NC STATE UNIVERSITY

Architecture Design Process (Applied to the Expansion Shield)

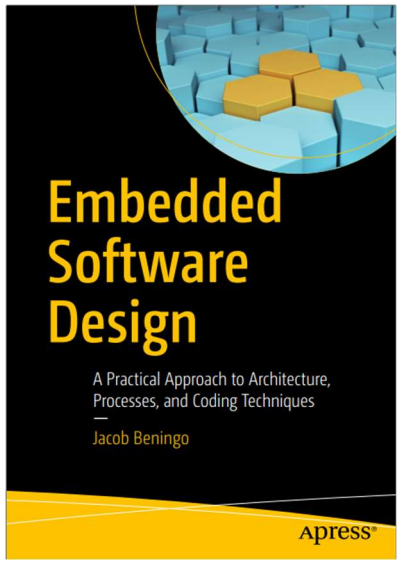
rev. 10/22/2024

NC STATE UNIVERSITY

Overview

- What is an architecture? High-Level Design
  - Components (that we care about)
  - Their interactions (that we care about)
- Overview
  - Decomposition

- References
  - **Embedded Software Design: A Practical Approach to Architecture, Processes, and Coding Techniques**, Jacob Beningo

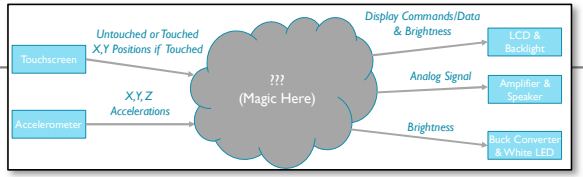


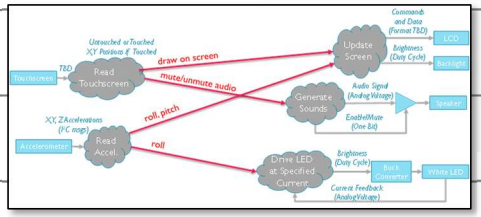
NC STATE UNIVERSITY

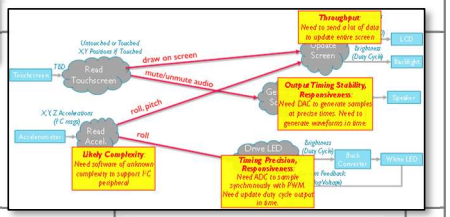
Architecture Design Process Overview

High-Level Design

- Identify Major Components
  - Identify Input/Output Components and Data
    - Refine I/O Data Comm/Sync with Critical Data Requirements and Formats (as known)
  - Identify Major Separate Processing Activities
  - Identify Known Comm/Sync between Major Activities
  - Look for Difficult Performance Requirements and Other Risks in Processing Activities, Synchronization and Communication

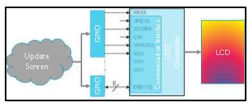


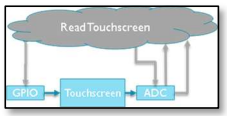


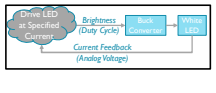


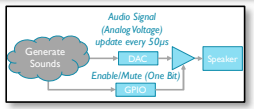
Detailed Design

- Decompose Each Major Activity into Processes in HW (peripherals) and SW
  - Refine Design for Each Major Activity: Process characteristics, comm/sync (internal, external)
  - Update Difficult Performance Requirements and Other Risks

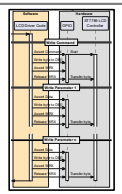


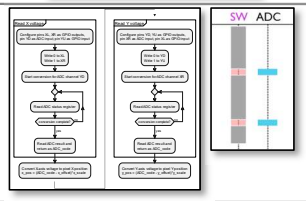


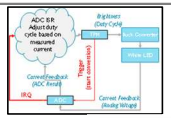


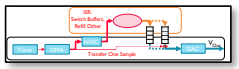


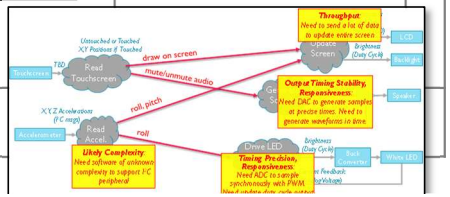
?











NC STATE UNIVERSITY

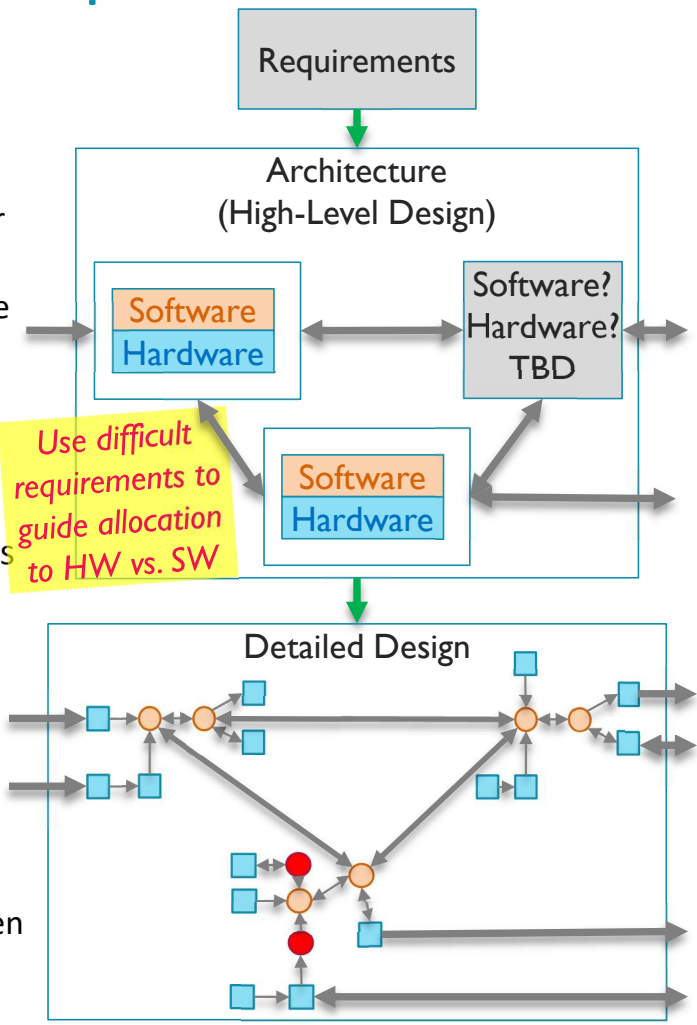
Architecture Design Process Overview

High-Level Design	Identify Major Components					
	Identify Input/Output Components and Data					
	Refine I/O Data Comm/Sync with Data Requirements and Formats (as known)					
	Identify High-Level (Major) Separate Processing Activities	Read Touchscrn.	Generate Sounds	Read Accel.	CC LED Driver	Update LCD
	Identify High-Level Comm/Sync between Major Activities) (as known)					
	Look for Difficult Performance Requirements and other Risks at High-Level in Activities, Comm/Sync					
Detailed Design	Decompose Each Major Activity into Processes in Hardware (peripherals) and Software	ADC GPIO				
	Refine Design for Each Major Activity – Process characteristics, communication/synchronization (internal, external)					
	Update Difficult Performance Requirements and other Risks	OH				

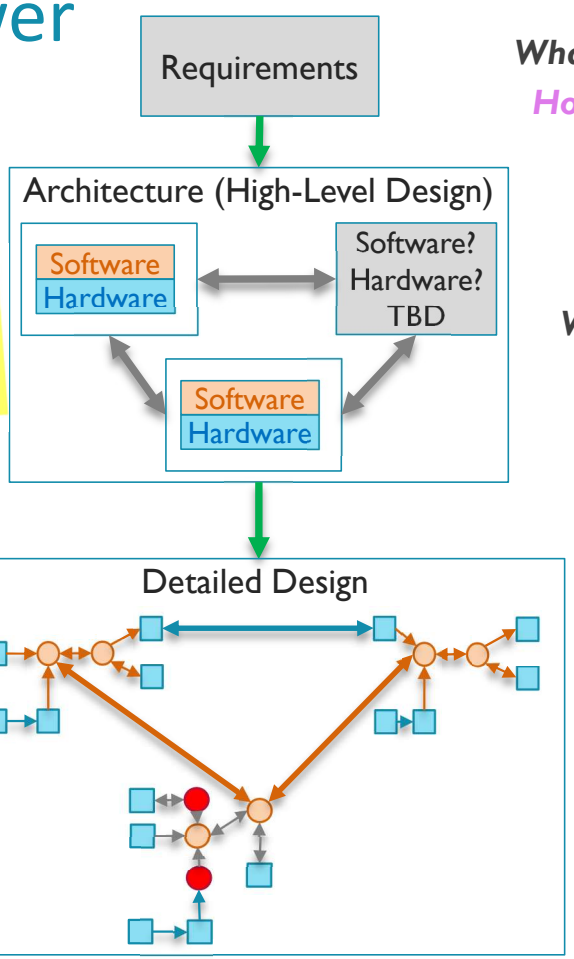
How to Create an Architecture Using Decomposition

Outside-In Approach, similar to Top-Down

- Determine requirements
- Define **major components**
  - Input components and data
  - Output components and data
  - Refine with requirements, formats if known
- Identify **major processing activities** per component
- Identify high-level **synchronization & communication** behavior
  - Where? System inputs, between activities, system outputs
  - What? Triggering, data flow, restrictions (dependency, mutual exclusion)
- Look for **risky areas**
  - **Performance**: raw compute speed, throughput (compute + more activities)
  - **Timing**: stability (e.g. periodicity), synchronization with events (including input-to-output response time)
  - **Internal synchronization** preventing running code straight through. Wait for HW (e.g. conversion done?)
  - **Sharing resources:** which might require waiting: data variables, peripheral HW
  - **Novelty**: Anything you haven't done before
- **Split** ("*decompose*") activities and allocate into HW and/or SW processes
  - Select HW or SW based on HW capabilities and risks found above
- Refine sync/comm relationships
  - Between major processing activities
    - Triggering, data sharing, etc.
  - Between sub-activities (processes) within each activity
    - Triggering, data sharing, sync between HW & SW



Questions to Answer



Develop architecture, identify key actors and interactions, allocate to software or hardware

Use difficult requirements to guide allocation to HW vs. SW

Refine architecture, add critical details. Repeat as needed.

What must the system do?
What are the inputs, processing activities and outputs?
How do we represent this? Can we do it graphically?
Equations, transfer functions, flow charts, state diagrams, sequence diagrams

What are the parts, and how do they interact?
Which parts are software and which are hardware?

What software parts (actors) will be used?
Threads, ISRs, state machines, callback functions

How will software parts interact (synchronizing activities and resource use, sharing data)? Buffer events?
Shared variable, mutex, event flag, semaphore, message queue

How will software parts share the CPU(s) time?
ISRs, scheduling approach (priority, preemption)

Which hardware peripherals (actors) will be used, and in which modes?

How will the peripherals interact (synchronizing activities and resource use, sharing data)? Buffer events?
Hardware trigger signals, interrupt requests

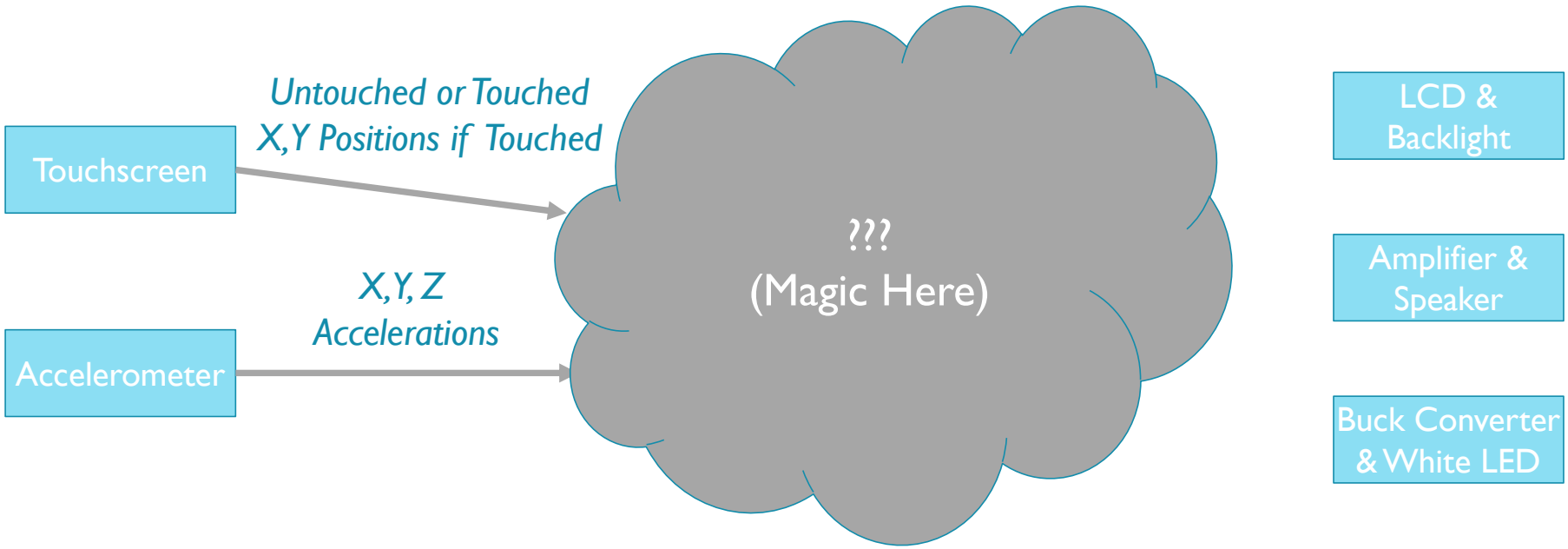
How will hardware and software interact?
Interrupts and ISRs, peripheral control/status registers

