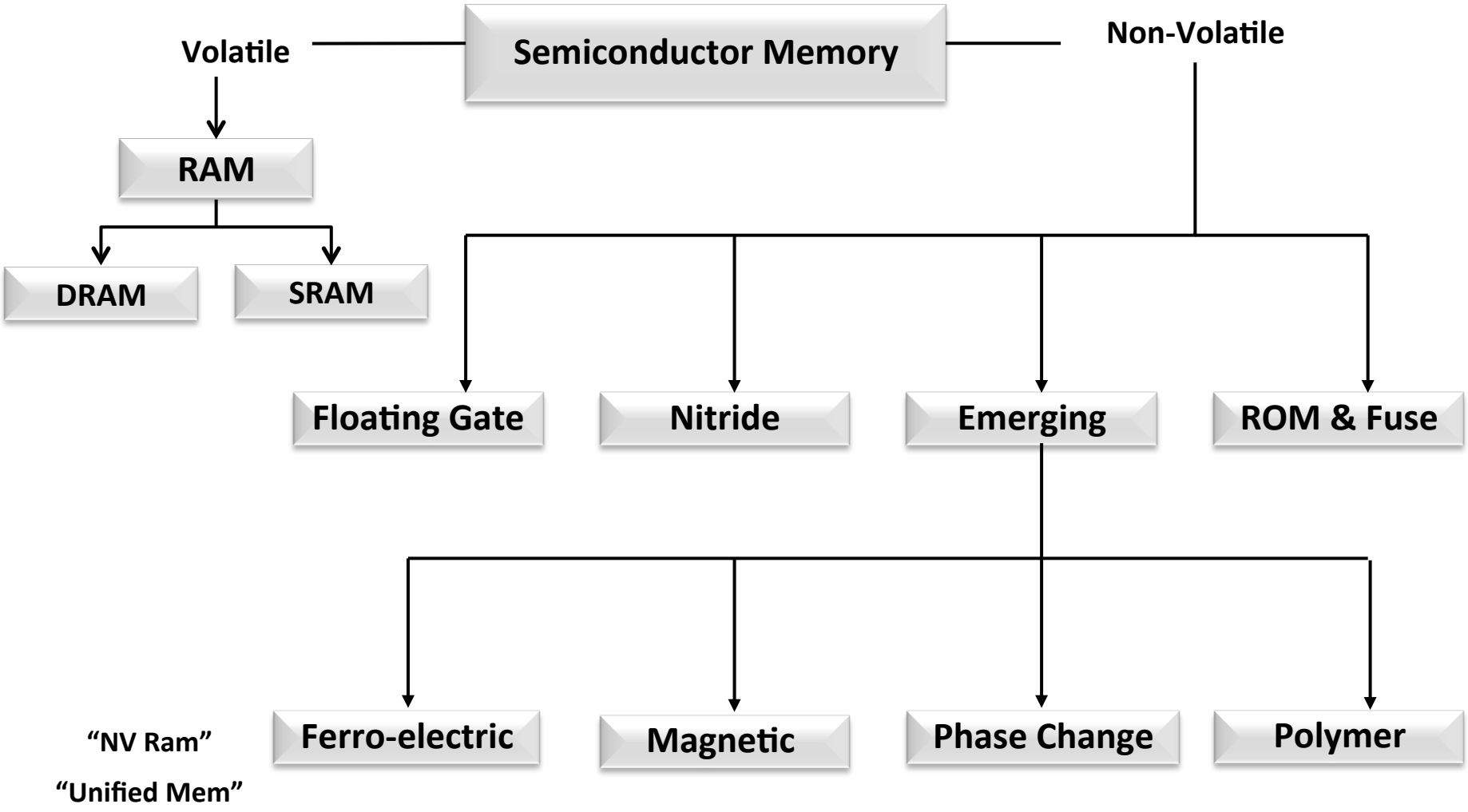
Flash Memory Technology and Flash Based File Systems

Mark McDermott

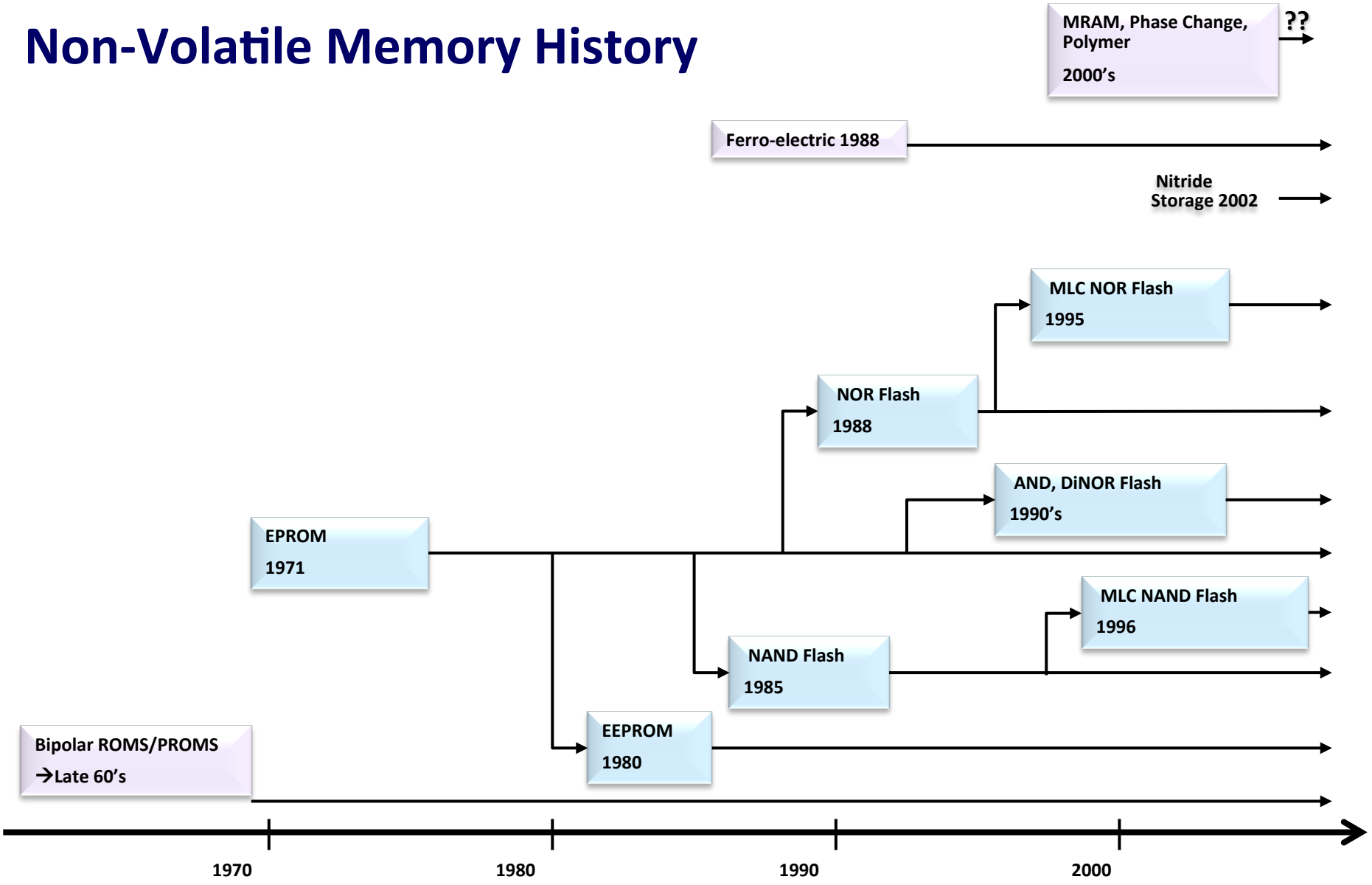
Flash Memory Technology

Semiconductor Memory Types



Courtesy Intel

Non-Volatile Memory History



Courtesy Intel

Non-Volatile Memory Terminology

- Program: Store charge on a floating gate
- Erase: Remove charge from the floating gate
- Data Retention: The longest time for the NVM to keep the stored information
- Endurance: The number of Program/Erase Cycles the memory can withstand
- Disturbs: A mechanism that corrupts the stored data in the memory cell
- Memory Cell: Device that contains the memory storage element
- Memory Array: Arrangement of Memory cells

Courtesy Intel

Flash memory

- **Extension of EEPROM**
  - Same floating gate principle
  - Same write ability and storage permanence
- **Fast erase**
  - Large blocks of memory erased at once, rather than one word at a time
  - Blocks typically several thousand bytes large
- **Flash memory wears out during writing.**
  - Early memories lasted for 10,000 cycles.
  - Modern memories last for 1 million cycles.
- **Two types of flash:**
  - NOR flash operates similar to RAM.
  - NAND is faster, may dominate in future.
- **Writes to single words may be slower**
  - Entire block must be read, word updated, then entire block written back

