

Lecture 15

Nonvolatile Memory & Energy Management

CE346 – Microcontroller System Design
Branden Ghena – Spring 2026

Some slides borrowed from:
Josiah Hester (Northwestern), Prabal Dutta (UC Berkeley)

Administrivia

- Project check-ins this Friday
 - Signup linked on Ed and Canvas
- Come prepared to talk about
 - Project overview/goals
 - Progress so far
 - What still needs to be worked on
 - Any major concerns
 - Hardware you still need to order
- Project demos in two weeks from today!

Today's Goals

- Discuss uses of memory, especially nonvolatile memory, in embedded systems
- Introduce internal flash peripheral and SD cards
- Discuss matters of energy on embedded systems
 - Where to gain energy?
 - How much does the Microbit use?
 - How do we write software for *very* low energy systems?

Outline

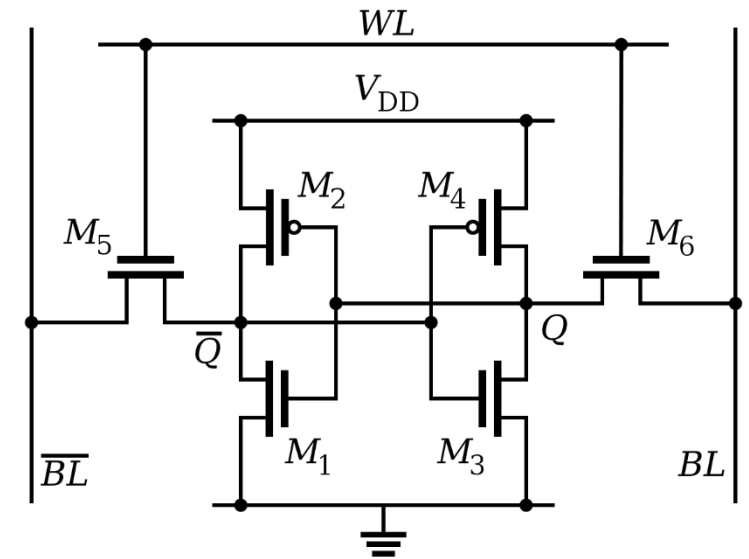
- **Nonvolatile Memory**
 - **Memory in Computing**
 - nRF52 Non-Volatile Memory Controller
 - SD Card Interface
- Energy Management
 - Energy Sources
 - Microbit Energy Use
 - Intermittent Computing

Memory in computing

- Various different memories serve different purposes in computing
- Needs
 - Fast, infinite-lifetime memory to keep things like stack memory
 - Nonvolatile memory that can be read from
- Desires
 - Fast, infinite-lifetime nonvolatile memory

Register technology: SRAM

- Static RAM (SRAM)
 - Each cell stores a bit in a bi-stable circuit, typically a six-transistor circuit
 - Static – no need for periodic refreshing; keeps data while powered
 - Relatively insensitive to disturbances such as electrical noise
 - Energetic particles (alpha particles, cosmic rays) can flip stored bits
- Fastest memory on computer
 - Also most expensive and takes up most space per bit
 - Typically used for registers and cache memories



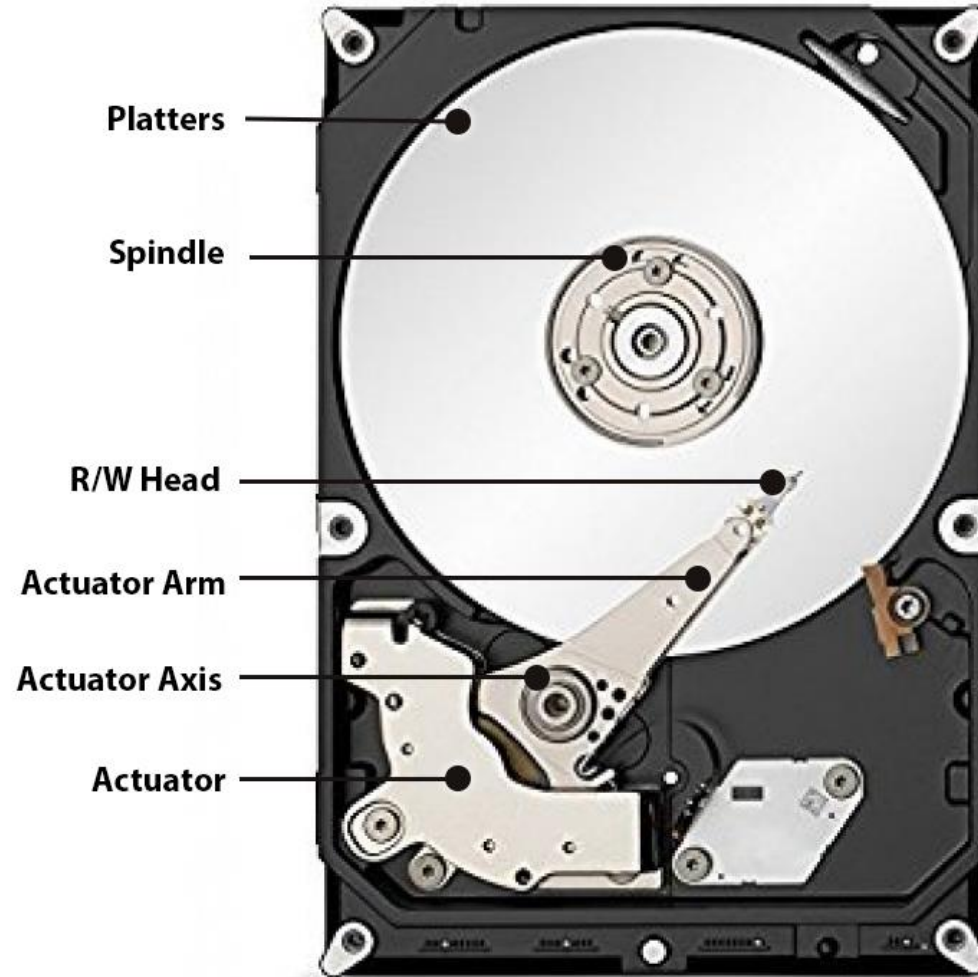
SRAM can be used a permanent memory in a pinch

- Gameboy and Gameboy Color used batteries to save state
- Gameboy Advanced games used batteries for an internal clock
- PSA: old Gameboy games have likely lost their save files



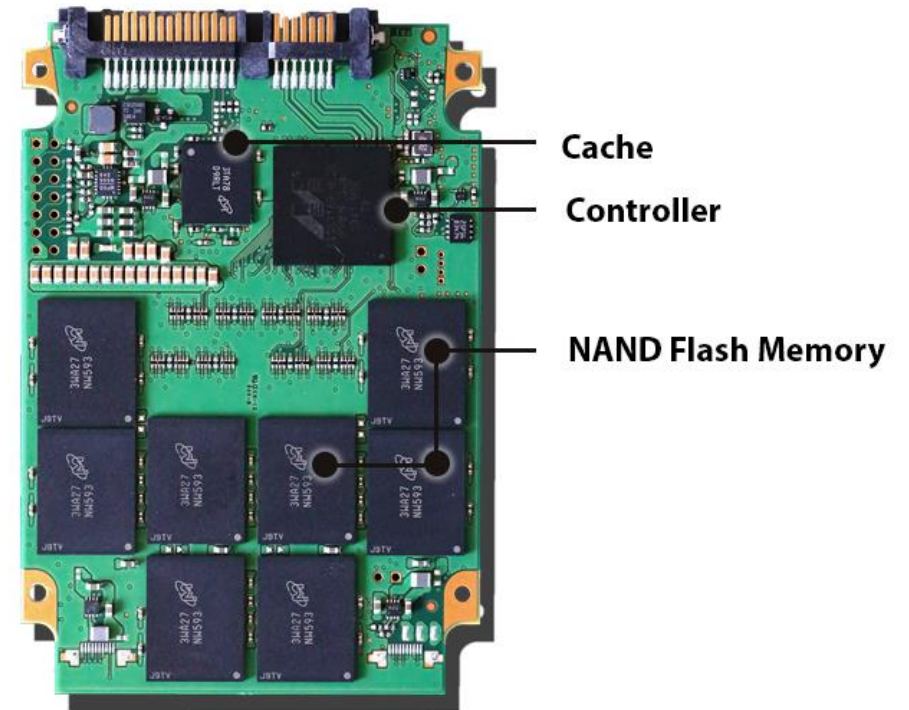
Disk drive storage

HDD
3.5"



Shock resistant up to 55g (operating)
Shock resistant up to 350g (non-operating)

SSD
2.5"



Shock resistant up to 1500g
(operating and non-operating)

