges of memory which contain no allocated blocks and breaking its segment into two with the free memory being returned to the system. This algorithm works well if dlmalloc is the sole user of the VM system, however if dlmalloc is used simultaneously with another allocator then dlmalloc's free space coalescer can fail to correctly identify opportunities for free memory release.

glibc implementation of ptmalloc2 differs from dlmalloc in this case as it generally requests extra memory from the kernel using mmap to allocate 1Mb aligned chunks, which its source code refers to as *arenas*. ptmalloc2 tries to ensure separate arenas per execution thread, thus permitting concurrency within the memory allocator.

ptmalloc3 improves significantly on ptmalloc2 by making the smallbins of dlmalloc (described above) per-thread. This allows ptmalloc3 to offer lock free concurrency for smaller blocks while still allowing separate arenas for largebin allocations. Allocations beyond the mmap threshold still route exclusively through mmap().

nedmalloc is similar to ptmalloc2 in its support of multiple per-thread arenas, but it also adds a separate per-thread lookaside cache for smaller sized blocks which avoids processor serialisation for typical C++ usage patterns much as ptmalloc3 does. nedmalloc, like Hoard referred to below, is able to patch Microsoft Windows binaries to replace the system allocator with itself within a given process. The most recent versions of nedmalloc implement a user mode page allocator which replaces mmap(), thus allowing memory pages to be held in a lookaside cache and therefore greatly improving the speed of large allocations.

All of dlmalloc, ptmalloc2, ptmalloc3 and nedmalloc are licensed under an open source licence and are therefore available for student study.

FreeBSD's and NetBSD's `jemalloc`

Since FreeBSD 7.0 and NetBSD 5.0, old `malloc` implementation (phkmalloc) was replaced by jemalloc, written by Jason Evans. The main reason for this was a lack of scalability of phkmalloc in terms of multithreading. In order to avoid lock contention, jemalloc uses separate "arenas" for each CPU. Experiments measuring number of allocations per second in multithreading application have shown that this makes it scale linearly with the number of threads, while for both pkhmalloc and dlmalloc performance was inversely proportional to the number of threads.

jemalloc is used as the default allocator, in Windows and Linux, of Firefox 3 beta4pre and later instead of the one provided by the operating system, except for Mac OS X. This improves performance and lowers memory consumption, due to less fragmentation.

OpenBSD's `malloc`

OpenBSD's implementation of the `malloc` function makes use of `mmap`. For requests greater in size than one page, the entire allocation is retrieved using `mmap`; smaller sizes are assigned from memory pools maintained by `malloc` within a number of "bucket pages," also allocated with `mmap`. On a call to `free`, memory is released and unmapped from the process address space using `munmap`. This system is designed to improve security by taking advantage of the address space layout randomization and gap page features implemented as part of OpenBSD's `mmap` system call, and to detect use-after-free bugs—as a large memory allocation is completely unmapped after it is freed, further use causes a segmentation fault and termination of the program.

Hoard's `malloc`

The Hoard memory allocator is an allocator whose goal is scalable memory allocation performance. Like OpenBSD's allocator, Hoard uses `mmap` exclusively, but manages memory in chunks of 64 kilobytes called superblocks. Hoard's heap is logically divided into a single global heap and a number of per-processor heaps. In addition, there is a thread-local cache that can hold a limited number of superblocks. By allocating only from superblocks on the local per-thread or per-processor heap, and moving mostly-empty superblocks to the global heap so they can be reused by other processors, Hoard keeps fragmentation low while achieving near linear scalability with the number of threads.

Thread-caching malloc `tcmalloc`

Every thread has local storage for small allocations. For large allocations mmap or sbrk can be used. Tcmalloc has garbage-collection for local storage of dead threads. The TCmalloc is considered to be more than twice as fast as glibc's ptmalloc for multithreaded programs.

In-kernel

Operating system kernels need to allocate memory just as application programs do. The implementation of `malloc` within a kernel often differs significantly from the implementations used by C libraries, however. For example, memory buffers might need to conform to special restrictions imposed by DMA, or the memory allocation function might be called from interrupt context . This necessitates a `malloc` implementation tightly integrated with the virtual memory subsystem of the operating system kernel.

Allocation size limits

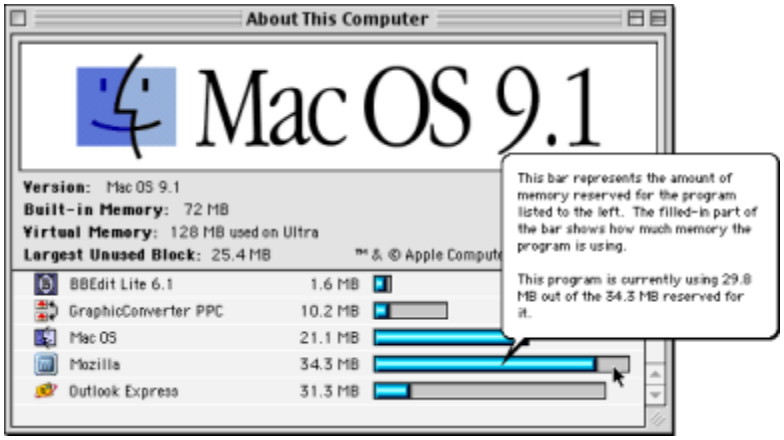
The largest possible memory block `malloc` can allocate depends on the host system, particularly the size of physical memory and the operating system implementation. Theoretically, the largest number should be the maximum value that can be held in a *size t* type, which is an implementation-dependent unsigned integer representing the size of an

area of memory. The maximum value is $2^{CHAR\_BIT*sizeof(size\_t)} - 1$, or the constant `SIZE_MAX` in the C99 standard.

Chapter 18

Mac OS Memory Management and Bank Switching

Mac OS memory management



"About This Computer" Mac OS 9.1 window showing the memory consumption of each open application and the system software itself.

Historically, the Mac OS used a form of memory management that has fallen out of favour in modern systems. Criticism of this approach was one of the key areas addressed by the change to Mac OS X.

The original problem for the designers of the Macintosh was how to make optimum use of the 128 KB of RAM that the machine was equipped with. Since at that time the machine could only run one application program at a time, and there was no fixed secondary storage, the designers implemented a simple scheme which worked well with those particular constraints. However, that design choice did not scale well with the development of the machine, creating various difficulties for both programmers and users.

Fragmentation

The primary concern of the original designers appears to have been fragmentation - that is, the repeated allocation and deallocation of memory through pointers leads to many small isolated areas of memory which cannot be used because they are too small, even though the total free memory may be sufficient to satisfy a particular request for memory. To solve this, Apple designers used the concept of a relocatable handle, a reference to memory which allowed the actual data referred to be moved without invalidating the handle. Apple's scheme was simple - a handle was simply a pointer into a (non relocatable) table of further pointers, which in turn pointed to the data. If a memory request required compaction of memory, this was done and the table, called the master pointer block, was updated. The machine itself implemented two areas in the machine available for this scheme - the system heap (used for the OS), and the application heap. As long as only one application at a time was run, the system worked well. Since the entire application heap was dissolved when the application quit, fragmentation was minimized.