NC STATE UNIVERSITY

Identify Major Components

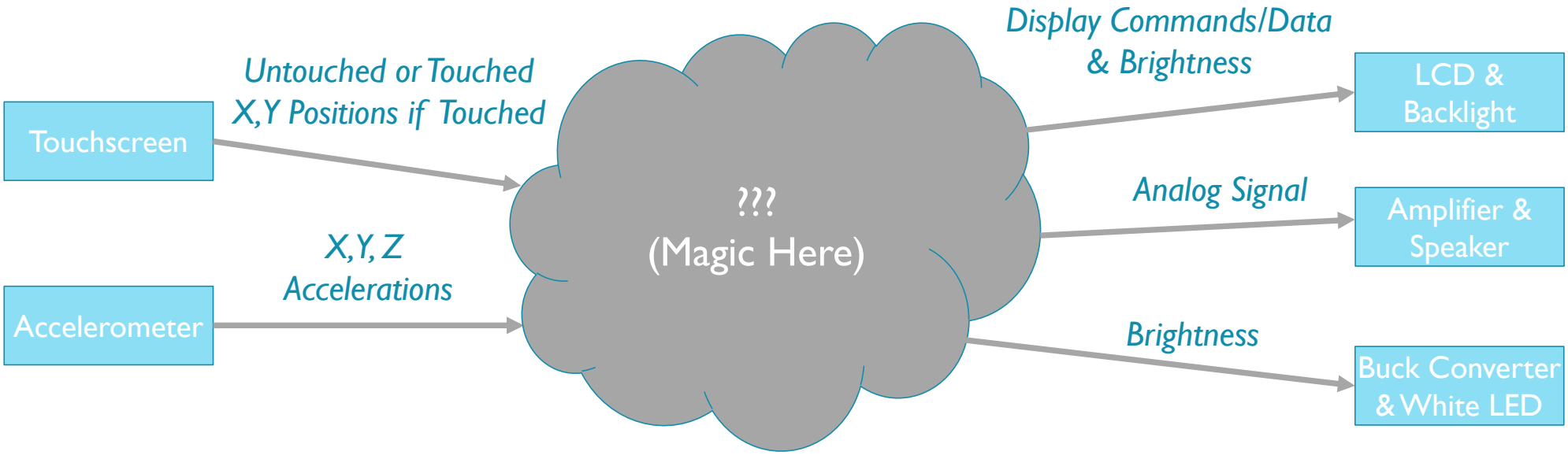
NC STATE UNIVERSITY

Identify Inputs and Describe Data

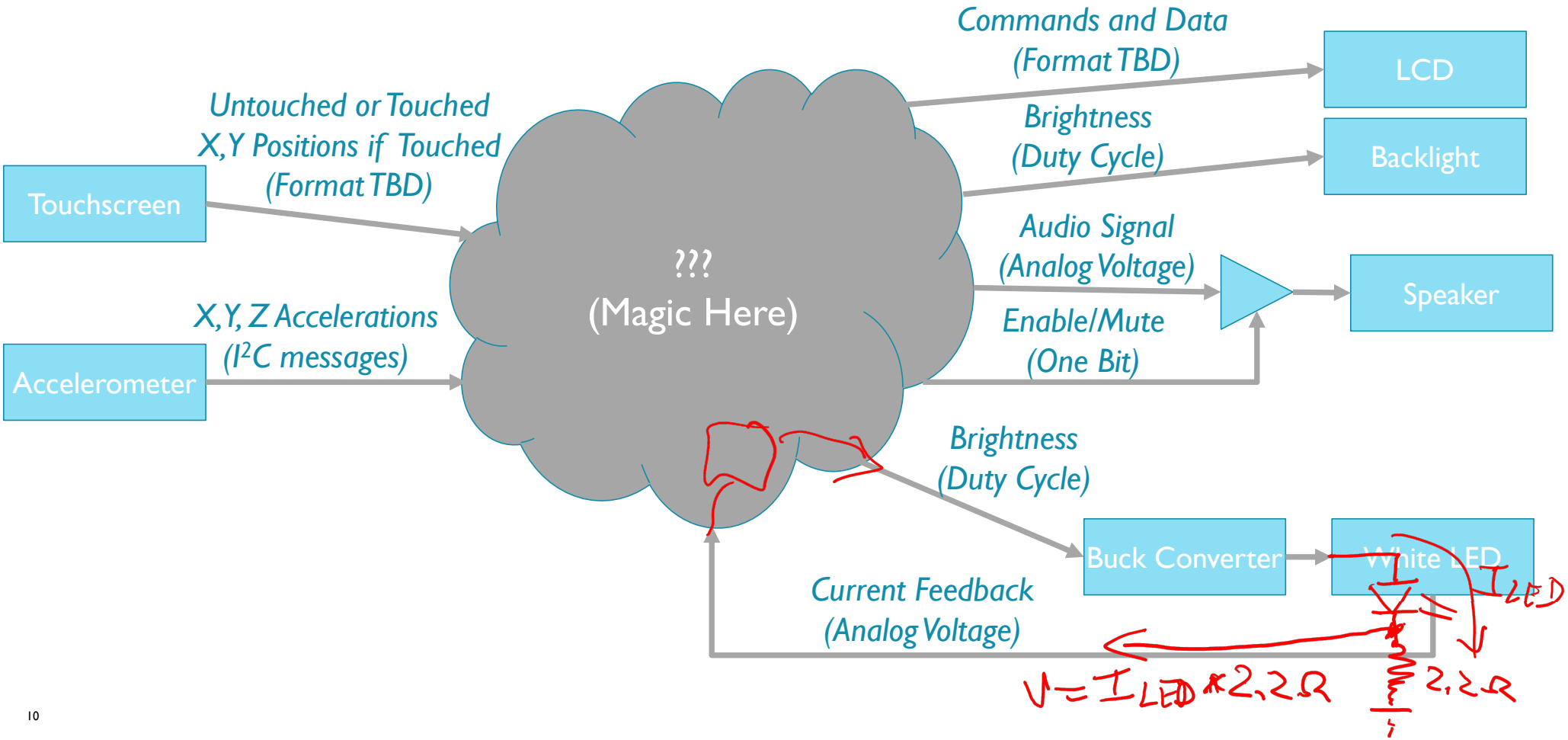


NC STATE UNIVERSITY

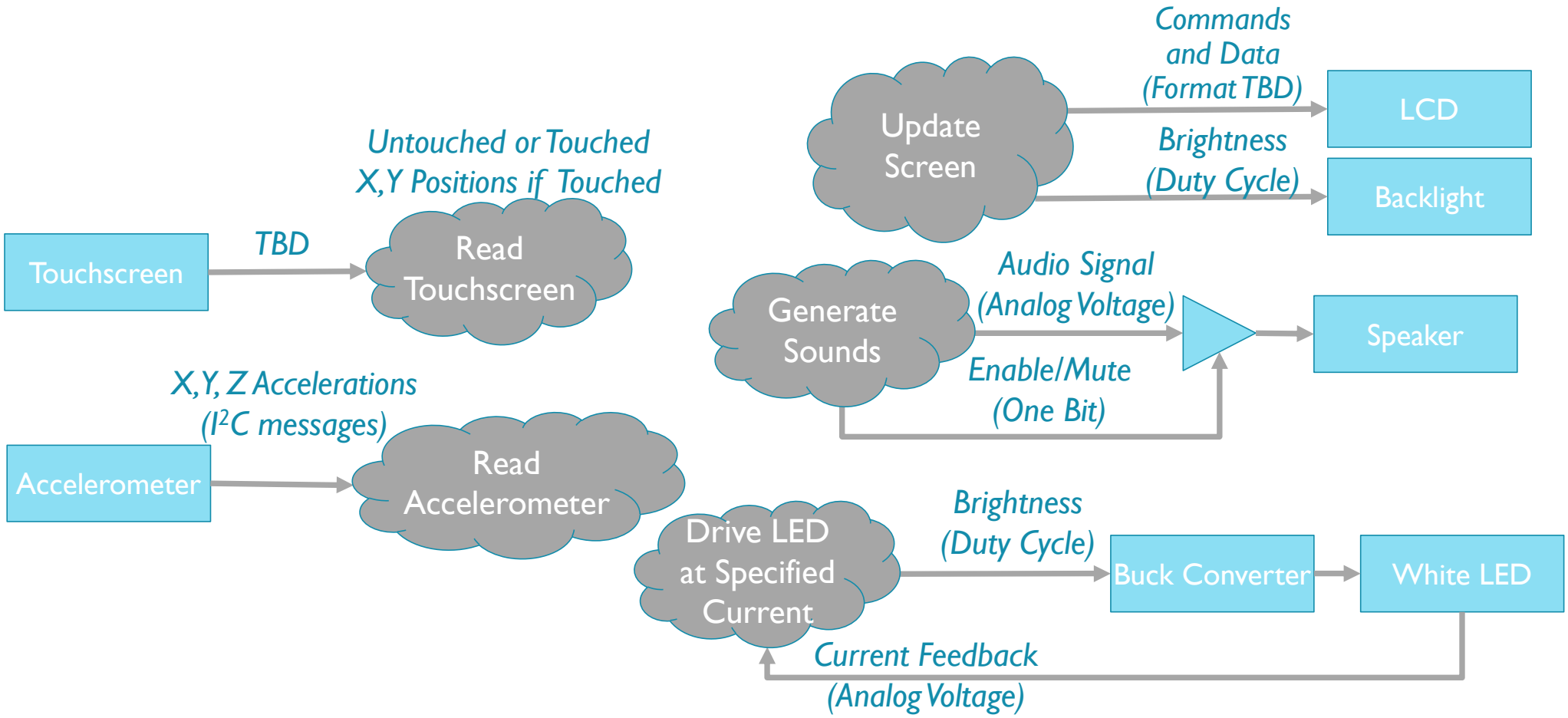
Identify Outputs and Describe Data



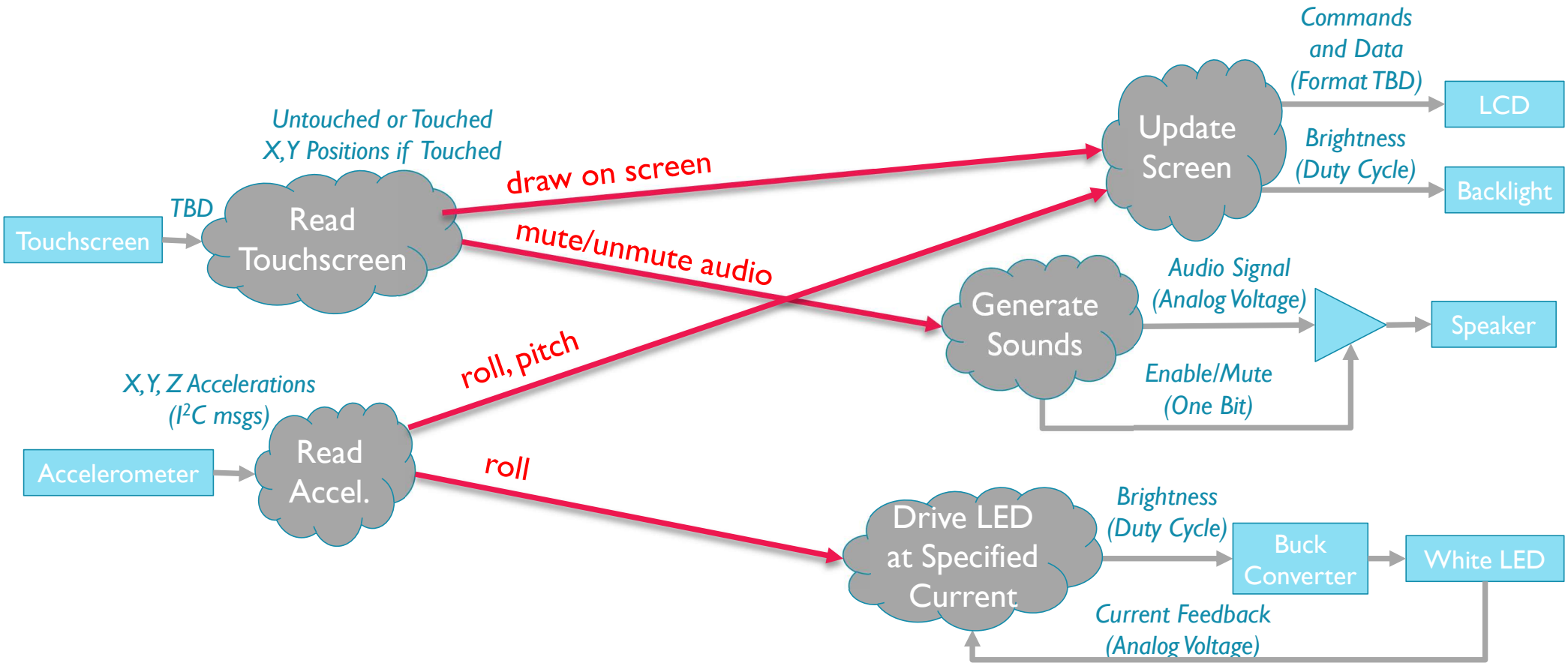
Describe I/O Comm/Sync: Known Data Requirements, Formats



Identify Major Separate Processing Activities



Identify High-Level Synchronization and Data Communication



Look for Difficult Performance Requirements and Other Risks in Processing and Comm/Sync

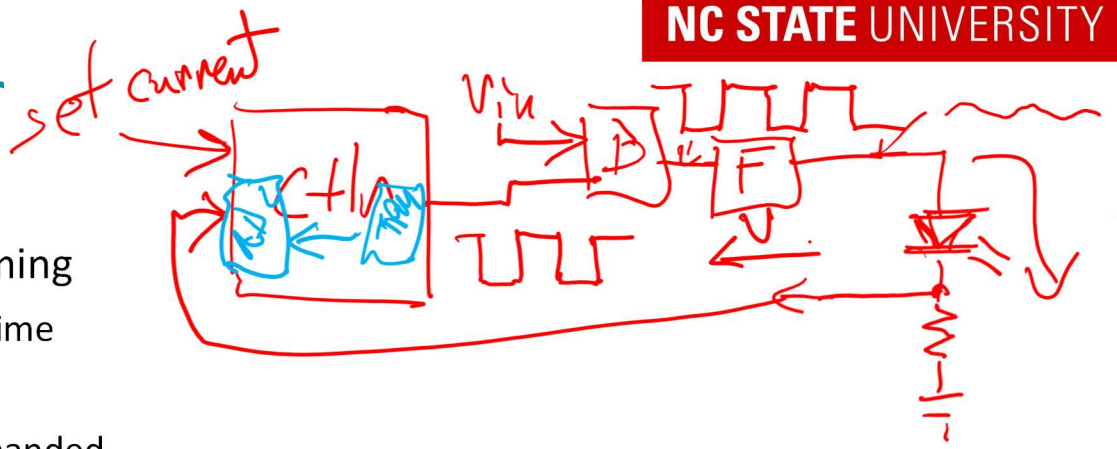
Identify any risky areas early and prioritize them

NC STATE UNIVERSITY

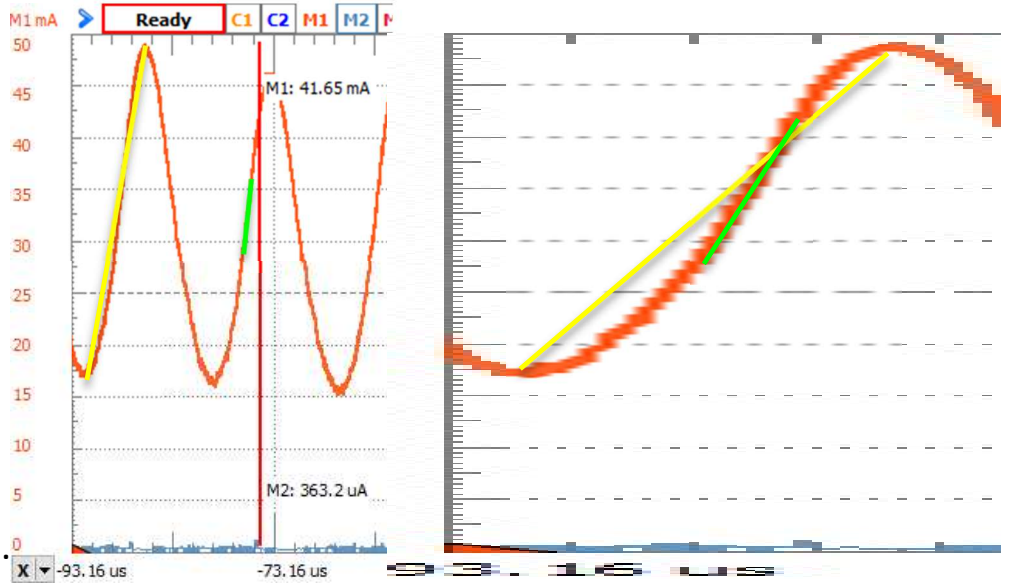
- Read Touchscreen
  - Response within 50 ms probably ok
  - How much position accuracy is needed? Depends:
    - Application: big pushbuttons vs. drawing program
    - How touched: finger vs. stylus
- Generate Sounds
  - Update DAC with 12 bit sample every 50 µs.
  - How much timing error is acceptable? Depends on response of amplifier and speaker, listener, etc.
- Accelerometer
  - Interfaces via I^2C bus. (up to 1 Mbps+), but this accelerometer maxes out at 400 kbps.
  - Part of protocol implemented in software, so will need to look closer
  - Required response time, throughput?
    - Depend on application: Tilt sensor, game controller, vibration frequency analysis

Constant-Current LED Driver

Identify any risky areas early and prioritize them

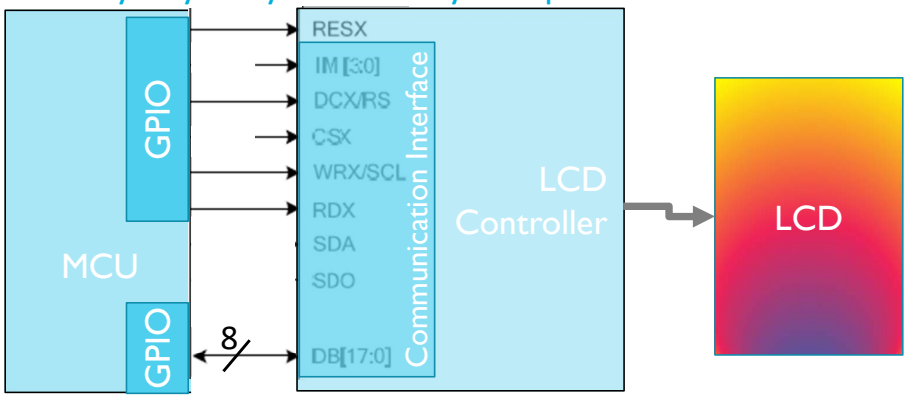


- Control system designed to run with steady timing
  - Want to **sample input** on time, **update output** on time
  - Timing errors reduce performance of controller
    - Slower response, never reaching set point (commanded value), going unstable and even destroying hardware
- Switch-mode signals have switching ripple
  - Example: Buck converter's 80 kHz switching means LED current signal changes quickly.
    - 33 mA average current has 32 mA of ripple!
    - Initial rough estimate: Peak-to-peak
      - 17 mA to 49 mA = 32 mA in ½ period (6 µs)
      - Up to 5 mA of error per µs early/late
    - Worse near middle: 7 mA in 1 µs
  - Need to sample synchronously with switching.
  - May also have other noise to avoid (ringing, etc.).



LCD Interface

Identify any risky areas early and prioritize them



- What are required update rate, throughput? Depends on application
  - Clock, thermostat, digital photo frame, video player, game?
- LCD Module includes LCD glass and controller IC
  - LCD Controller IC uses parallel memory bus for speed
    - Up to 18 data bits wide, matches max pixel color depth (6 bits each for red, green, blue)
    - t_{write_min}, t_{read_min} = 66 ns (15 MHz)
- No external memory bus on KL25Z MCU, so make interface with "bit-banging"
  - Software uses GPIO port bits (8 data, read/write, control/data, reset) requires 5-10 instructions for read or write operation
- Quick estimates of performance bounds
  - LCD is 240 x 320 pixels (76,800),
  - Use just 16 color bits (not 18) for speed, simplicity
  - Must write 76,800 pixels * 2 bytes per pixel = 153,600 data bytes to refresh screen
  - Refresh entire LCD in 1 second? At most have 48 MHz/153,600 bytes = 312.5 CPU clock cycles per byte on average
  - Refresh in 0.1 sec? Only ~31 cycles per byte – harder
- If refresh is not fast enough with software bit-banging, examine using DMA to accelerate transfers
  - Also lets CPU do something else during transfer