Technology Comparison

	Production			Research		
	NOR Flash	NAND Flash	Nitride	Phase Change	MRAM	FeRam
Cost Cell Size	$10\lambda^2$	$6\lambda^2$	$6\lambda^2$	$8\text{-}15\lambda^2$	$8\text{-}15\lambda^2$	$10\text{-}20\lambda^2$
Read Characteristics						
Cell Read Latency	10's ns	10's us	10's ns	10's ns	10's ns	10's ns
Cell Read Bandwidth	100's cells	1000's cells	100's cells	10's-100's cells	100's cells	100's cells
Write Characteristics						
Cell Write Time	100's ns	100's us	100's ns	10's ns	10's ns	10's ns
Cell Write Bandwidth	10's cells	1000's cells	10's cells	10's cells	10's cells	10's cells

λ represents minimum feature size for any technology

Feature size=process lithography capability

Example: 0.12u lithography, $10\lambda^2$ cell size yields a cell area of $0.144u^2$

Courtesy Intel

Memory Access Capabilities

	SDRAM (external)	Internal-SRAM	NOR-FLASH	NAND-FLASH	ROM
Non-volatile	No	No	Yes	Yes	Yes
Does NOT need initialization	No	Yes	Yes	Yes	Yes
Random Read	Yes	Yes	Yes	No	Yes
Random write	Yes	Yes	No	No	- - -

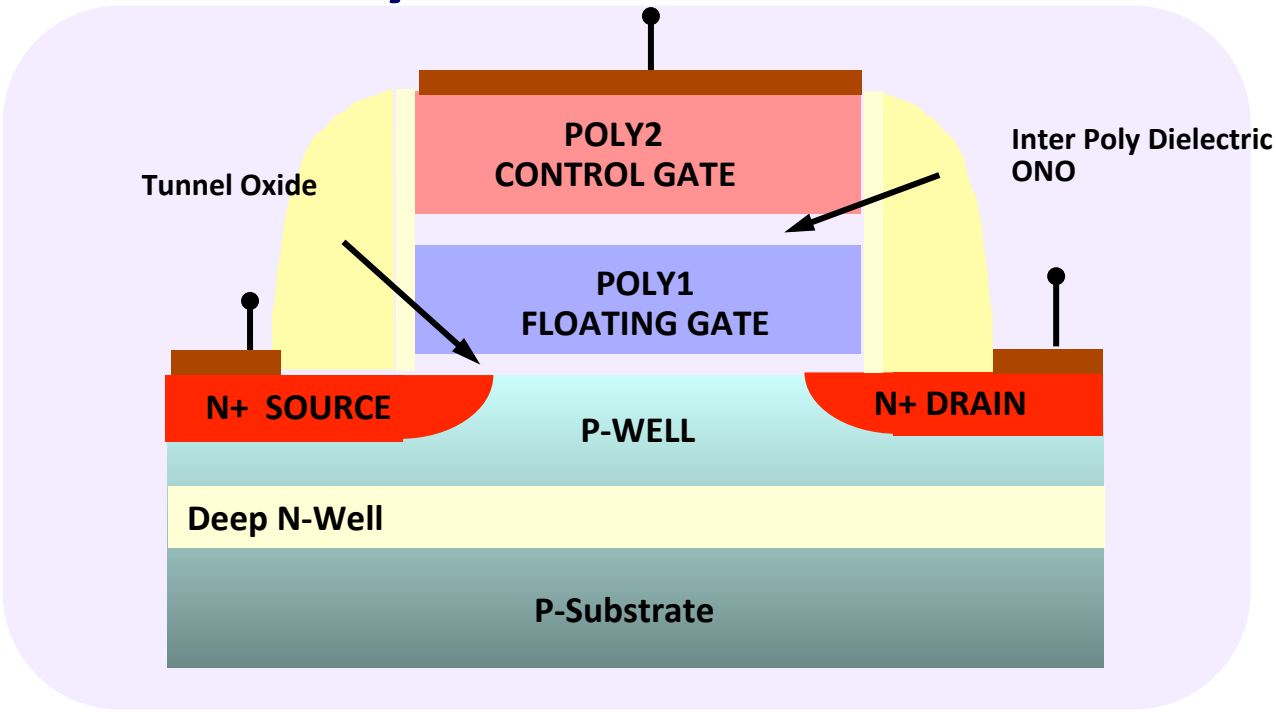
NOR vs. NAND Specifications

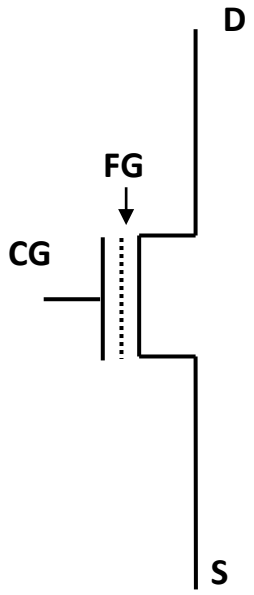
	READ		PROGRAM		ERASE
	Random	Throughput	Random	Throughput	Throughput
NOR	98 ns	266 MB/s (x16)	500 us (512 bytes)	1 MB/s	0.128 MB/s (1 s per block)
NAND	25 us (1^{st} byte)	37 MB/s (x16)	300 us (2048 bytes)	5 MB/s	64 MB/s (2 ms per block)

Block Size = 128KB

Courtesy Intel

Flash Memory Device – NAND & NOR

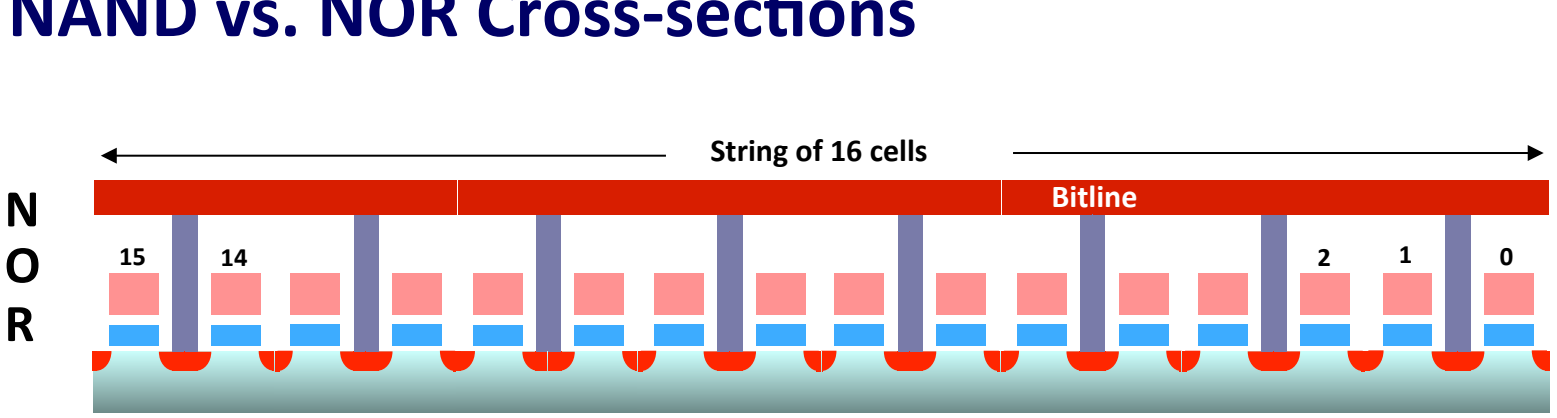


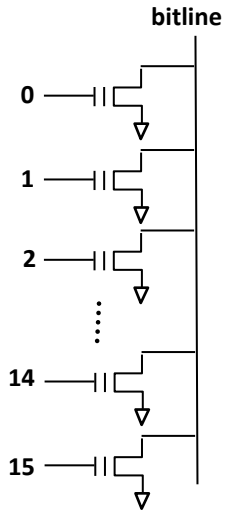


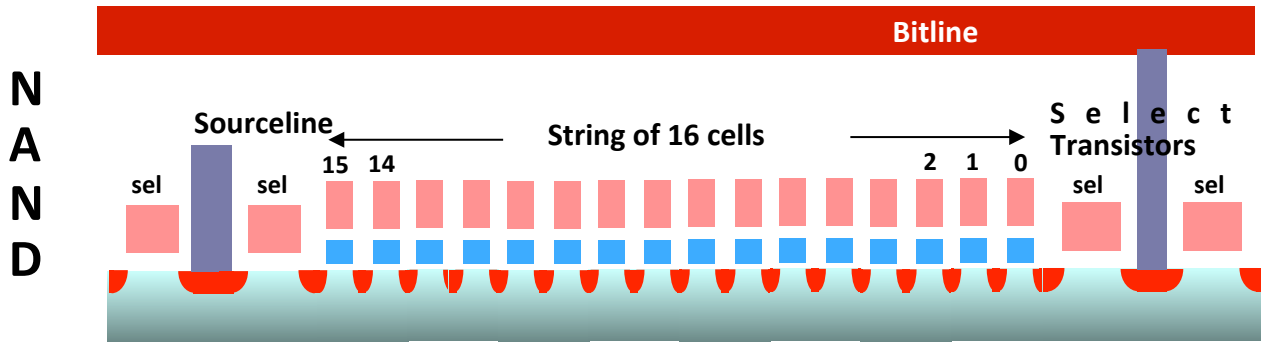
- **Stacked Gate NMOS Transistor**
  - Poly1 Floating Gate for charge storage
  - Poly2 Control Gate for accessing the transistor
  - Tunnel-oxide for Gate oxide
  - Oxide-Nitride-Oxide (ONO) for the inter Poly Dielectric
  - Source/Drain Junctions optimized for Program/Erase/Leakage