However, in addressing the fragmentation problem, all other issues were overlooked. The system heap was not protected from errant applications, and this was frequently the cause of major system problems and crashes. In addition, the handle-based approach also opened up a source of programming errors, where pointers to data within such relocatable blocks could not be guaranteed to remain valid across calls that might cause memory to move. In reality this was almost every system API that existed. Thus the onus was on the programmer not to create such pointers, or at least manage them very carefully. Since many programmers were not generally familiar with this approach, early Mac programs suffered frequently from faults arising from this - faults that very often went undetected until long after shipment.

Palm OS and 16-bit Windows use a similar scheme for memory management. However, the Palm and Windows versions make programmer error more difficult. For instance, in Mac OS to convert a handle to a pointer, a program just de-references the handle directly. However, if the handle is not locked, the pointer can become invalid quickly. Calls to lock and unlock handles are not balanced; ten calls to HLock are undone by a single call to HUnlock. In Palm OS and Windows, handles are opaque type and must be de-referenced with MemHandleLock on Palm OS or Global/LocalLock on Windows. When a Palm or Windows application is finished with a handle, it calls MemHandleUnlock or Global/LocalUnlock. Palm OS and Windows keeps a lock count for blocks; after three calls to MemHandleLock, a block will only become unlocked after three calls to MemHandleUnlock.

Switcher

The situation worsened with the advent of Switcher, which was a way for the Mac to run multiple applications at once. This was a necessary step forward for users, who found the one-app-at-a-time approach very limiting. However, because Apple was now committed to its memory management model, as well as compatibility with existing applications, it

was forced to adopt a scheme where each application was allocated its own heap from the available RAM. The amount of actual RAM allocated to each heap was set by a value coded into each application, set by the programmer. Invariably this value wasn't enough for particular kinds of work, so the value setting had to be exposed to the user to allow them to tweak the heap size to suit their own requirements. This exposure of a technical implementation detail was very much against the grain of the Mac user philosophy. Apart from exposing users to esoteric technicalities, it was inefficient, since an application would grab (unwillingly) all of its allotted RAM, even if it left most of it subsequently unused. Another application might be memory starved, but was unable to utilise the free memory "owned" by another application.

Switcher became MultiFinder in System 4.2, which became the Process Manager in System 7, and by then the scheme was utterly entrenched. Apple made some attempts to work around the obvious limitations – temporary memory was one, where an application could "borrow" free RAM that lay outside of its heap for short periods, but this was unpopular with programmers so it largely failed to solve the problem. There was also an early virtual memory scheme, which attempted to solve the issue by making more memory available by paging unused portions to disk, but for most users with 68K Macintoshes, this did nothing but slow everything down without solving the memory problems themselves. By this time all machines had permanent hard disks and MMU chips, and so were equipped to adopt a far better approach. For some reason, Apple never made this a priority until Mac OS X, even though several schemes were suggested by outside developers that would retain compatibility while solving the overall memory management problem. Third party replacements for the Mac OS memory manager, such as Optimem, showed it could be done.

32-bit clean

Originally the Macintosh had 128 kB of RAM, with a limit of 512 kB. This was increased to 4 MB upon the introduction of the Macintosh Plus. These Macintosh computers used the 68000 CPU, a 32-bit processor, but only had 24 physical address lines. The 24 lines allowed the processor to address up to 16 MB of memory (2^{24} bytes), which was seen as a sufficient amount at the time. However, the RAM limit in the Macintosh design was 4 MB of RAM and 4 MB of ROM, because of the structure of the memory map. This was fixed by changing the memory map with the Macintosh II and the Macintosh Portable, allowing up to 8 MB of RAM.

Because memory was a scarce resource, the authors of the Mac OS decided to take advantage of the unused byte in each address. The original Memory Manager (up until the advent of System 7) placed flags in the high 8 bits of each 32-bit pointer and handle. Each address contained flags such as "locked", "purgeable", or "resource", which were stored in the master pointer table. When used as an actual address, these flags were masked off and ignored by the CPU.

While a good use of very limited RAM space, this design led to problems once Apple introduced the Macintosh II, which used the 32-bit Motorola 68020 CPU. The 68020 had

32 physical address lines and could address up to 4GB (2^{32} bytes) of memory. The flags that the Memory Manager stored in the high byte of each pointer and handle were significant now, and could lead to addressing errors.

In theory, the architects of the Macintosh system software were free to change the "flags in the high byte" scheme to avoid this problem, and they did. For example, on the Macintosh II, HLock() was rewritten to implement handle locking in a way other than flagging the high bits of handles. However, many Macintosh application programmers — and a great deal of the Macintosh system software code itself, even the code in the ROMs (until the IIci, which made the ROM 32-bit clean) – accessed the flags directly rather than using the APIs, such as HLock(), which had been provided to manipulate them. By doing this they rendered their applications incompatible with true 32-bit addressing, and this became known as not being "32-bit clean".

In order to stop continual system crashes caused by this issue, System 6 and earlier running on a 68020 or a 68030 would force the CPU into 24-bit mode, and would only recognize and address the first 8 megabytes of RAM, an obvious flaw in machines whose hardware was wired to accept up to 128MB RAM – and whose product literature advertised this capability. With System 7, the Mac system software was finally made 32-bit clean, but there were still the problem of dirty ROMs. The problem was that the decision to use 24-bit or 32-bit addressing has to be made very early in the boot process, when the ROM routines initialized the Memory Manager to set up a basic Mac environment where NuBus ROMs and disk driver are loaded and executed. Older ROMs did not have this support and so was not possible to boot into 32-bit mode. Surprisingly, the first solution to this flaw was published by software utility company Connectix, whose 1991 product MODE32 reinitialized the Memory Manager and repeated early parts of the Mac boot process, allowing the system to boot into 32-bit mode and enabling the use of all the RAM in the machine. Apple licensed the software from Connectix later in 1991 and distributed it for free. The Macintosh IIci and later Motorola based Macintosh computers had 32-bit clean ROMs. (Even later models were based on the 64-bit AIM PowerPC processor and using New World ROM. The most recent models use 32-bit Intel x86-686 or 64-bit Intel x86-64 processors, connected to an EFI.)

However it was quite a while before applications were updated to remove all 24-bit dependencies, and System 7 provided a way to switch back to 24-bit mode if application incompatibilities were found. By the time of migration to the PowerPC and System 7.1.2, 32-bit cleanliness was mandatory for creating native applications.

Object orientation

The rise of object-oriented languages for programming the Mac – first Object Pascal, then later C++ – also caused problems for the memory model adopted. At first, it would seem natural that objects would be implemented via handles, to gain the advantage of being relocatable. However, these languages, as they were originally designed, used pointers for objects, which would lead to fragmentation issues. A solution, implemented by the THINK (later Symantec) compilers, was to use Handles internally for objects, but

use a pointer syntax to access them. This seemed a good idea at first, but soon deep problems emerged, since programmers could not tell whether they were dealing with a relocatable or fixed block, and so had no way to know whether to take on the task of locking objects or not. Needless to say this led to huge numbers of bugs and problems with these early object implementations. Later compilers did not attempt to do this, but used real pointers, often implementing their own memory allocation schemes to workaround the Mac OS memory model.

