3. Memory Management for Embedded Systems

Roadmap for Section 3

- Classical Memory Management Approaches
  - Segmentation
  - Paging / Paging
  - Virtual Memory
- Problems of Classical MM-Approaches
- Memory and Real-Time Programming
- Dynamic Memory Allocation
  - Bitmap based, Linked lists, Buddy algorithm
- Memory Management in selected RTOSes
- Memory Types/Access in RT-Systems

Motivation

- One major responsibility of an operating system is memory management
  - Memory allocation - give each tasks memory it needs
  - Memory mapping - map addresses used in tasks to real memory
  - Memory protection - Take appropriate actions when a task uses memory that it has not allocated
- Memory allocation and access has impact on execution time

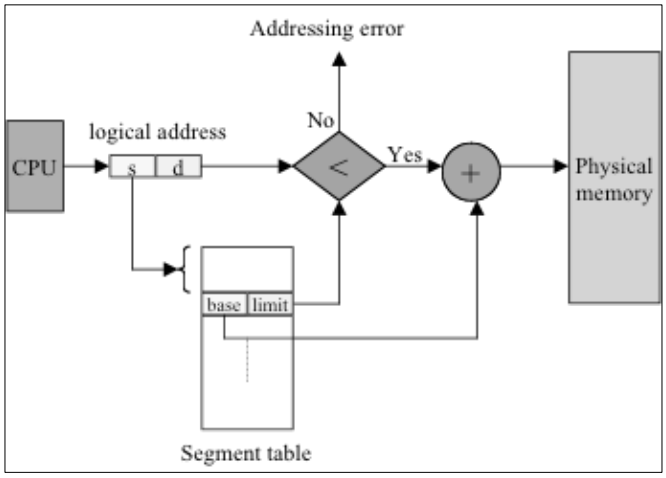
Background

- CPU utilization can be improved by using multiple parallel processes
- Parallel processes provide high abstraction to handle complex problems
- Several processes have to be kept in memory
- Programs are executed by fetching instructions from memory using addresses generated by compilers
- Translation Logical vs. Virtual vs. Physical Addresses

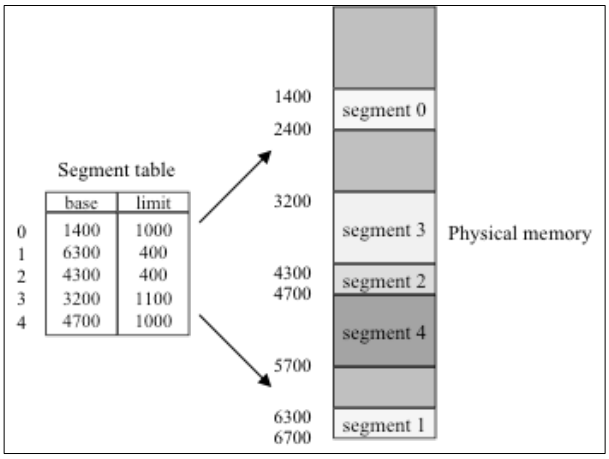
Segmentation

- System memory divided into variable-sized *segments*
- Each segment has name (address) and length
- Mapping off two-dimensional user-defined addresses into one-dimensional physical addresses using *segment table*
- Logical address consist of segment number and offset (two dimensions)
- Segments limited by segment limit in segment table

Segmentation cont.



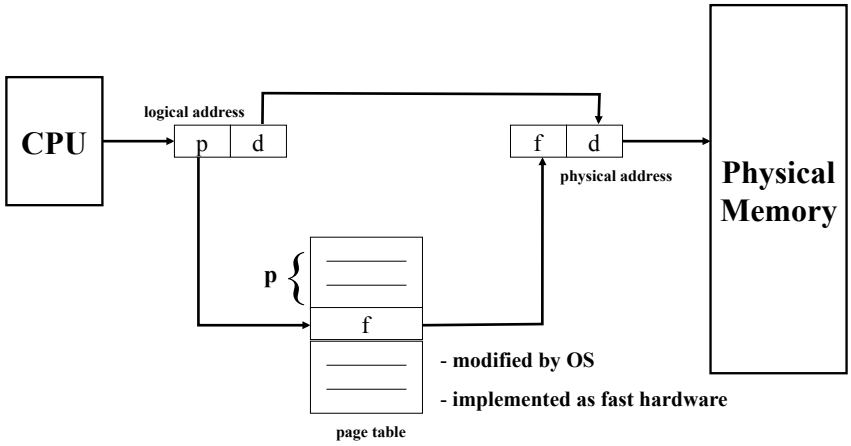
Segment Table



Paging

- Physical memory divided into fixed size blocks called *frames*
- Logical memory divided into blocks of same size – called *pages*
- Users have contiguous memory space
- Permits physical-address space of a process to be noncontiguous
- Memory scattered through physical memory
- Mapping of logical to physical memory kept in *frame table* data structure
- Lot of hardware support available