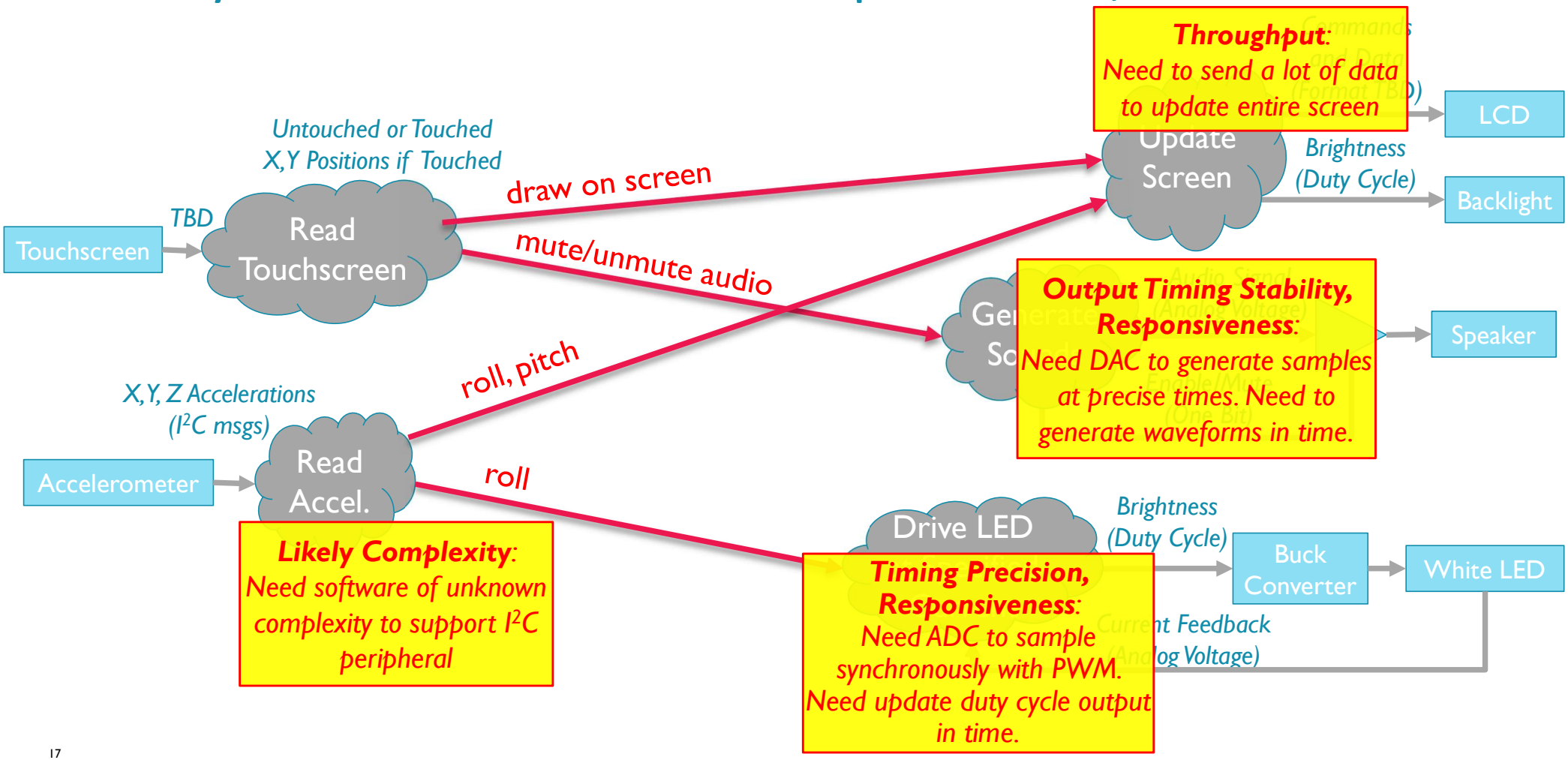
NC STATE UNIVERSITY

SD Card

Identify any risky areas early and prioritize them

- SD Card (Not used here)
  - Interface via SPI (legacy mode)
  - Card and bus support 50+ Mbps
  - KL25Z SPI peripheral limited to 12 Mbps
  - What are required throughput, response time? Depend on application:
    - digital photo frame (how long to read longest file)
    - data logger (how many bytes/second of data need to be saved)

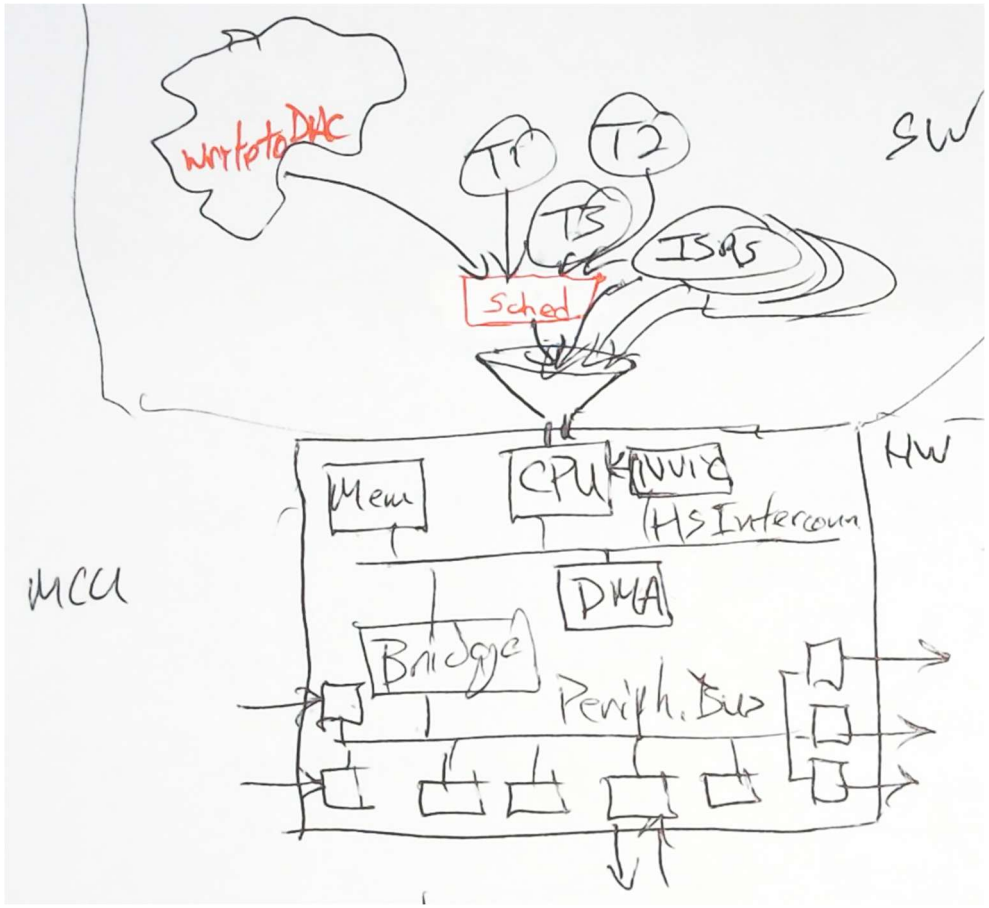
Identify Difficult Performance Requirements, Other Risks



Skip: Summary of Processing Activities

	Read Touchscreen	Generate Sounds	Read Accelerometer	CC LED Driver	Update LCD
Trigger Method	Periodic timer	Periodic timer	Periodic timer	PWM timer (periodic)	Periodic Timer
Triggering Frequency, Period	Loose.10 Hz,100 ms	20 kHz, 50 µs = 2,400 cycles	50 Hz, 20 ms	Every PWM cycle (80 kHz, 12.5 µs = 600 cycles	50 Hz, 20 ms
Timing Stability		Update DAC: Error < 50 µs. How tight?		Sample ADC: Error <<12.5 µs. Update PWM before next cycle	
Compute Speed		Calculate next sample within 50 µs		Calculate control effort, update PWM before next cycle	
Expected Throughput (Speed * data volume)		Large		Large	Large, depending on quantity, frequency of pixel changes
S/W and HW Allocation (Design Approach)	Mostly S/W. Use GPIO and ADC for interfacing	H/W Timer triggers DMA to transfer data to DAC, DMA ISR requests buffer refill. Update buffer: Software + peripherals. Sound Manager: TBD	Mostly S/W. Use I2C peripheral to talk with Accelerometer	H/W Timer (TPM) triggers ADC conversion, ADC ISR computes new control effort, updates duty cycle	Mostly S/W. Control LCD with parallel bus implemented on GPIO. Use DMA if SW is not fast enough.
Internal Delays/Blocking	Start and wait for 2 A/D conversions, ~3 µs (~150 cycles) each. ADC conversion completion flag or ADC IRQ	none	I2C communications: Limited transmission speed makes CPU wait one 26 µs (~1200 CPU cycles) chunk per byte. 9 bytes/message -> ~11,000 CPU cycles. Triggered by I2C byte completions	none	none

Decompose Activities into Hardware and Software



- Software – Thread(s) and ISRs
  - Software gives very flexible functionality
  - Stable, precise timing is expensive and difficult
    - Interrupt system and scheduler (if any) determine **what** software the CPU runs and **when**
    - Software threads very vulnerable to timing interference
    - ISRs have better timing stability, but not perfect
- Hardware – Peripherals, interconnect and DMA
  - Very stable timing
  - Functionality limited to what is built into hardware and your creativity with configuration

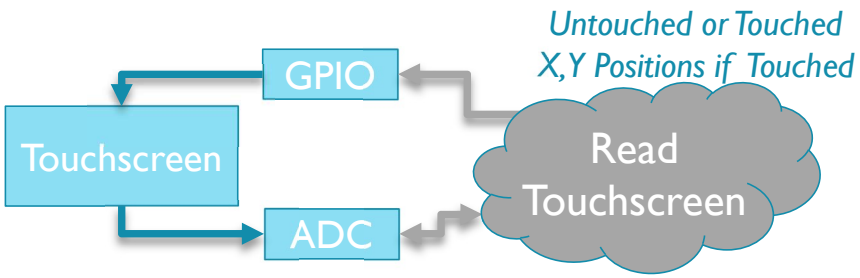
Decomposition

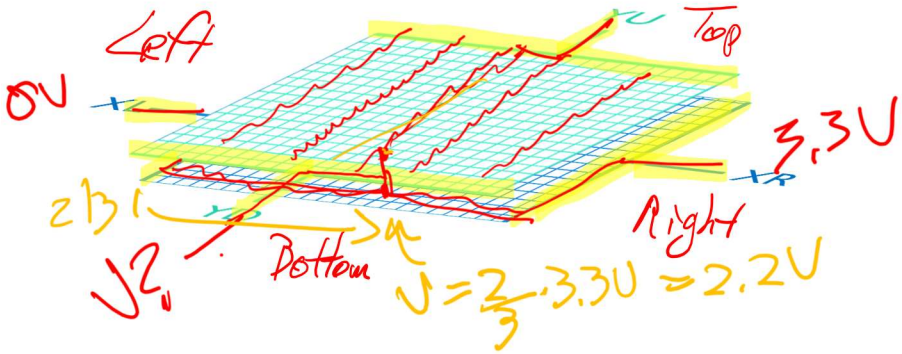
Peripheral Hardware, ISRs and Software Threads

- Key characteristics
  - Functionality: What can it do?
  - Speed: How fast does it do the work?
  - Timing stability: how predictable and steady is the timing?
  - Responsiveness: How soon does it respond to an input event?
- Use hardware peripherals where practical

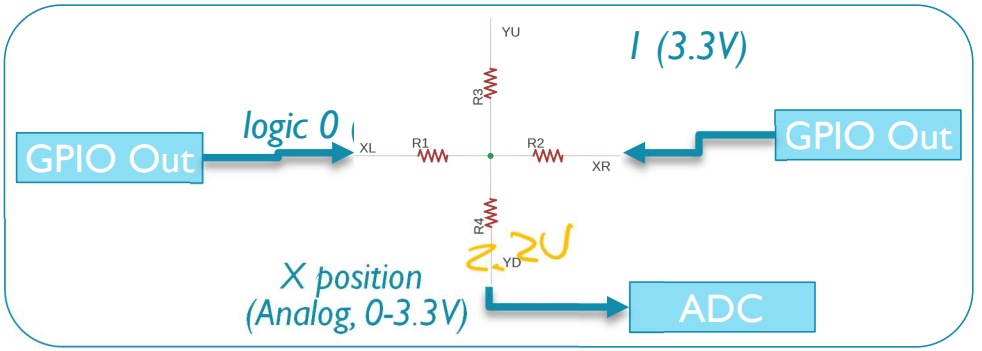
	Hardware	Software
Functionality	Limited to specific digital and analog functions	Anything you can code up (digital processing only)
Speed	Very fast	Much slower. CPU must execute instructions
Timing Stability	Excellent. Independent of code running on CPU	Poor. timing interference from other code running on CPU
Responsiveness	Excellent. Event-driven with hard-wired connections.	Poor. Polling, but can improve with interrupts, better task structure/scheduling
Configurability	Limited to hardware options, but peripherals have many application-specific features/extensions	Infinitely configurable (digital processing only)
Debuggability	Less visibility, but is probably working as designed	More visibility, but more bugs and Heisenbugs

Refine: Resistive Touchscreen

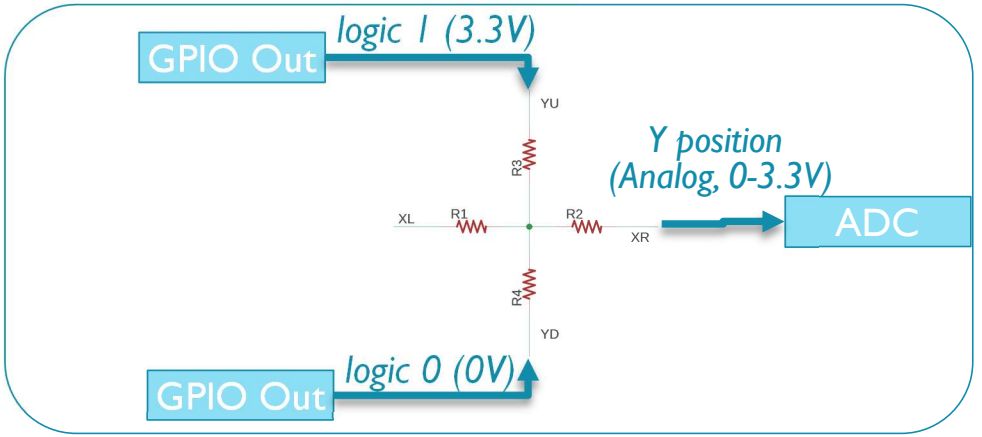




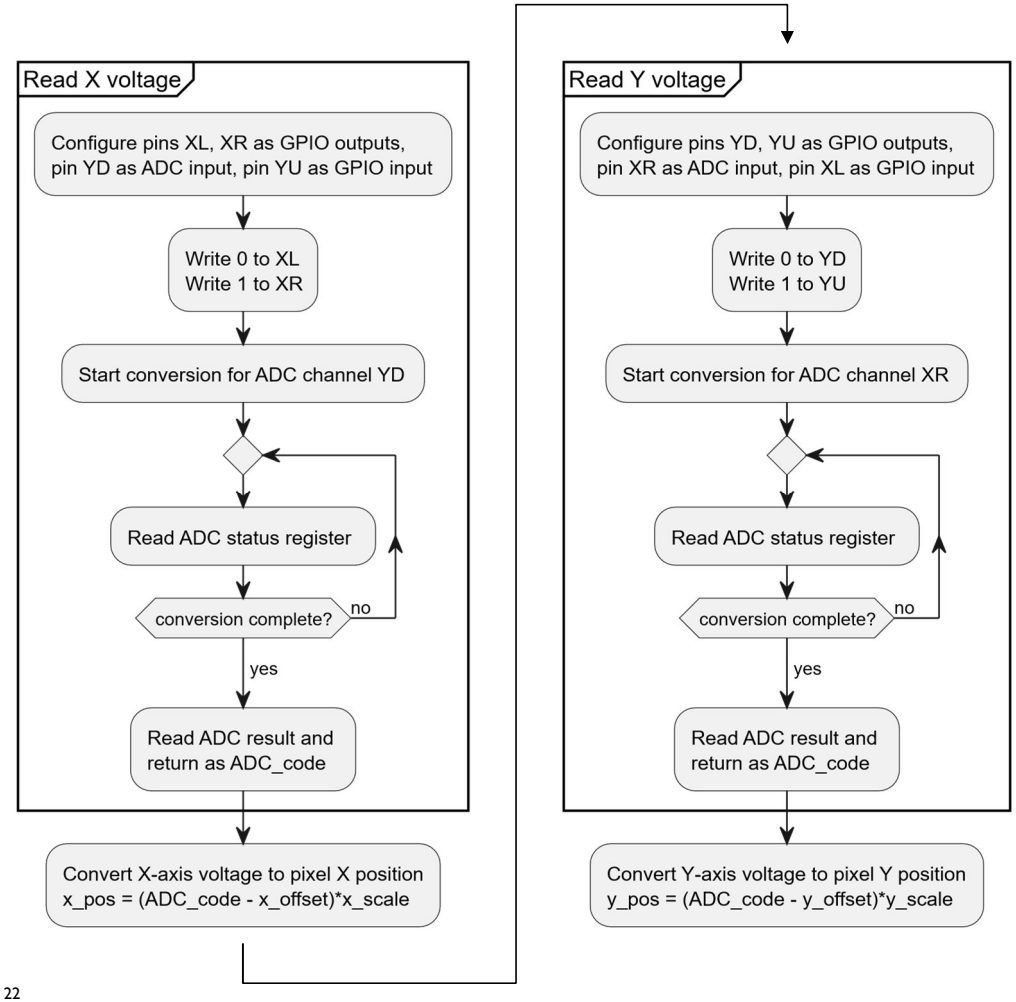
Read X Position



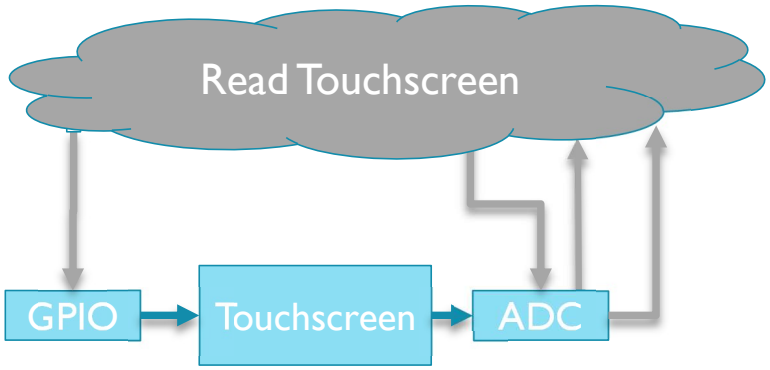
Read Y Position



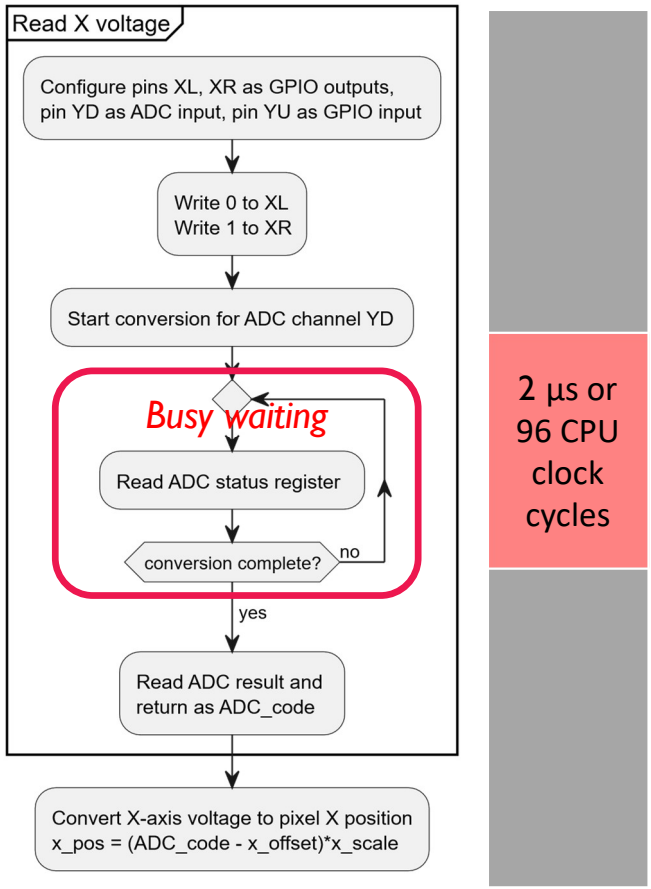
Software Overview



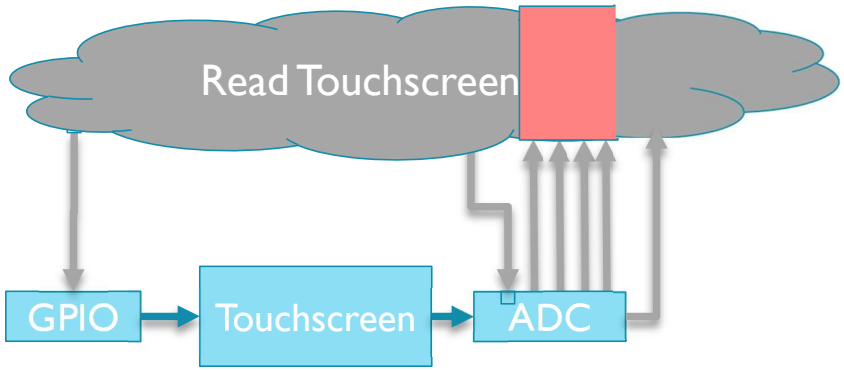
- Later enhancement: Determine if touchscreen is pressed without using ADC, saving time and power