While the Mac OS memory model, with all its inherent problems, remained this way right through to Mac OS 9, due to severe application compatibility constraints, the increasing availability of cheap RAM meant that by and large most users could upgrade their way out of a corner. The memory wasn't used efficiently, but it was abundant enough that the issue never became critical. This is perhaps ironic given that the purpose of the original design was to maximise the use of very limited amounts of memory. Mac OS X finally does away with the whole scheme, implementing a modern sparse virtual memory scheme. A subset of the older memory model APIs still exist for compatibility as part of Carbon, but map to the modern memory manager (a threadsafe malloc implementation) underneath. Apple recommends that Mac OS X code use malloc and free "almost exclusively".

Bank switching

Bank switching is a technique to increase the amount of usable memory beyond the amount directly addressable by the processor. It can be used to configure a system differently at different times; for example, a ROM required to start a system from diskette could be switched out when no longer needed. Many modern microcontrollers and microprocessors use bank switching to manage random-access memory, non-volatile memory, input-output devices and system management registers in small embedded systems. The technique was common in 8-bit microcomputer systems. Bank-switching may also be used to work around limitations in address bus width, where some hardware constraint prevents straightforward addition of more address lines.

Unlike memory management by "paging", data is not exchanged with a mass storage device like disk memory. Data remains in quiescent storage in a memory area that is not currently accessible to the processor, (although it may be accessible to other systems).

Technique

Bank switching can be considered a way of extending the address bus of a processor with some external register that is used to control how external memory devices respond to the processor address bus. For example, a processor with a 16-bit external address bus can only address 2^{16} = 65536 memory locations. If an external latch was added to the system,

it could be used to control which of two sets of memory devices, each with 65536 addresses, could be accessed. The processor could change which set is in current use by setting or clearing the latch bit.

The latch can be set or cleared by the processor in several ways; a particular memory address may be decoded and used to control the latch, or, in processors with separately-decoded I/O addresses, an output address may be decoded. Several bank-switching control bits could be gathered into a register, approximately doubling the available memory spaces with each additional bit in the register.

Because the external bank-selecting latch (or register) is not directly connected with the program counter of the processor, it does not automatically change state when the program counter overflows; this cannot be detected by the external latch since the program counter is an internal register of the processor.

There are other limitations. The extra memory is not seamlessly available to programs. Internal registers of the processor remain at their original length, so the processor cannot directly span all of bank-switched memory by, for example, incrementing an internal register. Instead the processor must explicitly do a bank-switching operation to access large memory objects. Generally a bank-switching system will have one block of program memory that is common to all banks; no matter which bank is currently active, for part of the address space only one set of memory locations will be used. This area would be used to hold code that manages the transitions between banks, and also to process interrupts.

Unlike a virtual memory scheme, bank-switching must be explicitly managed by the running program or operating system; the processor hardware cannot automatically detect that data not currently mapped into the active bank is required. The application program must keep track of which memory bank holds a required piece of data, and then call the bank-switching routine to make that bank active. However, bank-switching can access data much faster than, for example, retrieving the data from disk.

A hypothetical memory map of bank-switched memory for a processor that can only address 64 kB. This scheme shows 200 kB of memory, of which only 64 kb can be accessed at any time by the processor. The operating system must manage the bank-switching operation to ensure that program execution can continue when part of memory is not accessible to the processor.

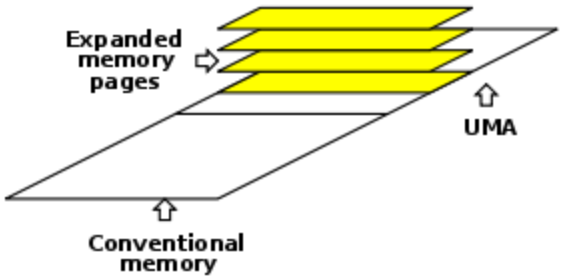
Home computers

Processors with 16-bit addressing (Z80, 6502, 6809, etc.) commonly used in home computers can directly address only 64 KB. Systems with more memory had to divide the address space into a number of blocks that could be dynamically mapped into parts of a larger address space. Blocks of various sizes were switched in and out via bank select registers or similar mechanisms. Some blocks typically were always enabled. Some

caution was required in order not to corrupt the handling of subroutine calls, interrupts, the machine stack, and so on. While the contents of memory temporarily switched out from the CPU was inaccessible to the processor, it could be used by other hardware, such as video display, DMA, I/O devices, etc.

Bank switching allowed extra memory and functions to be added to a computer design without the expense and incompatibility of switching to a processor with a wider address bus. For example, the C64 used bank switching to allow for a full 64KB of RAM and still provide for ROM and memory-mapped I/O as well. The Atari 130XE could allow its two processors (the 6502 and the ANTIC) to access separate RAM banks, allowing programmers on both machines to make large playfields and other graphic objects without using up the memory visible to the CPU.

The IBM PC



Expanded memory in the IBM PC

In 1988 the companies Lotus, Intel and Microsoft agreed on a specification called Expanded Memory System (EMS, also stated as LIM-EMS) for use in IBM PC compatible computers running MS-DOS. It is a form of bank switching technique that allows more than the 640 KB of RAM defined by the original IBM PC architecture, by letting it appear piecewise in a 64 KB "window" located in the Upper Memory Area. The 64 KB is divided into four 16 KB "pages" which can each be independently switched. Some computer games made use of this, and though EMS is obsolete, the feature is nowadays emulated by later Microsoft Windows operating systems to provide backwards compatibility with those programs.

The later eXtended Memory Specification (XMS), also now obsolete, is a standard for, in principle, simulating bank switching for memory above 1 MB (called "extended memory"), which is not directly addressable in the Real Mode of x86 processors in which MS-DOS runs. XMS allows extended memory to be copied anywhere in conventional memory, so the boundaries of the "banks" are not fixed, but in every other way it works like the bank switching of EMS, from the perspective of a program that uses it. Later versions of MS-DOS (starting circa version 5.0) included the EMM386 driver, which simulates EMS memory using XMS, allowing programs to use extended memory even if they were written for EMS. Microsoft Windows emulates XMS also, for those programs that require it.