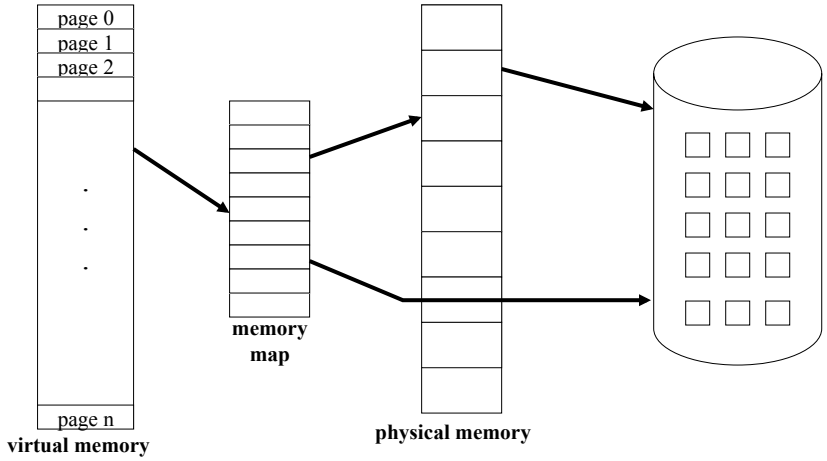
Paging
Paging Hardware



Virtual Memory

- Separates user logical memory from physical memory
- Virtual Memory allows the execution of processes that may not be completely in memory
- Each program has large virtual address space not limited by physical memory
- Implemented using demand paging, segmentation or hybrid techniques

Virtual Memory with Swapping/Paging



Problems of classical approaches used in Embedded Systems

- Not deterministic !
- Most embedded systems miss a lot of hardware support (MMU,TLB)
- No secondary storages available (for swapping)
- Classical approaches require overhead for page-/segmentation tables but we have only small memories
- High-end embedded system use classical approaches, but not for real-time tasks

Real Time with Virtual Memory Memory Locking / Pinning

- Controls demand paging of operating system
- Swapping is non-deterministic and has to be deactivated
- Real-Time POSIX compliant systems provide:
  - *mlockall()* locks all pages of a task
  - *mlock()* locks a specified *preallocated* region of address space
  - *munlock()* unlocks a specified region
  - *mlockall()* unlocks all pages of a process
  - Superuser privileges required
- Windows NT – All pages of a thread can be pinned in memory by specifying a Flag in the CreateThread() system call

Real-Time Programming with Virtual Memory

- Perform non-realtime tasks, such as opening files or allocating memory
- Lock the address space of the process calling mlockall() function
- Perform real time tasks
- Release resources and exit
- Don’t ever increase memory usage!

Real-Time Memory Management

- Fast and deterministic memory management
  - "The fastest and most deterministic approach to memory management is no memory management at all"
- Only an option for very small embedded systems
- At least memory allocation and deletion through system calls supported by most RTOS
- Often allocation and deallocation of memory performed before time critical operation
- (future) Real-Time Systems require predictable memory allocation / deallocation / garbage collection mechanisms
  - Real-Time Java, J2ME ...

Memory Mapping

- POSIX system call mmap()
- Peripheral devices often mapped into address space of memory
- Memory mapping is no real time activity
- Shared memory :
  - used for inter process communication
  - Mapping of identical physical memory into user process address space
  - Typical real-time communication pattern

Memory Allocation

- static allocation
- linked list
- bitmap allocator
- buddy systems
- segregated free lists

Static Memory Allocation

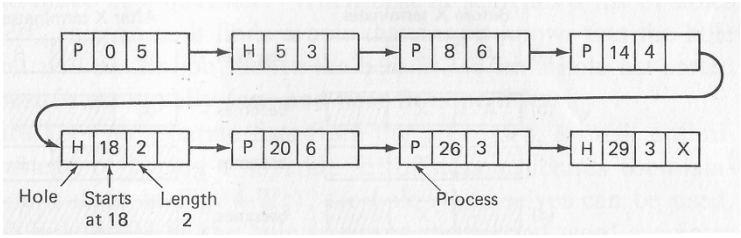
- Segmentation – each tasks gets fixed static region of memory
- No dynamic increase during runtime
- Very predictable, but inflexible
- Number of possible tasks restricted
- Size of all data structures must be known before runtime
- Suitable for deeply embedded systems