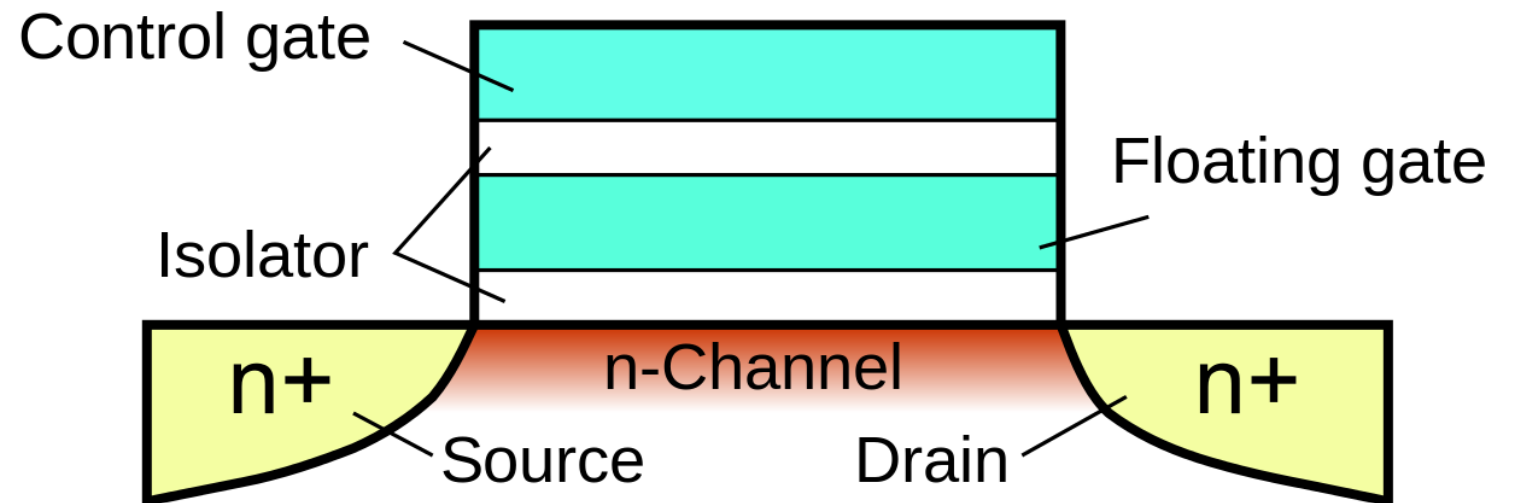
Necessity breeds creativity

- Original iPod used a small disk drive



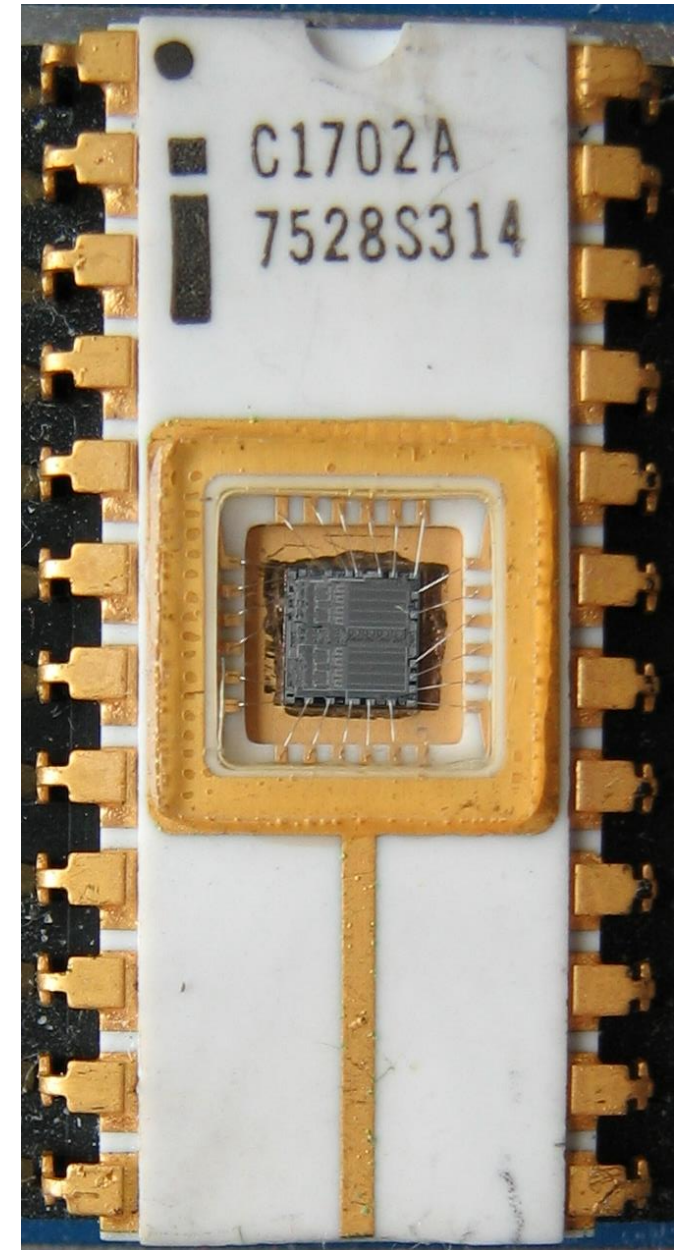
Floating-gate transistors

- Concept behind transistor-based non-volatile memory
 - EPROM, EEPROM, and Flash
 - High voltage on control gate creates charge on floating gate
 - Charge on floating gate activates/deactivates transistor
- High voltage degrades the structure, leading it to eventually fail after enough writes



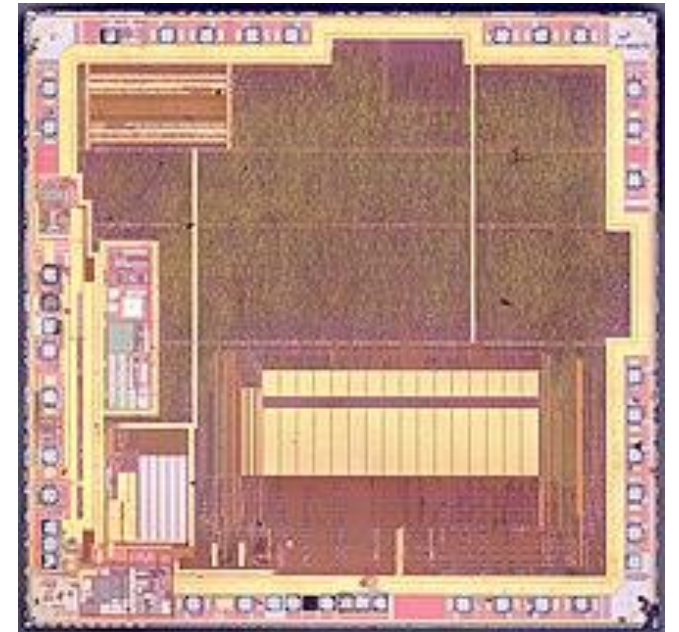
EPROM

- Erasable programmable read-only memory
- Erasable
 - If you shine UV light directly on the IC
 - Needed a window to expose the IC
- Programmable
 - With high voltage (25-50 volts)
- Typically acted as read-only memory in circuits



EEPROM

- Electrically-erasable programmable read-only memory
- Same concept as EPROM, but includes internal circuitry to allow rewriting under normal conditions
 - Slow and high-power to write
 - Has a longer lifetime compared to flash, ~100k writes
- Can be built into other ICs
 - Example: AT90USB162 microcontroller (512 bytes)



Flash

- Similarly based on floating-gate transistors
 - But with a different design that allows for faster erase of entire blocks
 - More limited lifetime, $\sim 1\text{k}-100\text{k}$ writes (10k common for embedded)
- Cannot erase individual bytes, must erase in units of blocks
 - Read can happen in units of bytes though
- Heavily used in commercial devices
 - Flash drives
 - SSDs
 - Smartphone storage
 - Microcontroller non-volatile storage!

More exotic memories

- FRAM and MRAM are both rising protentional Flash replacements
 - Non-volatile
 - Writable at the byte level
 - Very high to infinite write/erase cycles
 - Lower energy costs for writing and reading
- They use unrelated magnetic techniques for data storage
- Starting to appear in microcontrollers
 - TI MSP430s have used 16 kB FRAM
 - Apollo4 (ARM Cortex-M4F) has 2 MB of MRAM

Outline

- **Nonvolatile Memory**

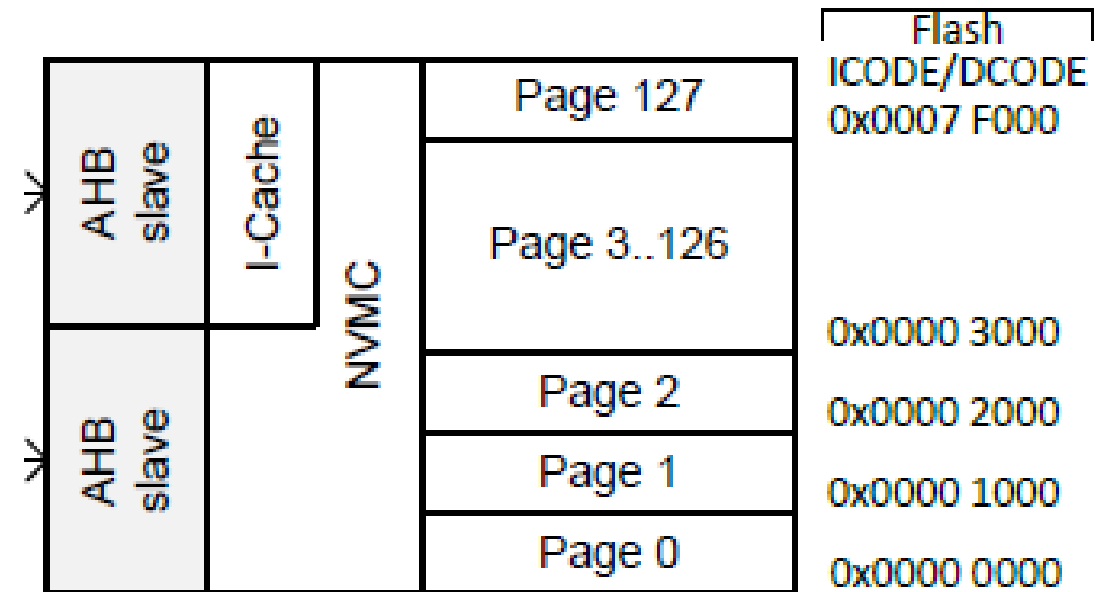
- Memory in Computing
- **nRF52 Non-Volatile Memory Controller**
- SD Card Interface

- Energy Management

- Energy Sources
- Microbit Energy Use
- Intermittent Computing

Flash memory on the nRF52833

- 512 kB total Flash memory
 - 128 pages each 4 kB in size
- Non-Volatile Memory Controller (NVMC) controls access
 - Enables writing to flash
 - Enables erasing flash
 - Manages status of flash



Writing to Flash

- Configurable, disabled by default
 - Enable with configuration register
- Rules for writing to Flash
 - Must write word-aligned 32-bit values
 - Can only write 0 values, not ones
 - Can only write 2 times before erasing (even if there are still 1 bits)
- Takes 42.5 μ s to write a 32-bit word
 - 64 MHz clock $\Rightarrow$ 2720 cycles per 32-bit write

Erasing Flash

- Lifetime: 10000 erase cycles per page
 - At once-a-second writes, that's <3 hours...
- Options
 - Erase a single page (4 kB): 87.5 ms
 - Erase all of flash (512 kB): 173 ms
- CPU is halted if executing code from Flash during the erase
 - That's 5.6 million cycles...
 - Code can execute from SRAM instead
 - Can also be split into a series of partial erases
 - Which must add up to a complete erase time before writing

Factory Information Configuration Registers

- Read-only memory
- Chip-specific information and configuration
 - Code size
 - Unique device ID
 - Production IDs
 - Temperature conversion functions

User Information Configuration Registers

- Additional Flash memory for non-volatile user configurations
 - Writable and erasable through NVMC processes described earlier
- 32 words of customer information (128 bytes total)
- Special configurations
 - Reset pin
 - NFC pin enable/disable
 - Debug configuration

Break + Question

- Could you run a system entirely within Flash?
- Could you run a system entirely within RAM?