Alternative and successor techniques

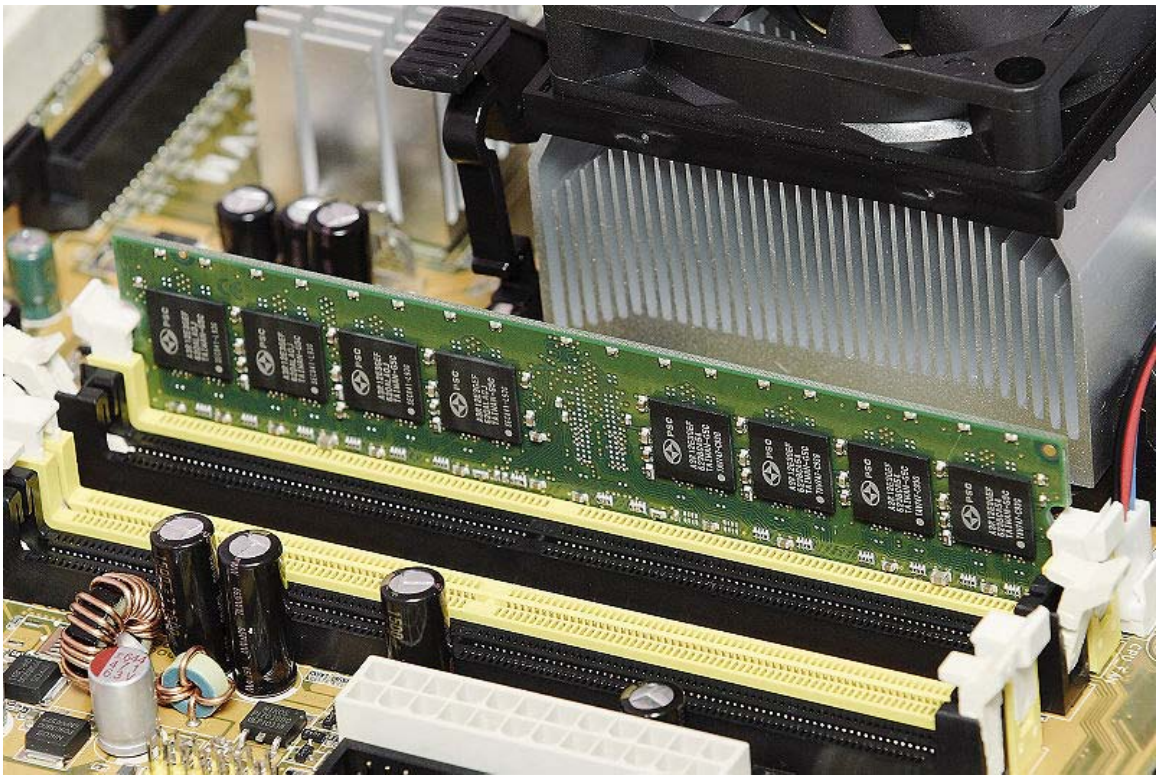
Bank switching was later supplanted by segmentation in many 16-bit systems, which in turn gave way to paging memory management units. In embedded systems, however, bank switching is still often used, for its simplicity, low cost, and often better adaptation to those contexts than to general purpose computing.

Video processing

In some types of computer video displays, the related technique of double buffering may be used to improve video performance. In this case, while the processor is updating the contents of one set of physical memory locations, the video generation hardware is accessing and displaying the contents of a second set. When the processor has completed its update, it can signal to the video display hardware to swap active banks, so that the transition visible on screen is free of artifacts or distortion. In this case, the processor may have access to all the memory at once, but the video display hardware is bank-switched between parts of the video memory. If the two(or more) banks of video memory contain slightly different images, rapidly cycling (page-flipping) between them can create animation or other visual effects that the processor might otherwise be too slow to carry out directly.

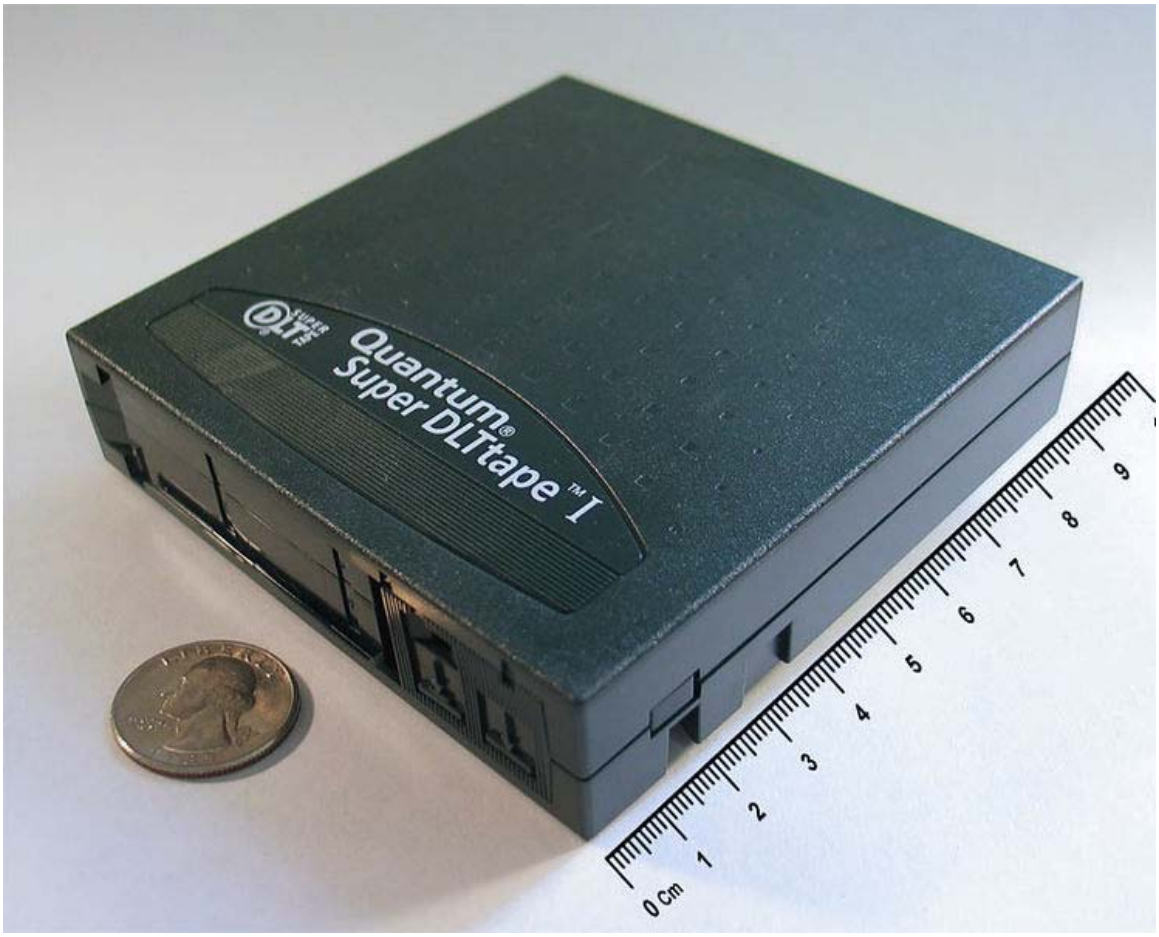
Chapter 19

Computer Data Storage



1 GB of SDRAM mounted in a personal computer. An example of *primary* storage.

40 GB PATA hard disk drive (HDD); when connected to a computer it serves as *secondary* storage.



160 GB SDLT tape cartridge, an example of *off-line* storage. When used within a robotic tape library, it is classified as *tertiary* storage instead.

Computer data storage, often called **storage** or **memory**, refers to computer components and recording media that retain digital data used for computing for some interval of time. Computer data storage provides one of the core functions of the modern computer, that of information retention. It is one of the fundamental components of all modern computers, and coupled with a central processing unit (CPU, a processor), implements the basic computer model used since the 1940s.

In contemporary usage, *memory* usually refers to a form of semiconductor storage known as random-access memory, typically DRAM (Dynamic-RAM) but *memory* can refer to other forms of fast but temporary storage. Similarly, *storage* today more commonly refers to storage devices and their media not directly accessible by the CPU (secondary or tertiary storage) — typically hard disk drives, optical disc drives, and other devices slower than RAM but more permanent. Historically, *memory* has been called *main memory*, *real storage* or *internal memory* while storage devices have been referred to as *secondary storage*, *external memory* or *auxiliary/peripheral storage*.