Dynamic Memory Allocation in Embedded Systems

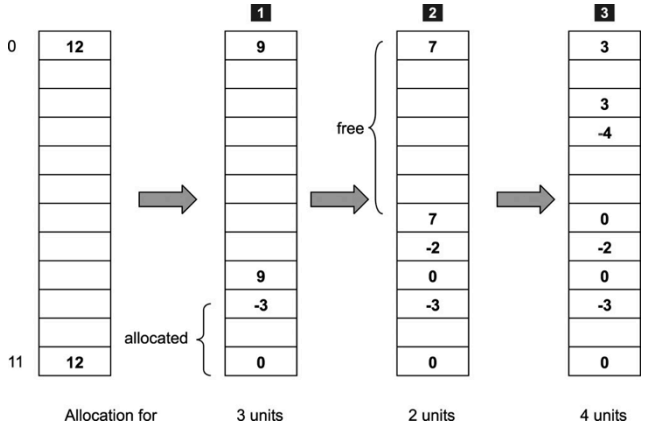
- Task's memory needs change during lifetime
- *Memory allocators* keep track of which parts of memory are used and which are free
- Memory allocated from a global heap memory
- Most RTOS support no timely bounded online allocation of memory
- Predictable memory allocators needed for online allocation

Memory Management with Linked Lists

- Each allocated and free block referenced in a linked list
- Allocating a new block goes through the list and finds a free block (First-fit, Best-fit …)
- Each allocated block has a header containing list pointers and block length

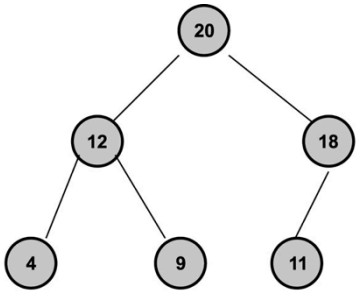


Implementing malloc() using a static allocation array



start address = offset + unit_size*index

Finding Free Blocks Quickly

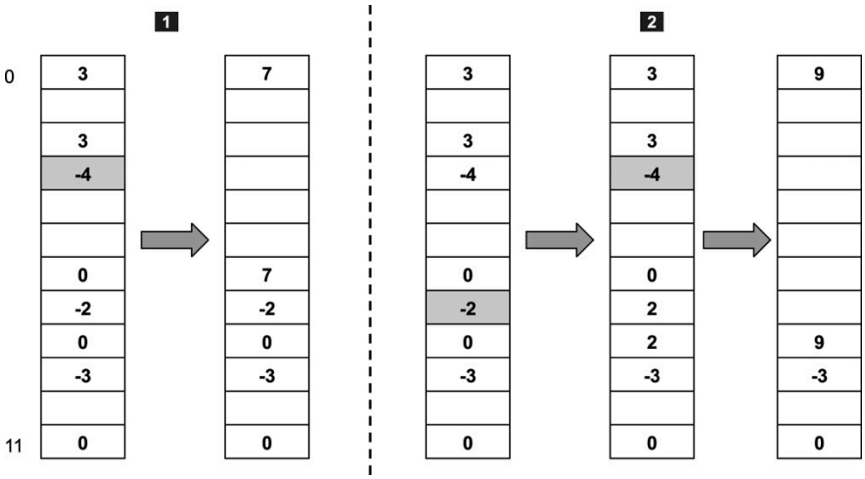


Implementing the Heap using a static array

1	2	3	4	5	6	.....	.....
20	12	18	4	9	11		

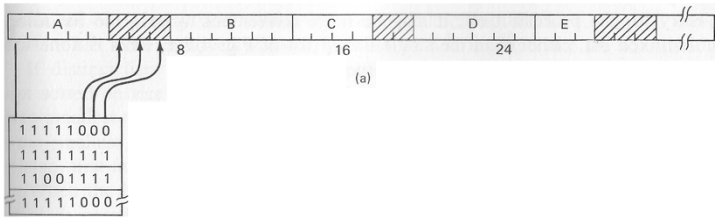
The left child of a node k is at position 2k.
The right child of a node k is at position 2k+1.