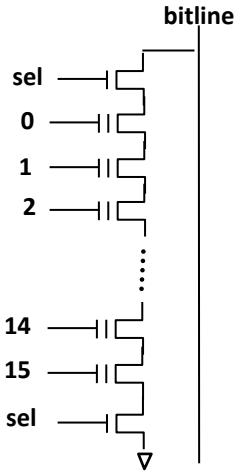
Courtesy Intel

NAND vs. NOR Cross-sections







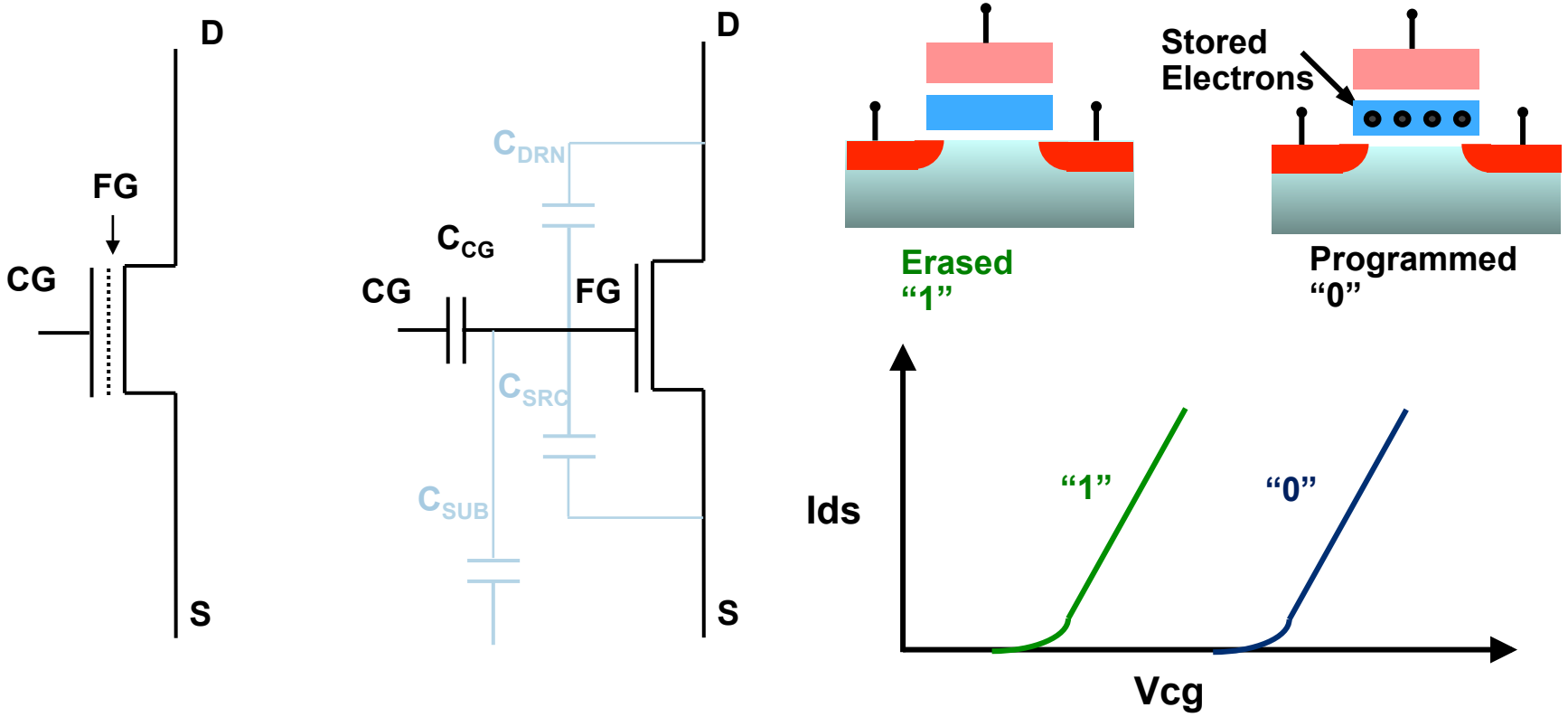


Two of the Most Popular Flash Memory Types

- Both have Dual Gate NMOS with charge storage in Poly1 floating gate
- Lack of contacts in NAND cell makes it inherently smaller in size

Courtesy Intel

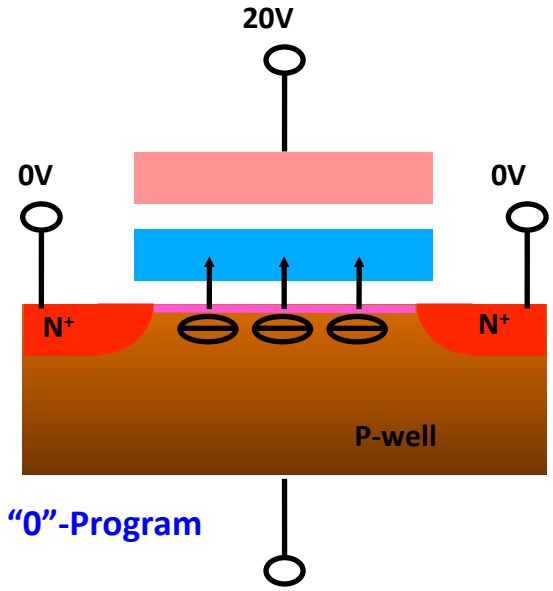
Flash Memory Device – Basic Operation

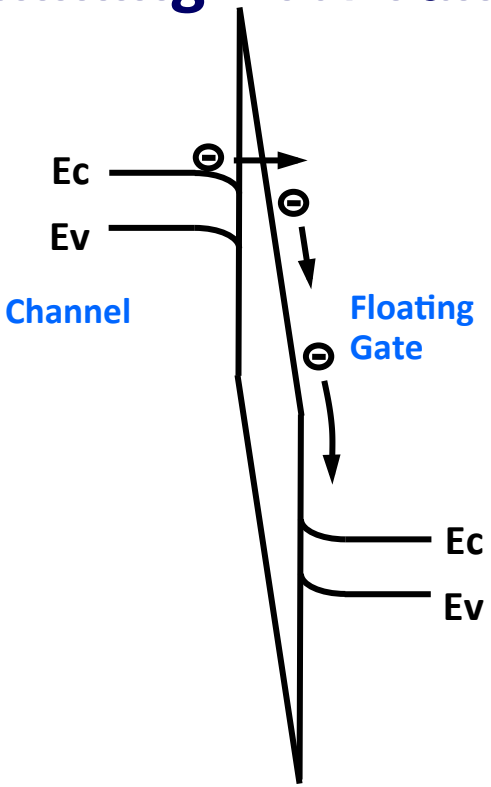


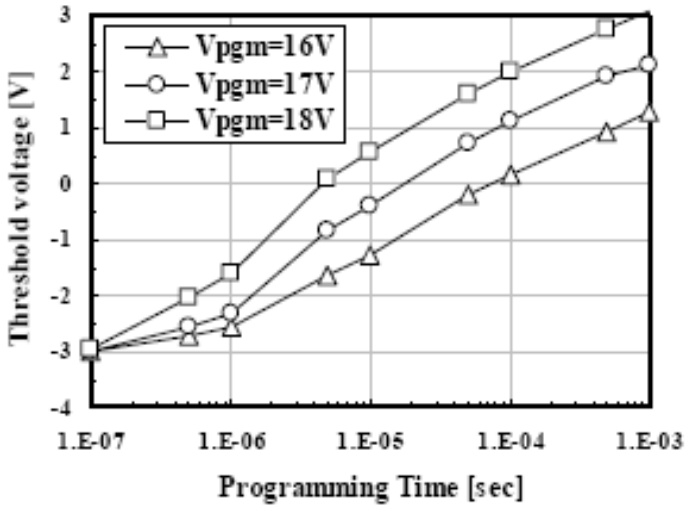
- Programming = Electrons Stored on the FG = High Vt
- Erasing = Remove electrons from the FG = Low Vt
- Threshold Voltage shift = $\Delta Q_{FG}/C_{CG}$