The contemporary distinctions are helpful, because they are also fundamental to the architecture of computers in general. The distinctions also reflect an important and significant technical difference between memory and mass storage devices, which has been blurred by the historical usage of the term *storage*.

Many different forms of storage, based on various natural phenomena, have been invented. So far, no practical universal storage medium exists, and all forms of storage have some drawbacks. Therefore a computer system usually contains several kinds of storage, each with an individual purpose.

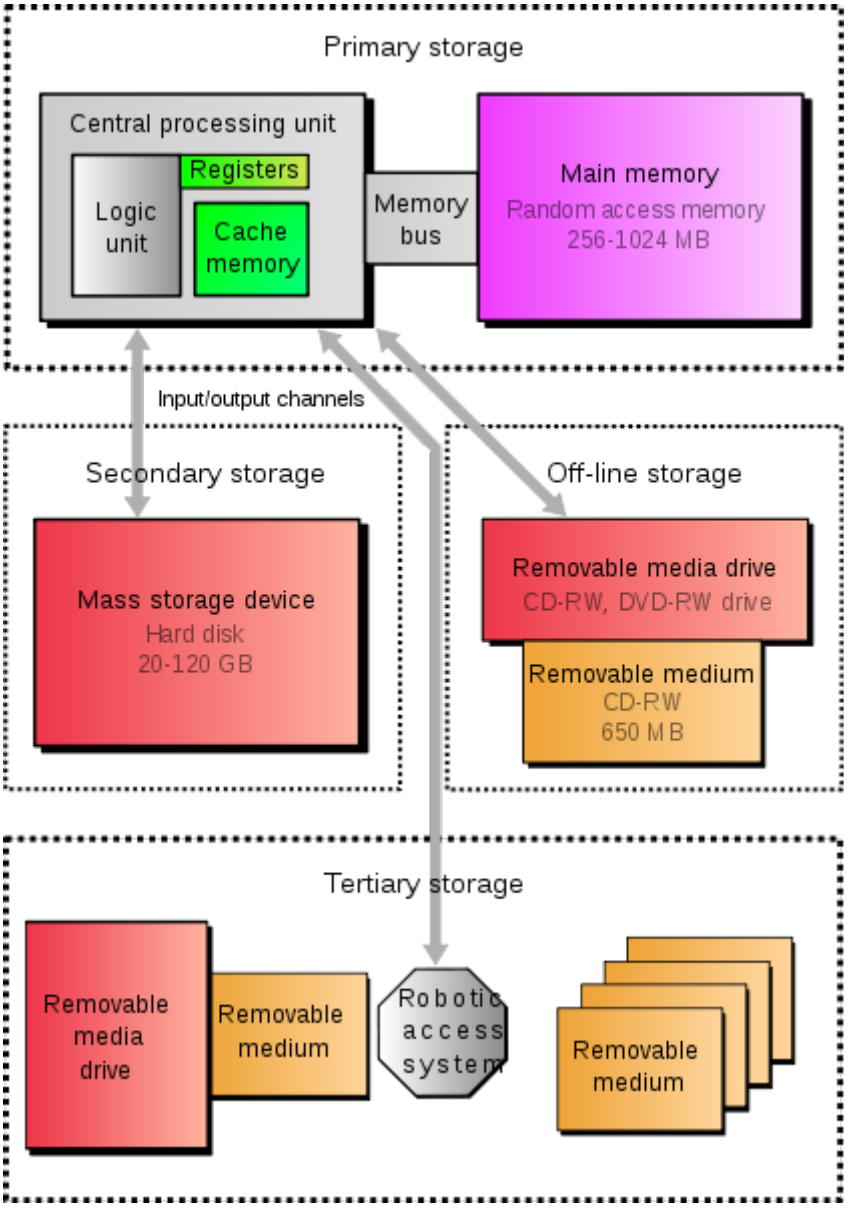
A digital computer represents data using the binary numeral system. Text, numbers, pictures, audio, and nearly any other form of information can be converted into a string of bits, or binary digits, each of which has a value of 1 or 0. The most common unit of storage is the byte, equal to 8 bits. A piece of information can be handled by any computer whose storage space is large enough to accommodate *the binary representation of the piece of information*, or simply data. For example, using eight million bits, or about one megabyte, a typical computer could store a short novel.

Traditionally the most important part of every computer is the central processing unit (CPU, or simply a processor), because it actually operates on data, performs any calculations, and controls all the other components.

Without a significant amount of memory, a computer would merely be able to perform fixed operations and immediately output the result. It would have to be reconfigured to change its behavior. This is acceptable for devices such as desk calculators or simple digital signal processors. Von Neumann machines differ in that they have a memory in which they store their operating instructions and data. Such computers are more versatile in that they do not need to have their hardware reconfigured for each new program, but can simply be reprogrammed with new in-memory instructions; they also tend to be simpler to design, in that a relatively simple processor may keep state between successive computations to build up complex procedural results. Most modern computers are von Neumann machines.

In practice, almost all computers use a variety of memory types, organized in a storage hierarchy around the CPU, as a trade-off between performance and cost. Generally, the lower a storage is in the hierarchy, the lesser its bandwidth and the greater its access latency is from the CPU. This traditional division of storage to primary, secondary, tertiary and off-line storage is also guided by cost per bit.

Hierarchy of storage



Various forms of storage, divided according to their distance from the central processing unit. The fundamental components of a general-purpose computer are arithmetic and logic unit, control circuitry, storage space, and input/output devices. Technology and capacity as in common home computers around 2005.

Primary storage

Primary storage (or **main memory** or **internal memory**), often referred to simply as **memory**, is the only one directly accessible to the CPU. The CPU continuously reads instructions stored there and executes them as required. Any data actively operated on is also stored there in uniform manner.

Historically, early computers used delay lines, Williams tubes, or rotating magnetic drums as primary storage. By 1954, those unreliable methods were mostly replaced by magnetic core memory. Core memory remained dominant until the 1970s, when advances in integrated circuit technology allowed semiconductor memory to become economically competitive.

This led to modern random-access memory (RAM). It is small-sized, light, but quite expensive at the same time. (The particular types of RAM used for primary storage are also volatile, i.e. they lose the information when not powered).

As shown in the diagram, traditionally there are two more sub-layers of the primary storage, besides main large-capacity RAM:

- Processor registers are located inside the processor. Each register typically holds a word of data (often 32 or 64 bits). CPU instructions instruct the arithmetic and logic unit to perform various calculations or other operations on this data (or with the help of it). Registers are the fastest of all forms of computer data storage.
- Processor cache is an intermediate stage between ultra-fast registers and much slower main memory. It's introduced solely to increase performance of the computer. Most actively used information in the main memory is just duplicated in the cache memory, which is faster, but of much lesser capacity. On the other hand it is much slower, but much larger than processor registers. Multi-level hierarchical cache setup is also commonly used—*primary cache* being smallest, fastest and located inside the processor; *secondary cache* being somewhat larger and slower.