Break + Question

- Could you run a system entirely within Flash?
 - Yes, but it would go _very_ slowly
 - Local variables would be pretty hard to manage
 - 87.5 ms of code pause every time you write to a variable...
- Could you run a system entirely within RAM?
 - Yes, but code would need to be loaded from somewhere else
 - Need initial state that is nonvolatile
 - Would run just as fast and be lower energy, actually

Outline

- **Nonvolatile Memory**

- Memory in Computing
- nRF52 Non-Volatile Memory Controller
- **SD Card Interface**

- Energy Management

- Energy Sources
- Microbit Energy Use
- Intermittent Computing

SD card references

- ChaN

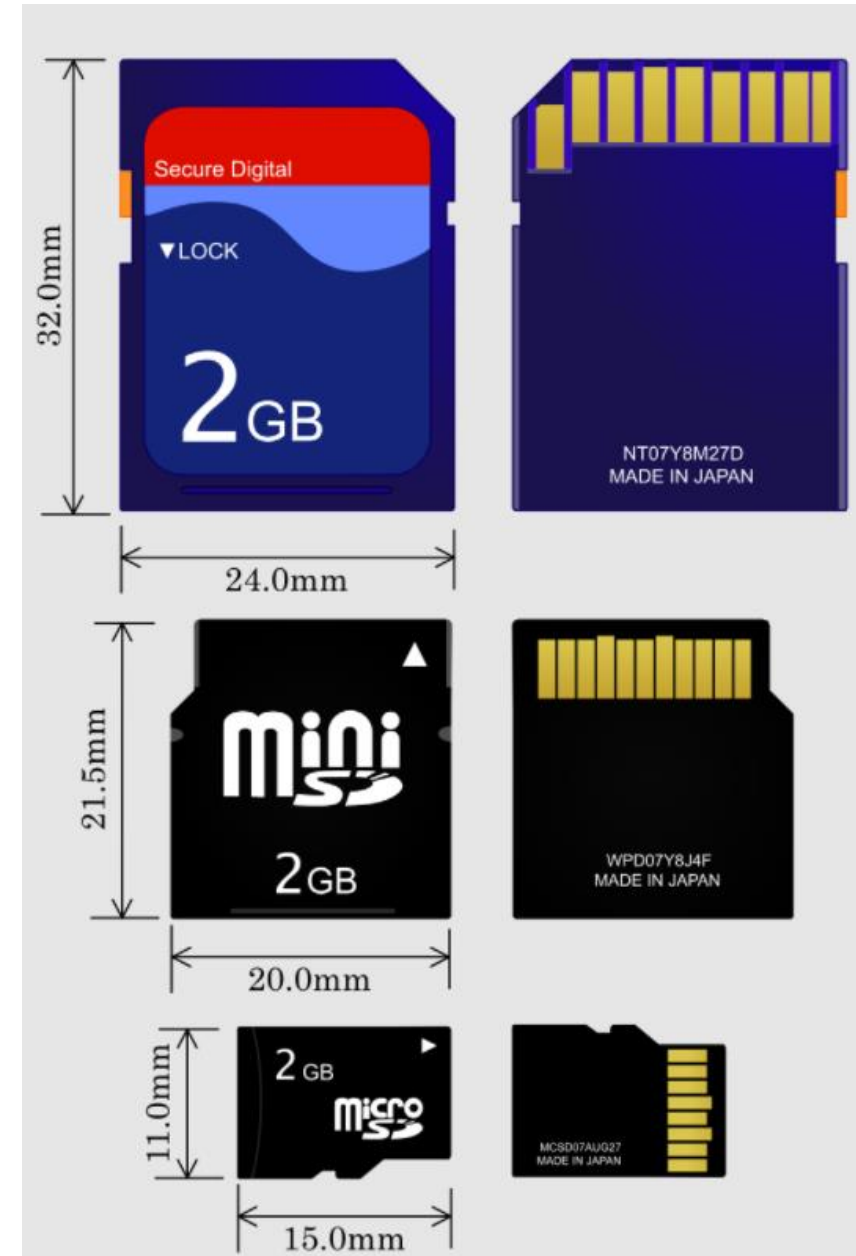
- Embedded systems engineer in Japan (and is amazing)
- http://elm-chan.org/docs/mmc/mmc_e.html
- http://elm-chan.org/fsw/ff/00index_e.html

- Various others

- http://users.ece.utexas.edu/~gerstl/ee445m_s15/lectures/Lec08.pdf
- http://alumni.cs.ucr.edu/~amitra/sdcard/Additional/sdcard_appnote_foust.pdf
- <https://luckyresistor.me/cat-protector/software/sdcard-2/>
- http://users.ece.utexas.edu/~valvano/EE345M/SD_Physical_Layer_Spec.pdf
- <https://github.com/tock/tock/blob/master/capsules/src/sdcard.rs>

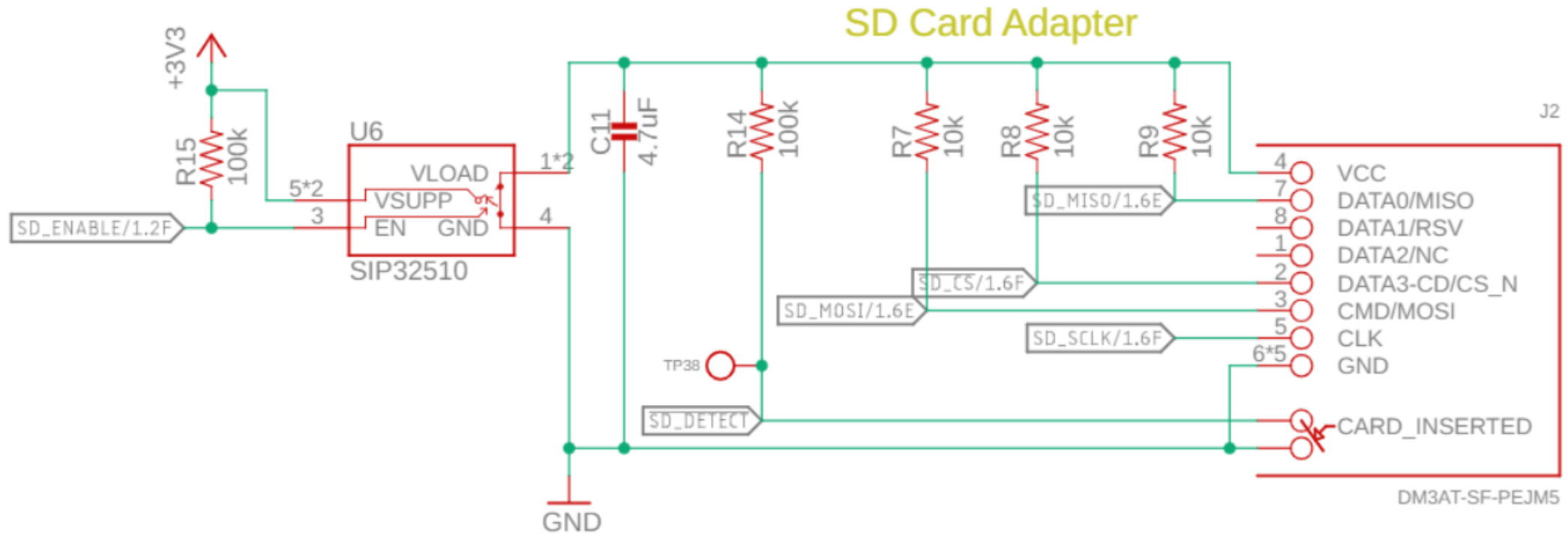
SD cards

- “Secure Digital” Card
 - Includes various formfactors
 - Flash memory
 - Capacities from 8 MB to 128 TB
 - 512 byte blocks
- Supports 1-bit SPI interface
 - As well as 4-bit SD bus protocol
- Easy to support in embedded systems
 - Cheap but high power



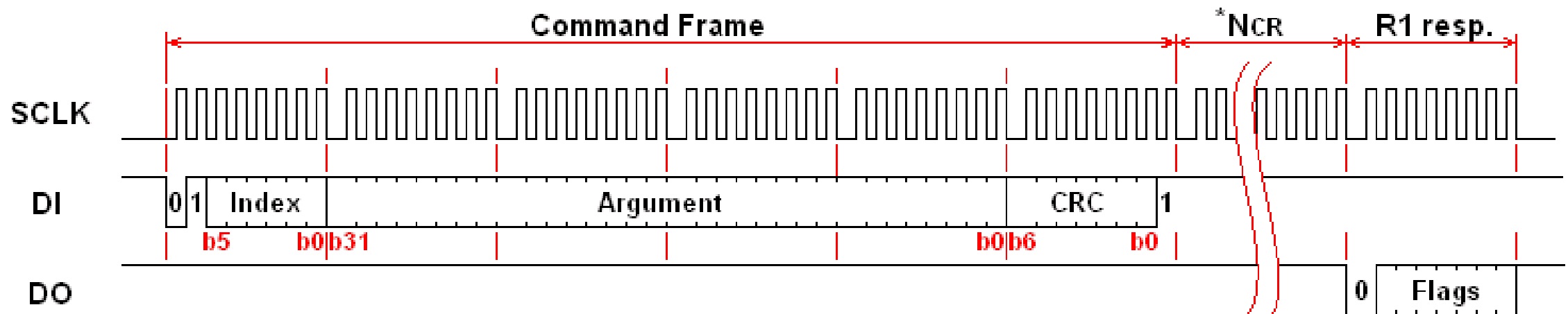
Electrical connections for an SD card

- SD Card connections
 - SPI SDI, SDO, CS, SCLK
 - Plus a switch to enable/disable the SD card and a detect signal



Controlling the SD card

- Index: 6-bit value of command being sent
- Argument: 32-bit value that may be arguments to commands
- CRC: checks for bit errors
- Response (after delay)



