Firmware Validation: Challenges & Opportunities

Jim Grundy

THIS PRESENTATION CONTAINS THE GENERAL INSIGHTS AND OPINIONS OF INTEL CORPORATION (INTEL). THE INFORMATION IN THIS PRESENTATION IS PROVIDED FOR INFORMATION ONLY AND IS NOT TO BE RELIED UPON FOR ANY OTHER PURPOSE THAN EDUCATIONAL. USE AT YOUR OWN RISK! INTEL MAKES NO REPRESENTATIONS OR WARRANTIES REGARDING THE ACCURACY OR COMPLETENESS OF THE INFORMATION IN THIS PRESENTATION. INTEL ACCEPTS NO DUTY TO UPDATE THIS PRESENTATION BASED ON MORE CURRENT INFORMATION. INTEL IS NOT LIABLE FOR ANY DAMAGES, DIRECT OR INDIRECT, CONSEQUENTIAL OR OTHERWISE, THAT MAY ARISE, DIRECTLY OR INDIRECTLY, FROM THE USE OR MISUSE OF THE INFORMATION IN THIS PRESENTATION.

Firmware is everywhere

Image © 2008 Diego Correa
Used under the terms of the
Creative Commons – Attribution 2.0 License

Image © 2012 Intel Corporation

Image © 2013 Intel Corporation

Image © 2013 Michael Sheehan
Used under the terms of the
Creative Commons – Attribution 2.0 License

Image © 2012 Aaron Miszalski
Used under the terms of the
Creative Commons – Attribution 2.0 License

But … is anything really "firmware"?

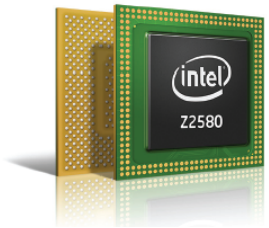
Smart devices sport capable processors and general purpose operating systems.

Is software for smart devices different to traditional software?

PRODUCT BRIEF
Intel® Atom™ Processors Z2580, Z2560, and Z2520 for Smartphones and Tablets

Intel® Atom™ Processors Z2580, Z2560, and Z2520 for Smartphones and Tablets

Industry-leading dual-core performance for incredible mobile experiences, rich graphics, fast apps, advanced camera features, and stunning HD video.



Overview

The pace of mobile life is accelerating, and mobile users want smartphones and tablets with the performance to keep up. The Intel® Atom™ processors Z2580 (2.0 GHz), Z2560 (1.6 GHz), and Z2520[Δ] (1.2 GHz) deliver dual-core performance built on Intel's leading 32-nanometer process technology. Intel® Hyper-Threading Technology[1] supports four simultaneous application threads for smooth and seamless multitasking and responsive Web browsing.

And the great experiences don't stop here.

Table 1. Technical Specifications

High-performance Processors	• 2.0 GHz (Z2580) • 1.6 GHz (Z2560)
Intel® Atom™ Microarchitecture	• Intel® Smart Cache, 512 KB per core • Advanced 32nm, dual-core Intel architecture …
System Memory Interface	• Total capacity up to 2 GB • Supports up to 1066 MT/s (533MHz) data rate