Main memory is directly or indirectly connected to the central processing unit via a *memory bus*. It is actually two buses (not on the diagram): an address bus and a data bus. The CPU firstly sends a number through an address bus, a number called memory address, that indicates the desired location of data. Then it reads or writes the data itself using the data bus. Additionally, a memory management unit (MMU) is a small device between CPU and RAM recalculating the actual memory address, for example to provide an abstraction of virtual memory or other tasks.

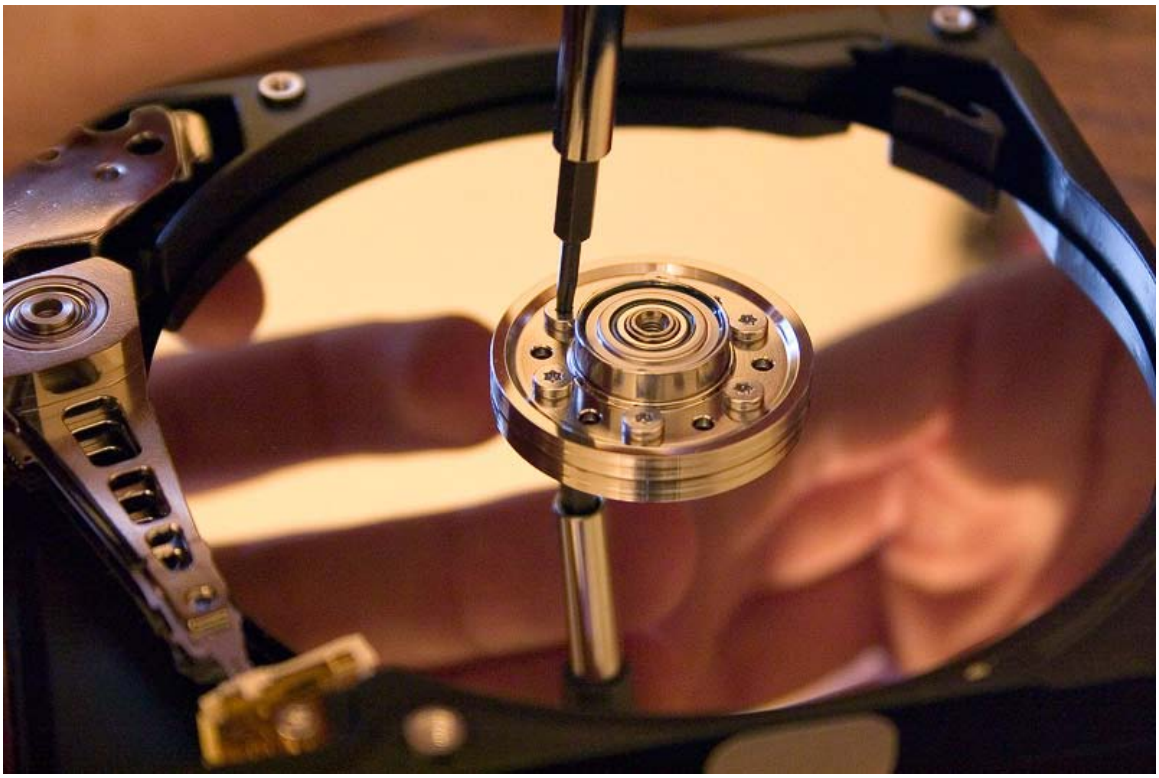
As the RAM types used for primary storage are volatile (cleared at start up), a computer containing only such storage would not have a source to read instructions from, in order to start the computer. Hence, non-volatile primary storage containing a small startup program (BIOS) is used to bootstrap the computer, that is, to read a larger program from non-volatile *secondary* storage to RAM and start to execute it. A non-volatile technology used for this purpose is called ROM, for read-only memory (the terminology may be somewhat confusing as most ROM types are also capable of *random access*).

Many types of "ROM" are not literally *read only*, as updates are possible; however it is slow and memory must be erased in large portions before it can be re-written. Some embedded systems run programs directly from ROM (or similar), because such programs are rarely changed. Standard computers do not store non-rudimentary programs in ROM,

rather use large capacities of secondary storage, which is non-volatile as well, and not as costly.

Recently, *primary storage* and *secondary storage* in some uses refer to what was historically called, respectively, *secondary storage* and *tertiary storage*.

Secondary storage



A hard disk drive with protective cover removed

Secondary storage (also known as external memory or auxiliary storage), differs from primary storage in that it is not directly accessible by the CPU. The computer usually uses its input/output channels to access secondary storage and transfers the desired data using intermediate area in primary storage. Secondary storage does not lose the data when the device is powered down—it is non-volatile. Per unit, it is typically also two orders of magnitude less expensive than primary storage. Consequently, modern computer systems typically have two orders of magnitude more secondary storage than primary storage and data is kept for a longer time there.

In modern computers, hard disk drives are usually used as secondary storage. The time taken to access a given byte of information stored on a hard disk is typically a few thousandths of a second, or milliseconds. By contrast, the time taken to access a given byte of information stored in random access memory is measured in billionths of a second, or nanoseconds. This illustrates the significant access-time difference which distinguishes solid-state memory from rotating magnetic storage devices: hard disks are

typically about a million times slower than memory. Rotating optical storage devices, such as CD and DVD drives, have even longer access times. With disk drives, once the disk read/write head reaches the proper placement and the data of interest rotates under it, subsequent data on the track are very fast to access. As a result, in order to hide the initial seek time and rotational latency, data is transferred to and from disks in large contiguous blocks.

When data reside on disk, block access to hide latency offers a ray of hope in designing efficient external memory algorithms. Sequential or block access on disks is orders of magnitude faster than random access, and many sophisticated paradigms have been developed to design efficient algorithms based upon sequential and block access . Another way to reduce the I/O bottleneck is to use multiple disks in parallel in order to increase the bandwidth between primary and secondary memory.

Some other examples of secondary storage technologies are: flash memory (e.g. USB flash drives or keys), floppy disks, magnetic tape, paper tape, punched cards, standalone RAM disks, and Iomega Zip drives.

The secondary storage is often formatted according to a file system format, which provides the abstraction necessary to organize data into files and directories, providing also additional information (called metadata) describing the owner of a certain file, the access time, the access permissions, and other information.

Most computer operating systems use the concept of virtual memory, allowing utilization of more primary storage capacity than is physically available in the system. As the primary memory fills up, the system moves the least-used chunks (*pages*) to secondary storage devices (to a swap file or page file), retrieving them later when they are needed. As more of these retrievals from slower secondary storage are necessary, the more the overall system performance is degraded.

Tertiary storage

Large tape library. Tape cartridges placed on shelves in the front, robotic arm moving in the back. Visible height of the library is about 180 cm.

Tertiary storage or **tertiary memory**, provides a third level of storage. Typically it involves a robotic mechanism which will *mount* (insert) and *dismount* removable mass storage media into a storage device according to the system's demands; this data is often copied to secondary storage before use. It is primarily used for archival of rarely accessed information since it is much slower than secondary storage (e.g. 5–60 seconds vs. 1-10 milliseconds). This is primarily useful for extraordinarily large data stores, accessed without human operators. Typical examples include tape libraries and optical jukeboxes.