SD card SPI commands

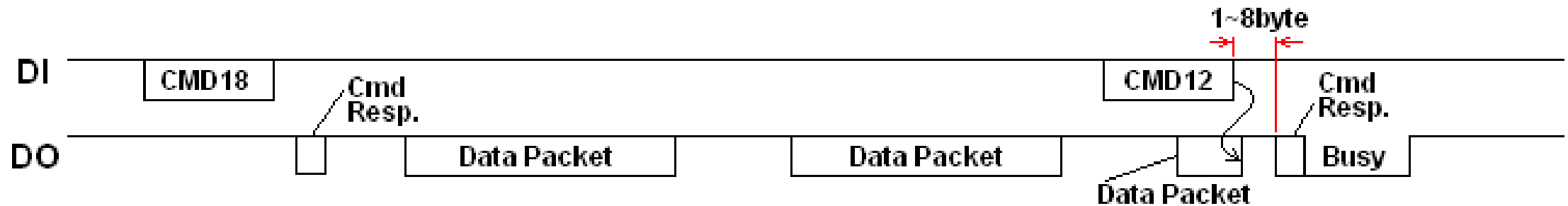
Command Index	Argument	Response	Data	Abbreviation	Description
CMD0	None(0)	R1	No	GO_IDLE_STATE	Software reset.
CMD1	None(0)	R1	No	SEND_OP_COND	Initiate initialization process.
ACMD41(*1)	*2	R1	No	APP_SEND_OP_COND	For only SDC. Initiate initialization process.
CMD8	*3	R7	No	SEND_IF_COND	For only SDC V2. Check voltage range.
CMD9	None(0)	R1	Yes	SEND_CSD	Read CSD register.
CMD10	None(0)	R1	Yes	SEND_CID	Read CID register.
CMD12	None(0)	R1b	No	STOP_TRANSMISSION	Stop to read data.
CMD16	Block length[31:0]	R1	No	SET_BLOCKLEN	Change R/W block size.
CMD17	Address[31:0]	R1	Yes	READ_SINGLE_BLOCK	Read a block.
CMD18	Address[31:0]	R1	Yes	READ_MULTIPLE_BLOCK	Read multiple blocks.
CMD23	Number of blocks[15:0]	R1	No	SET_BLOCK_COUNT	For only MMC. Define number of blocks to transfer with next multi-block read/write command.
ACMD23(*1)	Number of blocks[22:0]	R1	No	SET_WR_BLOCK_ERASE_COUNT	For only SDC. Define number of blocks to pre-erase with next multi-block write command.
CMD24	Address[31:0]	R1	Yes	WRITE_BLOCK	Write a block.
CMD25	Address[31:0]	R1	Yes	WRITE_MULTIPLE_BLOCK	Write multiple blocks.
CMD55(*1)	None(0)	R1	No	APP_CMD	Leading command of ACMD<n> command.
CMD58	None(0)	R3	No	READ_OCR	Read Operations Condition Register (OCR). Indicates supported working voltage range.

Reading from the SD card

- Single block read: CMD17

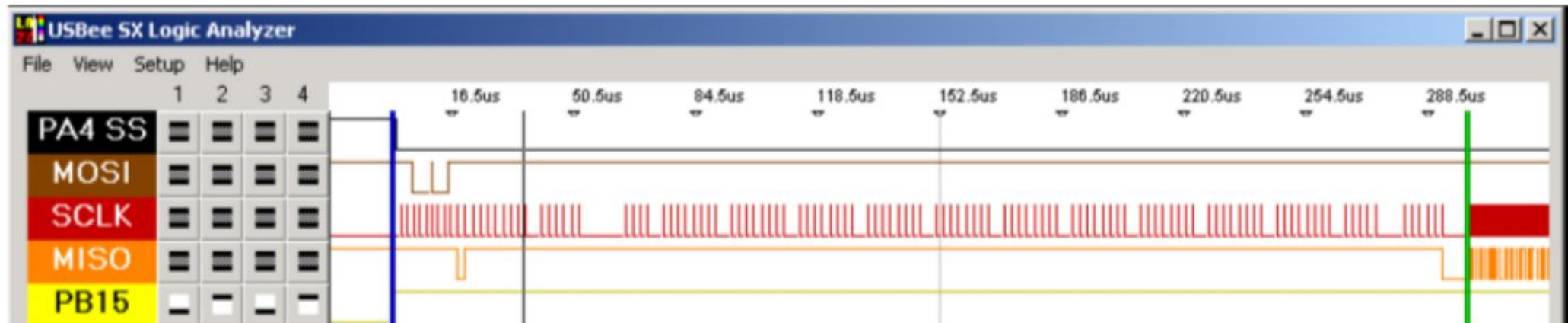
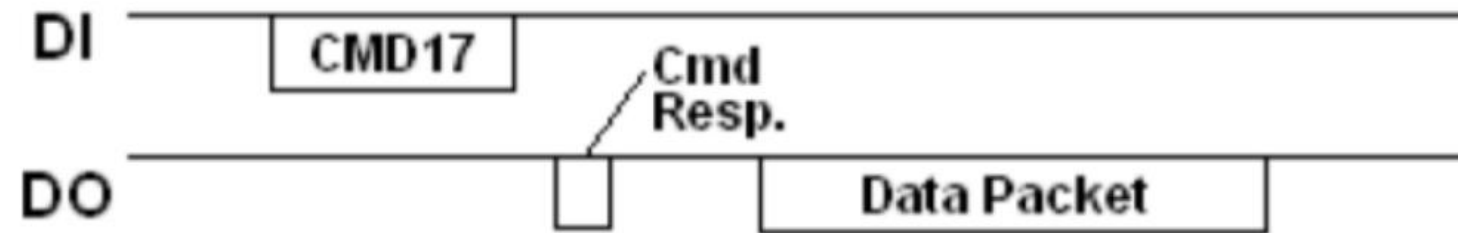


- Multiple block read: CMD18 (CMD12 – Stop Transmission)



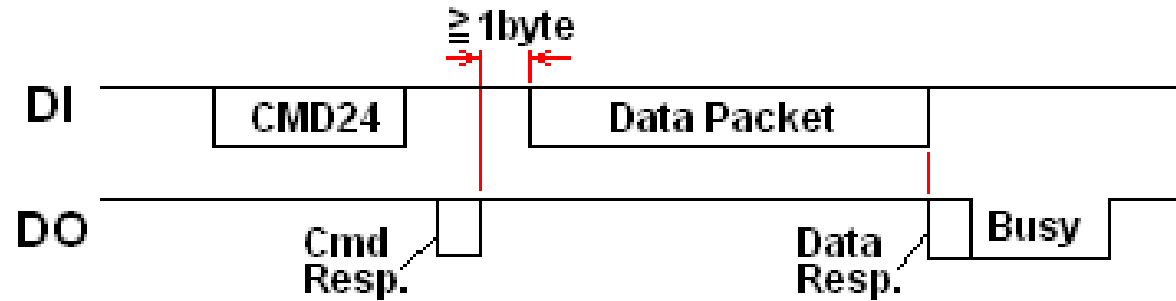
SD card delays can be significant

- Performing a single byte read
 - Almost 300 μs before the SD card *starts* sending data
 - $\sim 200 \mu\text{s}$ additional time to send the 512 bytes (20 Mbps data, 8 Mbps total)

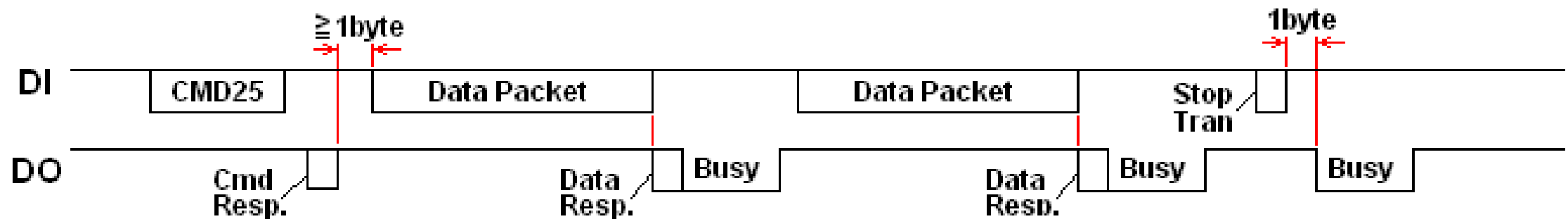


Writing to the SD card

- Single block write: CMD24



- Multiple block write: CMD25



SD Card Example Generic Driver

- <https://github.com/1nv1/ulibSD>
 - Handles the logic of SD card interactions
 - Based on a generic SPI device
 - User has to provide the following functions
 - void SPI_Init();
 - BYTE SPI_RW(BYTE);
 - void SPI_CS_LOW()
 - void SPI_CS_HIGH();
 - ...
- Idea: provide some narrow interface to allow reuse of software
 - Code below are microcontroller-specific SPI implementations
 - Code above is generic SD card control logic

Hardware Abstraction Layers (HALs)

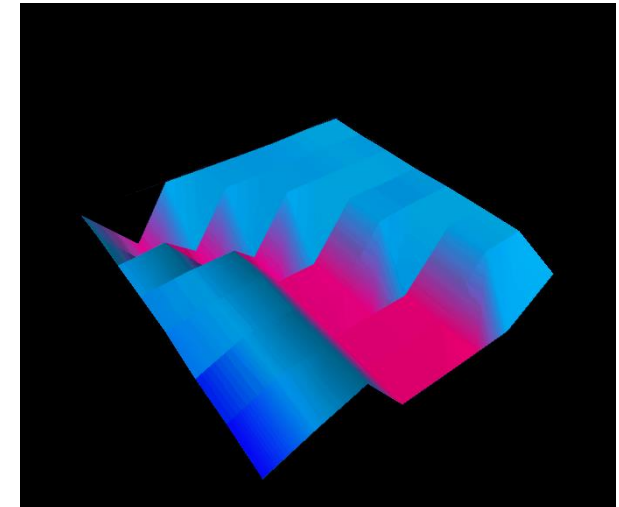
- Idea: enable hardware-agnostic software on top of hardware-aware implementations
- Hardware Abstraction Layers allow code reuse on multiple platforms by decoupling higher-level software from the implementation specifics
- Example: generic LED
 - Generic functionality: Activate or Deactivate
 - Hardware-specific implementation:
 - Active-high or active-low
 - Simple LED or complex NeoPixel RGB LED

Higher-level software uses a HAL

- I2C devices are great examples of higher-level software
 - There is device-specific code that decides which registers to read or write
 - But the code does *not* rely on any particular microcontroller
 - As long as you've got an I2C system that can read/write
- We can reuse device drivers across various platforms
 - At the cost of needing to separate out microcontroller-specific behavior