Courtesy Intel

NAND Flash Programming – FN Tunneling





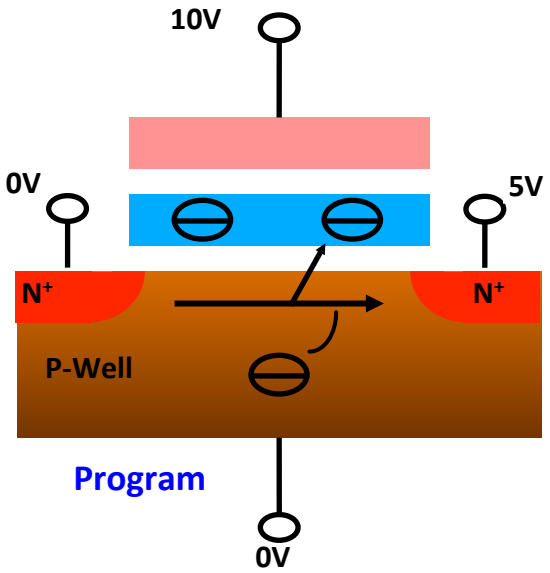


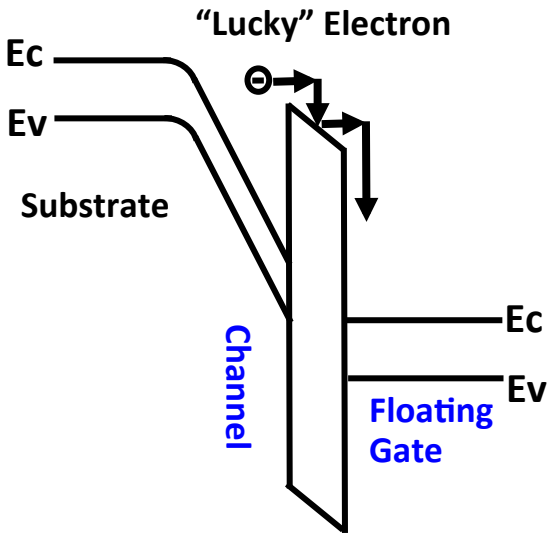
Y. S. Yim, et al. IEDM 2003

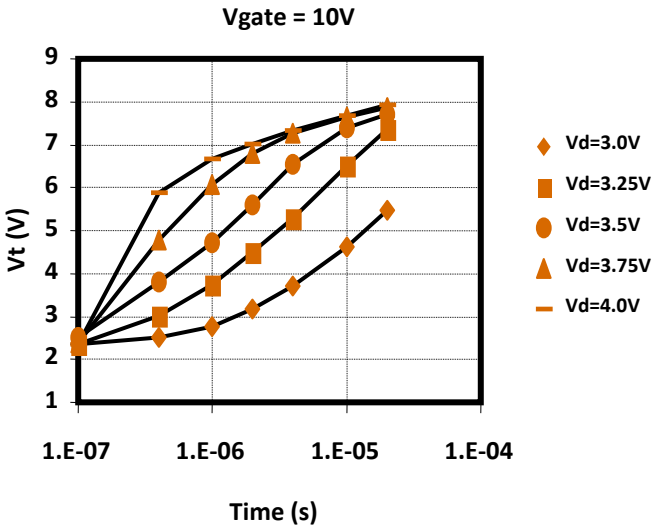
- Tunnel Programming from channel by biasing the Top Gate positive with respect to the ground
- Program Time ~300us
- Program current ~ Displacement plus Tunneling current. Low current allows large parallelism.

Courtesy Intel

NOR Flash Programming – Channel Hot Electron



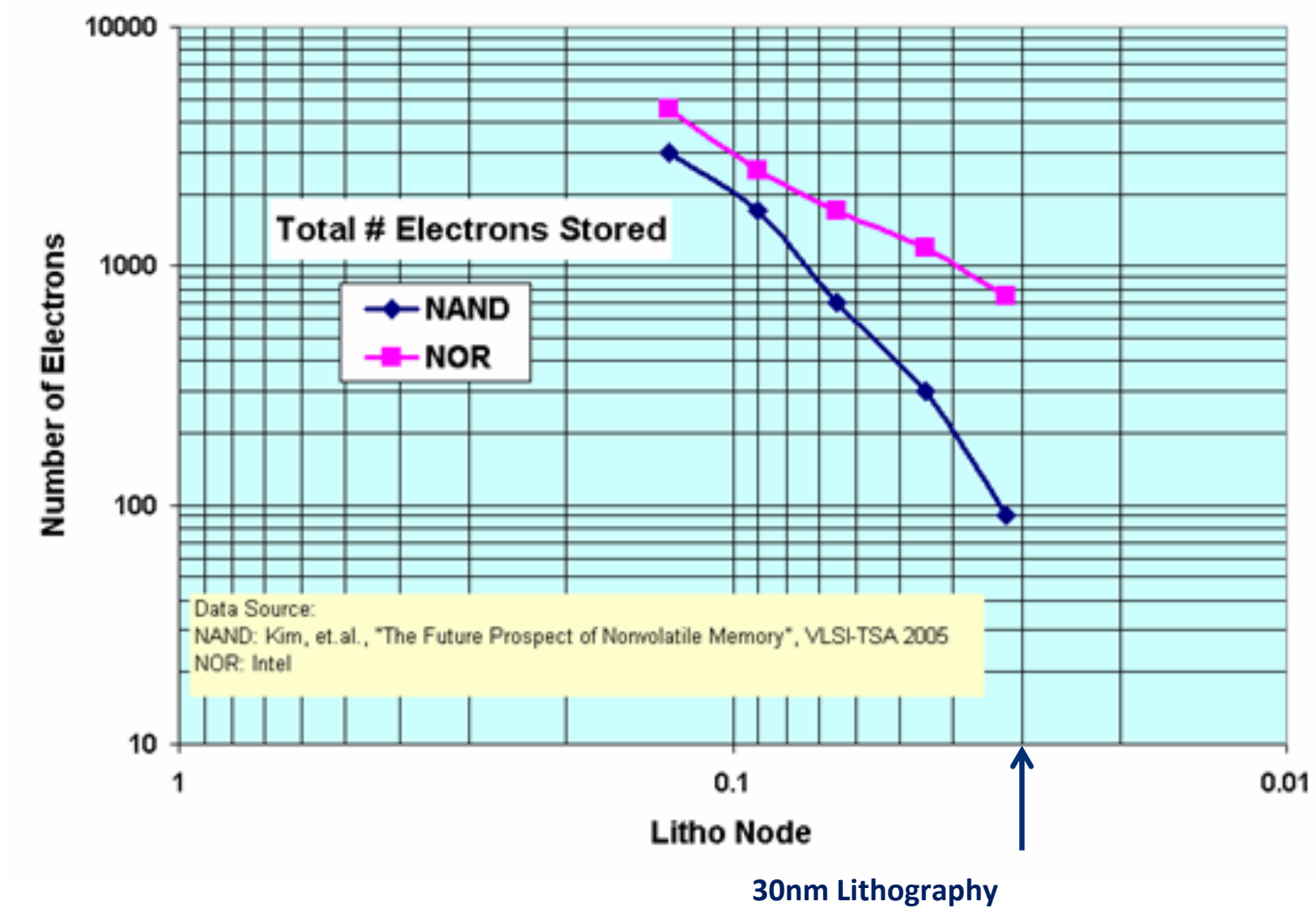




- Channel Hot Electron Programming - Gate voltage inverts channel; drain voltage accelerates electrons towards drain; gate voltage pulls them to the floating gate
- In Lucky Electron Model, electron crosses channel without collision, gaining > 3.2eV, hits Si atom, bounces over barrier
- Program Time ~ 0.5-1ms. Program current ~50mA/cell