When a computer needs to read information from the tertiary storage, it will first consult a catalog database to determine which tape or disc contains the information. Next, the computer will instruct a robotic arm to fetch the medium and place it in a drive. When the computer has finished reading the information, the robotic arm will return the medium to its place in the library.

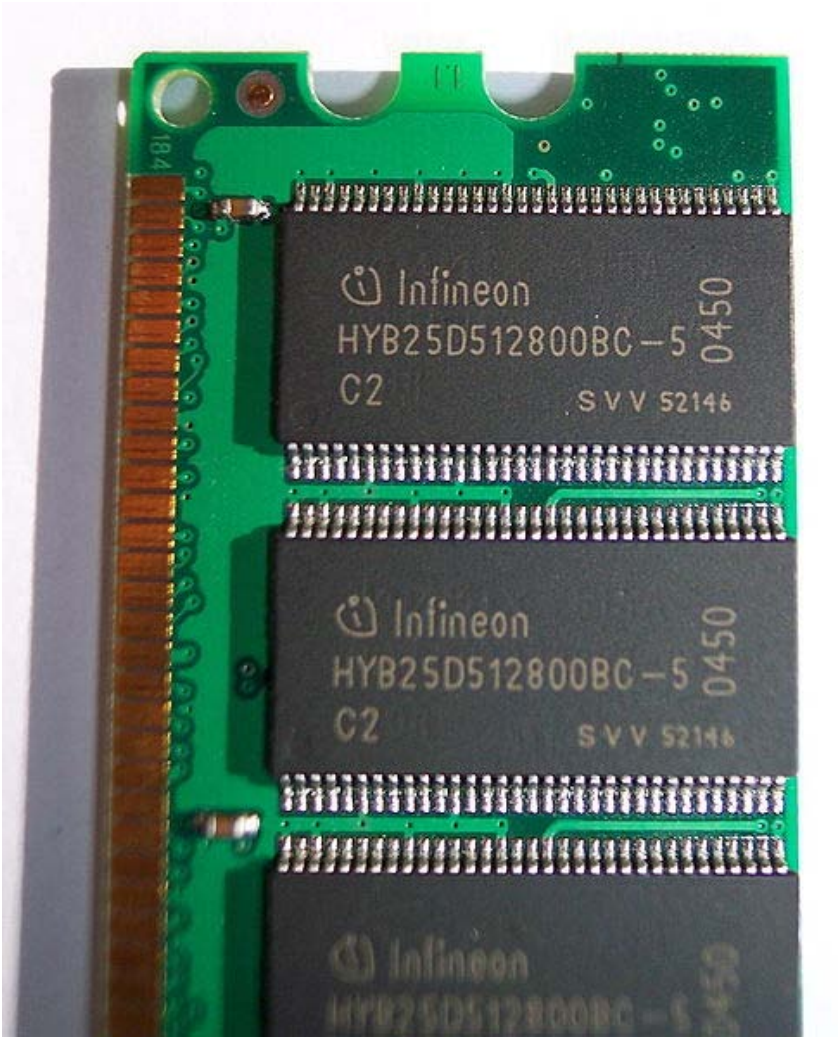
Off-line storage

Off-line storage is a computer data storage on a medium or a device that is not under the control of a processing unit. The medium is recorded, usually in a secondary or tertiary storage device, and then physically removed or disconnected. It must be inserted or connected by a human operator before a computer can access it again. Unlike tertiary storage, it cannot be accessed without human interaction.

Off-line storage is used to transfer information, since the detached medium can be easily physically transported. Additionally, in case a disaster, for example a fire, destroys the original data, a medium in a remote location will probably be unaffected, enabling disaster recovery. Off-line storage increases general information security, since it is physically inaccessible from a computer, and data confidentiality or integrity cannot be affected by computer-based attack techniques. Also, if the information stored for archival purposes is accessed seldom or never, off-line storage is less expensive than tertiary storage.

In modern personal computers, most secondary and tertiary storage media are also used for off-line storage. Optical discs and flash memory devices are most popular, and to much lesser extent removable hard disk drives. In enterprise uses, magnetic tape is predominant. Older examples are floppy disks, Zip disks, or punched cards.

Characteristics of storage



A 1GB DDR RAM memory module (detail)

Storage technologies at all levels of the storage hierarchy can be differentiated by evaluating certain core characteristics as well as measuring characteristics specific to a particular implementation. These core characteristics are volatility, mutability, accessibility, and addressibility. For any particular implementation of any storage technology, the characteristics worth measuring are capacity and performance.

Volatility

Non-volatile memory
: Will retain the stored information even if it is not constantly supplied with electric power. It is suitable for long-term storage of information.

Volatile memory
: Requires constant power to maintain the stored information. The fastest memory technologies of today are volatile ones (not a universal rule). Since primary storage is required to be very fast, it predominantly uses volatile memory.

Differentiation

Dynamic random access memory

A form of volatile memory which also requires the stored information to be periodically re-read and re-written, or refreshed, otherwise it would vanish.

Static memory

A form of volatile memory similar to DRAM with the exception that it never needs to be refreshed as long as power is applied. (It loses its content if power is removed).

Mutability

Read/write storage or mutable storage

Allows information to be overwritten at any time. A computer without some amount of read/write storage for primary storage purposes would be useless for many tasks. Modern computers typically use read/write storage also for secondary storage.

Read only storage

Retains the information stored at the time of manufacture, and **write once storage** (Write Once Read Many) allows the information to be written only once at some point after manufacture. These are called **immutable storage**. Immutable storage is used for tertiary and off-line storage. Examples include CD-ROM and CD-R.

Slow write, fast read storage

Read/write storage which allows information to be overwritten multiple times, but with the write operation being much slower than the read operation. Examples include CD-RW and flash memory.

Accessibility

Random access

Any location in storage can be accessed at any moment in approximately the same amount of time. Such characteristic is well suited for primary and secondary storage.

Sequential access

The accessing of pieces of information will be in a serial order, one after the other; therefore the time to access a particular piece of information depends upon which piece of information was last accessed. Such characteristic is typical of off-line storage.

Addressability

Location-addressable

Each individually accessible unit of information in storage is selected with its numerical memory address. In modern computers, location-addressable storage usually limits to primary storage, accessed internally by computer programs, since location-addressability is very efficient, but burdensome for humans.

File addressable

Information is divided into *files* of variable length, and a particular file is selected with human-readable directory and file names. The underlying device is still location-addressable, but the operating system of a computer provides the file

system abstraction to make the operation more understandable. In modern computers, secondary, tertiary and off-line storage use file systems.

Content-addressable
: Each individually accessible unit of information is selected based on the basis of (part of) the contents stored there. Content-addressable storage can be implemented using software (computer program) or hardware (computer device), with hardware being faster but more expensive option. Hardware content addressable memory is often used in a computer's CPU cache.

Capacity

Raw capacity
: The total amount of stored information that a storage device or medium can hold. It is expressed as a quantity of bits or bytes (e.g. 10.4 megabytes).

Memory storage density
: The compactness of stored information. It is the storage capacity of a medium divided with a unit of length, area or volume (e.g. 1.2 megabytes per square inch).

Performance

Latency
: The time it takes to access a particular location in storage. The relevant unit of measurement is typically nanosecond for primary storage, millisecond for secondary storage, and second for tertiary storage. It may make sense to separate read latency and write latency, and in case of sequential access storage, minimum, maximum and average latency.