Software Synchronization 1: Internal Delays



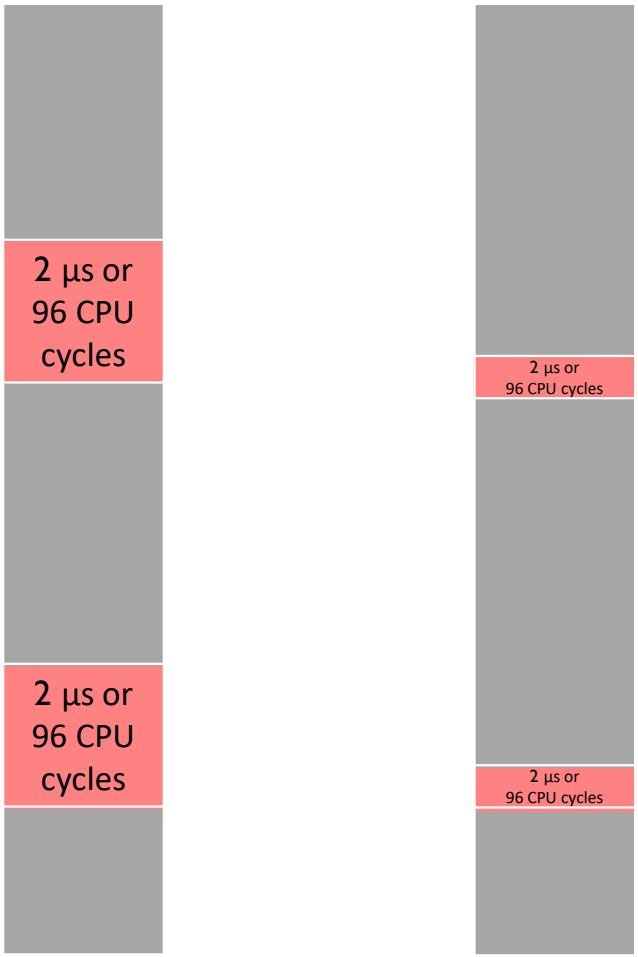
2 µs or 96 CPU clock cycles



- Any internal delays?
  - Software busy-waits for ADC conversion to complete. Example of synchronization
  - Typical A/D conversion time: ~2 µs * 48 MHz = 96 clock cycles
    - Might need more settling time for high impedance signals

NC STATE UNIVERSITY

Software Synchronization 1: Internal Delays

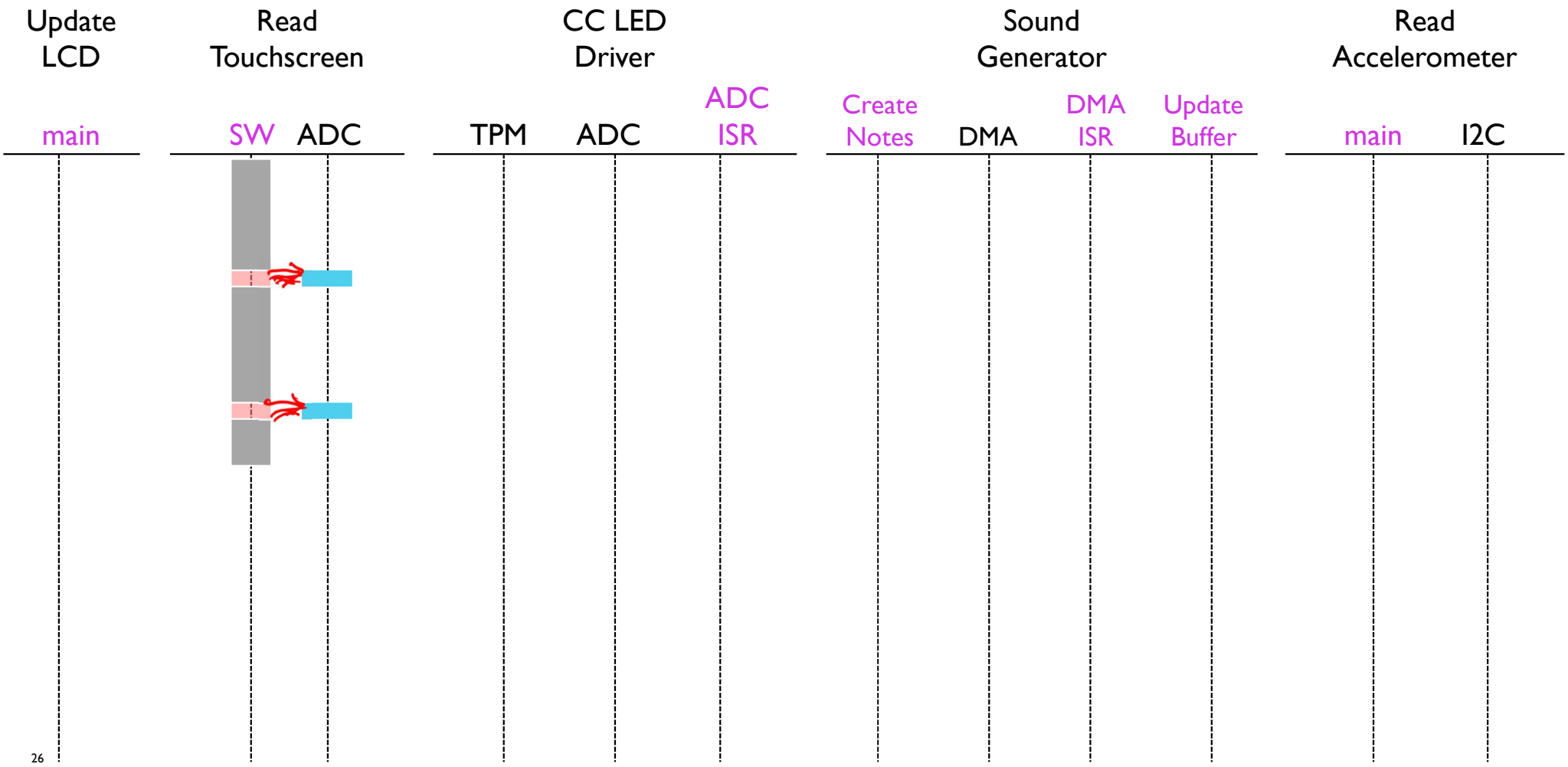


- Two conversions, 2*96 = 192 clock cycles per touchscreen read
- Is this big enough to be significant? Depends on how long the other code (gray) takes to run
  - 10 cycles? Total is 192+10 = 202 cycles. Significant, because 192/202 is large
  - 10000 cycles? Total is 192+10000 = 10192 cycles. Not significant, 192/10192 is small
- Assume reading touchscreen 10 times per second
  - Wasting 1920 clock cycles each second on busy waiting
- Try to recover that idle time for use by other processing? Probably not worthwhile.
  - #1: 1920 cycles / 48 MHz is very small part of **CPU capacity**: 0.004%
  - #2: 96 cycle chunk of idle time is not long enough compared with **scheduling overhead**
    - Interrupts: 15 (entry) + ~5 (exit) clock cycles
    - RTOS Context switch: 2 * 100+ clock cycles

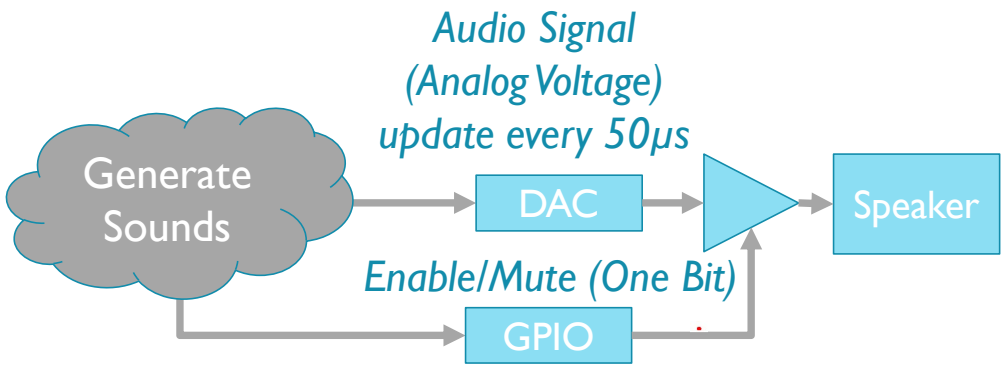
Software Synchronization 2

- Triggering question: **When** to run code to read touchscreen?
- Raises issues of **software process scheduling**
- Preview of issues covered later in scheduling
  - General Approaches: polling vs. event-driven
    - Polling runs code even if not needed
      - Loose response time (e.g. 100 ms) allows slow polling rate
      - Time/power/energy are wasted if screen not touched when code runs
    - Event-driven runs code only when needed (pressed)
      - Need scheduling mechanism to detect screen press and trigger code to run
      - Interrupts? How to use touchscreen circuit and available peripherals to generate an interrupt?
      - Scheduler with polling code?
  - Is event-driven worthwhile?
    - Since TS reading code is so short, usually not much benefit from reducing CPU cycles if other code is running
    - However, can be very useful when this code dominates compute time. Example: using touchscreen to CPU wake up from sleep mode

Timelines of Major Processing Activities and Synchronization

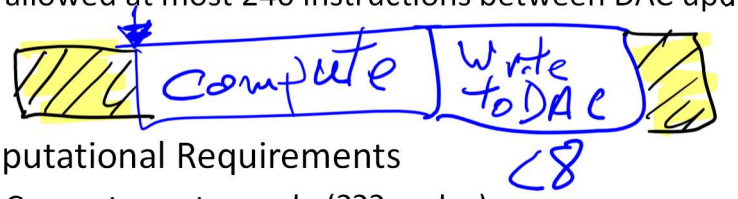


Refine: Audio Generation Requirements and Constraints



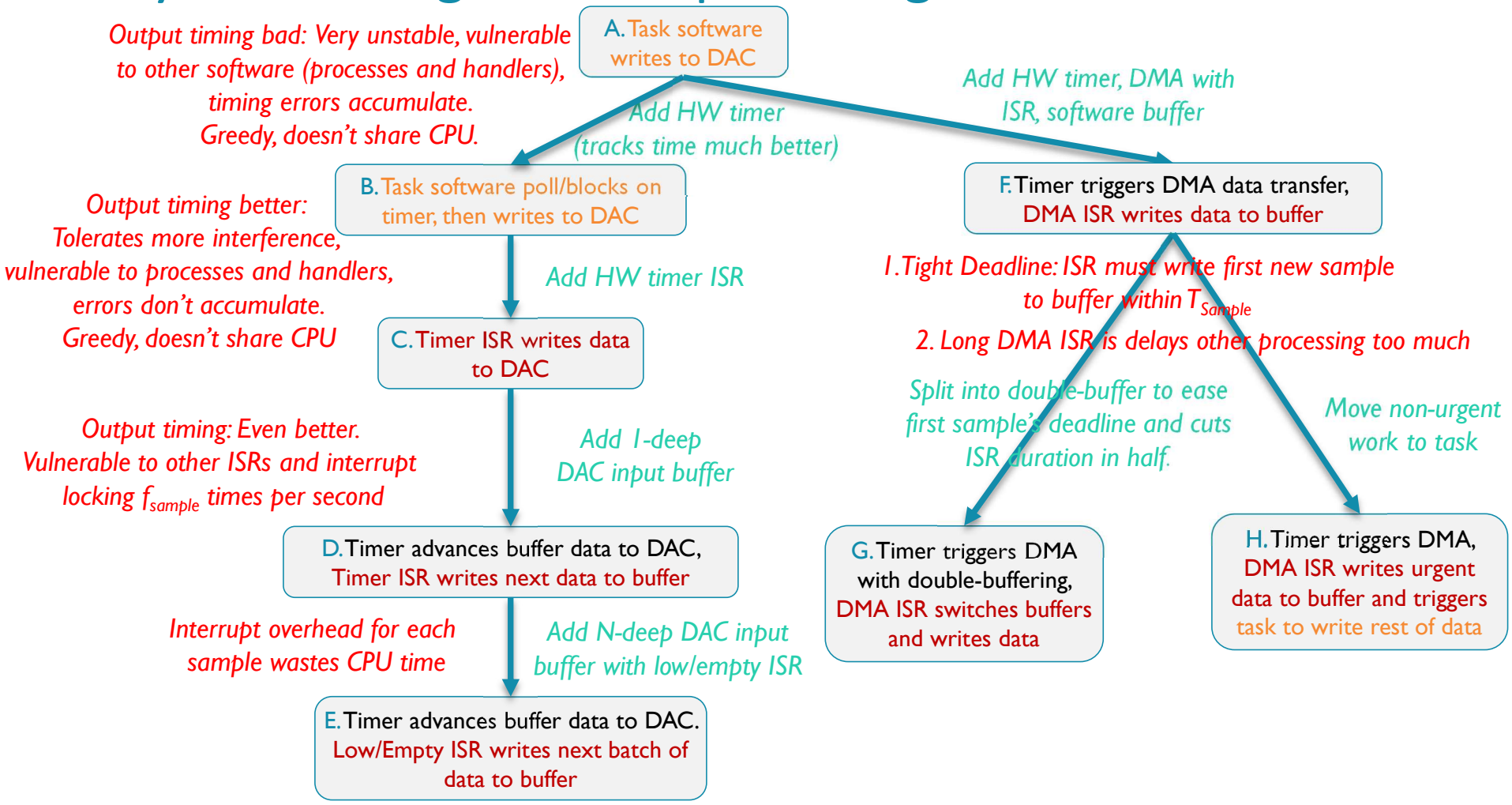
- Requirements:
  - Generate audio samples. How? TBD, defer.
  - Use DAC to update analog output (0-3.3 V) every 50 µs (20 kHz sample rate)
  - Use digital output bit to enable/disable amplifier

- Design
  - Allocate hardware: Use MCU's 12-bit DAC
  - Evaluate timing on KL25Z, f_{CPU} = 48 MHz
  - Timing Budget
    - 50 µs * 48 MHz = 240 clock cycles
    - Since MCU can execute up to 1 instruction per cycle, allowed at most 240 instructions between DAC updates

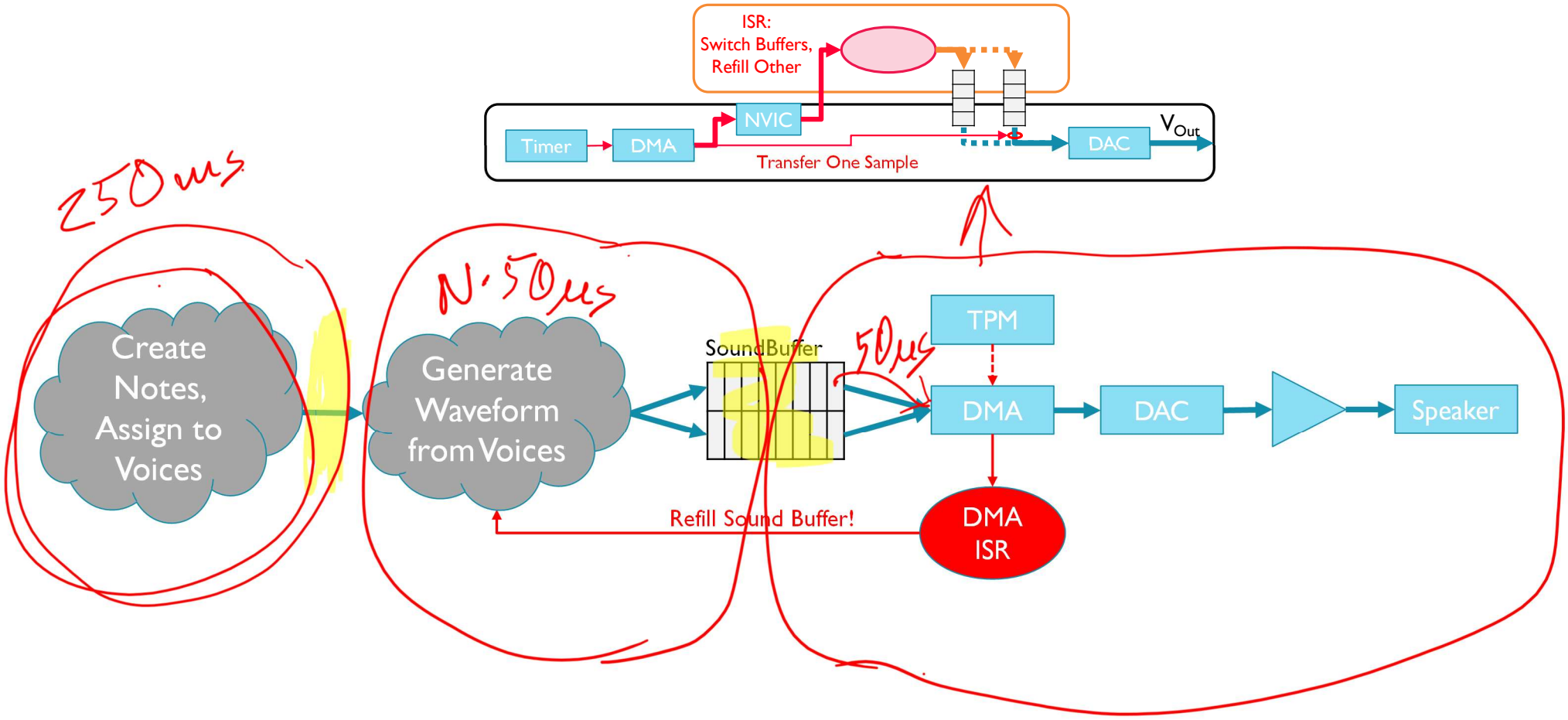


  - Computational Requirements
    - Compute next sample (??? cycles)
    - Write next sample to DAC (well under 8 cycles)
    - Overhead of switching contexts: interrupt, RTOS
- Evaluate different implementation options
  - Try to leverage hardware to simplify software to meet tough requirements: timing stability (-> HW and buffering), throughput (-> deep buffers for batch processing), responsiveness, etc.