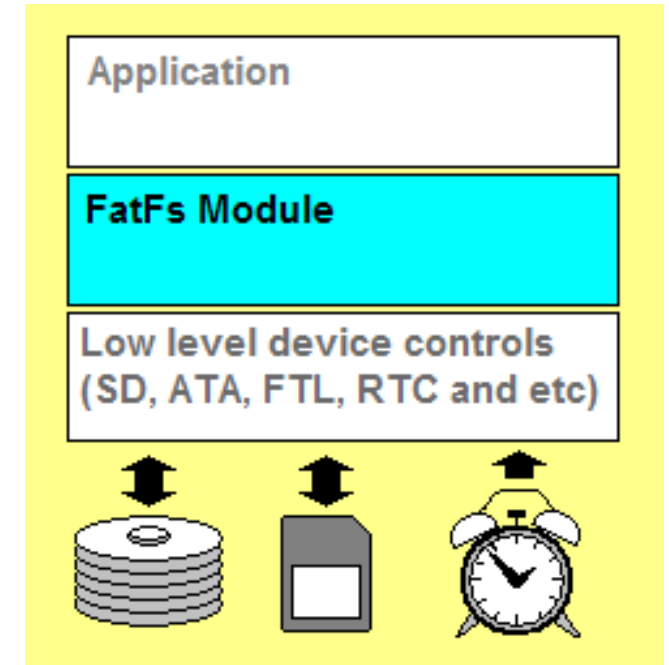
Abstraction layer example

- Example of abstraction layer: Infrared Time-of-Flight Ranging chip
- Rather complicated device that can provide an 8x8 array of distance measurements
- Chip manufacturer created a generic C driver for the chip
- User has to implement
 - Reset sensor
 - Wait milliseconds
 - I2C read
 - I2C write



Layering a filesystem on top of an SD card

- FatFs library implements the filesystem agnostic of application and storage medium
- Enables the use of file system calls:
 - Open, Close, Read, Seek
- Connects to generic interface for low-level implementation
 - disk_status, disk_init, disk_read, disk_write



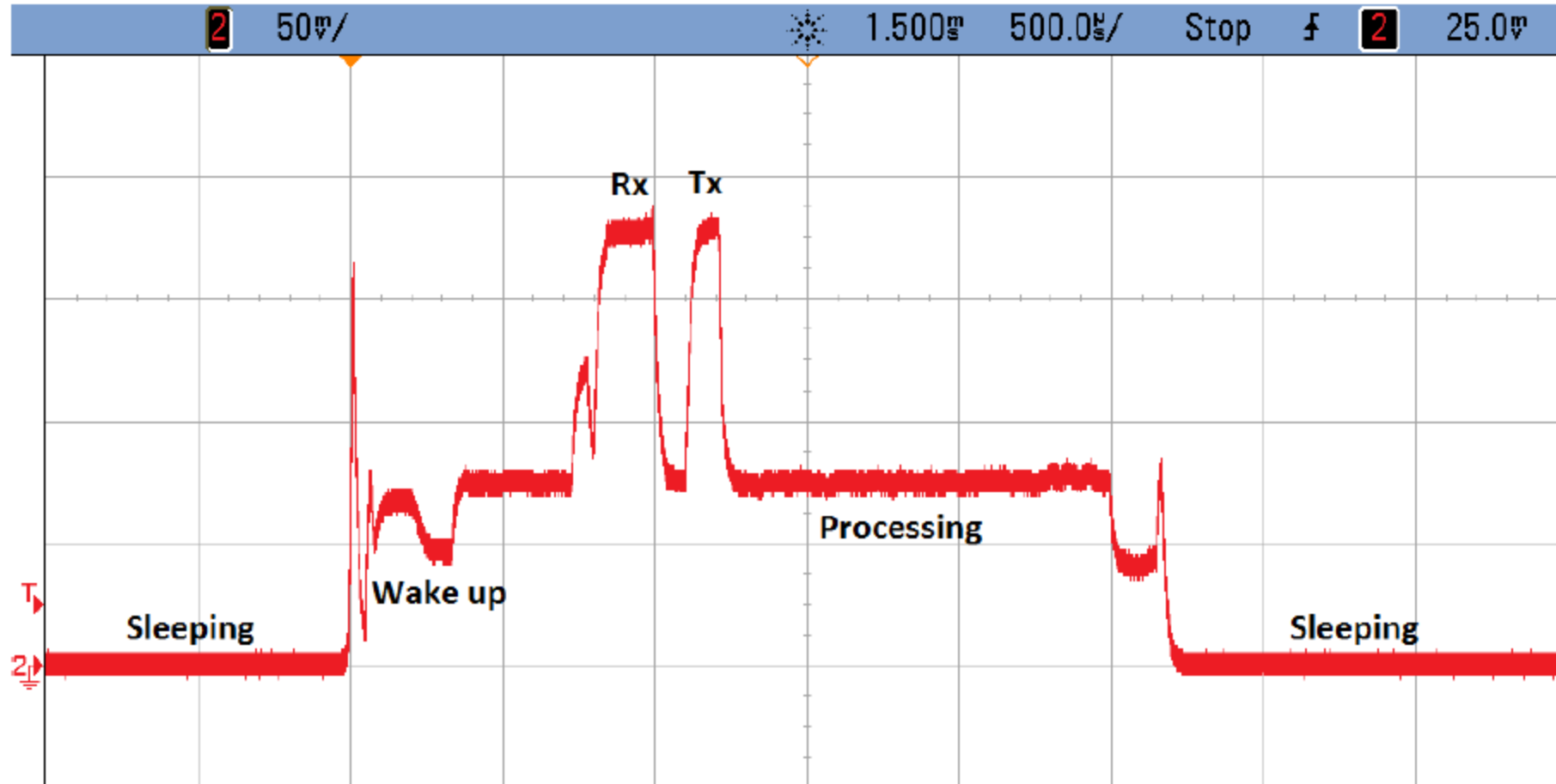
Outline

- Nonvolatile Memory
 - Memory in Computing
 - nRF52 Non-Volatile Memory Controller
 - SD Card Interface
- **Energy Management**
 - **Energy Sources**
 - Microbit Energy Use
 - Intermittent Computing

Measuring energy use

- Base equations
 - $\text{Power} = \text{Current} * \text{Voltage}$ (Watts)
 - $\text{Energy} = \text{Power} * \text{Time}$ (Joules)
- $\text{Energy} = \text{volts} * \text{amps} * \text{seconds}$
 - Voltage is *usually* constant for a system
 - Time is how long you are running for / measurement period
- Current changes based on activities being done
 - Often energy is presented as a current draw
 - Maybe an average current draw
 - With Voltage and Time implicit

Example current trace during wireless communication



Current Consumption versus Time during a single Connection Event

Wired power through USB

- Provides 5v at up to 500 mA (USB 2.0) or 900 mA (USB 3.0)
 - Or power delivery specifications, which can do far more power
- Must be converted to different voltage to use
 - Voltage regulator takes in 5v and spits out 3.3v
 - Has its own maximum current!
- System is limited by the minimum of USB or regulator power
 - Microbit: regulator gives 3.3v at up to 600 mA = 1.98 W
 - USB 2.5 Watts, Regulator 1.98 Watts $\Rightarrow$ System 1.98 Watts
 - This is a max! Stay 15-30% below regulator limit

Thinking about energy

- Batteries often list energy in mA*h (milliamp – hours)
 - Coin cell battery: 3v at 220 mAh
 - 2x AA battery: 3v at 2000 mAh
 - iPhone 11 battery: 3.7v at 3000 mAh
- Also usually limited by regulator
 - Sometimes just directly connected to system
 - We can run at 3v just fine! (3.7v is no good though)
- Voltage can vary with charge
 - But only a little, right before battery is depleted
 - Example: coin cell goes down to ~ 2.7 volts

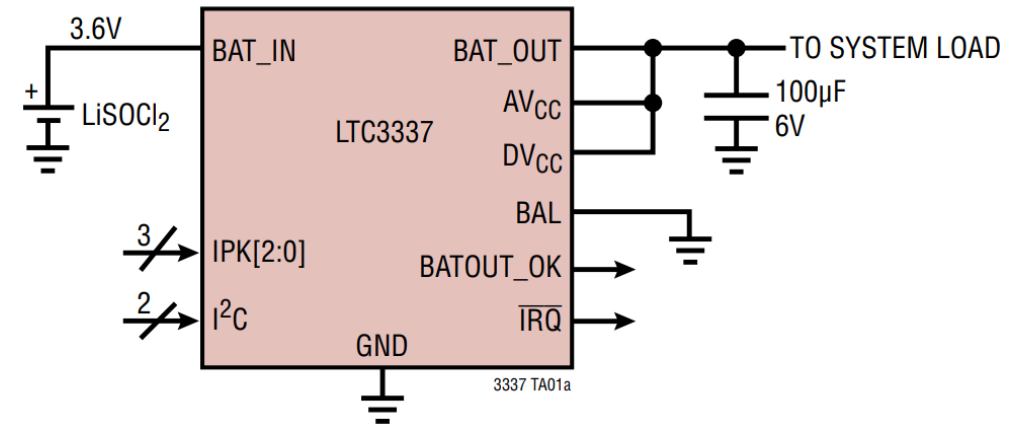


How are batteries measured?

- Measuring energy remaining is a difficult problem
 - Many questions to be handled
 - How much did it start with?
 - How much energy has been used?
 - What type of battery is it?
 - Energy is not as constant a quantity as one would hope
 - Pulling out lots at once has an overhead penalty

- Coulomb Counter (aka Battery Fuel Gauge)
 - Designed for a specific battery "chemistry"
 - Monitors charge flowing in each direction
 - I2C interface for reading battery state

- Accuracy is not exact, more of an educated guess



How are batteries managed?