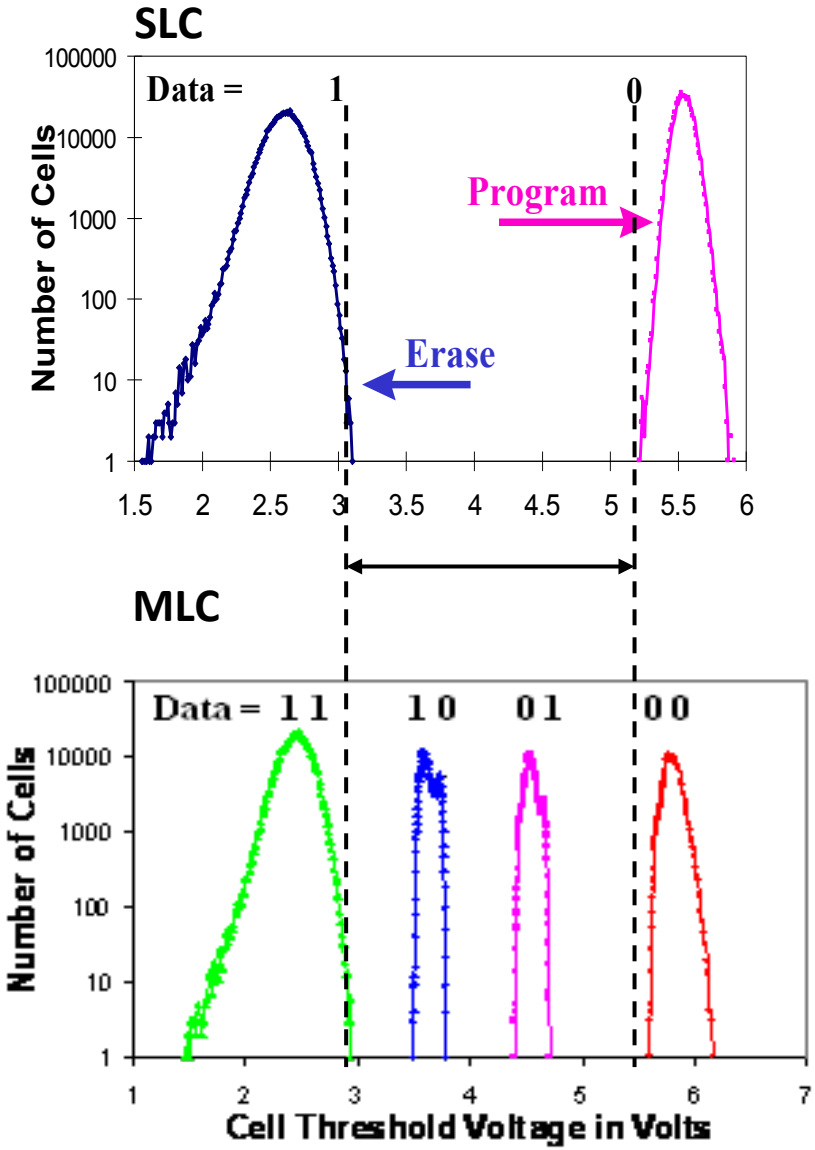
Courtesy Intel

Floating Gate Electrons vs. Lithography

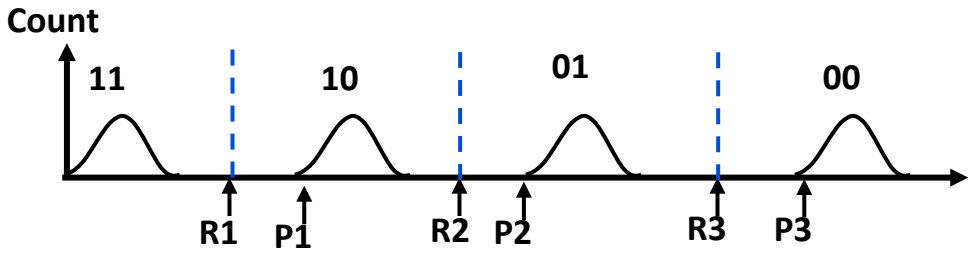


Courtesy Intel

Single-Level Cell (SLC) vs. Multi-Level Cell (MLC)



- Take advantage of the threshold voltage difference between the erased and programmed states of the single-level-cell case
  - Two levels = 1 bit/cell
  - Four levels = 2 bits/cell
  - In general: n bits/cell = $\log_2$(#levels)
- Need additional reference cells for program / read
  - One read reference cell for 1 bit/cell
  - Three read reference cells for 2bits/cell
  - N-1 reference cells for n bits/cell
  - Corresponding reference cells for program



Courtesy Intel

Multi-Level Cell Design

- **Why do MLC? Cost**
  - Effectively cuts cell area per bit in half
  - Provides the same cost improvement from an array area perspective as a litho generation
- **Three Key MLC Considerations**
  - Precise Charge Placement (Programming)
    - Cell programming must be accurately controlled, which requires a detailed understanding of cell physics, voltage control and timing
    - Precision voltage generation for stable wordline and drain voltage
  - Precise Charge Sensing (Read)
    - MLC read operation is an analog to digital conversion of the charge stored in the cell
    - Device and capacitance matching, Collapsing sources of variation, Precision wordline and drain voltage generation, Low current sensing
  - Stable Charge Storage
    - Leakage rate needs to be less than one electron per day

Courtesy Intel

NVM Forecast

- **Floating Gate NVM**
  - More than 95% of semiconductor Non-Volatile memories shipped today
  - Shipping 8th generation of flash memory in high volume
  - Main NVM technology for the next 5-10 years
  - Current projections show scaling continues at 45nm
- **Emerging NVM**
  - Storage devices with new material types are being explored vigorously throughout the industry
    - Delta R type: (Phase change, MRAM)
    - Delta C type: (FeRAM)
  - New materials integrated into standard CMOS is challenging
  - High volume manufacturing has not yet been established on most emerging technologies
    - Technical challenges yet to be uncovered
    - Quality, reliability and failure mechanisms need to be quantified
    - Testers, and test methods

Courtesy Intel

Flash Based File Systems for Embedded Linux

Usage of Flash for Linux File Systems

- NAND and NOR flash were developed around the same time (NOR by Intel and NAND by Toshiba in the late 1980s), NOR caught up fast with embedded world because of its ease to use.
  - When embedded system evolved to have large storage (like media players and digital cameras), NAND flash became popular for data applications.
  - MTD (Memory Technology Device) layer also evolved initially around NOR but the support for NAND was added later.
- NOR chips come in two flavors: old non-CFI chips and newer CFI (Common Flash Interface), industry standard for ensuring compatibility among flash chips coming from the same vendor.
  - CFI was introduced that enable flash vendors to allow configuration data to be read from flash drivers.
- System software could interrogate flash devices and reconfigure itself
  - MTD supports CFI command sets from Intel and AMD
- NAND flash support was added in Linux 2.4 kernel. Along with NFTL (NAND Flash Translation Layer) it can be mounted as a regular file system
- Support for JFFS2 was added in the 2.6 kernel.

Derived from Building Embedded Systems

Flash Mapping

- On an embedded Linux system, flash is used for:
  - Storing boot-loader
  - Storing OS image
  - Storing application and application library images
  - Storing read-write files (having configuration data)
- The first three are generally read-only for most embedded systems
- Typical Flash map for a 4MB flash:
  - Boot-loader
  - OS images
  - Read only data in a CRAMFS
  - Read write data in a JFFS2

Derived from Building Embedded Systems

Flash Mapping Issues

- How do you partition the flash?
  - Do you have the OS, application and read-write in a single partition?
    - This increases the risk of corrupting entire system data
  - Do you put read-only and read-write into separate partitions?
    - read-only data is safe from any corruptions, but its partition size is fixed.
- How do you access the partition? Raw or as a file system?
  - Raw partitions are useful for boot loader
  - For partitions holding Linux data, it is safer to go via file systems
- How do you upgrade?
  - If upgrades involve changing only read-only data, it is better to partition into read-only and read-write partitions so that you will not have to do any backup and restore of read-write data