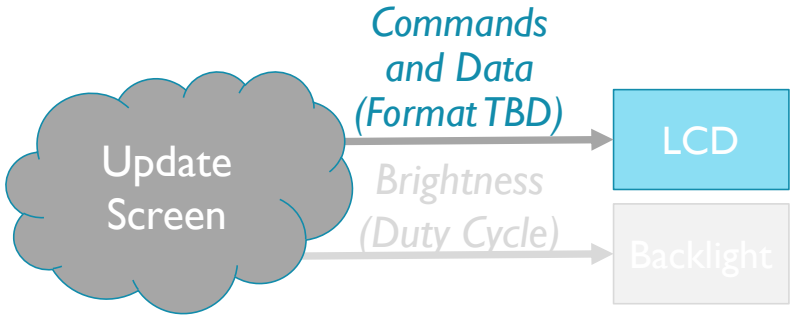
Summary of Refining DAC Output Design

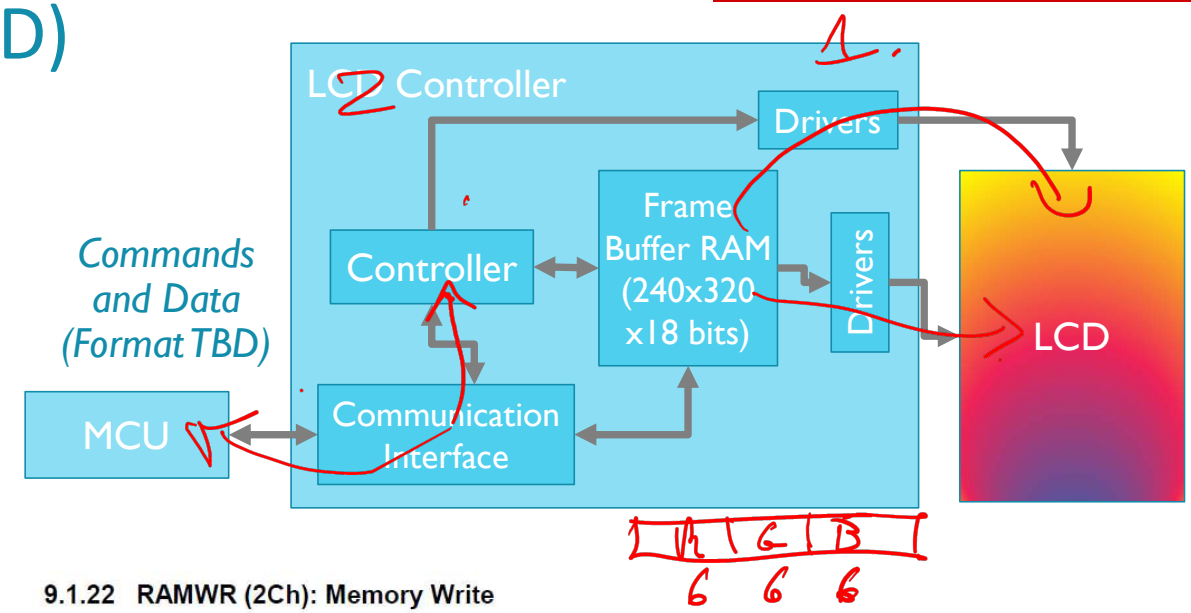


Refine: Audio Generation HW & SW



Refine: Update Screen (LCD)



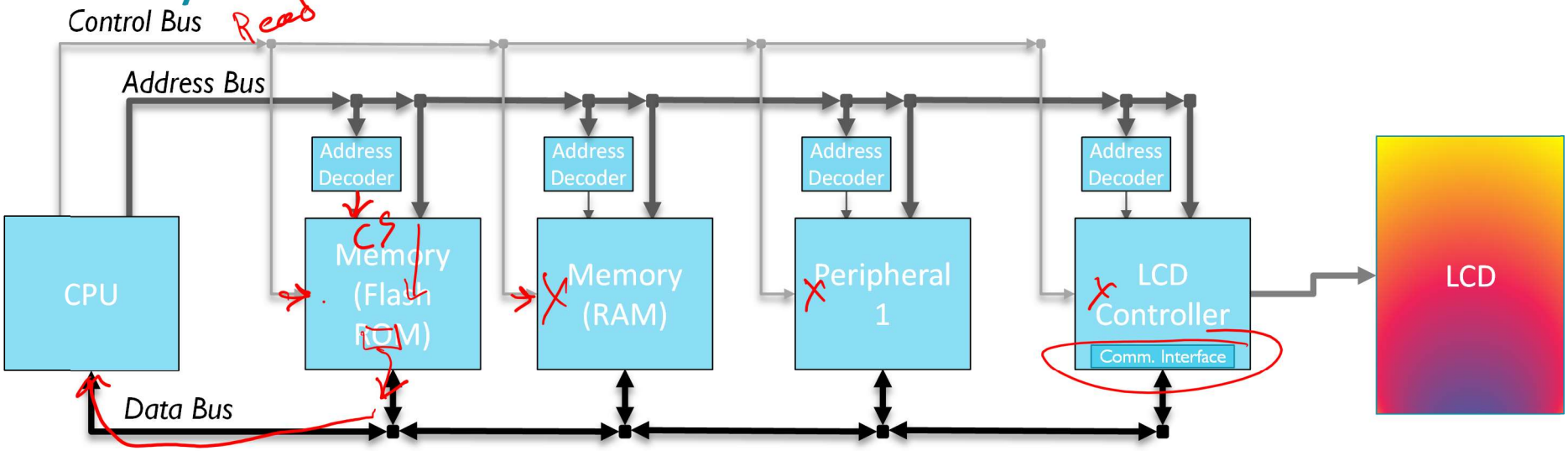


- How do we control the LCD?
- Examine documentation for ST7789 LCD Controller
  - Automatically refreshes display from internal frame buffer memory
  - Supports 83 commands
    - Function: configuration, reading status, accessing frame buffer
    - Format: 1 byte command, 0 or more parameters (data)

9.1.22 RAMWR (2Ch): Memory Write

2CH	RAMWR (Memory Write)												
Inst / Para	D/CX	WRX	RDX	D17-8	D7	D6	D5	D4	D3	D2	D1	D0	HEX
RAMWR	0	↑	1	-	0	0	1	0	1	1	0	0	(2Ch)
1st parameter	1	↑	1	D1[17]-1[8]	D1[7]	D1[6]	D1[5]	D1[4]	D1[3]	D1[2]	D1[1]	D1[0]	
…	1	↑	1	Dx[17]-x[8]	Dx[7]	Dx[6]	Dx[5]	Dx[4]	Dx[3]	Dx[2]	Dx[1]	Dx[0]	
N parameter	1	↑	1	Dn[17]-n[8]	Dn[7]	Dn[6]	Dn[5]	Dn[4]	Dn[3]	Dn[2]	Dn[1]	Dn[0]	
	-This command is used to transfer data from MCU to frame memory.												

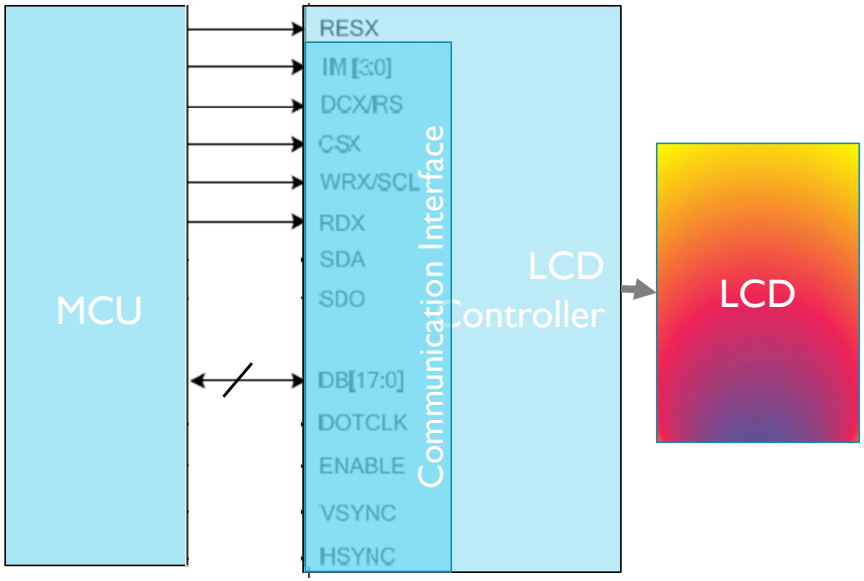
Memory Bus Architecture

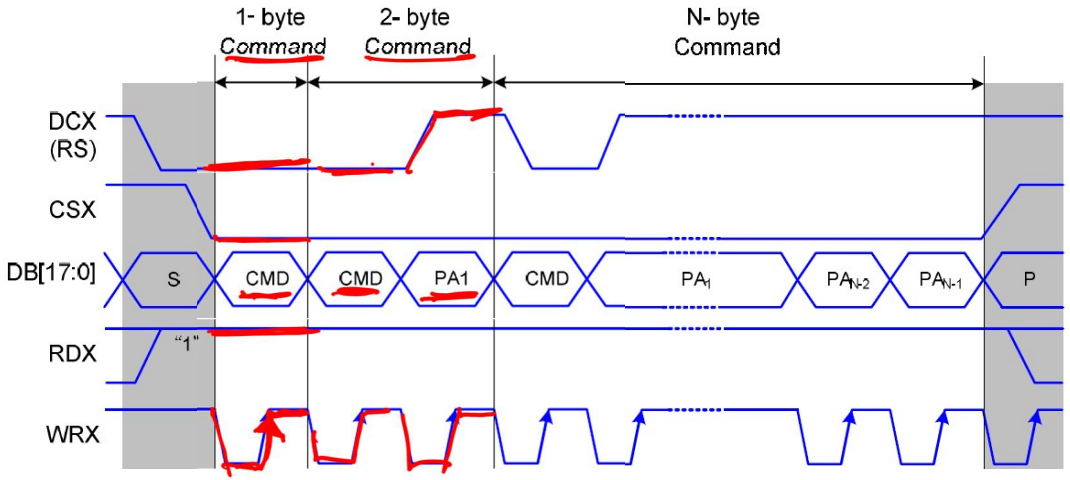


- CPU uses bus to access items in memory space
  - Fast! Parallel 32-bit data and 32-bit address buses
- Operation
  - Decoder matches **certain address bits** to find range of addresses this device will serve, asserts **chip select** line
  - Device uses some of **other address bits** to identify location within device
  - Control: Read? Write?
  - Data

- LCD Controller accepts commands through its communication interface
- Supports parallel and serial interfaces
  - 11 "parallel" interfaces: control lines + 8/9/16/18 data bits target MCU/MPU memory expansion bus
  - 5 single-bit serial interfaces
  - *Ignore rabbit holes of semantics*
    - *Serialization (1 vs. 8 vs. 9 bits), truncation, half- vs. full-duplex*

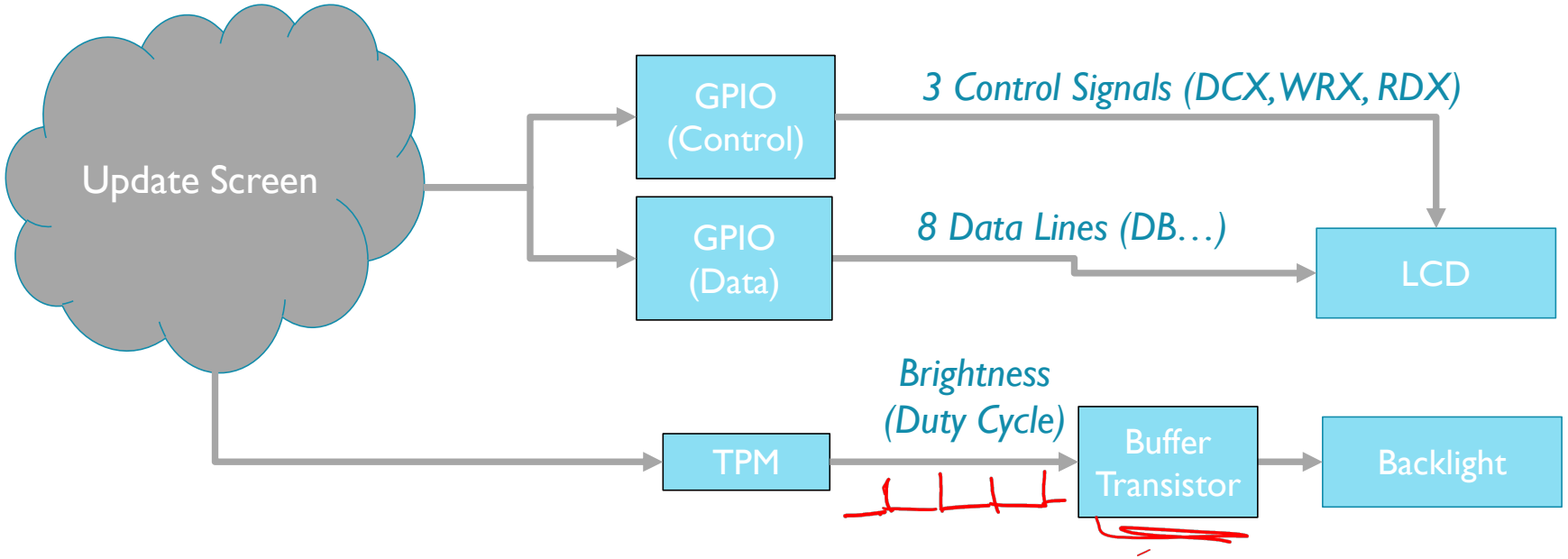
LCD Controller Memory Bus Interface





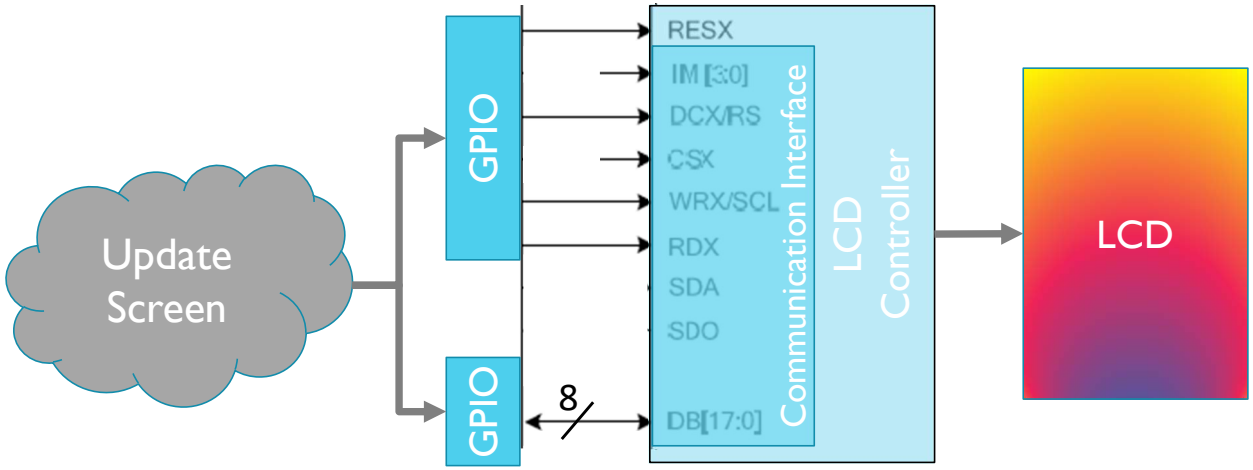
- LCD Controller interface for memory bus
  - Control lines (X means active low)
    - DCX data (1) or command (0)
    - RDX read (0)
    - WRX write (0)
  - Optional CSX chip select signal (if sharing bus)
  - Data bus DB[17:0] up to 18 bits wide
- Signal diagram shows typical operation
  - Specify command or data with **DCX**
  - (Optional: Select chip by asserting **CSX**)
  - Place command or parameter on data bus **DB[...]**
  - Write by asserting and releasing **WRX** (DB sampled on rising edge of WRX)