- Usually a dedicated IC for charging and managing battery packs
 - Recharges battery with appropriate amount of current
- Monitors issues of battery health
 - Various status monitoring
 - Overcharge, undercharge
 - Overcurrent
 - Overtemperature, undertemperature
 - Will go so far as to cut off the system to protect the battery
- Takeaway: complicated problem, approach with caution!
 - Best to reuse an existing design, if possible

Microbit only uses battery energy in a simple way

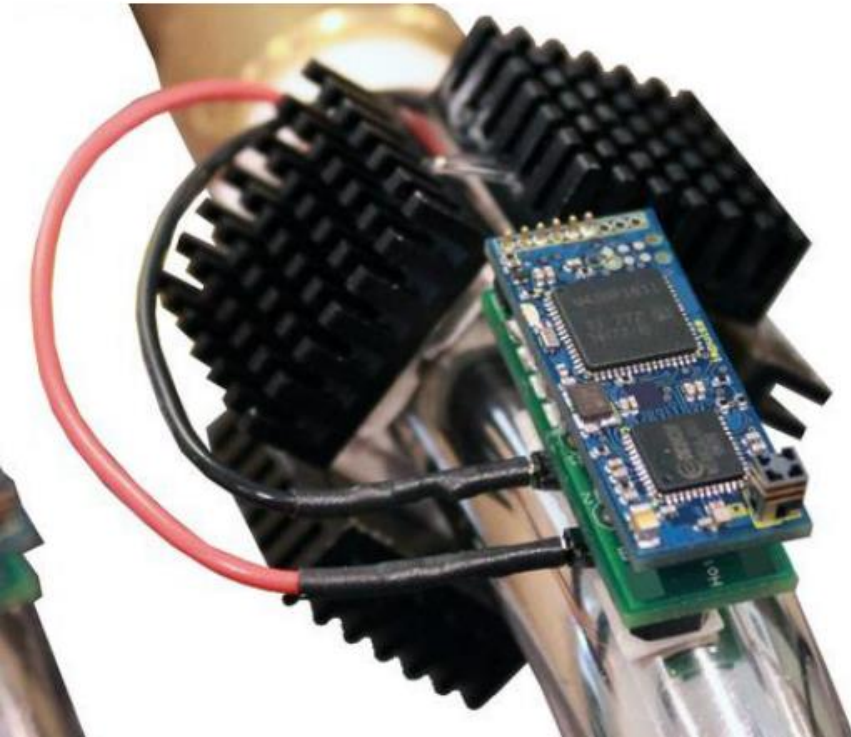
- Battery input connects directly to regulator
 - No protection for battery health
 - No battery charging capabilities
- Usually this is fine for simple, low-power systems
 - It means that the input voltage can vary though
 - Makes the reference voltages for the ADC/Comparator more important

Energy harvesting

- Grab energy from the environment and use that!
 - Could augment with a battery and use energy to recharge
 - Could go entirely batteryless and live on harvested energy alone
- Sources
 - Light (outdoor or indoor. most successful)
 - Airflow (outdoor or air vents)
 - Motion (on human body)
 - Temperature differential (difficult in practice)
 - RF (very low energy source)

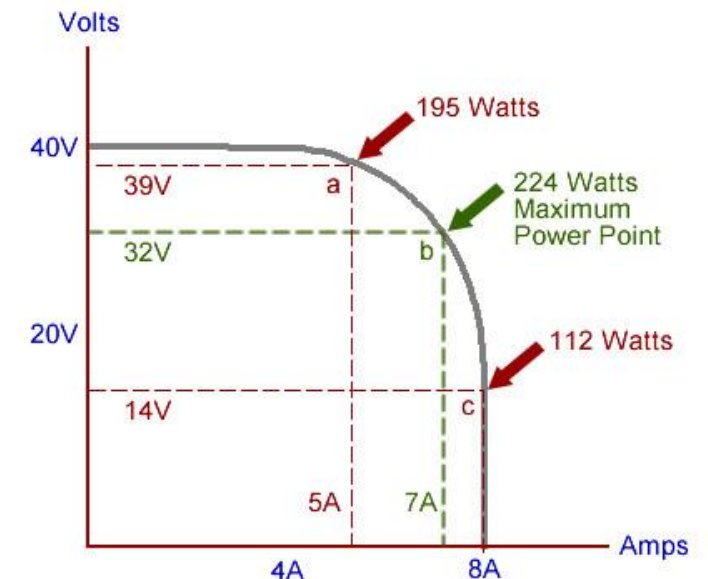
Temperature harvesting from hot pipes

- Peltier junctions create a voltage from temperature differential
 - Challenge: needs a large differential for more energy



Managing harvested energy

- Often uses an IC to pull in energy and provide to system
- Harvested voltage/current are often very small
 - Signal in millivolts is pretty common
 - Need to accumulate over time to power system
 - Fill up a capacitor
- Need particular load for maximum power
 - ICs often implement Maximum Power Point Tracking (MPPT)
 - Varies load automatically to always harvest the most possible energy



Break + Question

- System configuration
 - System current draw: 280 mA
 - Battery capacity: 1000 mAh
- How long can you run the system for?

Break + Question

- System configuration

- System current draw: 280 mA
- Battery capacity: 1000 mAh

Microbit max
current draw



Typical AAA
battery energy



- How long can you run the system for?

- $1000 \text{ mAh} / 280 \text{ mA} \approx 3.5 \text{ hours}$
- That's not great battery life. Would need a much bigger battery to keep this running for a long time, or need to reduce average current consumption

Outline

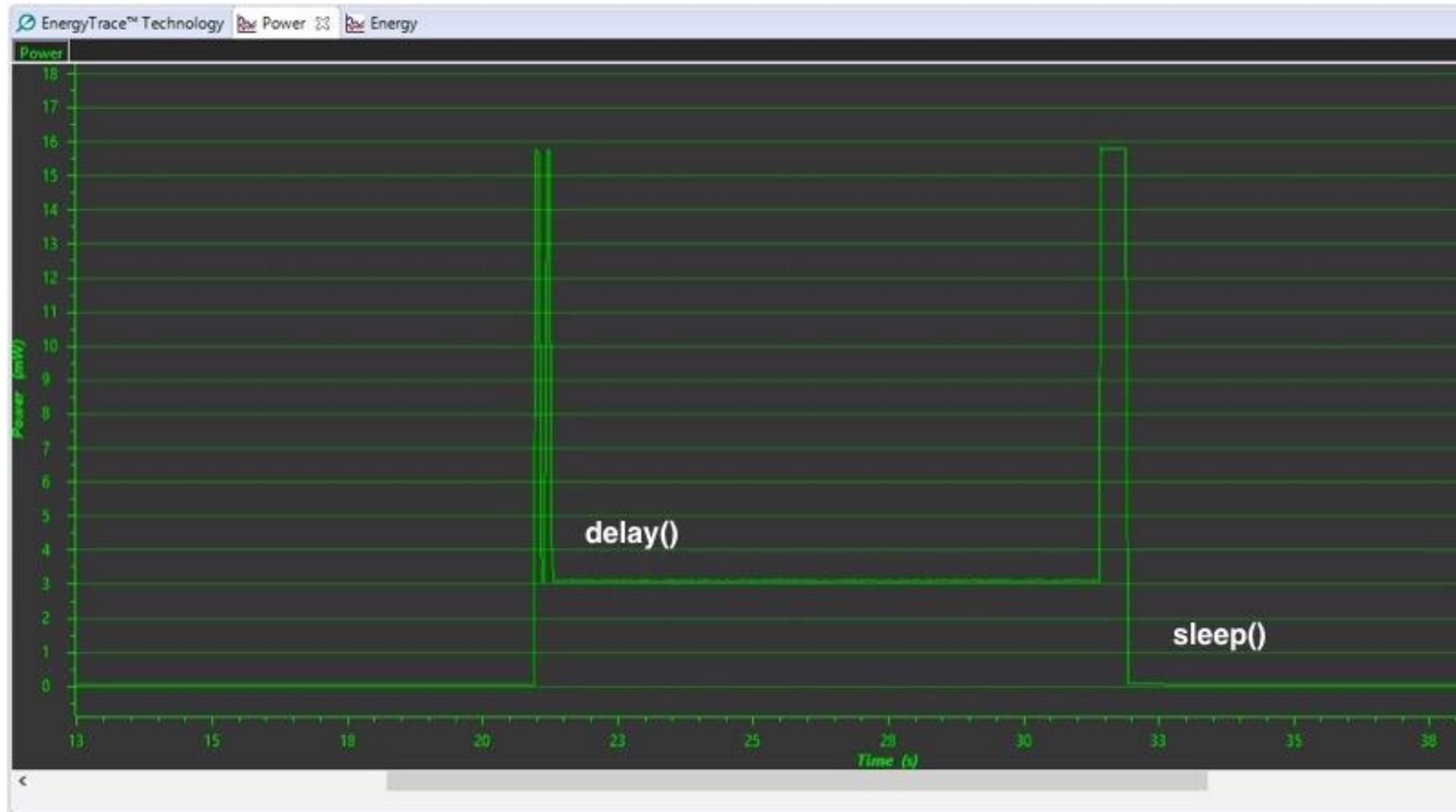
- Nonvolatile Memory
 - Memory in Computing
 - nRF52 Non-Volatile Memory Controller
 - SD Card Interface
- **Energy Management**
 - Energy Sources
 - **Microbit Energy Use**
 - Intermittent Computing

Thinking about energy

- Battery energy
 - Coin cell battery: 3v at 220 mAh
 - 2x AAA battery: 3v at 1000 mAh
 - 2x AA battery: 3v at 2000 mAh
 - iPhone 11 battery: 3.7v at 3000 mAh
- nRF52833 active current: 5.6 mA (at 3v)
 - Coin cell: 40 hours -> ~2 days
 - 2x AAA: 180 hours -> ~7 days
 - 2x AA: 360 hours -> ~15 days
 - iPhone 11: 535 hours -> ~22 days
- So how does any of this work???