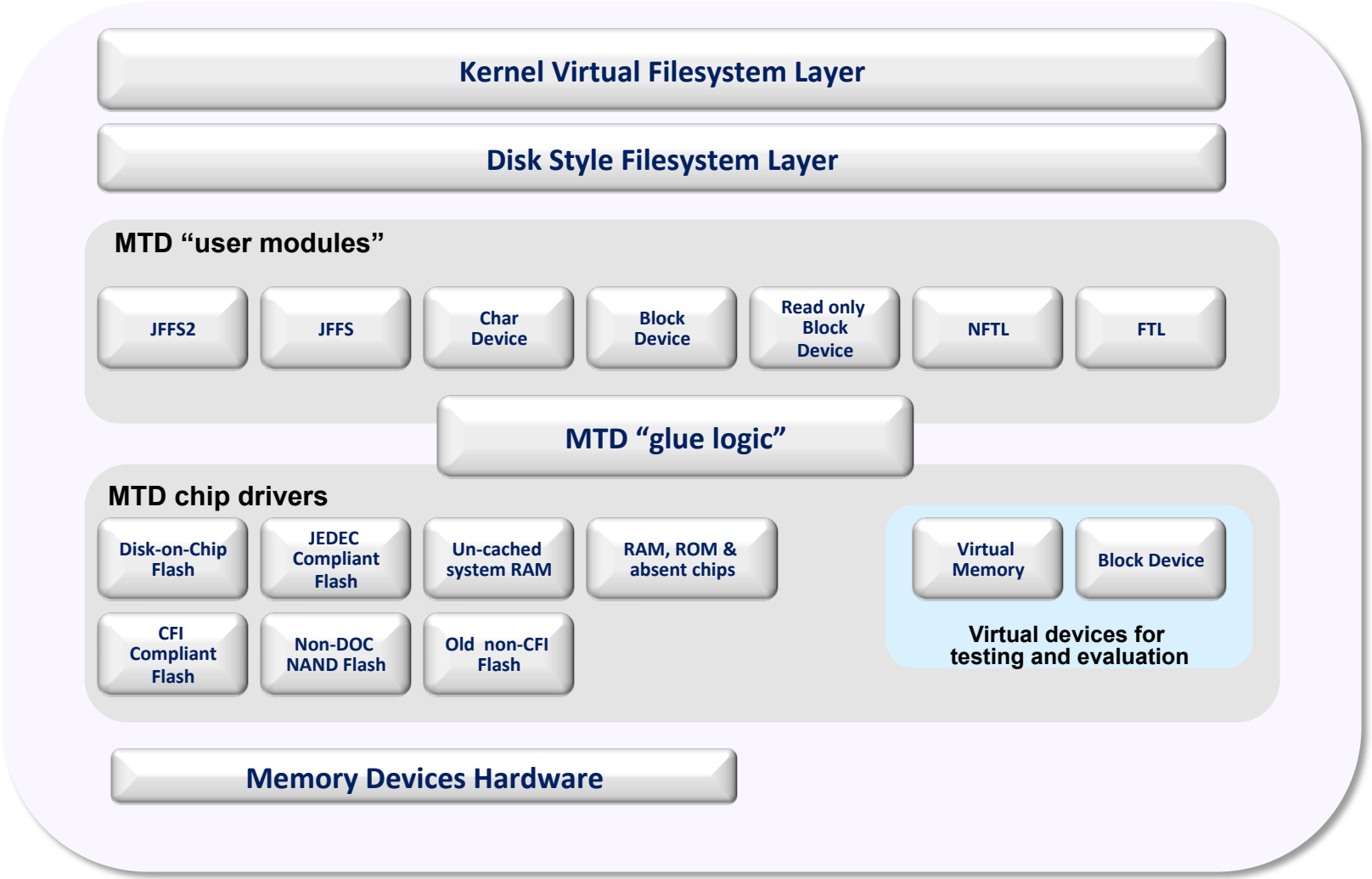
Derived from Building Embedded Systems

Memory Technology Devices (MTD)

- In Linux terminology, memory technology devices (MTDs) include all memory devices, such as conventional ROM, RAM, FLASH, and M-Systems' DiskOnChip (DOC)
- To program and use an MTD device in their systems, embedded system developers traditionally use tools and methods specific to that type of device.
  - The Linux kernel includes the MTD subsystem. This provides a unified and uniform layer that enables a seamless combination of low-level MTD chip drivers with higher-level interfaces called user modules.
  - These "user modules" should not be confused with kernel modules or any sort of user-land software abstraction. The term "MTD user module" refers to software modules within the kernel that enable access to the low-level MTD chip drivers by providing recognizable interfaces and abstractions to the higher levels of the kernel or, in some cases, to user space.

Derived from Building Embedded Systems

MTD System Block Diagram



Derived from Building Embedded Systems

MTD architectural block definitions

- **DiskOnChip (DOC)**
  - These are the drivers for M-Systems' DOC technology. Currently, Linux supports the DOC 1000, DOC 2000, and DOC Millennium
- **Common Flash Interface (CFI)**
  - CFI is a specification developed by Intel, AMD, and other flash manufacturers. All CFI-compliant flash components have their configuration and parameters stored directly on the chip. Hence, the software interfaces for their detection, configuration, and use are standardized. The kernel includes code to detect and support CFI chips.
- **JEDEC**
  - The JEDEC flash standard has been rendered obsolete by CFI, although some flash chips still feature JEDEC compliance. The MTD subsystem supports the probing and configuration of such devices.
- **RAM, ROM & Absent Chips**
  - The MTD subsystem provides drivers to access conventional RAM and ROM chips as MTD devices. Since some of these chips may be connected to the system using a socket or some similar connector that lets you remove the chip, the MTD subsystem also provides a driver that can be used to preserve the registration order of the MTD device nodes in case one of the devices is removed and is therefore absent from the system.

Derived from Building Embedded Systems

MTD Flash File System Model

- **Although flash devices are storage devices like hard disks, there are some fundamental differences between them.**
  - Normally hard disks have a sector of 512 byte. Flash chips have large sector size of 64K byte
  - Flash sectors normally have to be erased before writing to them. Write and erase operations can be independent depending on the software using flash
  - Flash chips have a limited lifetime which is defined in terms of number of times a sector is erased. So if a particular sector is getting written very often the lifespan gets shortened. To prevent this, the writes to a flash need to be distributed to all the sectors (this is called wear leveling and not supported by block devices)
  - Normal file system cannot be used on top of a flash because these go through the buffer cache. Normally disk IO is slow. To speed it up a cache in memory called buffer cache stores IO data to disk.
  - Generally need to halt the OS before powering down a PC based Linux system. However, embedded systems can be powered off without proper shutdown and still have consistent data.

Derived from Building Embedded Systems

MTD: Flash Access Mechanism

- **Flash Translation Layer (FTL) method**
  - Access flash to be via FTL (Flash Translation Layer)
    - This layer emulates a block device behavior on a flash to get regular file system to work on flash devices
    - Getting a new file system or a new flash driver working with FTL is cumbersome and this is why MTD subsystem was invented
  - MTD approach – treat memory devices as memory devices, not like disks. So instead of changing low level drivers or introducing a translation layer, change application to use memory devices as they are.
  - MTD is very much tied to the applications
  - MTD subsystem maps any device to both a character and a block device driver. When driver is registered with MTD subsystem, it exports the device in both these driver flavors.
    - Character device can let memory device be accessed directly using standard open/ read/write ioctl calls.
    - In the event you want to mount a regular file system on the memory device using traditional method, you can still mount it using block driver.

Derived from Building Embedded Systems

MTD: Flash Access Mechanism

- **Flash disks – ATA based**
  - Introduced for mass storage applications
    - ATA-based flash disks appear as IDE disks in the system
    - A controller sits on the same silicon as the flash, but does FTL implementation to map the flash to sectors. In addition, it implements disk protocol so that flash appears as a normal disk to the system
    - The approach taken by Compact-Flash designers
    - Advantage of this one is software compatibility but disadvantage of this one is its high cost
    - Linux treats these devices as regular IDE devices and the driver for these devices can be found in drivers/ide directory

Derived from Building Embedded Systems

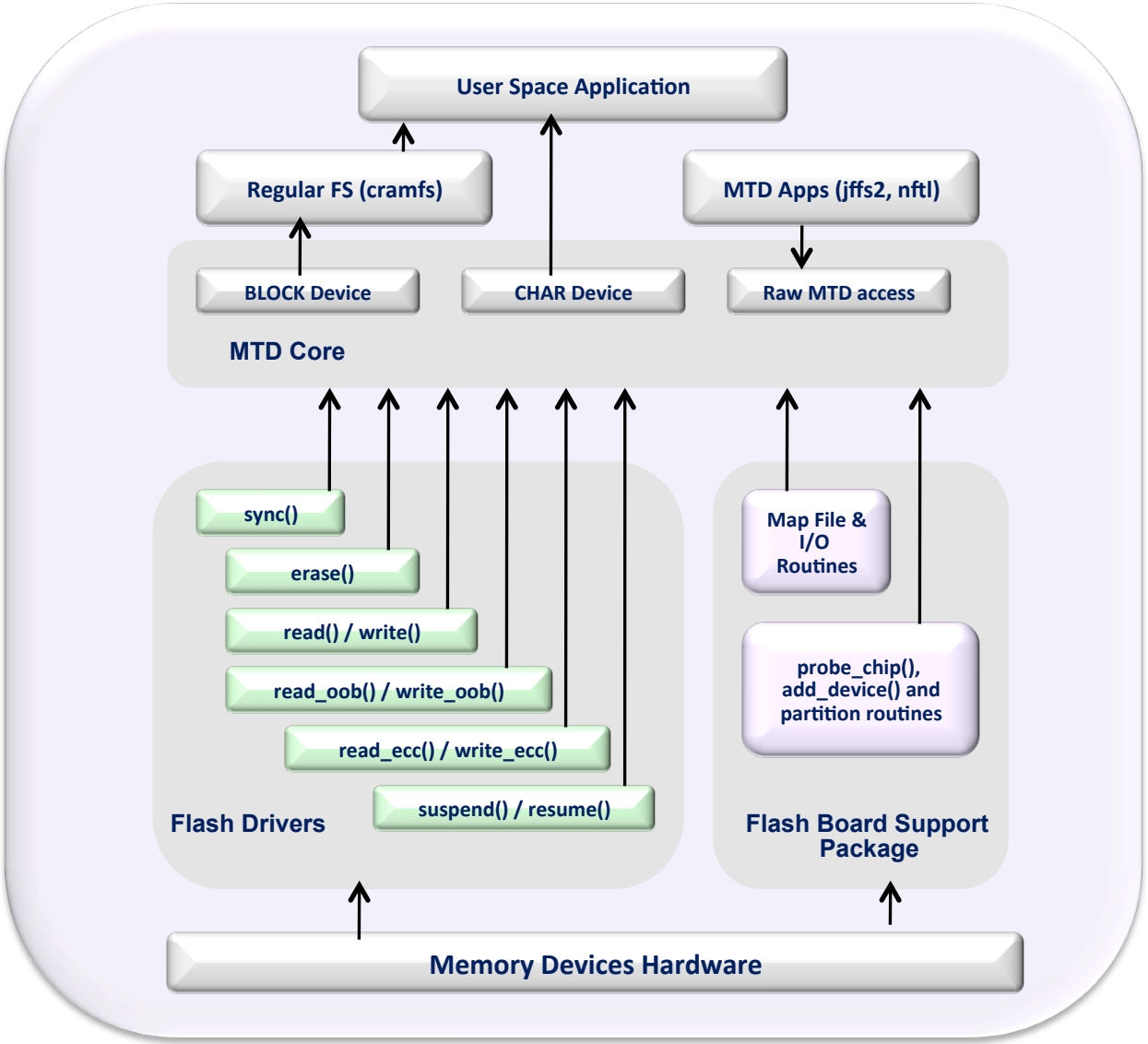
MTD Software Architecture

- **Need to answer the following questions when compiling Linux to work with a flash-based file system:**
  - **Does Linux support my flash driver, if not, how do I port the driver?**
  - **If Linux supports my flash driver, how do I make it detect the flash on my board and get its driver automatically installed ?**