Refine: Update Screen HW & SW



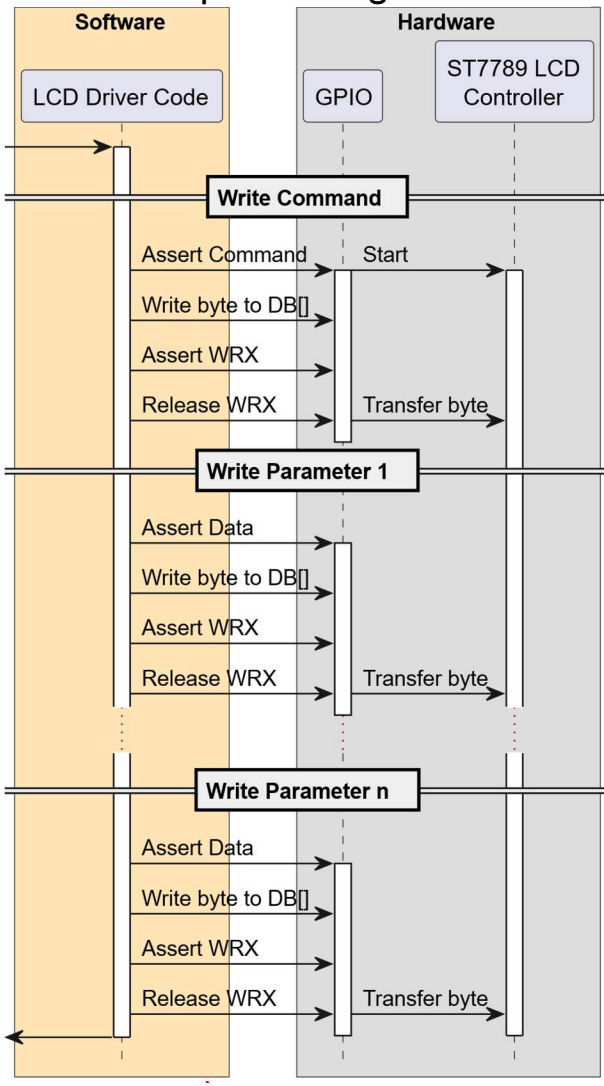
- Software-based approach is single thread, not complex.
- Drive GPIO with software? Prototype early to reduce risks
- If not fast enough, look into accelerating interface with DMA (may need to slow it down)

Refine: Update Screen (LCD)

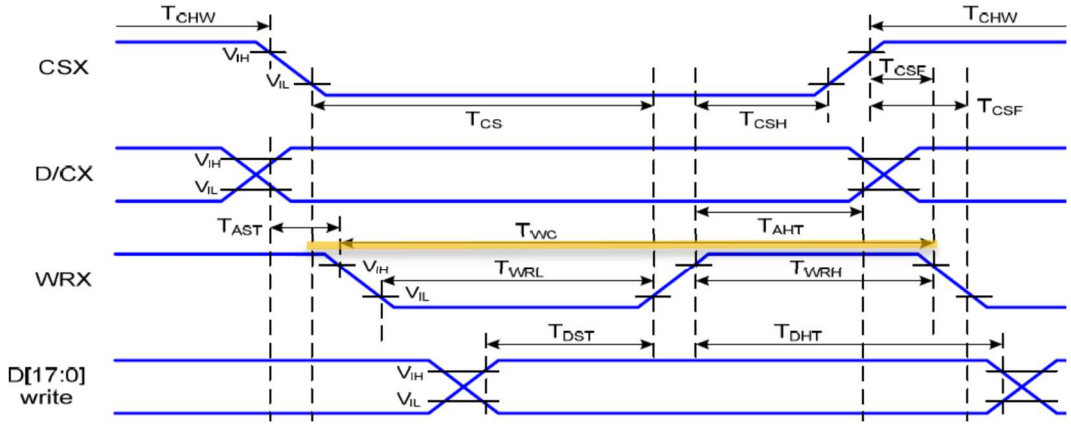


- Interface as implemented on shield
  - 3 control lines + 8 data bits: balance pin count with speed
  - KL25Z MCU lacks memory expansion bus, so **emulate bus** with GPIO port peripheral and software
  - CSX: Chip select hard wired to ground
  - RESX: Reset driven by GPIO bit
- Sequence diagram: "bus" operations to send **command** + **n parameters**
- Implementing bus operations
  - GPIO port operations must leave other bits **unchanged**
  - Take advantage of hardware support for bit masking: PSOR, PCOR, PTOR
  - Otherwise use software to perform read/modify/write

Sequence Diagram



LCD Controller Parallel Interface Timing



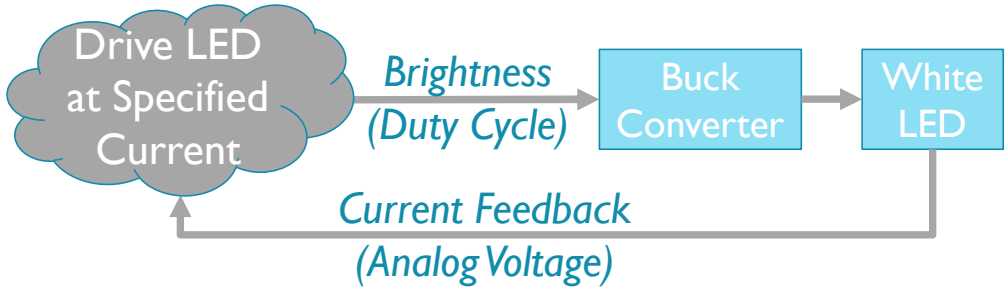
VDDI=1.65 to 3.3V, VDD=2.4 to 3.3V, AGND=D

Signal	Symbol	Parameter	Min	Max	Unit
D/CX	T_{AST}	Address setup time	0		ns
	T_{AHT}	Address hold time (Write/Read)	10		ns
CSX	T_{CHW}	Chip select "H" pulse width	0		ns
	T_{CS}	Chip select setup time (Write)	15		ns
	T_{RCS}	Chip select setup time (Read ID)	45		ns
	T_{RCSFM}	Chip select setup time (Read FM)	355		ns
	T_{CSF}	Chip select wait time (Write/Read)	10		ns
	T_{CSH}	Chip select hold time	10		ns
WRX	T_{WC}	Write cycle	66		ns
	T_{WRH}	Control pulse "H" duration	15		ns
	T_{WRL}	Control pulse "L" duration	15		ns

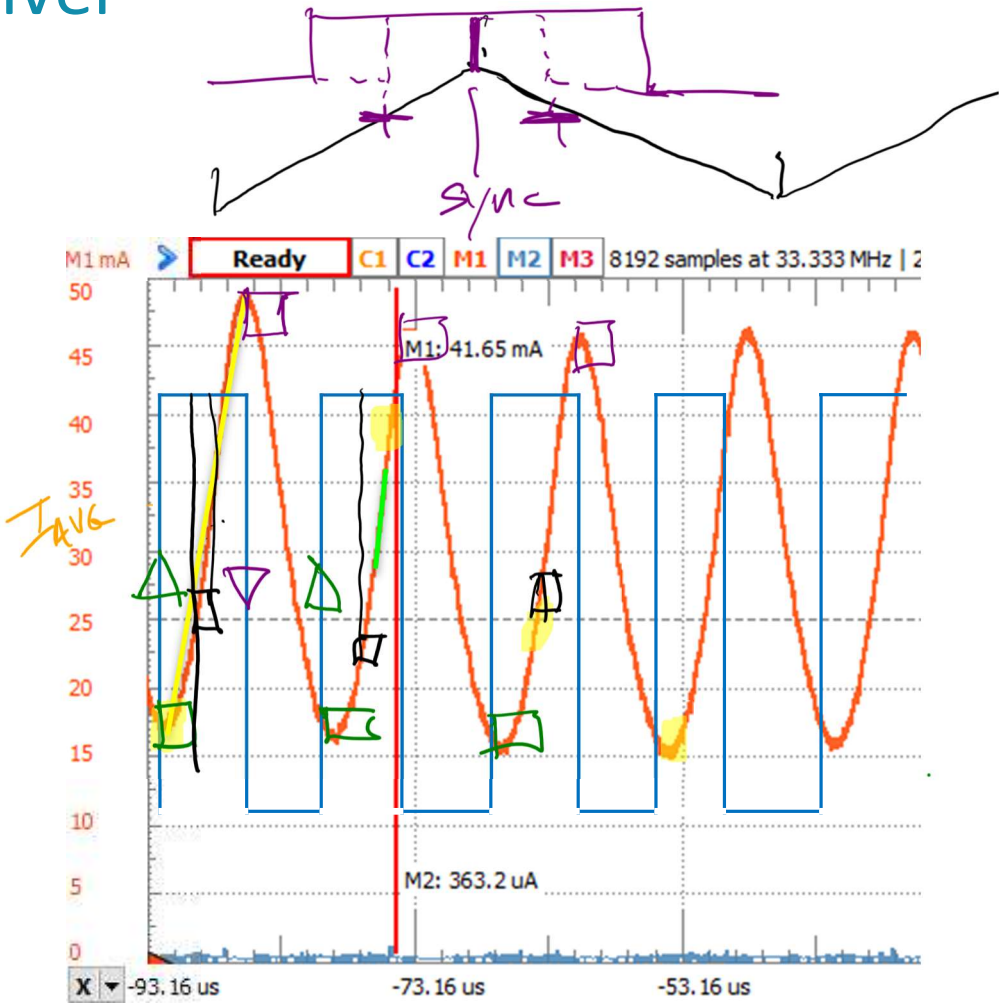
- Minimum write cycle time: 66 ns
- KL25Z running at 48 MHz: 48 MHz * 66 ns = 3.168 instruction cycles

- Estimate best possible performance (lower bound) for software–implemented bus
  - Sequence diagram: four operations per byte
  - Minimum one instruction per operation
  - Minimum one clock cycle per instruction
  - 4 * 1 * 1 = Minimum of four cycles.
- **Will not violate** minimum timing requirement of 3.168 cycles
  - If it did, we would need to slow down the code (synchronization!)

Refine: Constant-Current LED Driver

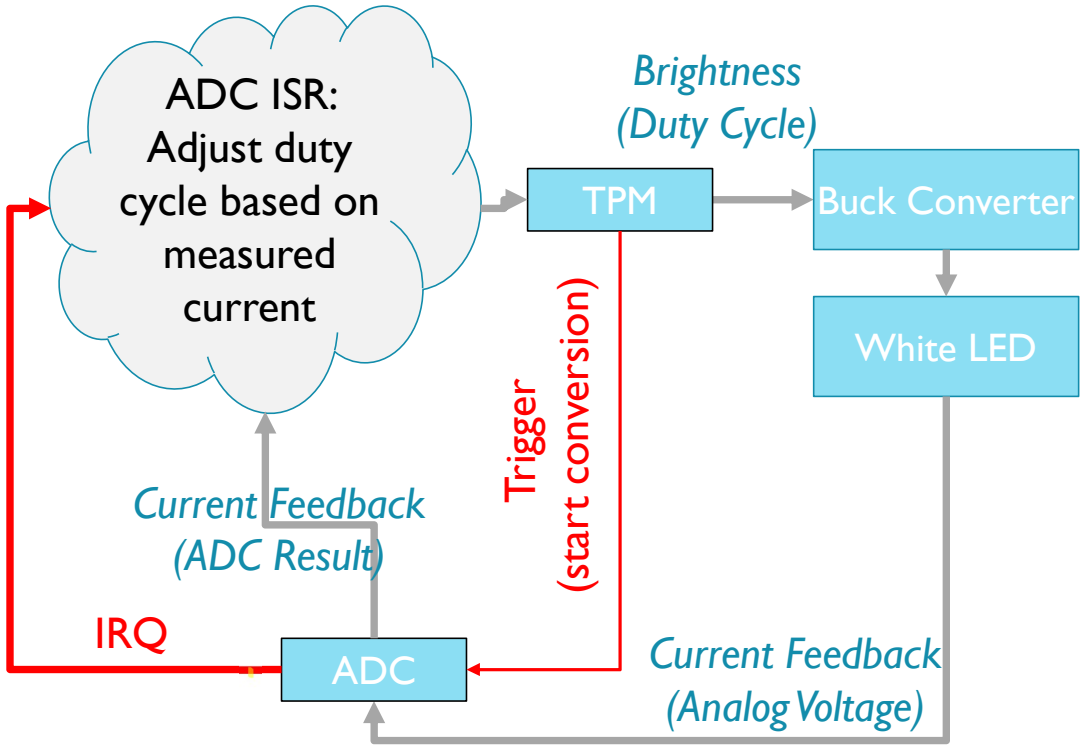


- Timing precision requirement?
  - Measurement Error: up to 7 mA from 1 µs. 7/32 = ~25% error
- Trigger ADC conversion **in phase** with PWM signal from timer (TPM)
  - Examine PWM signal
  - In what phase of TPM signal should ADC convert?
  - Hardware may allow triggering on rising edge, falling edge, or middle of pulse (center-aligned mode).



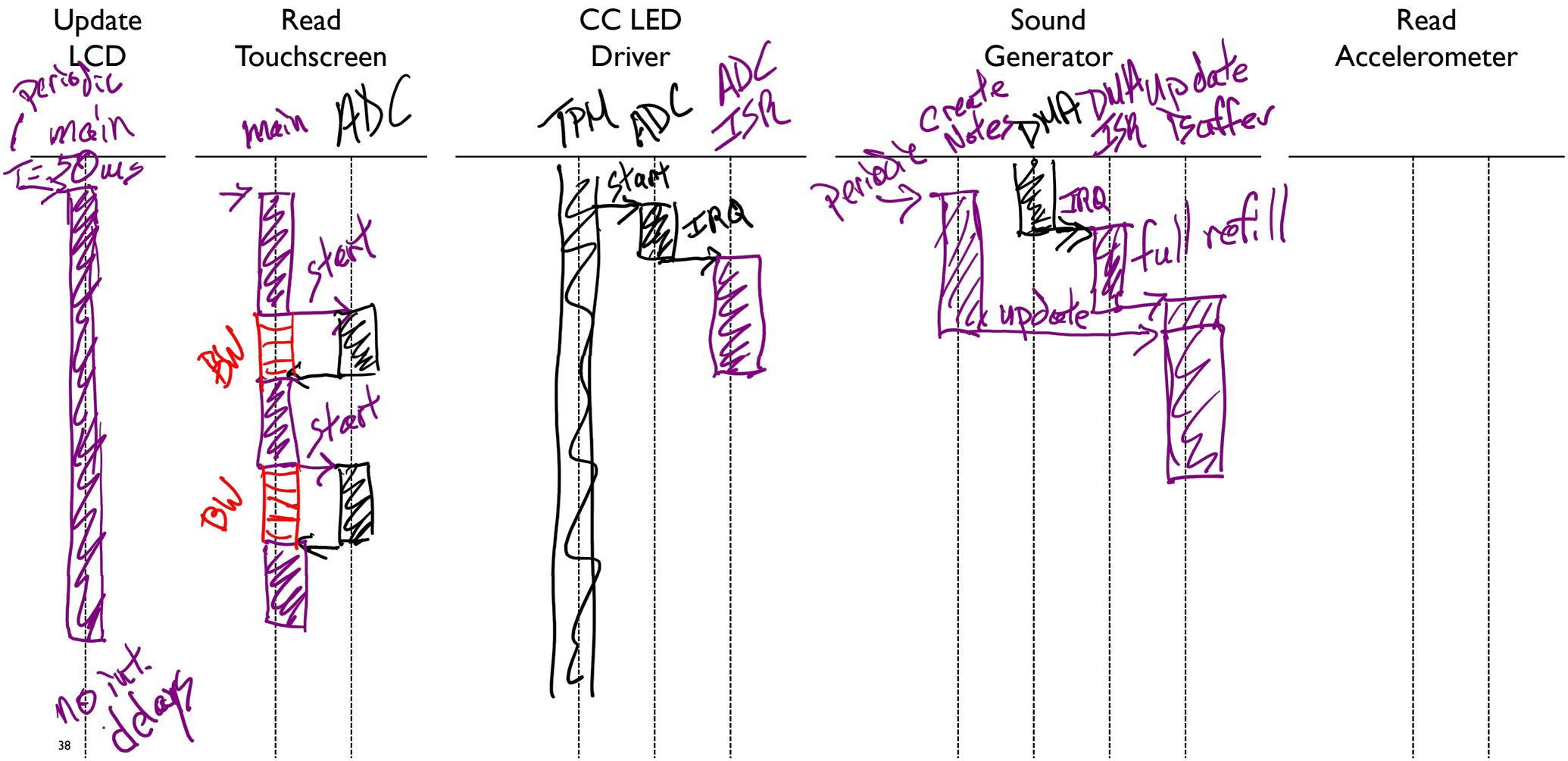
Refine: Constant-Current LED Driver HW & SW

- Use high priority ISR for timing precision
- ISR has simple software structure
  - Read ADC result
  - Calculate new duty cycle (output pulse width)
  - Write to TPM duty cycle register (CNV)
  - No blocking or delays
- Needs to be fast enough to…
  - Meet deadline, likely at most 1/80 kHz = 12.5 µs (depends on when duty cycle register is updated)
  - Leave enough CPU time for other processing