Sleep mode power draw



Microcontroller sleep modes

- Sleep mode
 - Processor stops running but memory values are preserved
 - Most peripherals are disabled
 - Continues until an interrupt occurs and wakes the microcontroller
 - Usually a timer or GPIO input
- nRF52833 sleep mode current: 1.8 μA (GPIO port event only)
 - Coin cell: 122222 hours -> ~5000 days -> ~14 years
- Low-power systems shoot for less than 1% duty cycle
 - Average current of ~100 μA or less
 - Warning: other stuff on the board counts!!
 - LEDs are 1-10 mA each... Power is not a concern of the Microbit

Microbit current draw (microcontroller)

- Active CPU
 - 5.6 mA (executing from Flash)
 - 1.8 μ A (sleep mode with RAM retention)
- Transmitting RF packet
 - 15.5 mA (+8 dBm)
- Other peripherals
 - SAADC: 1.37 mA
 - Timer: 729 μ A (for any Timer peripheral)
 - I2C: 6.6 mA (pull-down resistors when transmitting 0 bit)
 - Everything else is handfuls of μ A

Microbit current draw (non-microcontroller)

- KL27 (JTAG interface microcontroller)
 - 2 μ A sleep, 8 mA active
- Speaker
 - 0-27.5 mA (changes with input signal)
- Microphone
 - 0-120 μ A (activated with GPIO pin)
- Accelerometer/Magnetometer
 - 2-212 μ A (depends on sensing rates, 200 is magnetometer)
- LEDs
 - 0-230 mA (can be activated individually)
- Everything else
 - 0-1 mA (mostly due to pull-up resistors)

Max and min current for Microbit

- Maximum current: 280 mA at 3.3 volts (~ 1 W)
 - With *everything* active
 - Well within limits of regulator
- Minimum current
 - ~ 15 mA (always-on power LED)
 - If you removed the power LED:
 - < 100 μ A (with everything off)

nRF52 sleep mode

- Triggered with assembly instruction
 - WFI (Wait For Interrupt) or WFE (Wait For Event)
- Stops processor until woken by interrupt, exception, or event
- On nRF52 automatically disables high frequency clock if unneeded

```
__attribute__((always_inline)) __STATIC_INLINE void __WFI(void)
{
    __ASM volatile ("wfi");
}
```

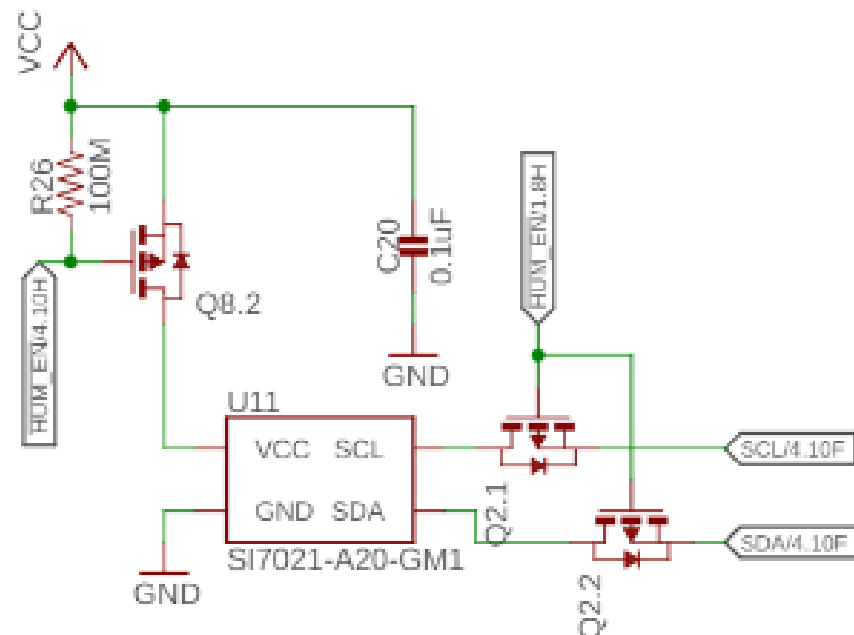
Outline

- Nonvolatile Memory
 - Memory in Computing
 - nRF52 Non-Volatile Memory Controller
 - SD Card Interface
- **Energy Management**
 - Energy Sources
 - Microbit Energy Use
 - **Intermittent Computing**

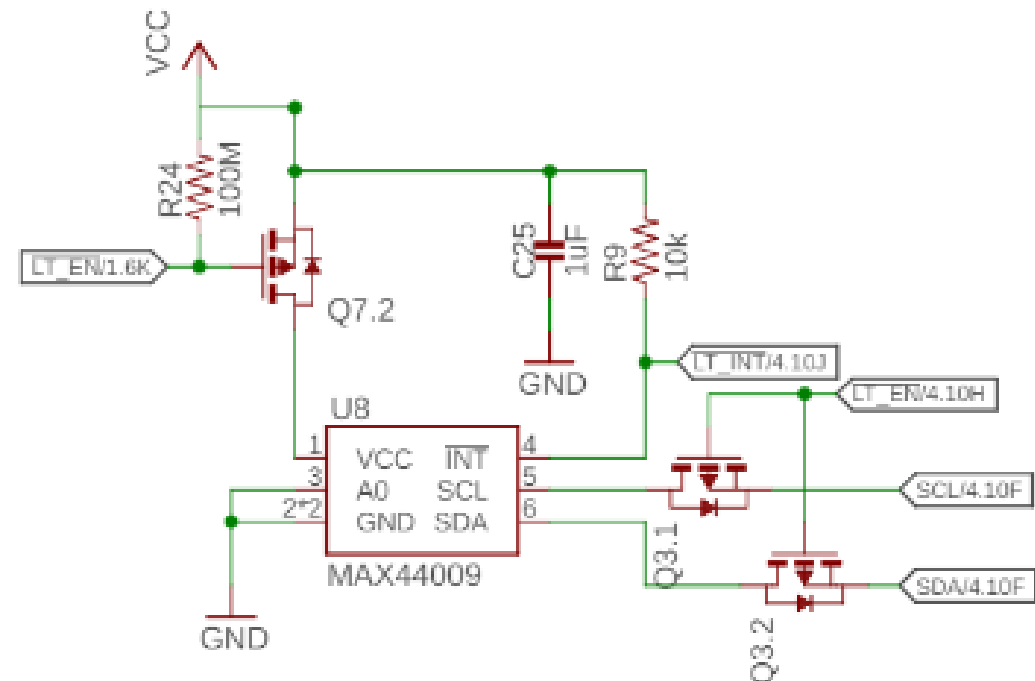
Reducing energy consumption even further

- If sleep isn't enough, you can power things off completely
 - Transistor can be used to turn off the sensor

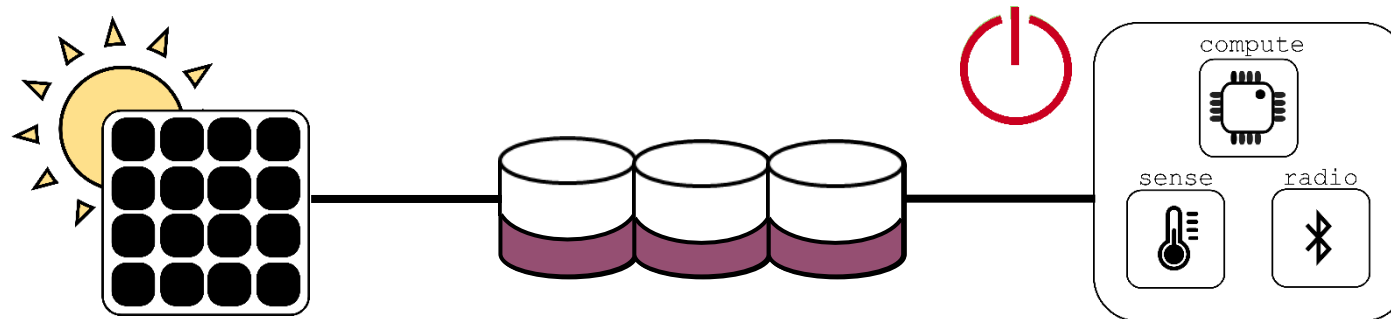
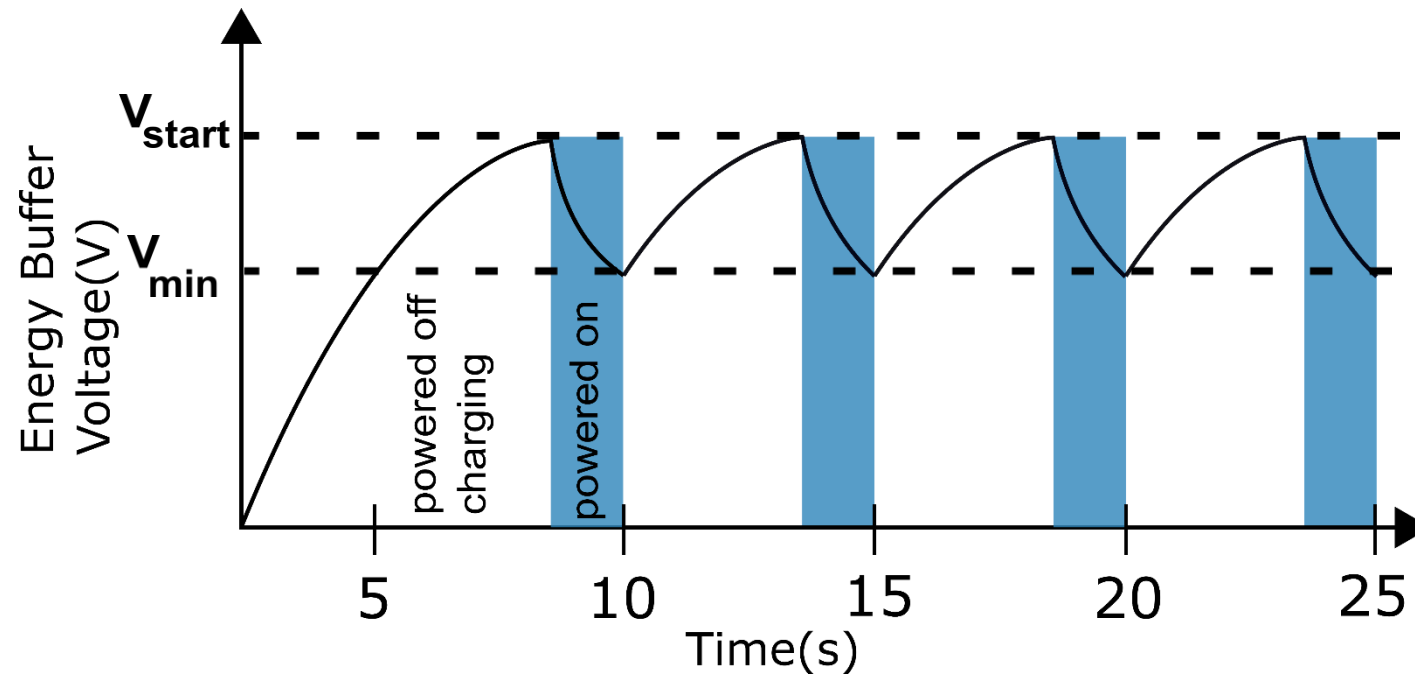
Humidity



Ambient Light



Energy harvesting can lead to intermittent computing



Disabling the microcontroller

- Even 2 μA sleep current might be too much for energy harvesting
 - Can turn off microcontroller periodically
 - Enable it again once VCC returns
- Problem: how do you write software to deal with intermittency?
 - Run-to-completion with relatively quick code
 - Initialize, sample sensor, send packet, turn off again
 - Code checkpointing
 - Save state from code and restore when power resumes
 - Might be as little as which state the system is in, plus some samples
 - Might be as much as saving entire stack state
 - Needs low-energy, nonvolatile storage (FRAM or MRAM help!)