Free Operation



Memory Management with Bit Maps

- Memory divided into allocation units
- Each allocation unit corresponds to a bit in a Bitmap
  - 0 if unit is free
  - 1 if unit is occupied

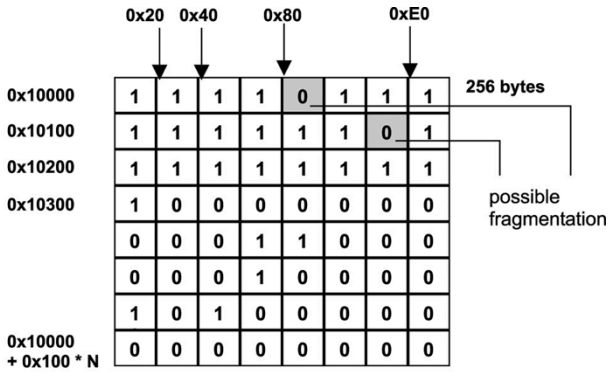


Memory Management with Bitmaps

	0 0 0 0 0 0 0 0	256 bytes	
1	1 1 1 1 0 0 0 0	A = malloc (120)	1 free block = 128 bytes
2	1 1 1 1 1 0 0 0	B = malloc (20)	1 free block = 96 bytes
3	1 1 1 1 1 1 1 0	C = malloc (50)	1 free block = 32 bytes
4	1 1 1 1 1 1 1 1	D = malloc (32)	No free blocks left
5	1 1 1 1 0 1 1 1	free(B)	1 free block = 32 bytes
6	1 1 1 1 0 1 1 0	free(D)	2 free blocks, 32 bytes each
7	1 1 1 1 0 0 0 0	free(C)	1 free blocks = 128 bytes

0 – the block is free
1 – the block is in use

Fragmentation



0 – the block is free
1 – the block is in use

Fragmentation

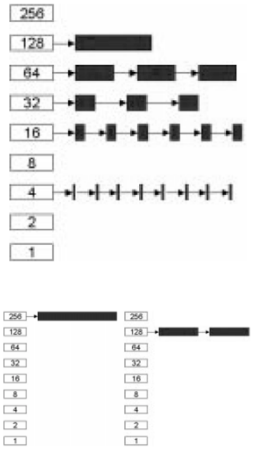
- **Internal Fragmentation** : unused space within a partition (e.g. if there are 32 Byte blocks of memory and 20 Bytes allocated : 12 Bytes are lost
- **External Fragmentation** : memory that is unused and available but too small for requested memory size
- **Memory Compaction** : allocated regions moved and put together to a contiguous free memory area

Dynamic Memory Allocation Buddy Systems

- Knuth 1973, Knowlton 1965
- Each memory request is resolved to a block size of 2^k for some positive, integral value of k
- The buddy algorithm has high fragmentation, but is bounded in time (allocation and deallocation)
- Also called binary allocator / binary buddy
- Relatively high fragmentation (max. 50 % external)
- Add a factor of 1.5 to memory size and internal fragmentation doesn't matter

Buddy System Algorithm

- Translate request of size s into size of 2k-x, k=⌈log2s⌉
- Consult free-list at index k for an available block
- If no block of 2k is available, two blocks can be obtained through bisection of 2k+1
- Recursively apply this strategy to increasingly larger block until a block to bisect is found



Buddy System Allocation

	0	128k	256k		512k		1024k
start	1024k						
A=70K	A	128		256		512	
B=35K	A	B	64	256		512	
C=80K	A	B	64	C	128	512	
A ends	128	B	64	C	128	512	
D=60K	128	B	D	C	128	512	
B ends	128	64	D	C	128	512	
D ends	256			C	128	512	
C ends	512					512	
end	1024k						

Buddy System
Allocation in bounded time

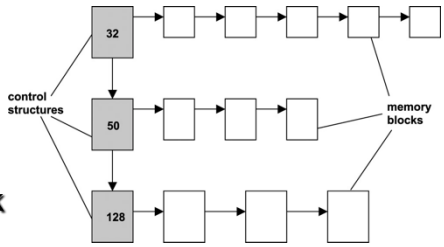
- Steps to allocate a block of size 2^K
  1. Starting at index k, search upwards for an available block (takes log n steps)
  2. Recursively bisect the discovered block unit a block of size 2^K is obtained (takes log n steps)
  3. Return the address of the block
- Size of allocation list (memory) is known a priori