Understanding MTD software architecture answers these questions

Derived from Building Embedded Systems

MTD Software Architecture Block Diagram



Derived from Building Embedded Systems

MTD Software Architecture Components

- **MTD core**
  - Provides interface between low level flash drivers and the user space application.
    - Implements the character and block device mode
- **Low level flash drivers**
  - NOR and NAND-based flash chips only
- **Board Support Package (BSP) for flash**
  - BSP layer provides details that allow flash driver work with any board/ processor.
  - The user has to provide details of how the flash is mapped on the board.
  - This piece of the code is called as flash mapping driver
- **MTD applications**
  - Kernel sub-modules such as JFFS2 or NFTL, or user-space application such as upgrade manager

Derived from Building Embedded Systems

MTD Software Architecture (cont)

- **mtd_info Data Structure**
  - Heart of MTD software
  - Defined in: …/include/linux/mtd/mtd.h
  - The software driver on detecting a particular flash fills up this structure with pointers to all the required routines (such as erase, read, write) that are used by MTD core and MTD applications
  - List of mtd_info structures for all devices added is kept in a table called mtd_table[]
- **Interface between MTD core and low-level flash drivers**
  - Low level flash driver exports the following functions to MTD core
  - Functions common to NAND and NOR flash chips
    - Read()/write()
    - Erase()
    - Lock()/unlock()
    - Sync()
    - Suspend()/resume()
  - Functions for NAND
    - Read_ecc()/write_ecc()
    - Read_oob()/write_oob()

Derived from Building Embedded Systems

MTD Software Architecture (cont)

- If you have a CFI-enabled NOR flash or a standard IO device-mapped 8 bit NAND chip, your driver is ready. Otherwise you need to implement MTD driver.
- Driver functions include:
  - Lock() and unlock() – a portion of flash can be write or erase protected to prevent accidental overwriting of images. These are exported to user applications using ioctls MEMLOCK and MEMUNLOCK
  - Sync() – this gets called when a device gets closed or released and it makes sure that the flash is in a safe state
  - Suspend() and resume() – useful only when you turn on CONFIG_PM option on building the kernel
  - Read_ecc() and write_ecc() – apply for NAND only. ECC is error-correction code
  - Read_oob() and write_oob() – apply for NAND only. Every NAND flash is divided into either 256 or 512 byte pages. Each of these pages contains an additional 8- or 16-byte spare area called out of band data, which stores ECC, bad block information, and any file system dependent data. These functions are used to access the out of band data.

Derived from Building Embedded Systems

Flash Mapping Drivers

- Irrespective of the type of device (NAND or NOR), the basis of mapping driver operations is to get mtd_info structure populated (by calling the appropriate probe routine) and then register with MTD core
- Process of flash-mapping driver can be split into
  - Creating and populating mtd_info structure
  - Registering mtd_info with MTD core

Derived from Building Embedded Systems

Flash Mapping Drivers (cont)

- Populating mtd_info for NOR flash chip
  - Address to which flash is memory mapped
  - Size of flash
  - Bus width: 8, 16, or 32 bit bus
  - Routines to do 8, 16, or 32 bit read/write
  - Routine to do bulk copy
- NOR flash map is defined in map_info data structure and database for various board configuration is found in drivers/mtd/maps directory
  - Once map_info structure is filled, function do_map_probe() is invoked with map_info as an argument. This functions returns a pointer to mtd_info structure filled with function pointer for operating on flash map.

Derived from Building Embedded Systems

Flash Mapping Drivers (cont)

- **Populating the mtd_info data structure for NAND Flash Chip**
  - NAND-based mapping driver allocates mtd_info structure whereas NOR flash calls do_map_probe() to allocate mtd_info structure
    - Allocate mtd_info structure
    - Allocate a nand_chip structure and fill up required fields
    - Make mtd_info's priv field point to nand_chip structure
    - Call nand_scan(), which will probe for NAND chip, and fill mtd_info structure with NAND operation
    - Register mtd_info structure with MTD core
  - NAND parameters that are stored in nand_chip structure
    - Mandatory parameter
      - IO_ADDR_R, IO_ADDR_W – for accessing IO lines of NAND chip
      - Hwcontrol() – implement board-specific mechanism for setting and clearing CLE, ALE and CE pins
      - Eccmode – ecc type (no ecc, software ecc, hardware ecc)
    - Nonmandatory
      - Dev_ready() – used to find state of flash
      - Cmdfunc() – sending commands to flash
      - Waitfunc() – invoked after a write or erase is done
      - Chip_delay – delay for transferring data from NAND array to its registers, default value is 20micrisec

Derived from Building Embedded Systems

Flash Mapping Drivers (cont)

- **Registering mtd_info**
  - Partitioning – allow multiple partitions on a flash to be created and added into different slots in the mtd_table[] array. The partitions can be exported as multiple devices to the application
  - Key to partitioning is mtd_partition data structure

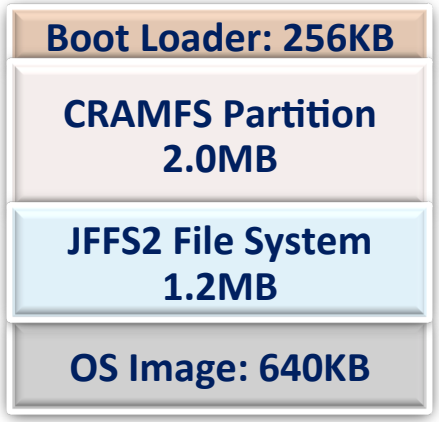
```
Structure mtd_partition partition_info[]=
{ { .name='part1", .offset=0, .size=1*1024*1024},
  { .name='part2", .offset=0, .size=3*1024*1024} }
```

  - Concatenation – merge separate devices into a single device

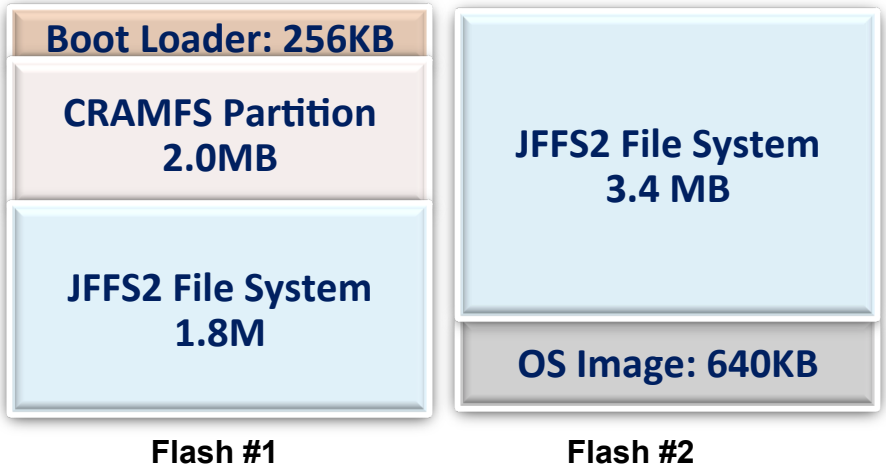
Derived from Building Embedded Systems

Flash Partition Examples

- **Flash map for a single 4MB flash:**
  - Boot-loader: 256KB
  - OS image: 640KB
  - Read only data in a CRAMFS: 2.0MB
  - Read write data in a JFFS2: 1.2MB



- **Flash map for dual 4MB flash:**
  - Boot-loader: 256KB
  - OS image: 640KB
  - Read only data in a CRAMFS: 2.0MB
  - Read write data in a JFFS2: 5.2MB
    - 1.8MB + 3.4MB