Programs may not finish

```
int process() {  
    count++;  
    buf[count] = accel();  
    avg = sum(buf)/count;  
    transmit(avg);  
}
```

```
count++  
buf[count] = accel()  
Power fail
```

Execution Time



Programs may not finish

```
int process() {  
    count++;  
    buf[count] = accel();  
    avg = sum(buf)/count;  
    transmit(avg);  
}
```

```
count++  
buf[count] = accel()  
Power fail
```

```
count++;  
buf[count] = accel()  
Power fail
```

▪
▪
▪

Execution Time



Programs may not finish

```
int process() {  
    count++;  
    buf[count] = accel();  
    avg = sum(buf)/count;  
    transmit(avg);  
}
```

```
count++  
buf[count] = accel()  
Power fail
```

```
count++;  
buf[count] = accel()  
Power fail
```

⋮

**Need to latch execution
state periodically!**

Execution Time



Checkpointing enables progress

```
int process() {  
    count++;  
    buf[count] = accel();  
    avg = sum(buf)/count;  
    transmit(avg);  
}
```

Execute with
checkpoints

```
count++  
buf[count] = accel()  
Power fail
```

```
count++;  
buf[count] = accel()  
Power fail
```

⋮

**Need to latch execution
state periodically!**

Execution Time

```
count++  
Checkpoint  
buf[count] = accel()  
Power fail
```

Checkpointing enables progress

```
int process() {  
    count++;  
    buf[count] = accel();  
    avg = sum(buf)/count;  
    transmit(avg);  
}
```

Execute with
checkpoints

```
count++  
buf[count] = accel()  
Power fail
```

```
count++;  
buf[count] = accel()  
Power fail
```

⋮

Need to latch execution
state periodically!

Execution Time

```
count++  
Checkpoint  
buf[count] = accel()  
Power fail
```

```
buf[count] = accel()  
avg = sum(buf)/count  
Checkpoint  
transmit-  
Power fail
```

Checkpointing enables progress

```
int process() {  
    count++;  
    buf[count] = accel();  
    avg = sum(buf)/count;  
    transmit(avg);  
}
```

Execute with
checkpoints

```
count++  
buf[count] = accel()  
Power fail
```

```
count++;  
buf[count] = accel()  
Power fail
```

⋮

Need to latch execution
state periodically!

Execution Time

```
count++  
Checkpoint  
buf[count] = accel()  
Power fail
```

```
buf[count] = accel()  
avg = sum(buf)/count  
Checkpoint  
transmit-  
Power fail
```

```
transmit(avg)
```

Checkpointing goals

- Have the compiler automatically insert checkpoints as needed
 - Human doesn't have to think about them when programming
- Limit checkpointing overhead while maximizing forward progress
 - Checkpointing will take time to perform, so want to do it rarely
 - Rarer checkpoints mean more progress is lost in average outage
 - Need to compromise on the two based on available energy

Outline

- Nonvolatile Memory
 - Memory in Computing
 - nRF52 Non-Volatile Memory Controller
 - SD Card Interface
- Energy Management
 - Energy Sources
 - Microbit Energy Use
 - Intermittent Computing

Outline

- **Bonus: Task/Event Chaining with PPI**

Software stops when the processor does, but peripherals continue

- Problem: when the processor is off, no code is running
- Solutions
 - Peripherals can wake it up again
 - Can probably go for milliseconds to minutes without any actions
 - Timer interrupt can wake processor to do things
 - Have hardware handle some parts in the background without the processor's involvement
 - DMA
 - PPI

Controlling peripherals while processor sleeps

- DMA (Direct Memory Access)
 - Set up a pointer to memory and a length
 - Peripheral can load/store memory without processor's involvement
 - Usually use completion interrupt to wake processor
- PPI (Programmable Peripheral Interconnect)
 - Any Event can be tied to any Task within the nRF52
 - Allows for complicated actions to be chained together

nRF52 Tasks and Events

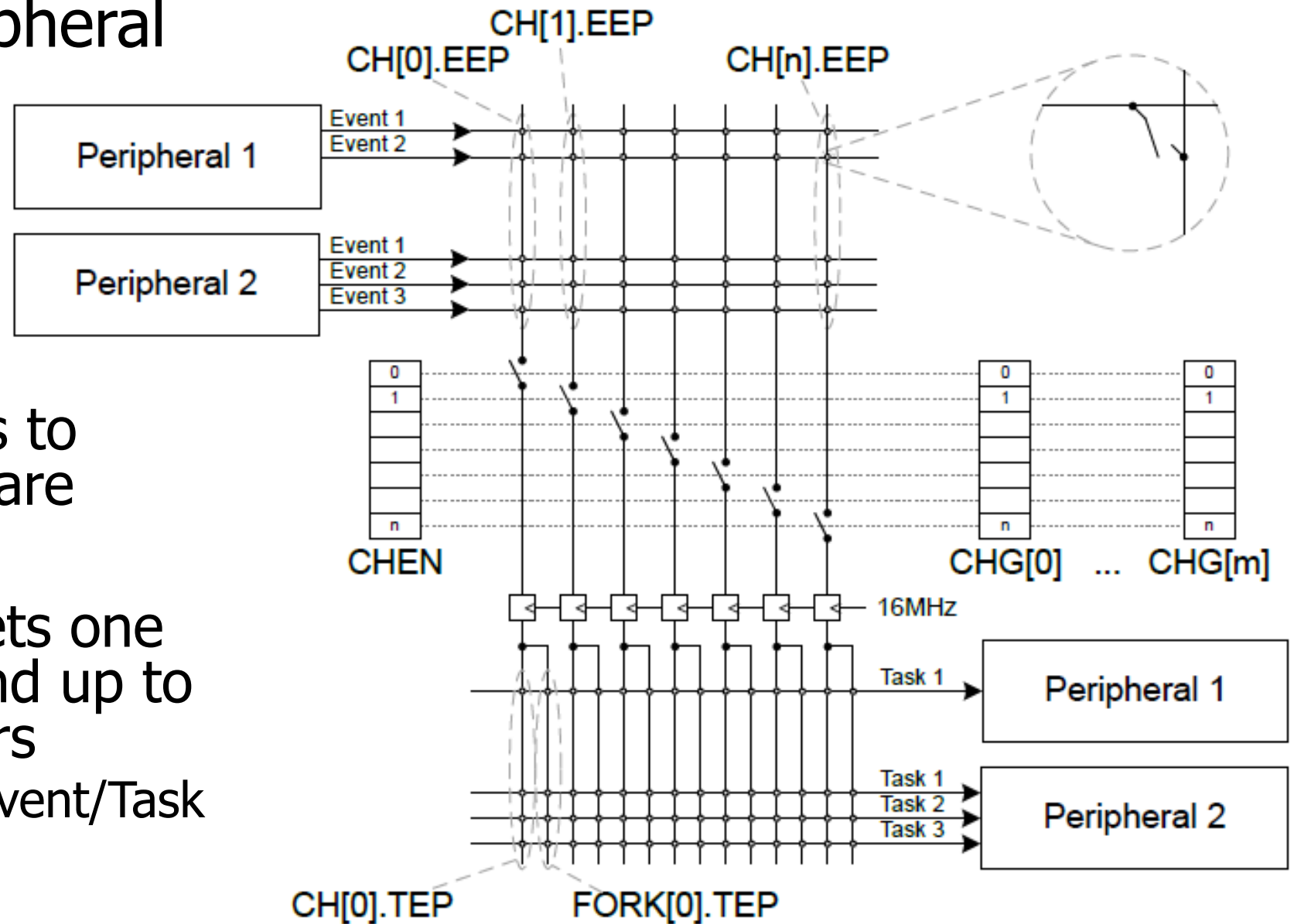
- Tasks are used to perform some operation
 - Often written to by software
- Events change value when some change in status occurs
 - Often used to trigger interrupts
- PPI peripheral can connect any TASK to any EVENT

Example: Timer peripheral

Register	Offset	Description
TASKS_START	0x000	Start Timer
TASKS_STOP	0x004	Stop Timer
TASKS_COUNT	0x008	Increment Timer (Counter mode only)
TASKS_CLEAR	0x00C	Clear time
TASKS_SHUTDOWN	0x010	Shut down timer
TASKS_CAPTURE[0]	0x040	Capture Timer value to CC[0] register
TASKS_CAPTURE[1]	0x044	Capture Timer value to CC[1] register
TASKS_CAPTURE[2]	0x048	Capture Timer value to CC[2] register
TASKS_CAPTURE[3]	0x04C	Capture Timer value to CC[3] register
TASKS_CAPTURE[4]	0x050	Capture Timer value to CC[4] register
TASKS_CAPTURE[5]	0x054	Capture Timer value to CC[5] register
EVENTS_COMPARE[0]	0x140	Compare event on CC[0] match
EVENTS_COMPARE[1]	0x144	Compare event on CC[1] match
EVENTS_COMPARE[2]	0x148	Compare event on CC[2] match
EVENTS_COMPARE[3]	0x14C	Compare event on CC[3] match
EVENTS_COMPARE[4]	0x150	Compare event on CC[4] match
EVENTS_COMPARE[5]	0x154	Compare event on CC[5] match

nRF52 PPI peripheral

- Connects Events to Tasks via hardware
- Each channel gets one Event pointer and up to two Task pointers
 - Must point to Event/Task registers



Example PPI use case

- Automatic high-speed ADC sampling
- Software configures and sleeps
 - ADC (buffer and enable)
 - Timer (prescaler, compare value, short from compare to clear, and start)
- PPI: When Timer fires (EVENTS_COMPARE[0]),
 - Sample ADC (TASKS_SAMPLE)
- PPI: When ADC buffer full (EVENTS_END),
 - Stop Timer (TASKS_STOP)
 - Fork: wake processor (via software interrupt from EGU)