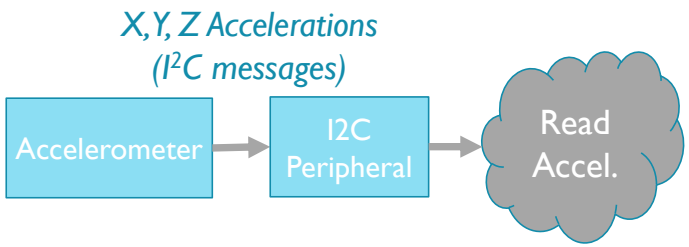


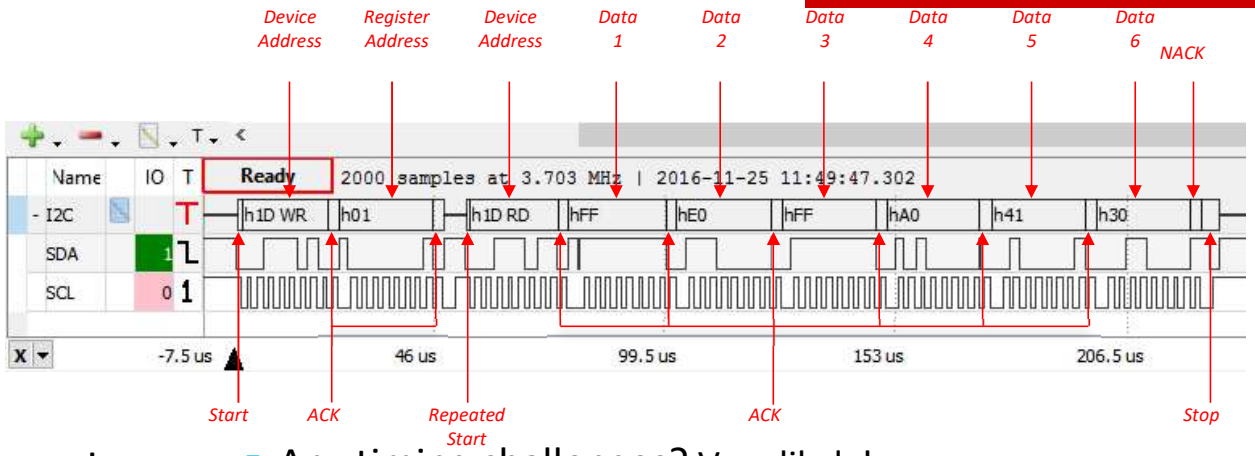
NC STATE UNIVERSITY

Timelines of Major Processing Activities and Synchronization



Refine: Accelerometer





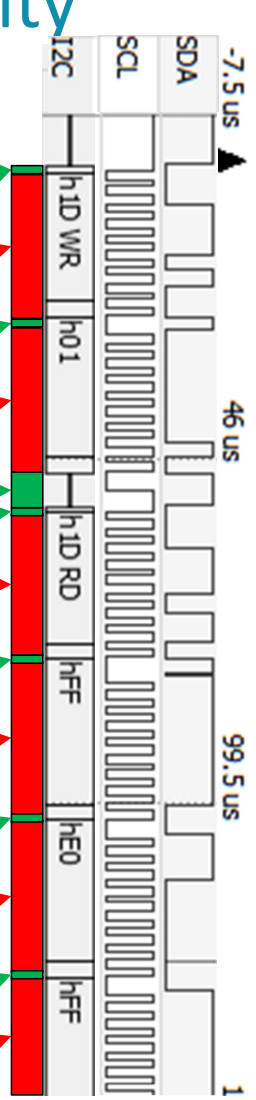
- Send I²C message to read acceleration, then compute roll & pitch and share
- How does I²C communication work?
  - Byte-oriented protocol:
    - Data register holds one byte, so must run some software for each byte sent or received.
  - Message format:
    - Fields: framing, addressing, command, data, acknowledgements
    - Steps: Send start condition, send address byte, send data byte, etc. …. send stop condition

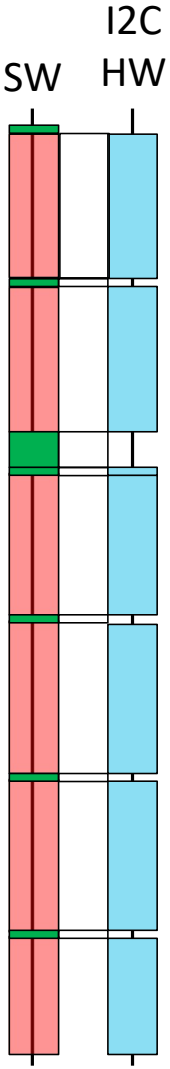
- Any timing challenges? Very likely!
  - Must ***synchronize*** software and hardware: Can't send byte until previous byte has been sent
  - I²C bus runs much slower than CPU, so can't just run straight through all the code at full speed.
    - 400 kilobits/second -> 1 bit takes 2.5 µs, 1 byte takes 20 µs
    - For CPU, 20 µs at 48 MHz = 960 instruction cycles

Timing Analysis: Linking Code and I²C Bus Activity

```
void read_full_xyz()
{
  int i;
  uint8_t data[6];

  i2c_start();
  i2c_read_setup(MMA_ADDR , REG_XHI);

  for( i=0;i<5;i++)
    data[i] = i2c_repeated_read(0);
  data[i] = i2c_repeated_read(1);

  acc_X = (((int16_t) data[0])<<8) | data[1];
  acc_Y = (((int16_t) data[2])<<8) | data[3];
  acc_Z = (((int16_t) data[4])<<8) | data[5];
}
```

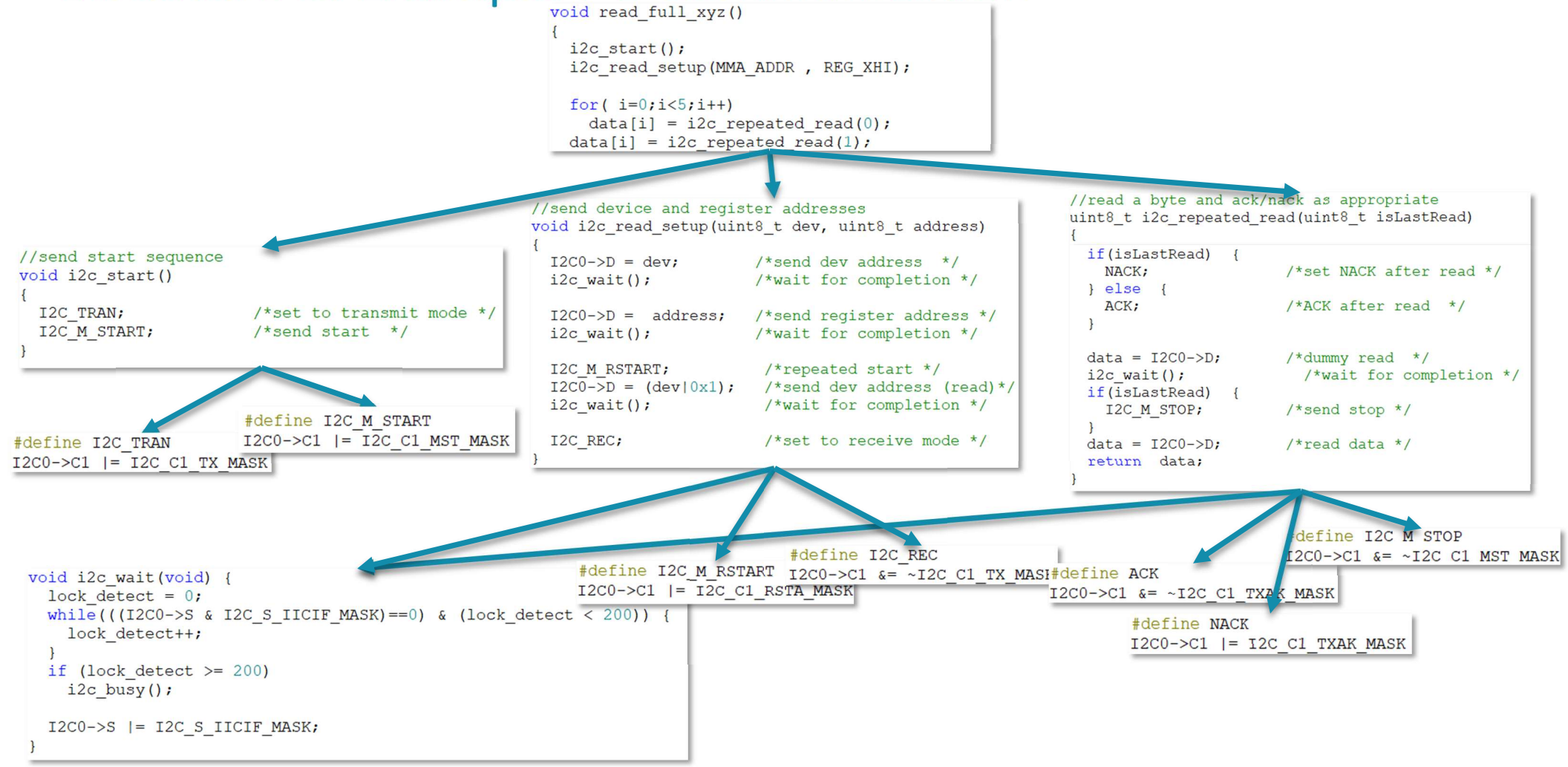
```
read_full_xyz())
    i2c_start()
        I2C_TRAN;              /*set to transmit mode */
        I2C_M_START;           /*send start          */
    i2c_read_setup(dev, address)
        I2C0->D = dev;         /*send dev address (write)*/
        i2c_wait()
            while ((I2C0->S&I2C_S_IICIF_MASK)==0);
            I2C0->S |= I2C_S_IICIC_MASK;
        I2C0->D = address;     /*send read address */
        i2c_wait();            /*wait for completion */
        I2C_M_RSTART;          /*repeated start */
        I2C0->D = (dev|0x1);   /*send dev address (read)*/
        i2c_wait();            /*wait for completion */
        I2C_REC;               /*set to receive mode */
    i2c_repeated_read(isLastRead)
        data = I2C0->D;        /*dummy read starts rx (if not
                                 already receiving) */
        i2c_wait();            /*wait for completion */
        data = I2C0->D;        /*read data, start next rx */
    i2c_repeated_read(isLastRead)
        data = I2C0->D;        /*dummy read starts rx (if not
                                 already receiving) */
        i2c_wait();            /*wait for completion */
        data = I2C0->D;        /*read data, start next rx */
    i2c_repeated_read(isLastRead)
```





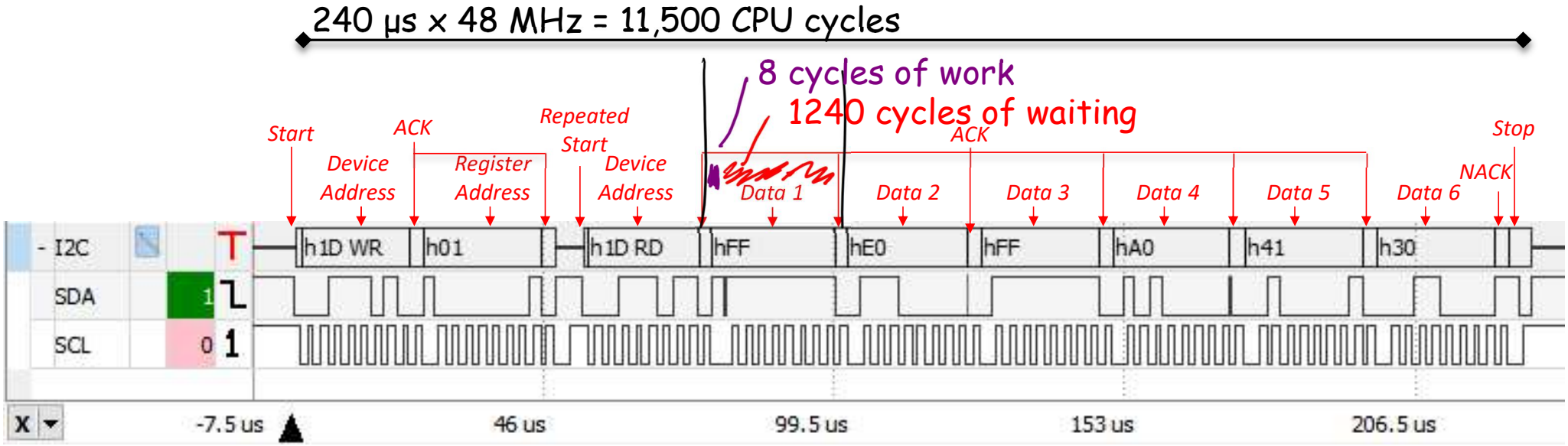
NC STATE UNIVERSITY

Structure of Example I²C Communication Code



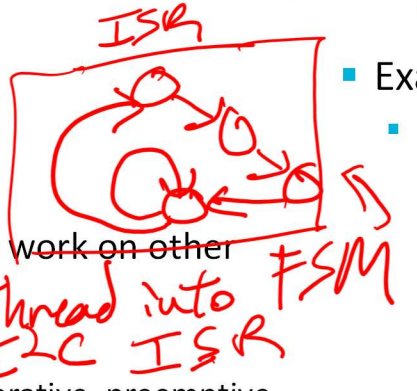
Looking at the Timing

- CPU could execute up to one instruction per cycle
- Work per byte (and its ACK and other overhead)
  - 26 us per byte * 48 MHz = 1248 CPU cycles
  - 8 cycles of work, 1240 cycles of waiting
- Work per message
  - Three address bytes, six data bytes
  - 11,500 cycles total
  - 1240 * 9 = 11,160 cycles waiting
- Is it worth recovering these 11,160 cycles per message?

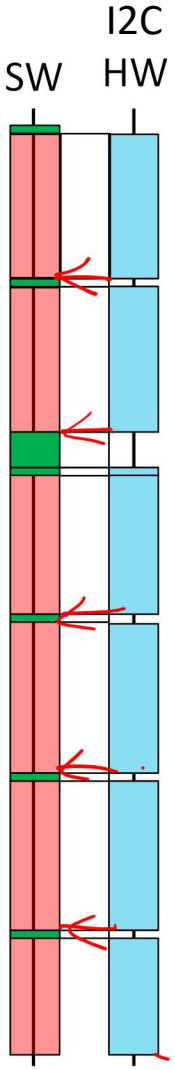


Try to Recover Time Wasted on Synchronization?

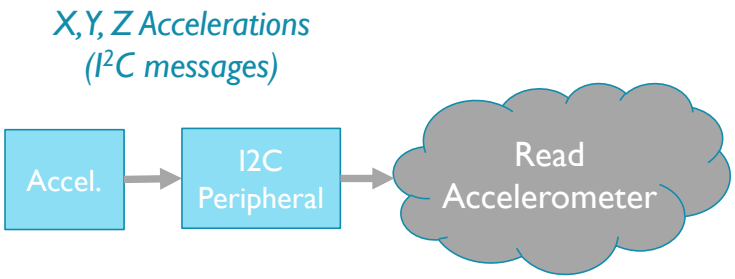
- How much time is used?
  - Add up all the pieces
- How can we recover it?
  - Use scheduler to reallocate CPU to work on other useful code
    - Interrupt system
    - Explicit software scheduler: cooperative, preemptive
    - Implicitly scheduled code: integrate **other** useful code into code doing synchronization
- Is it worth recovering?
  - Is there enough time?
    - Bound: How much relative to CPU speed?
    - Refined: How much relative to CPU's **currently available** free time?
  - Is granularity suitable?
    - Are time chunks big enough to be worth recovering, considering time overhead of recovery?
    - Must consider scheduler time overhead vs. granularity
      - ISR response overhead, OS context switch, etc.

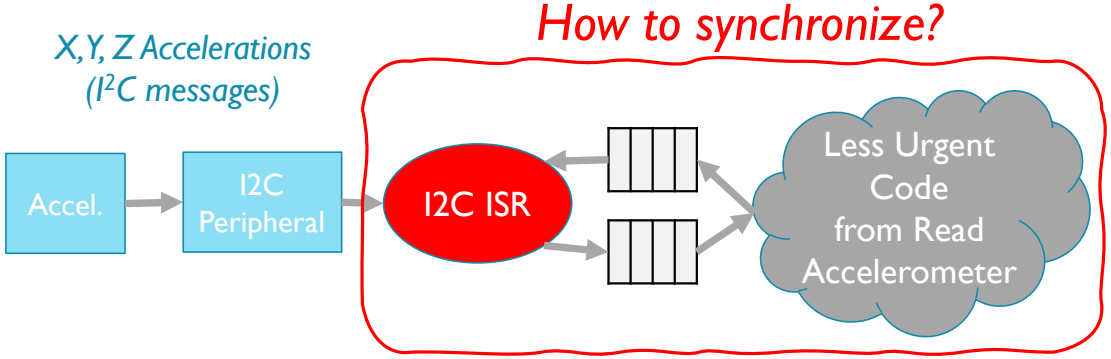


- Example Design Points for shield application
  - Read Accelerometer: I2C
    - Granularity: ~960 CPU cycles long
    - Total per event: ~11,160 CPU cycles per message
    - Total per second:
      50 Hz * 11,160 = 558,000 CPU cycles/second
      (out of 48,000,000) = ~1%
    - **Probably not worth recovering**
  - Read Touchscreen: ADC
    - Granularity: ~96 CPU cycles
    - Total per event: ~192 CPU cycles per TS read
    - Total per second:
      10 Hz * 192 = 1,920 CPU cycles/second
      (out of 48,000,000) << 1%
    - **Not worth recovering**