Buddy System Deallocation

- Coalescing of free blocks is a common problem for most memory management algorithms
- Bisected blocks of allocation generates 2 “buddies”
- Buddy’s can easily be computed by change of one bit in the address
- When blocks are returned, buddies are joined in order to create larger blocks

Fixed-size Memory Managent Segregated Free Lists

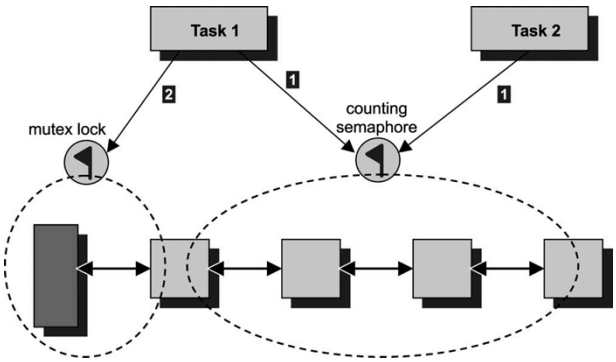
- Fixed-size memory pools
- Embedded networking code, embedded protocol stacks
- Allocated entry removed from memory pool
- Reduced internal fragmen-tation
- Used in predictable environments
- Initially known required block sizes



Blocking vs. Non-Blocking Memory Functions

- malloc() and free() normally not allow the calling task to block and wait for memory to become available
- Some tasks can tolerate allocation delay instead of complicated exception handling in case of allocation failure
- Allocation functions should permit blocking forever, blocking for a timeout period, or no blocking at all

Implementation of a blocking malloc()



Allocation

```
Acquire(Counting_Semaphore)
Lock(mutex)
Retrieve the memory block
from the pool
Unlock(mutex)
```

Deallocation

```
Lock(mutex)
Release the memory block
back to into the pool
Unlock(mutex)
Release(Counting_Semaphore)
```


Real Time Automatic Garbage Collection

- Lots of work around Real Time Java
- Reference counting problematic because deallocation of an object can cause a lot of referenced objects to be deallocated
- Real time implementations of mark-and-sweep available
  - Bounded times for allocation and garbage collection
  - Fragmentation can not be prevented

Literature

- "Dynamic Storage Allocation a Survey and Critical Review", Paul R. Wilson et al., University of Texas
- "Storage Allocation for Real-Time, Embedded System", Steven M. Donahue et al., Washington University
- "Guide to Realtime Programming" http://www.uccs.edu/~compsvcs/doc-cdrom/DOCS/HTML/APS33DTE/TOC.HTM
- "Real-Time and Embedded Guide", Herman Bruynickx, K.U.Leuven, Mechanical Engineering Leuven Belgium

QNX

- Microkernel real time operating system
- Smallest configuration 12 kByte
- Provides full POSIX compliant memory management functions
- Full memory protection if MMU present
- Dynamic memory allocation support, but no real time algorithms implemented
- No virtual memory, paging with 4 Kbyte pages

Windows Ce

- Windows Ce 3.0 350 KByte minimal footprint
- Supports paged virtual memory (requires CPU to support TLB)
- No page file support (read/write to backing store)
- 32 MB usable memory per task – for private data
- XIP – execution-in-place used for dynamic libraries (separate address space outside 32 MB)

Windows CE VirtualAlloc()

- Minimal allocation unit : 1 Page (1024 / 4096 Byte depending on CPU)
- Reserve and commit phase
- Reserved regions are 64 KByte aligned

```
LPVOID VirtualAlloc (LPVOID lpAddress,
                     DWORD dwSize,
                     DWORD flAllocationType,
                     DWORD flProtect);
```

- MEM_COMMIT, MEM_AUTO_COMMIT and MEM_RESERVE

Windows CE – VirtualAlloc() cont.

```
INT i;
PVOID pMem[512];
for (i = 0; i < 512; i++)
{
   pMem[i] = VirtualAlloc (0, PAGE_SIZE, MEM_RESERVE |
                           MEM_COMMIT, PAGE_READWRITE);
}
```

```
INT i;
PVOID pBase, pMem[512];
pBase = VirtualAlloc (0, 512*PAGE_SIZE, MEM_RESERVE,
                      PAGE_READWRITE);
for (i = 0; i < 512; i++)
{
   pMem[i] = VirtualAlloc (pBase + (i * PAGE_SIZE), PAGE_SIZE,
                           MEM_COMMIT, PAGE_READWRITE);
}
```