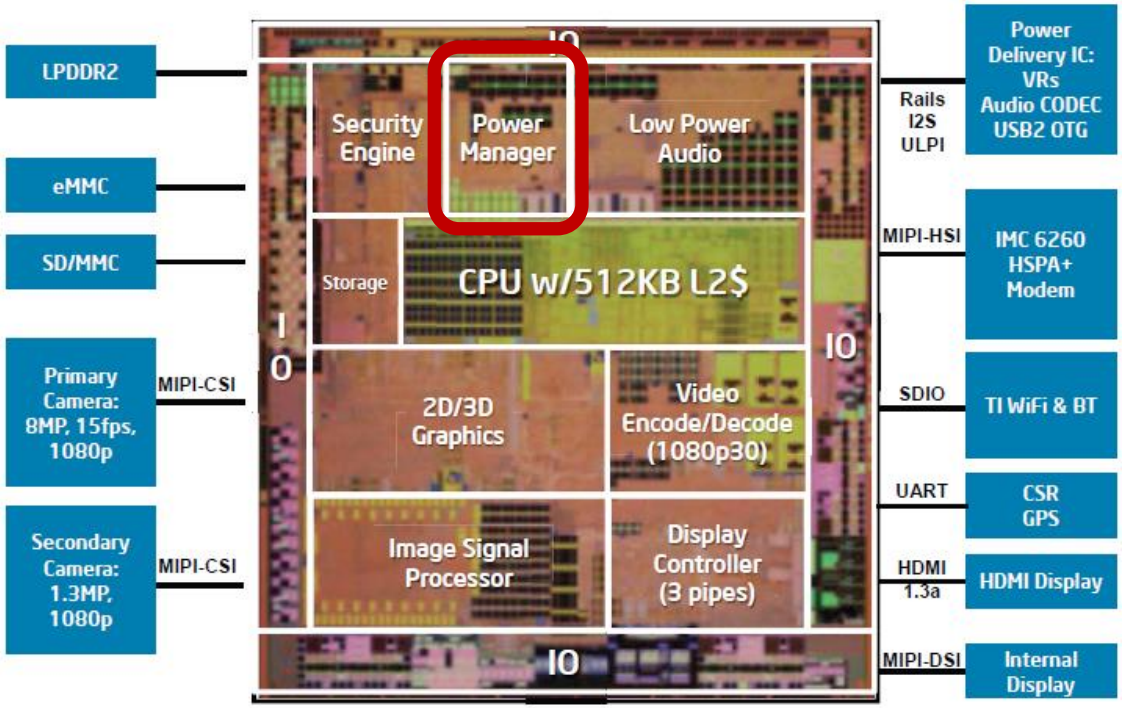
Source: Product Brief – Intel® Atom™ Processors Z2580, Z2560, Z2520 for Smartphones and Tablets

That isn't firmware

This is firmware…

"[a] micro-controller called the Power Manager handles … fine-grained control of the … power islands. The operating system … controls power only indirectly by specifying the desired … states to the Power Manager."

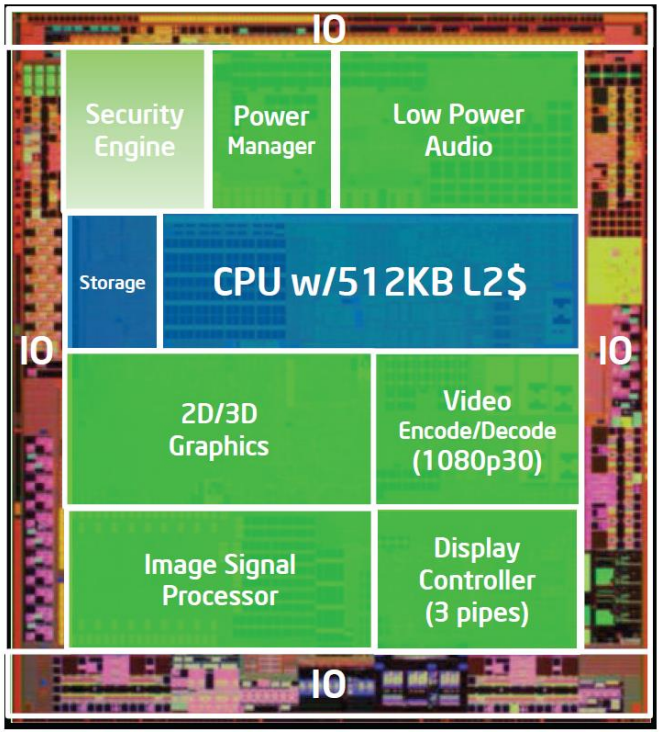
Intel® Atom™ Z2480 Processor Block Diagram