Throughput
: The rate at which information can be read from or written to the storage. In computer data storage, throughput is usually expressed in terms of megabytes per second or MB/s, though bit rate may also be used. As with latency, read rate and write rate may need to be differentiated. Also accessing media sequentially, as opposed to randomly, typically yields maximum throughput.

Energy use

- Storage devices that reduce fan usage, automatically shut-down during inactivity, and low power hard drives can reduce energy consumption 90 percent.
- 2.5 inch hard disk drives often consume less power than larger ones. Low capacity solid-state drives have no moving parts and consume less power than hard disks. Also, memory may use more power than hard disks.

Fundamental storage technologies

As of 2008, the most commonly used data storage technologies are semiconductor, magnetic, and optical, while paper still sees some limited usage. Some other fundamental storage technologies have also been used in the past or are proposed for development.

Semiconductor

Semiconductor memory uses semiconductor-based integrated circuits to store information. A semiconductor memory chip may contain millions of tiny transistors or capacitors. Both *volatile* and *non-volatile* forms of semiconductor memory exist. In modern computers, primary storage almost exclusively consists of dynamic volatile semiconductor memory or dynamic random access memory. Since the turn of the century, a type of non-volatile semiconductor memory known as flash memory has steadily gained share as off-line storage for home computers. Non-volatile semiconductor memory is also used for secondary storage in various advanced electronic devices and specialized computers.

Magnetic

Magnetic storage uses different patterns of magnetization on a magnetically coated surface to store information. Magnetic storage is *non-volatile*. The information is accessed using one or more read/write heads which may contain one or more recording transducers. A read/write head only covers a part of the surface so that the head or medium or both must be moved relative to another in order to access data. In modern computers, magnetic storage will take these forms:

- Magnetic disk
  - Floppy disk, used for off-line storage
  - Hard disk drive, used for secondary storage
- Magnetic tape data storage, used for tertiary and off-line storage

In early computers, magnetic storage was also used for primary storage in a form of magnetic drum, or core memory, core rope memory, thin-film memory, twistor memory or bubble memory. Also unlike today, magnetic tape was often used for secondary storage.

Optical

Optical storage, the typical optical disc, stores information in deformities on the surface of a circular disc and reads this information by illuminating the surface with a laser diode and observing the reflection. Optical disc storage is *non-volatile*. The deformities may be permanent (read only media), formed once (write once media) or reversible (recordable or read/write media). The following forms are currently in common use:

- CD, CD-ROM, DVD, BD-ROM: Read only storage, used for mass distribution of digital information (music, video, computer programs)
- CD-R, DVD-R, DVD+R, BD-R: Write once storage, used for tertiary and off-line storage
- CD-RW, DVD-RW, DVD+RW, DVD-RAM, BD-RE: Slow write, fast read storage, used for tertiary and off-line storage

- Ultra Density Optical or UDO is similar in capacity to BD-R or BD-RE and is slow write, fast read storage used for tertiary and off-line storage.

Magneto-optical disc storage is optical disc storage where the magnetic state on a ferromagnetic surface stores information. The information is read optically and written by combining magnetic and optical methods. Magneto-optical disc storage is *non-volatile*, *sequential access*, slow write, fast read storage used for tertiary and off-line storage.

3D optical data storage has also been proposed.

Paper

Paper data storage, typically in the form of paper tape or punched cards, has long been used to store information for automatic processing, particularly before general-purpose computers existed. Information was recorded by punching holes into the paper or cardboard medium and was read mechanically (or later optically) to determine whether a particular location on the medium was solid or contained a hole. A few technologies allow people to make marks on paper that are easily read by machine—these are widely used for tabulating votes and grading standardized tests. Barcodes made it possible for any object that was to be sold or transported to have some computer readable information securely attached to it.

Uncommon

Vacuum tube memory
: A Williams tube used a cathode ray tube, and a Selectron tube used a large vacuum tube to store information. These primary storage devices were short-lived in the market, since Williams tube was unreliable and Selectron tube was expensive.

Electro-acoustic memory
: Delay line memory used sound waves in a substance such as mercury to store information. Delay line memory was dynamic volatile, cycle sequential read/write storage, and was used for primary storage.

Optical tape
: is a medium for optical storage generally consisting of a long and narrow strip of plastic onto which patterns can be written and from which the patterns can be read back. It shares some technologies with cinema film stock and optical discs, but is compatible with neither. The motivation behind developing this technology was the possibility of far greater storage capacities than either magnetic tape or optical discs.

Phase-change memory
: uses different mechanical phases of Phase Change Material to store information in an X-Y addressable matrix, and reads the information by observing the varying electrical resistance of the material. Phase-change memory would be non-volatile, random access read/write storage, and might be used for primary, secondary and off-line storage. Most rewritable and many write once optical disks already use phase change material to store information.

Holographic data storage
: stores information optically inside crystals or photopolymers. Holographic storage can utilize the whole volume of the storage medium, unlike optical disc storage which is limited to a small number of surface layers. Holographic storage would be non-volatile, sequential access, and either write once or read/write storage. It might be used for secondary and off-line storage.

Molecular memory
: stores information in polymer that can store electric charge. Molecular memory might be especially suited for primary storage. The theoretical storage capacity of molecular memory is 10 terabits per square inch.

Related technologies

Network connectivity

A secondary or tertiary storage may connect to a computer utilizing computer networks. This concept does not pertain to the primary storage, which is shared between multiple processors in a much lesser degree.

- **Direct-attached storage** (DAS) is a traditional mass storage, that does not use any network. This is still a most popular approach. This term was coined lately, together with NAS and SAN.
- **Network-attached storage** (NAS) is mass storage attached to a computer which another computer can access at file level over a local area network, a private wide area network, or in the case of online file storage, over the Internet. NAS is commonly associated with the NFS and CIFS/SMB protocols.
- **Storage area network** (SAN) is a specialized network, that provides other computers with storage capacity. The crucial difference between NAS and SAN is the former presents and manages file systems to client computers, whilst the latter provides access at block-addressing (raw) level, leaving it to attaching systems to manage data or file systems within the provided capacity. SAN is commonly associated with Fibre Channel networks.

Robotic storage

Large quantities of individual magnetic tapes, and optical or magneto-optical discs may be stored in robotic tertiary storage devices. In tape storage field they are known as tape libraries, and in optical storage field optical jukeboxes, or optical disk libraries per analogy. Smallest forms of either technology containing just one drive device are referred to as autoloaders or autochangers.

Robotic-access storage devices may have a number of slots, each holding individual media, and usually one or more picking robots that traverse the slots and load media to built-in drives. The arrangement of the slots and picking devices affects performance. Important characteristics of such storage are possible expansion options: adding slots, modules, drives, robots. Tape libraries may have from 10 to more than 100,000 slots, and

provide terabytes or petabytes of near-line information. Optical jukeboxes are somewhat smaller solutions, up to 1,000 slots.

Robotic storage is used for backups, and for high-capacity archives in imaging, medical, and video industries. Hierarchical storage management is a most known archiving strategy of automatically *migrating* long-unused files from fast hard disk storage to libraries or jukeboxes. If the files are needed, they are *retrieved* back to disk.