RTLinux

- Uses standard POSIX memory management functions: mmap(), mlock(), malloc()
- No support for online memory allocation
- No memory protection between threads and the kernel
- “allocate all the memory that each thread will require before the threads are created”

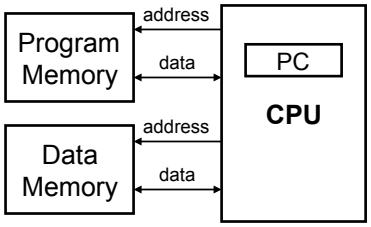
Palm OS

- 32-bit addressing, no virtual memory
- Memory cards 256 MB
- Card 0 stars at $1000000, card 1 starts at $2000000 and so on
- Dynamic RAM / storage RAM
- Dynamic RAM used as one single heap for dynamic memory allocations
- Memory managed in chunks of variable size (1Byte – 64 Kbyte) by a Palm OS memory manager
- Execution-in-place

Memory Access

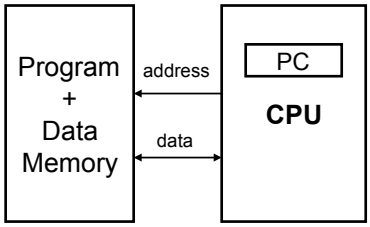
- Harvard / von Neumann architecture
- Types / characteristics of memory
- Synchronous / asynchronous memory interfaces
- Memory functional testing

Harvard vs. von Neumann



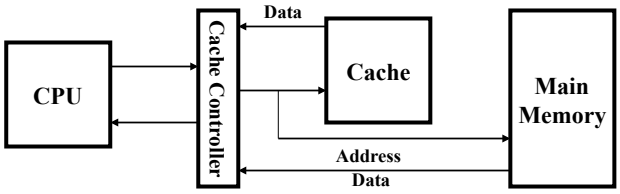
- separate bus for program and data
- parallel acess to program and data
- memory protection

- common data/program bus
- simple programming
- technical simpler

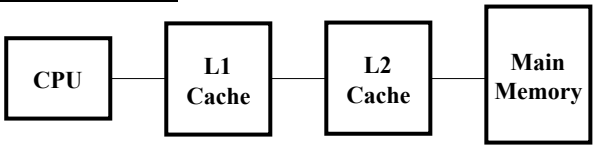


Effects of Caches

Cache in the memory system



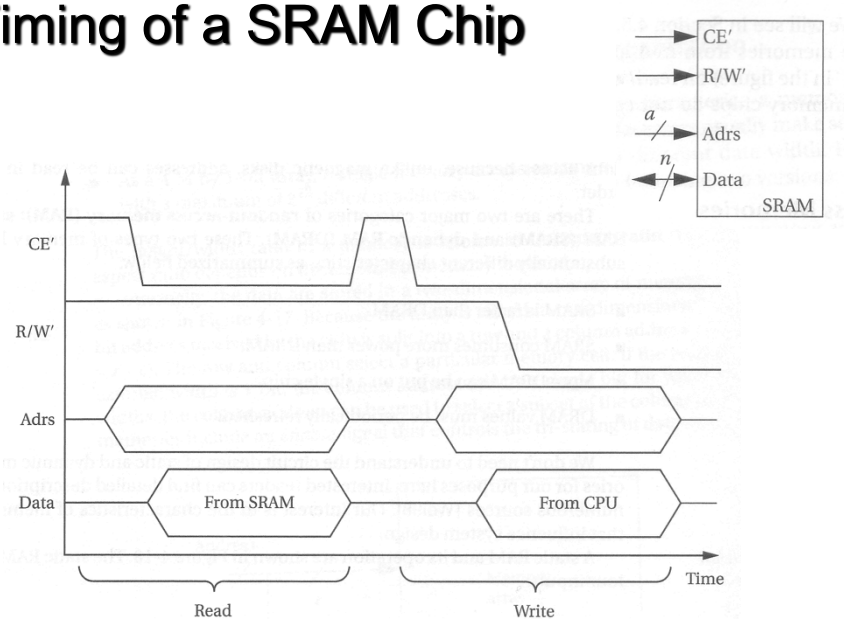
Two-level Cache



Memory Devices
Random Access Memory

- Static RAM (SRAM), dynamic RAM (DRAM)
- SRAM is faster than DRAM
- SRAM consumes more power than DRAM
- DRAM values must be periodically refreshed
  - Charge of capacitors leaks away
  - Typical lifetime : about a millisecond
  - Refreshed influence transfer time to CPU