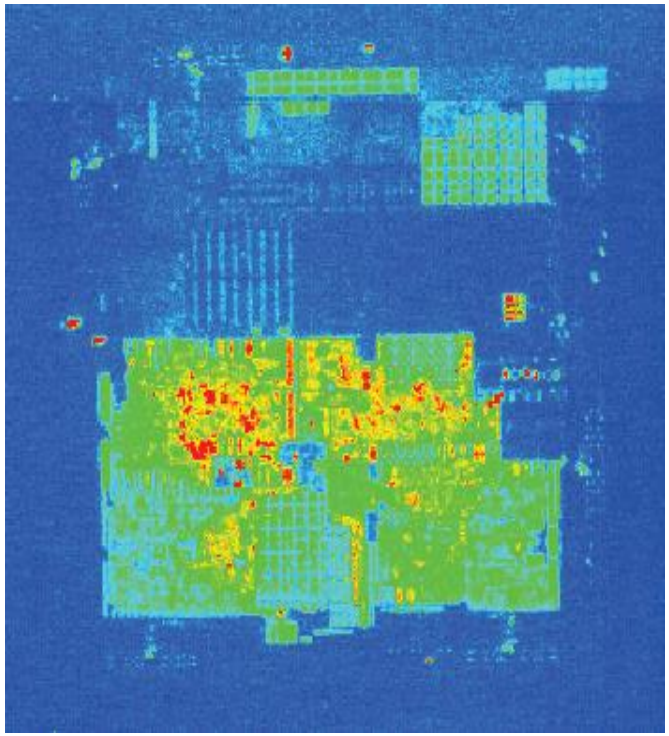


Source: R. Zhair, M. Ewert, H Seshardri. The Medfield Smartphone: Intel® Architecture in a Handheld Form Factor. *IEEE Micro*. 2013.

Firmware in action

Q: Media is playing … the CPU is powered off … how?

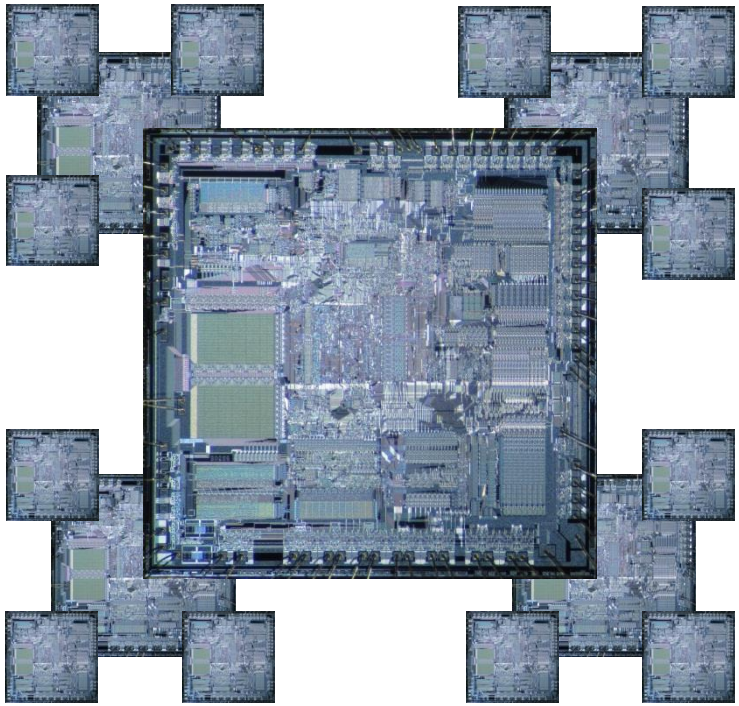




Source: R. Zhair, M. Ewert, H Seshardri. The Medfield Smartphone: Intel® Architecture in a Handheld Form Factor. *IEEE Micro*. 2013.

A: The devices are getting smarter.

The problem of programming tiny machines is not going away



Outline

- What is firmware?
- What makes firmware validation so important?
- What does firmware look like?
- Research needs
- Case studies for researchers

(non-technical) Definition of Firmware

A *software component that*

- *Must be functional before the associate hardware ships*
- *Is difficult/costly/undesirable to patch after shipping*

- So defined
  - Includes the lowest level software in a product
  - Software that interacts directly with hardware
  - May include some of the higher-level software as well

- Says nothing about what firmware looks like.
- But this definition is useful because
  - People get an immediate sense of its importance.
  - We can find technical commonalities in such software.

Outline

- What is firmware?
- What makes firmware validation so important?
- What does firmware look like?
- Research needs
- Case studies for researchers

Amount of firmware is growing fast

- Reduce design cost and add schedule predictability
  - Moving control to firmware simplifies the hardware.
  - Configurability allows hardware to function in many products.
  - Firmware is easily updated prior to shipping.
  - Makes first silicon more likely to be production silicon.
- More efficiently process data at the edge
  - Efficient to process data with silicon accelerators.
  - Minimizes traffic to locate those accelerators in the devices.
  - Requires richer control algorithms in the devices.
- More sophisticated behavior than possible through software visible interfaces.
  - HW/SW interfaces are standards. They change slowly.
  - OS power management is an example.