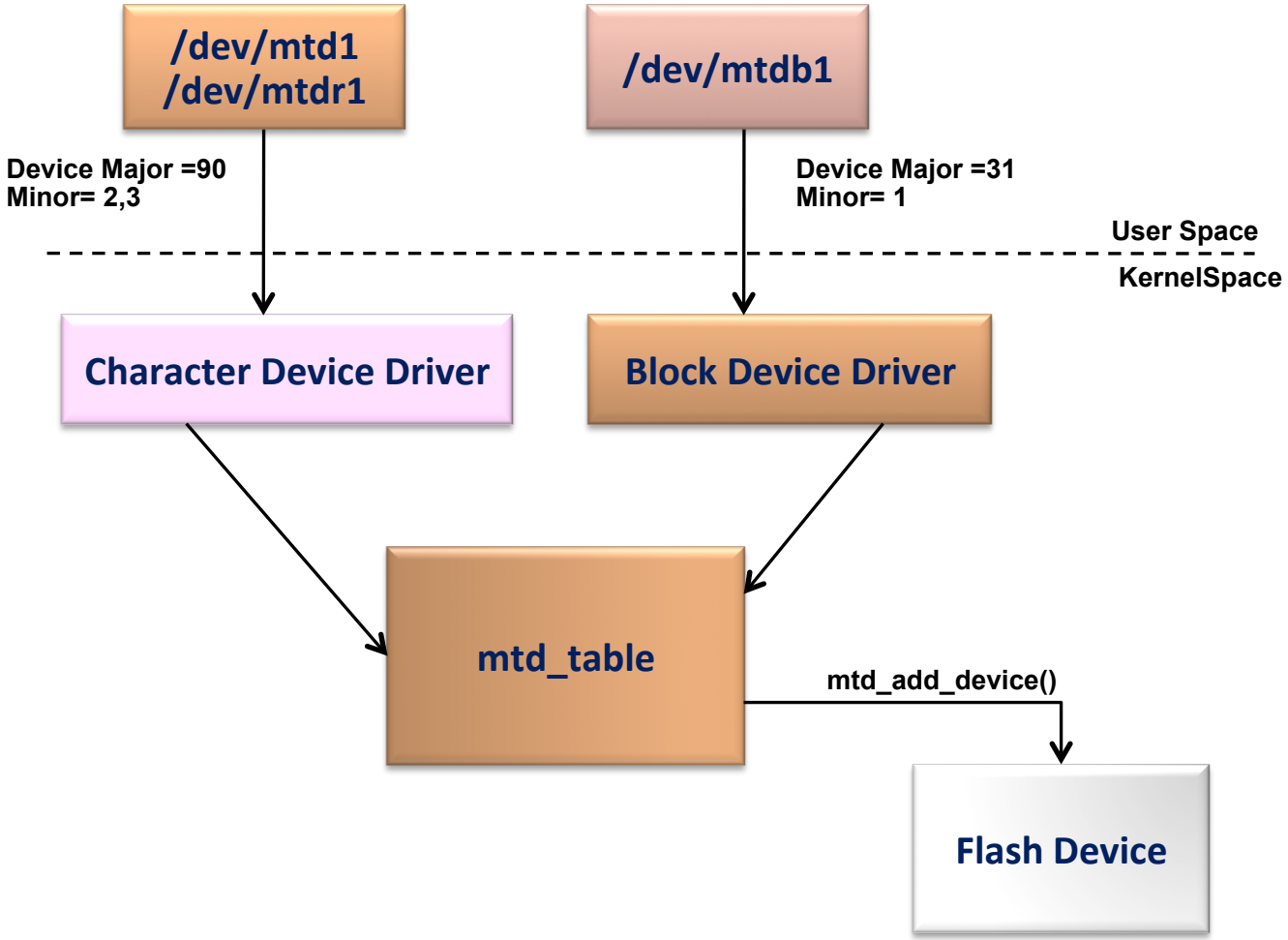
Derived from Building Embedded Systems

MTD block and character devices

- MTD devices are exported in two modes to user space: character and block device
- Character devices are represented using following device names
  - /dev/mtd0, /dev/mtdr1, /dev/mtd1, …..
  - Major number=90
- Block devices
  - /dev/mtdb1
  - Major number=31
- MTD devices having odd Minor numbers are exported as read only devices
- Following is the list of ioctls that are supported by MTD character device
  - MEMGETREGIONCOUNT – to pass number of erase regions back to user
  - MEMGETREGIONINFO – to get erase region information
  - MEMERASE – to erase specific sector of flash
  - MEMWRITEOOB – used for accessing out of band data
  - MEMLOCK/MEMUNLOCK – used for locking specified sectors if there is support from hardware

Derived from Building Embedded Systems

MTD Device Exported as a 'char' or 'block' Device



Derived from Building Embedded Systems

Mtdutils package

- A set of useful programs such as for creating file systems and testing device integrity
- Some are host based (such as tools to create a JFFS2 file image) and some can be used on the target (such as utilities to erase a flash device)
- Following is a description of individual utilities in the package
  - Erase – to erase a specified number of blocks at a given offset
  - Erase11 – to erase an entire device
  - Nft1_format – to create a NFTL(NAND Flash Translation Layer)
  - Nft1dump – to dump the NFTL partition
  - Doc_loadbios – to reprogram the disk on chip with a new firmware (such as GRUB)
  - Doc_loadip1 – to load an IPL(initialization code) into DOC flash
  - Ft1_format – to create a FTL partition
  - Nanddump – to dump NAND contents
  - Nandtest – to test NAND devices such as writing and reading
  - Nandwrite – to write a binary image into NAND flash
  - Mkfs.jffs – given a directory tree, this utility creates a JFFS image
  - Lock – to lock one or more sectors of flash
  - Unlock – unlocks all sectors of a flash device
  - Mtd_debug
  - fcp – to copy a file into a flash device

Derived from Building Embedded Systems

Embedded file systems

- **RAMDISKS**
  - Provide a mechanism for Linux to emulates a hard disk using volatile memory
    - Need a ramdisk when you do not have traditional storage device such as a hard disk or flash
  - Provide a mechanism by which you can load an actual file system into memory and use it as root file system
  - 'initrd' provides a mechanism by which a boot loader loads kernel image along with a root file system into memory

Derived from Building Embedded Systems

Embedded file systems (cont)

- **RAMFS (RAM File System)**
  - Typically an embedded system will have files that are not useful after a reboot. These file would be normally stored in /tmp directory
  - Storing these files in memory is a better option than storing them in non-volatile memory because writes are expensive and wear out the flash.

Derived from Building Embedded Systems

Embedded file systems (cont)

- **CRAMFS (Compressed RAM File System)**
  - Useful file system for flash storage
  - Introduced in 2.4 kernel
  - High-compression read only "regular" file system
    - Accessed via buffer cache using block device holding actual data
  - Enabled using MTD block device driver mode
  - CRAMFS uses 'zlib' routines for compression and it does compression for every 4 KB block
  - Using a program called mkcramfs, we can create CRAMFS image that needs to be burned in flash

Derived from Building Embedded Systems

Embedded file systems (cont)

- **Journaling Flash File System – JFFS and JFFS2**
  - A write to a file creates a log that records the offset in the file to where the data is written and size of data written along with actual data
  - When a file needs to be read, the logs are played back and using the data size and offset, the file gets recreated
  - Requires garbage collection. As time proceeds, certain logs become obsolete partially and/or totally and must be cleaned.

Derived from Building Embedded Systems

Embedded file systems (cont)

- **Main features of the JFFS2 file system**
  - Management of erase blocks
    - Erase blocks are placed in three list: clean, dirty and free. Clean list contains only valid logs. Dirty list contains obsolete logs to be purged when garbage collection is called. Free list contains no logs and will be used for storing new logs.
  - Garbage collection
    - Work as a separate thread when gets started when we mount a JFFS2 file system
    - For every 99 out of 100 times, this thread will pick up a block from dirty list and for the other 1 out of 100 will pick up a block from clean list to ensure wear leveling
    - JFFS2 reserves 5 blocks for doing garbage collection
  - Compression
    - JFFS2 uses various compression routines including zlib and rubin
  - JFSS and JFSS2 file system image is created using mkfs.jffs and mkfs.jffs2

Derived from Building Embedded Systems

Embedded file systems (cont)

- **NFS (Network File System)**
  - Popular desktop file system EXT2 and EXT3 are exportable to an embedded system using NFS
    - Access it as a root file system on embedded system
  - During debugging stage, often developers can make changes to root file system. In such a case writing to flash can be costly (because flash has limited write cycles) and time consuming. NFS can be a good choice provided the network driver is ready
  - With NFS you do not have any size restrictions because all the storage is done on the remote server.
  - Linux kernel provides a mechanism for automatically mounting a file system using NFS at boot time using following steps
    - config options CONFIG_NFS_FS, which allow you to mount a file system from a remote server and CONFIG_ROOT_NFS, which allows you to use NFS as root file system needs to be turned on at build time

Derived from Building Embedded Systems

Embedded file systems (cont)

- **PROC file system**
  - A logical file system used by kernel to export its information to external world
  - Proc file system can be used for system monitoring, debugging and tuning
  - Proc files are created on the fly when they get opened.
  - Proc files can be read-only or read-write depending on the information exported by files
  - Read-only proc files can be used to get some kernel status and cannot be modified at runtime. (ex) process status information
  - Read-write proc files contain information that can be changed at runtime. (ex) TCP keep-alive time.

  - Recall /proc/interrupts had the following entries when accessed

Int #	#INT's	Source
20:	117	IMX-uart
26:	15985	i.MX Timer Tick
55:	0	imx21-hc:usb1
224:	2	smsc911x

Derived from Building Embedded Systems