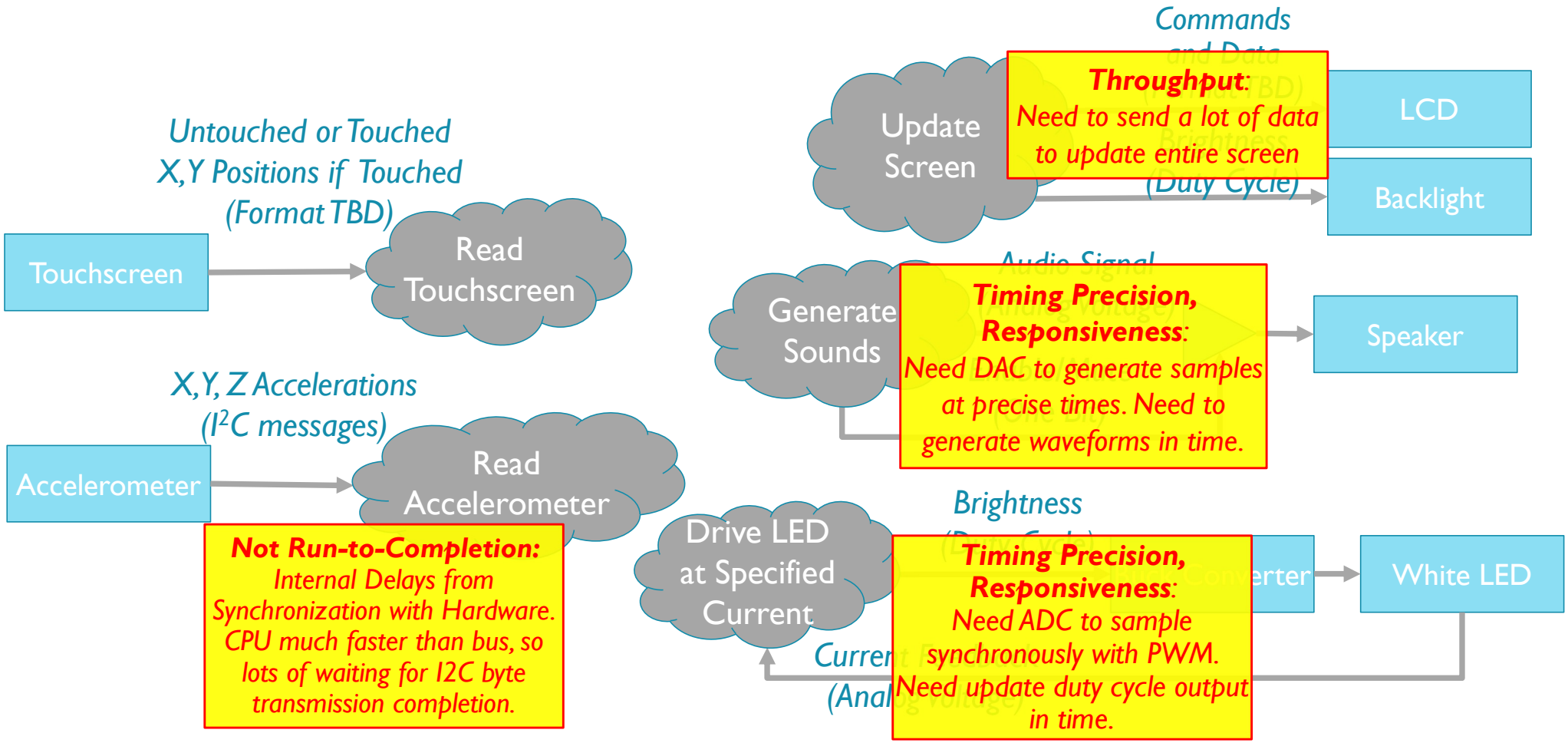
Refine: Accelerometer HW & SW





- Initial solution
  - Principle: Functionality First, Elegant Performance Later (if ever)
  - Use busy-waiting, and **don't try to recover idle time yet**
- Complex problem
  - Want to recover idle time while ensuring urgent code runs at right times
  - Generic solution puts urgent work into ISR, defers other work to thread. How do we split up this I2C driver code?
- Possible solutions – coming up soon
  - Finite State Machines
  - Task/Thread scheduler (heart of operating system)

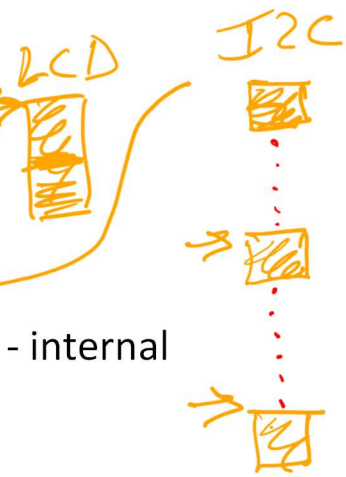
Updated Difficult Areas



Synchronizing Activities

- Interrupt System example
  - HW Peripheral generates IRQ, which triggers CPU to run ISR
- Timing relationships between activities
  - Triggering: does one trigger the other?
  - Order: Must one be first?
  - Concurrency: Overlap in time allowed?

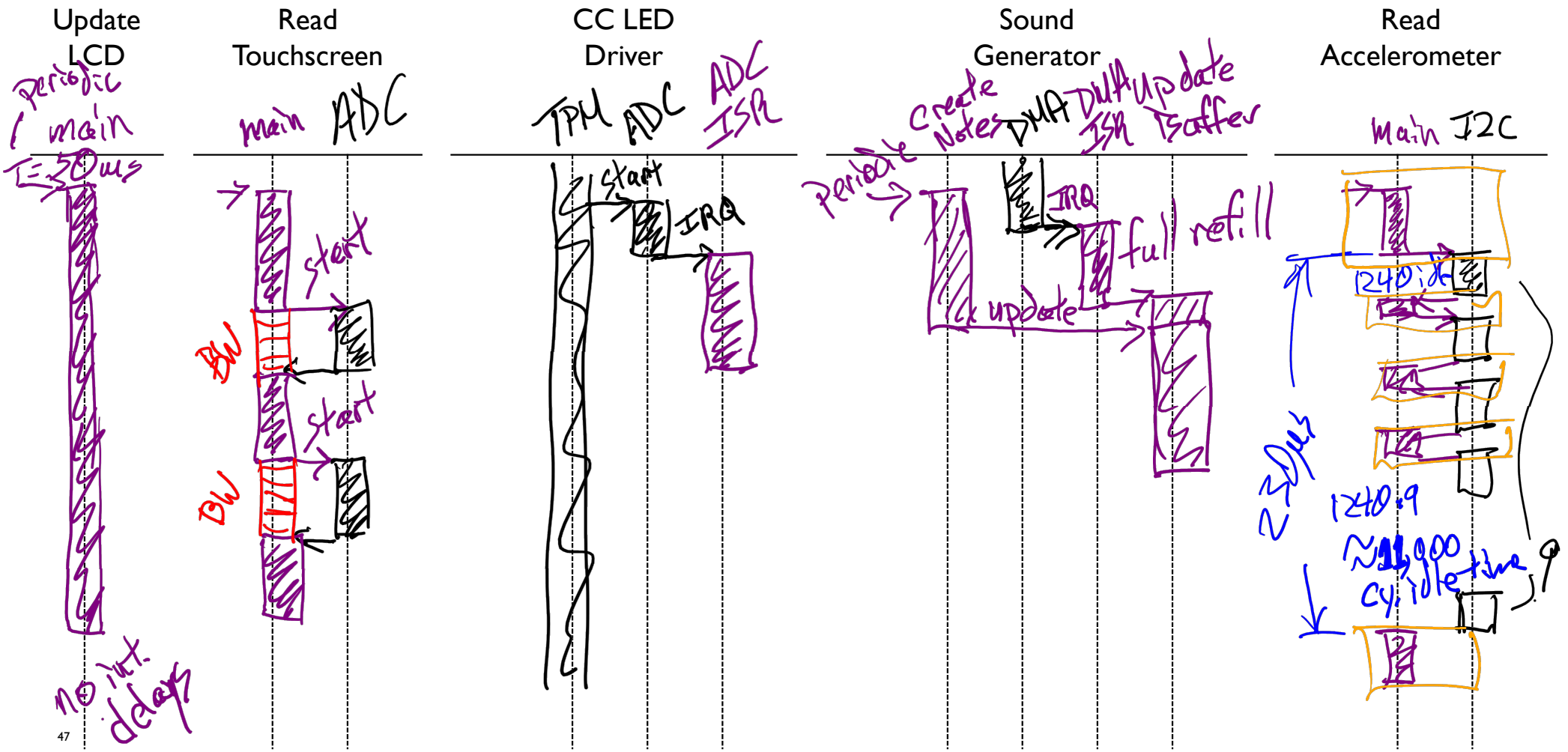
- General Examples
  - Neither:
    - A and B can overlap in any order
  - Concurrency:
    - A and B can't be concurrent. A must finish before B starts, or B must finish before A starts
    - A and B must be concurrent. A and B must overlap in time.
  - Order:
    - A must finish before B starts.



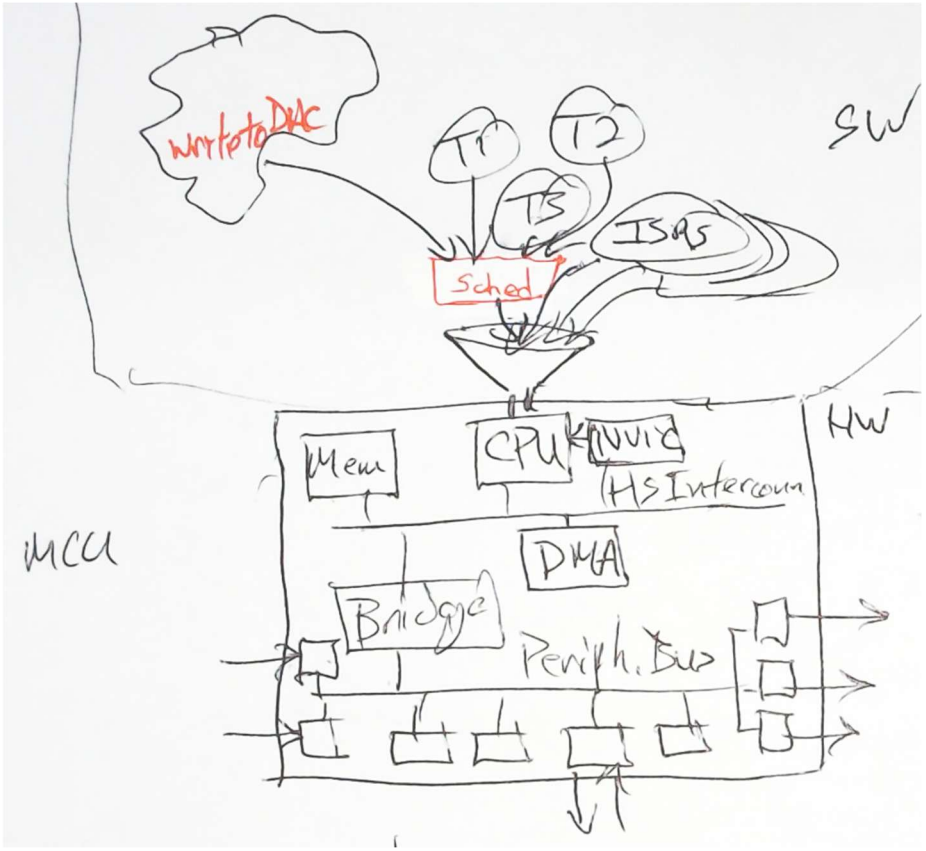
- Use synchronization to
  - ***start*** running code
    - thread - external
    - code within thread - internal
  - ***stop*** running code within thread - internal
- Synchronization Triggers
  - Event
    - When something happens, do something else. Event: IRQ -> Activity: ISR
  - Time
    - At absolute time T, do something
    - After time delay ΔT, do something

- What if some activities are in software, and others in hardware?

Timelines of Major Processing Activities and Synchronization



Synchronization between Software and Hardware



- If some processing is done in hardware and some in software, how do we synchronize them?

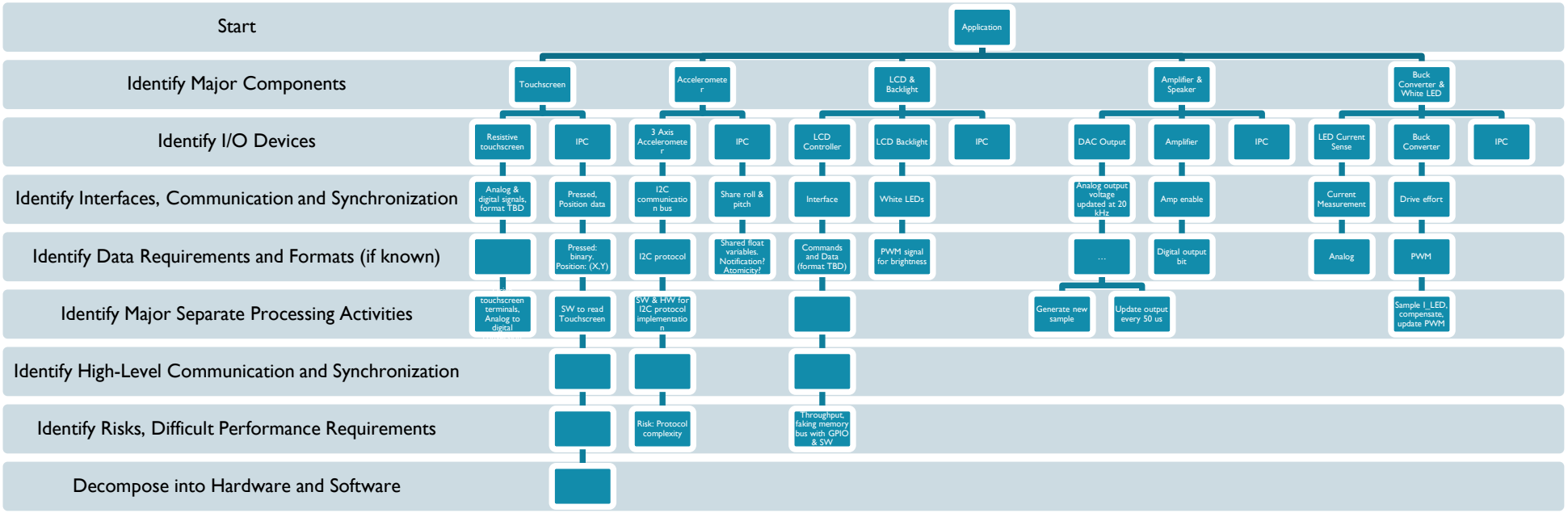
	… to Hardware	… to Software
From Software …	• Write to peripheral register	• Subroutine calls • Software interrupts • Polled shared variables Scheduler task control mechanisms: enable, disable, release, etc. • OS synch/comm. mechanisms: semaphore, mutex, event flag, message, etc.
From Hardware …	• Hardware signals	• Interrupt System: IRQ→ISR (handler) • SW Polling HW status

Software Overview

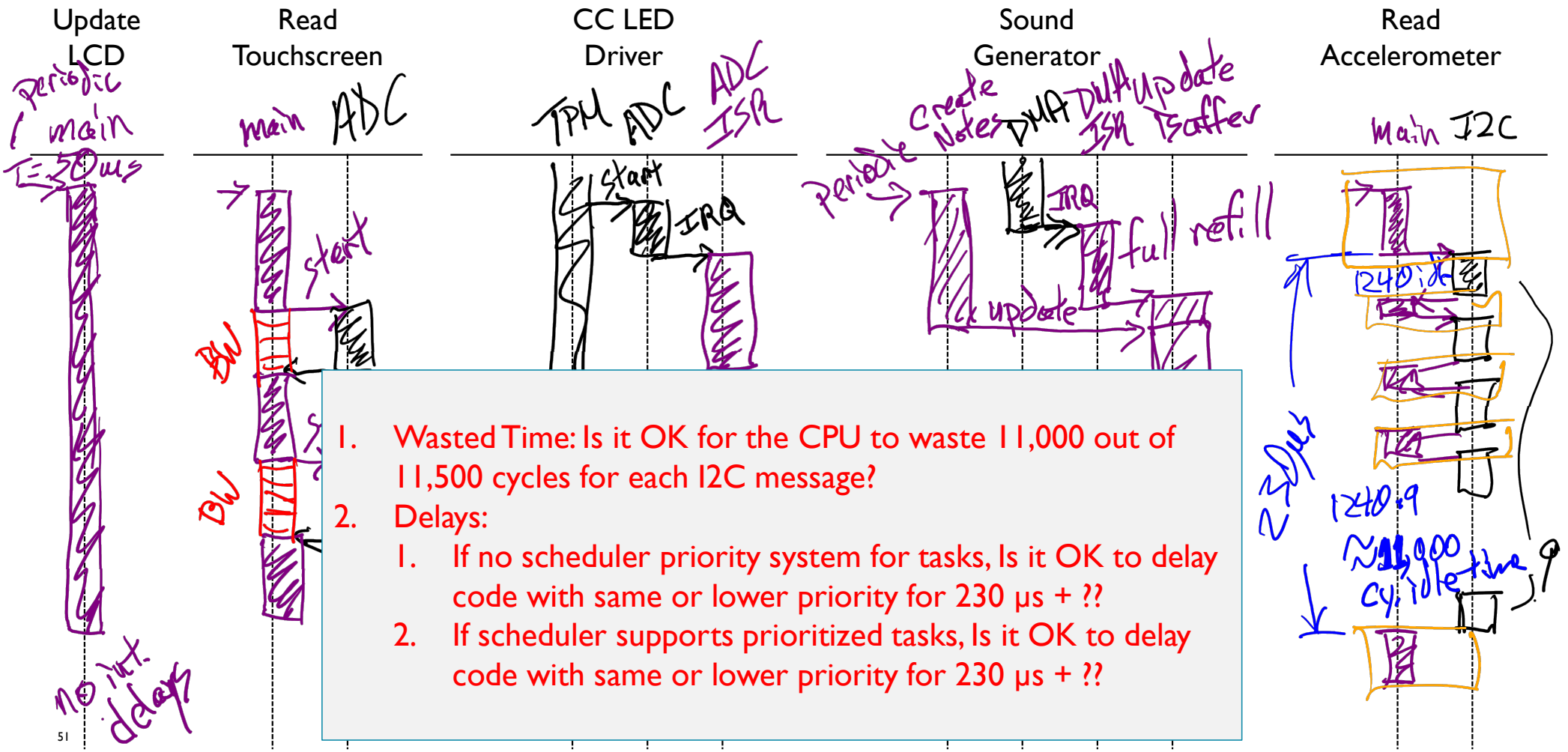
- Optional/Later: Return if screen isn't pressed
- Read X position
  - Configure GPIO XL and XR as outputs, YU as input
  - Configure GPIO YD as analog input
  - Write 0 to XL and 1 to XR
  - Start conversion on ADC channel connected to YD
  - Wait for conversion complete flag
  - Read ADC result
- Read Y position
  - Configure GPIO YD and YU as outputs, XL as input
  - Configure GPIO XR as analog input
  - Write 0 to YD and 1 to YU
  - Start conversion on ADC channel connected to XR
  - Wait for conversion complete flag
  - Read ADC result
- Convert X, Y voltages to positions

NC STATE UNIVERSITY

Design Refinement



Timelines of Major Processing Activities and Synchronization



1. Wasted Time: Is it OK for the CPU to waste 11,000 out of 11,500 cycles for each I2C message?
2. Delays:
   1. If no scheduler priority system for tasks, Is it OK to delay code with same or lower priority for 230 µs + ??
   2. If scheduler supports prioritized tasks, Is it OK to delay code with same or lower priority for 230 µs + ??