Firmware Validation Criticality is Higher Than Software

From the definition that firmware is software that:

- *Is difficult/costly/undesirable to patch after shipping*

- Then it is also the software you particularly want to get right before shipping

Why?

Q: Can't firmware be patched?

A: Yes, but firmware typically isn't patched once the product is in the customer's hands.

Q: My <insert-device> installs updates all the time. Aren't those firmware update?

A: Those are mostly software updates (by definition).

Why is firmware hard to update?

- A technical reason: Criticality
  - Failed updates are a remote but inescapable possibility.
  - Lower level failed updates are harder to recover from.
  - Lower level updates must meet a risk/reward analysis.
  - Often left to users to install rather than being pushed.

- The real reason: Rich supply chains
  - Devices reach customers though deep chains. My phone:
    - Is sold by a carrier that handles customer and network issues
    - Is made by a vendor
    - Runs an operating system from another vendor
    - On an SoC from yet another vendor
    - That integrates IP blocks from multiple vendors: radio, usb, …
  - Only the top layer(s) maintain a connection to the device.

Firmware Validation Schedule Criticality is Higher than Software

Schedule predictability is important for hardware and software projects. But, more critical for hardware:

- Threshold for software update is lower
  - Last features can be added and last bugs fixed post launch.
  - Post release update relieves schedule pressure.
- Hardware manufacturing capacity is expensive.
  - Efficient manufacturing capacity scheduling is vital.
  - Manufacturing efficiency increases schedule pressure.

From the definition that firmware is software that

- *Must be functional before the associate hardware ships*

Firmware schedule criticality is like that of hardware.

Opportunity to test firmware is smaller than software

- Pre-silicon: you must
  - Use a software model of the silicon – takes effort
  - Use a simulator – takes time
  - Use an emulator – takes money

- Post-silicon: you must
  - Resolve issues quickly
  - In an environment with less observability than software.

Firmware/Hardware interfaces are "dynamic"

When two software modules are integrated in a system and one hasn't followed the interface:

- The offending module is fixed
- The interface is unchanged

When hardware and firmware are integrated, and the hardware hasn't followed the interface:

- The interface is revised
- The firmware is "fixed"

Opportunity for Formal Methods (the non-technical ones)

- Firmware needs your research
  - Quality must be high.
  - Hard to achieve the needed quality through testing.

- The challenge is to develop methods that:
  - Deliver understandable confidence in predictable time.
  - Are fast to re-apply when there is change.

Opportunity for Formal Methods: Examples

- Static Analysis: Easiest technique to propagate
  - Needs little integration with development environment
  - Assurance achieved is understandable
  - Time to apply is predictable
  - Deals well with change
  - Maximizes value of pre-Si test by removing bugs first
- Dynamic Analysis: Can formal choose the best tests?
  - Pre-Si dynamic analysis is expensive
  - Can static analysis be used to increase the effectiveness?
  - Many practically challenges … see later slides.

Outline

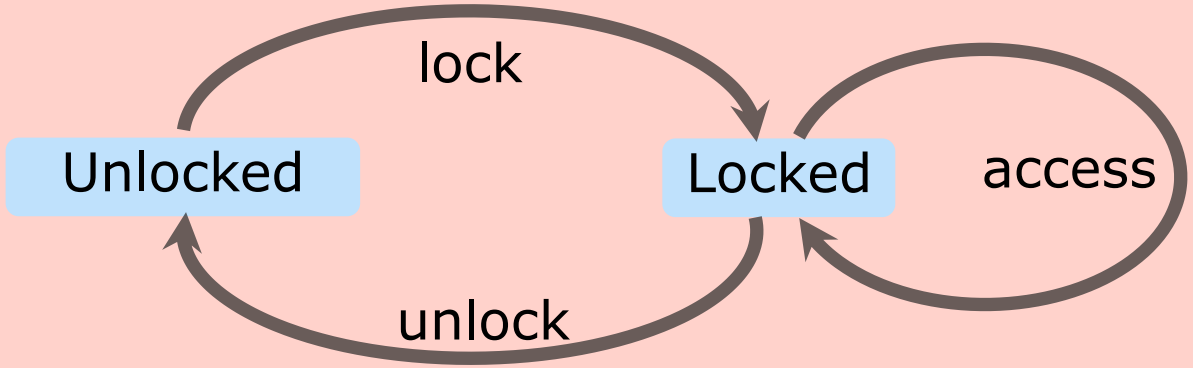
- What is firmware?
- What makes firmware validation so important?
- What does firmware look like?
- Research needs
- Case studies for researchers