Timing of a SRAM Chip



Synchronous DRAM

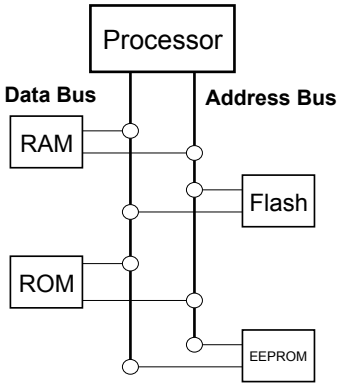
- DRAM/SRAM are asynchronous because they react on asynchronous events from CPU
- Introduction of a clock allows faster internal circuitry
- Refresh cycles integrated into clock frequency

Memory Device Characteristics

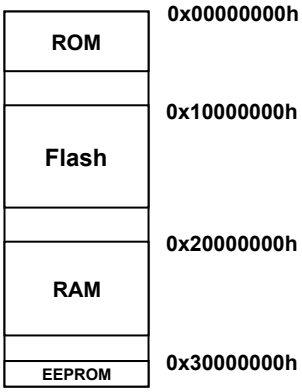
Memory Type	Volatile?	Writeable?	Erase Size	Erase Cycles	Relative Cost	Relative Speed
SRAM	yes	yes	byte	unlimited	expensive	fast
DRAM	yes	yes	byte	unlimited	moderate	moderate
Masked ROM	no	no	n/a	n/a	inexpensive	fast
PROM	no	once, with programmer	n/a	n/a	moderate	fast
EPROM	no	yes, with programmer	entire chip	limited (see specs)	moderate	fast
EEPROM	no	yes	byte	limited (see specs)	expensive	fast to read, slow to write
Flash	no	yes	sector	limited (see specs)	moderate	fast to read, slow to write
NVRAM	no	yes	byte	none	expensive	fast

Mapping Executable Images to Target Systems

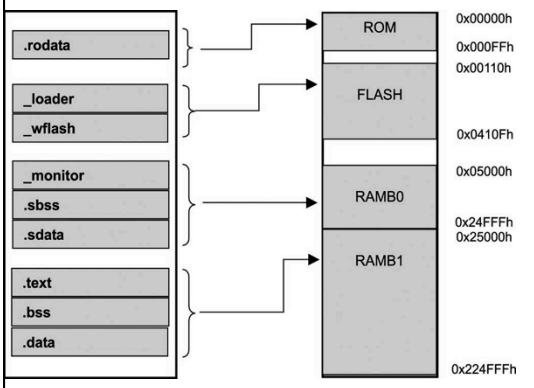
Schematic Target System



Memory Map



Mapping Executables

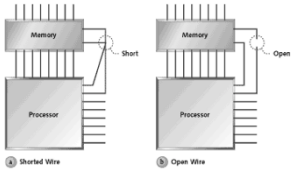



```
MEMORY {
      ROM:    origin = 0x00000h,    length = 0x000100h
      FLASH:  origin = 0x00110h,    length = 0x004000h
      RAMB0:  origin = 0x05000h,    length = 0x020000h
      RAMB1:  origin = 0x25000h,    length = 0x200000h
}
SECTION {
      .rodata  : > ROM
      _loader  : > FLASH
      _wflash  : > FLASH
      _monitor : > RAMB0
      .sbss  (ALIGN 4)  : > RAMB0
      .sdata (ALIGN 4)  : > RAMB0
      .text    : > RAMB1
      .bss  (ALIGN 4) : > RAMB1
      .data (ALIGN 4) : > RAMB1
}
```

Memory Functional Testing

- In order to guarantee stable functioning of embedded devices memory must be checked
- Online Tests integrated into hardware
  - Parity checker, Berger Codes
  - Hamming Codes, Error Correcting Codes
- Offline Memory Tests at system startup
  - Several algorithm available
    - Problem: Bigger memories, More complex faults
  - Test theory based on memory fault models

Memory Faults



- Memory cell faults
  - Stuck-at fault (SAF): cell or line s-a-0 or s-a-1
  - Stuck-open fault (SOF): open cell or broken line
  - Transition fault (TF): cell fails to transit
  - Data retention fault (DRF): cell fails to retain its logic value after some specified time due to, e.g., leakage, resistor opens, or feedback path opens.
  - Coupling fault (CF), Bridging Fault
  - Neighborhood Pattern Sensitive Fault (NPSF)
- Address decoder faults (AFs)
  - Open decoders cells not truly addressed.
  - Multiple writes more than one cell addressed.
  - Cell accessed by more than one address.

