Intro Embedded Operating Systems

- 1. Intro
 - Hardware
 - 2018: ~256k RAM, ~512k flash, ~ 80 MHz
 - 2008: ~10k RAM, ~48k flash, ~8 MHz
 - I2C, SPI, UART, ADC, DAC, PWM, etc.
 - 2 uA sleep, 10 mA active
 - Goals (why would programmers use an embedded OS)
 - Abstract hardware
 - Enable low power operation
 - Manage concurrency
 - Manage scheduling
 - Provide shared libraries
 - Virtualize hardware resources
 - Meet resource constraints
 - Not-really goals
 - Isolate processes
 - Dynamic configuration
 - Virtualized memory
 - Concurrency?
 - Not multi-core
 - Interrupts
 - Common design patterns
 - Modularity
 - Virtualized and non-virtualized resources
 - Long running operations
 - Event-driven versus threaded
 - Toolchain
 - Compile small apps with many shared components
- 2. Abstract Hardware
 - Layers

•	High-level interface	Readline Console

- Virtualized driver SharedUART
 ---- common interface ----
- MCU-specific driver MSP430 UART --- widely varying interface ---
- MMIO Peripherals
- 3. Enable Low Power
 - For long-term operation, node must be in sleep state a majority of the time (~99%)

UART1

- MCUs have different sleep states
 - Support different peripherals

- Depending on what is being used on certain sleep states may be available
- Always try to put the chip in lowest valid sleep state
- Provide wakeup sources
 - Interrupts!
 - Common:
 - Timers (i.e. wait X seconds and then do next operation)
 - Peripheral done (UART message sent)
 - External events (GPIO interrupt)
- Easy to mess up
 - One misconfigured driver can sabotage the system
- Still an open challenge
- 4. Manage Concurrency
 - Interrupts are essentially a second thread
 - Including all of the race condition and memory bugs
 - Two high-level approaches
 - Only a single active interrupt
 - Hard to make general purpose
 - Minimal code in the interrupt handler
 - Simply wait for main loop to recognize the event occurred
 - Potential latency issues
- 5. Manage Scheduling
 - Decide what order to execute things
 - Priorities
 - Usually very simple in practice
- 6. Provide Shared Libraries
 - Useful libraries make developing applications easier and faster
 - Examples
 - Logging utility
 - Networking stack
 - Crypto operations
 - Time synchronization
 - What abstractions are required? Does hardware generally support them?
- 7. Virtualize Hardware Resources
 - Enable limited hardware resources to be shared among multiple users
 - Policies for sharing
 - Exclusive access
 - Merging requests
 - Complete virtualization (timers)
 - Closer to mode where application is the only thing on the system
- 8. Meet Resource Constraints
 - No dynamic memory in the kernel
 - Fixed size buffers decided at compile time

• Include only the code that is needed by the application