Firmware: Language & Runtime

- Firmware is written in C
  - With essential use of embedded assembler
  - Pre-Si test rigs model hardware in a variety of languages
- Firmware runs directly on a microcontroller
  - Most applications have no OS or runtime
  - Usually only a single thread of execution
- Firmware runs with limited resources
  - Do not assume 32-bit μ-controllers
  - Do not assume the μ-controller is fast
  - May need more memory than μ-controller has address bits
  - Stack depth can be limited

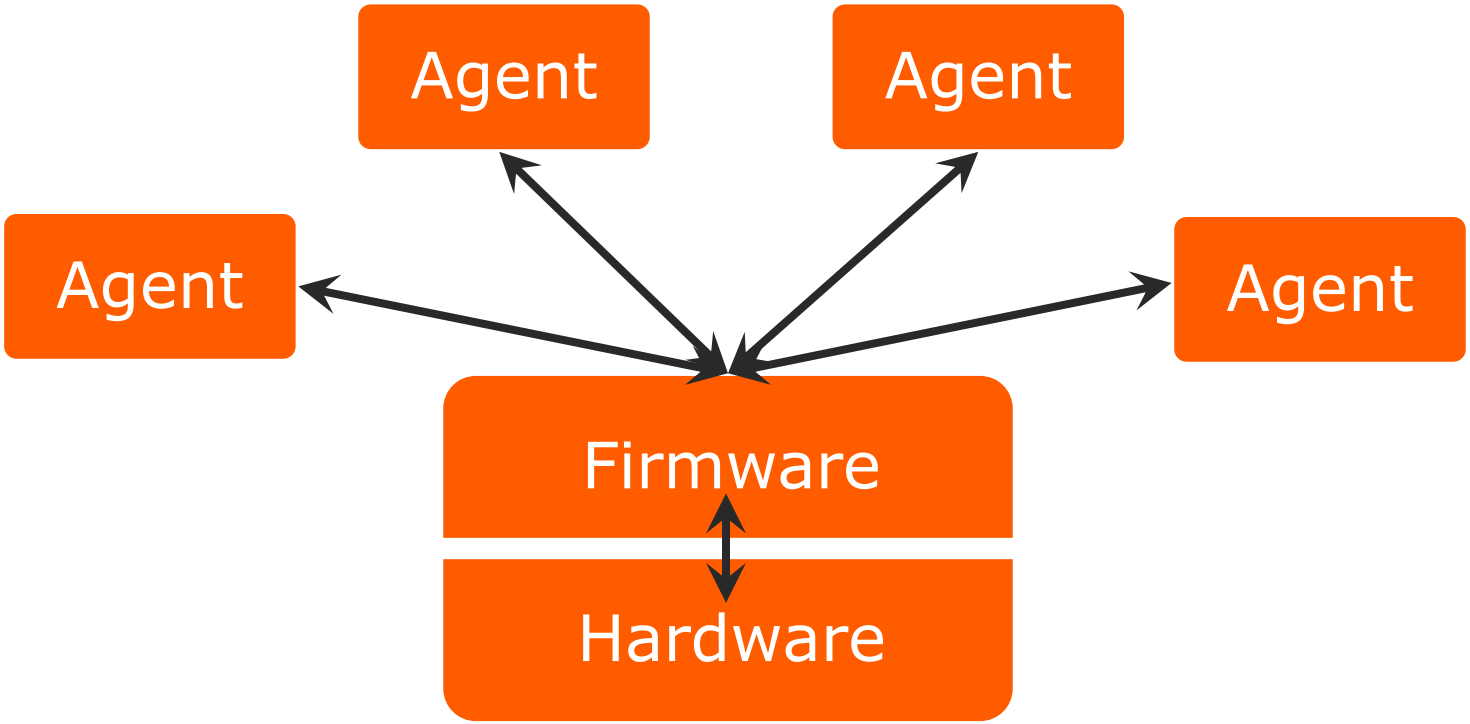
Firmware/Hardware interaction

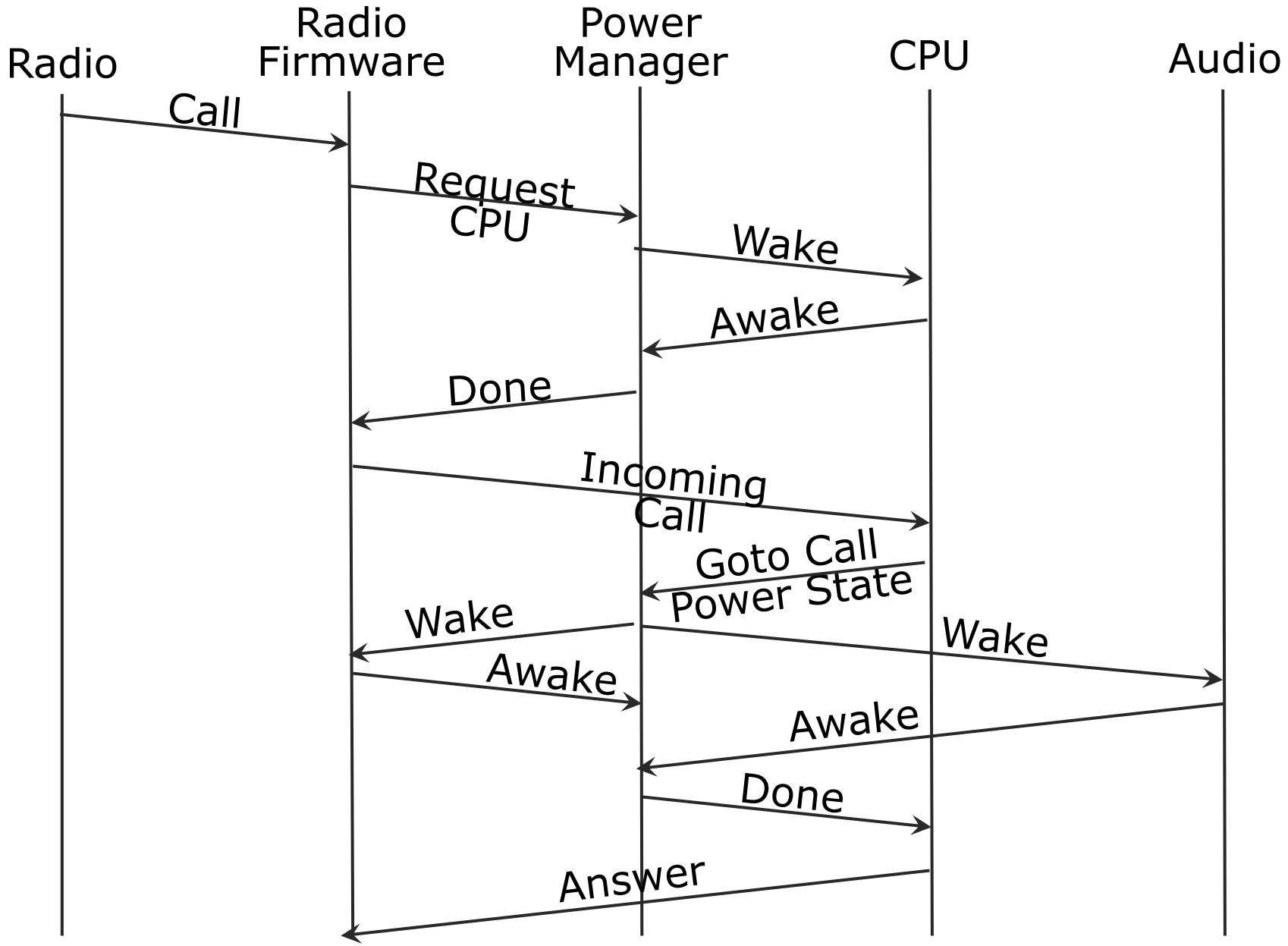
- Software uses standard abstractions to interact
  - Software/software uses standard abstraction: call/return
  - Compilers ensure consistent call/return for each interface
- No standard abstraction to encapsulate interaction
  - Many firmware/hardware interaction patterns exist: interrupts, memory mapped IO, special purpose registers
- Interactions typically have a temporal aspect
  - This occurs in software too – at the problematic interfaces

Easy	Hard
time_t time(time_t*)	

Firmware/Hardware Concurrency

- Firmware operates concurrently with its hardware.
- Firmware operates concurrently with other agents.