March Memory Testing

- March test : set of finite sequences of march elements
- March element : finite sequence of operations applied applied to every cell in a memory array
- Operation : write 0/1 into a cell,

  read expected 0/1
- Many test pattern published
  - Coverage vs. Complexity

1	SCAN [1]	4n	$\{\Uparrow (w0); \Uparrow (r0); \Uparrow (w1); \Uparrow (r0)\}$
2	MATS+ [16]	5n	$\{\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0)\}$
3	MATS++ [6]	6n	$\{\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0, r0)\}$

Size	Complexity			
n	n	$n \log n$	$n^{3/2}$	n^2
1K	0.0001s	0.001s	0.0033s	0.105s
4K	0.0004s	0.0048s	0.0262s	1.7s
16K	0.0016s	0.0224s	0.21s	27s
64K	0.0064s	0.1s	1.678s	7.17m
256K	0.0256s	0.46s	13.4s	1.9h
1M	0.102s	2.04s	1.83m	1.27d
4M	0.41s	9.02s	14.3m	20.39d
16M	1.64s	39.36s	1.9h	326d
64M	6.56s	2.843m	15.25h	14.3y
256M	26.24s	12.25m	5.1d	229y
1G	1.75m	52.48m	40.8d	3659y

Zero-One Test Pattern

```
Procedure ZERO-ONE
{
   1: write 0 in all cells;
   2: read all cells;
   3: write 1 in all cells;
   4: read all cells;
}
```

- The minimal test O(4n).
- Not all TFs are covered; not all CFs are covered.
- SAFs are covered if the address decoder is correct (not all AFs are covered).
- Also known as MSCAN

Checkerboard Pattern

- Writes 1's and 0's into alternate memory locations in a checkerboard pattern. Wait for several seconds and read. Repeat for complementary patterns.

```
Procedure Checkerboard
{
    while(i is odd && j is even)
    {
        write 0 in cell[i]; write 1 in cell[j];
        pause; read all cells;
        complement all cells;
        pause; read all cells;
    }
}
```


Checkerboard

- Time complexity is O(4n).
- For shorts between cells, data retention of SRAMs, SAFs, and half of the TFs.
- The starting point for pattern sensitivity test, but some CFs cannot be detected.
- Not good for AFs.
- Must create true physical checkerboard, not logical checkerboard (the engineer must obtain design information about the actual layout and then modify the test addressing accordingly).

Galloping (ping-pong) pattern (GALPAT)

- The base cell (BC) is read alternately with every other cell in its set

```
Procedure GALPAT
{
    write 0 in all cells;
    2: for BC = 0 to N-1
    {
        complement cell[BC];
        for OC = 1 to n, BC != OC { read BC; read OC;}
        complement cell[BC];
    }
    3: write 1 in all cells;
    4: replay Step 2;
}
```


GALPAT

- $O(4n^2)$, very long sequence (for characterization, not for production tests).
- A strong test for most faults.
- All AFs, TFs, CFs, and SAFs are detected and located.
- Set may be a column, a row, a diagonal, or all cells.

Functional Memory Tests

Test Pattern	AF	SAF	TF	CF	Others	Complexity
Zero-One	N	L	N	N		$4n$
Checkerboard	N	L	N	N	Refresh	$4n$
WALPAT	L	L	L	L	Sense amp. rec.	$2n^2$
GALPAT	L	L	L	L	Write rec.	$4n^2$
Galloping Diagonal	LS	L	L	N		$4n^{1.5}$
Butterfly	L	L	N	N		$5n \log n$
MATS	DS	D	N	N		$4n$
MATS+	D	D	N	N		$5n$
Marching 1/0	D	D	D	N		$14n$
MATS++	D	D	D	N		$6n$
March X	D	D	D	D	Unlinked CFin	$6n$
March C-	D	D	D	D	Unlinked CFin	$10n$
March A	D	D	D	D	Unlinked CF	$15n$
March Y	D	D	D	D	Linked TF	$8n$
March B	D	D	D	D	Linked CF	$17n$
MOVI	D	D	D	D	Read access time	$12n \log n$

N='no'; L='locate'; D='detect'; LS='locate some'; DS='detect some'