Radio
Radio Firmware
Power Manager
CPU
Audio
Call
Request CPU
Wake
Awake
Done
Incoming Call
Goto Call Power State
Wake
Wake
Awake
Awake
Done
Answer

So firmware is like software but…

- Like the software in one agent of a concurrent system.
- But with low-level interface code written by hand.
- Resources are in short supply, and you have to manage them manually.

TRANSITION TO OXFORD INTERFACE EXAMPLE

Interface Checking Example

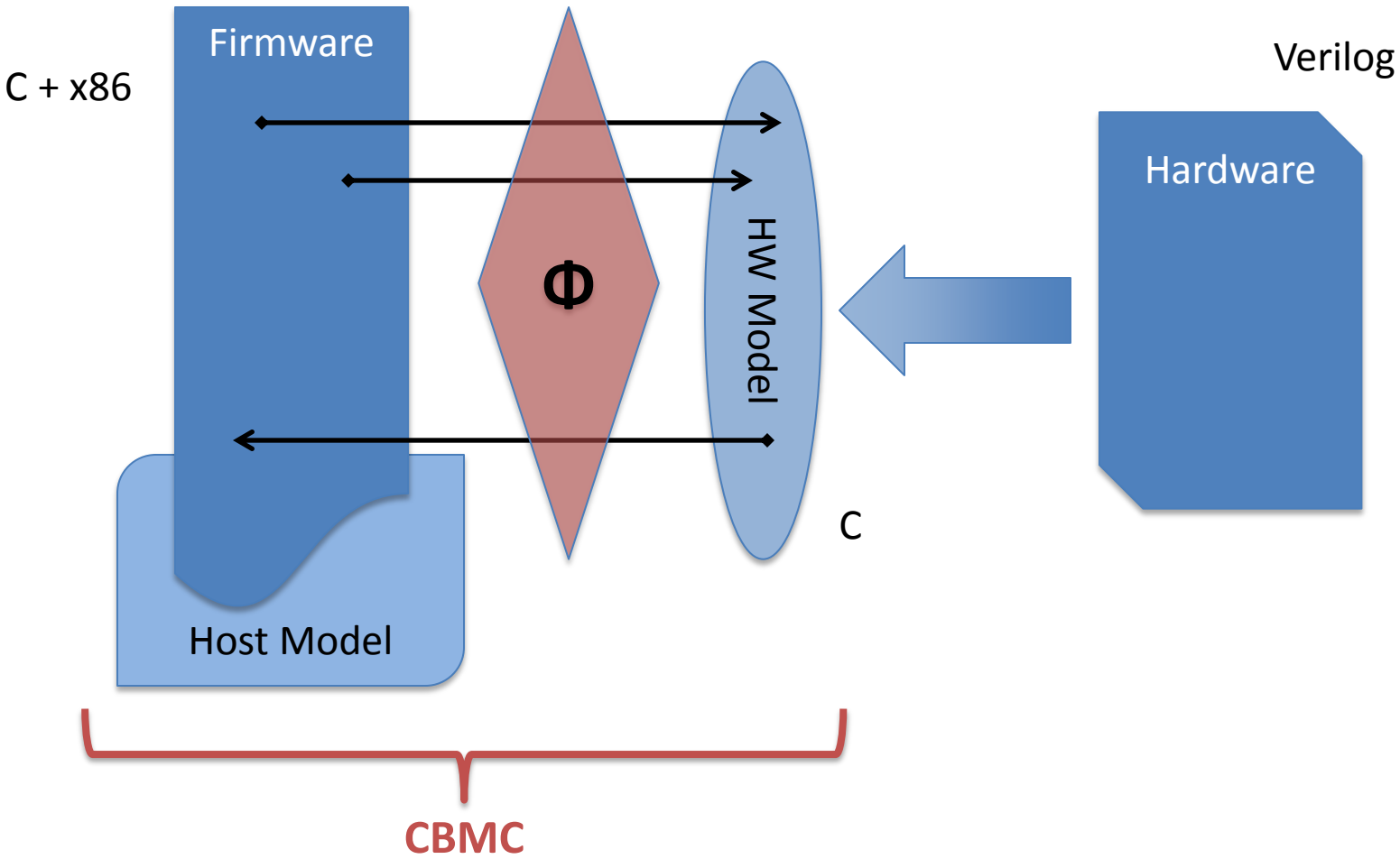
TMP 105 Temperature Sensor

Effective Validation of Firmware

Tom Melham

University of Oxford

Project Approach



TMP105 - Temperature Sensor

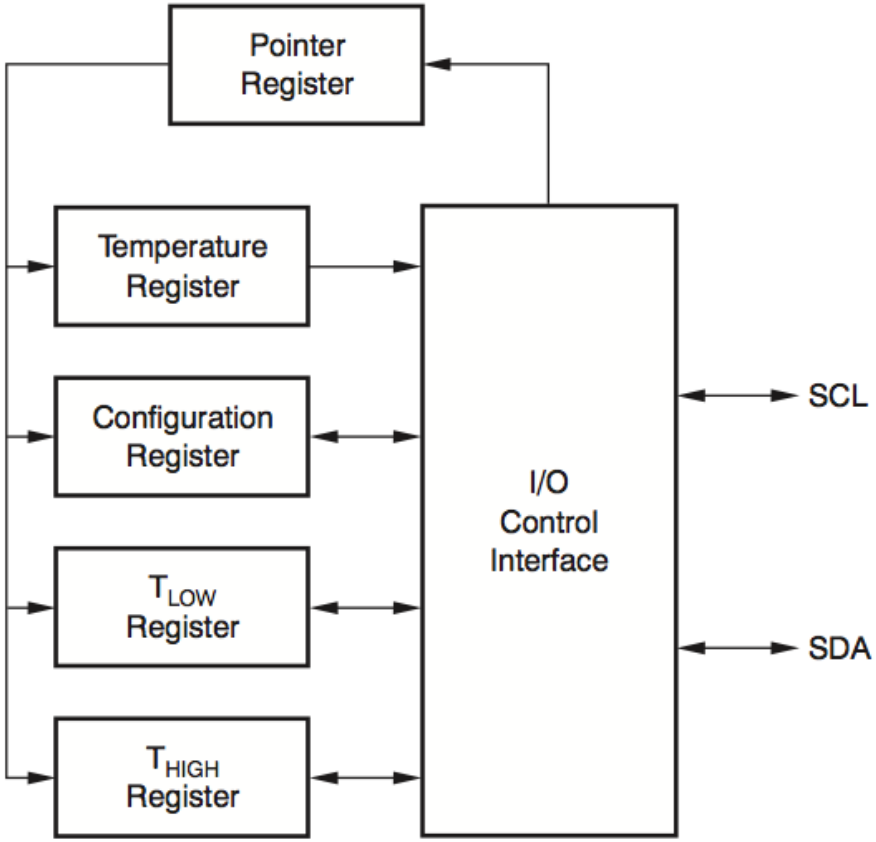
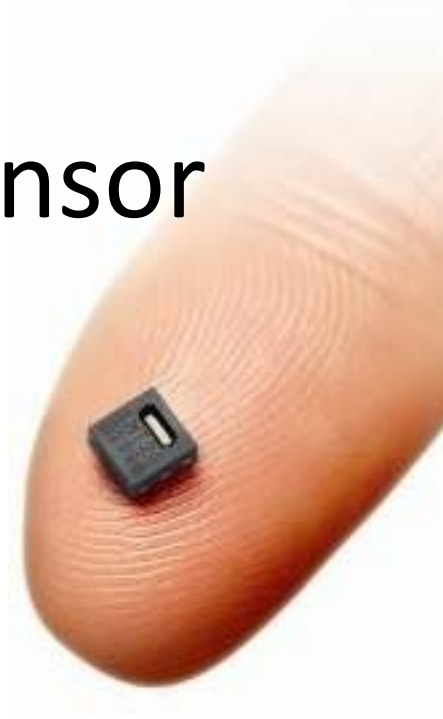


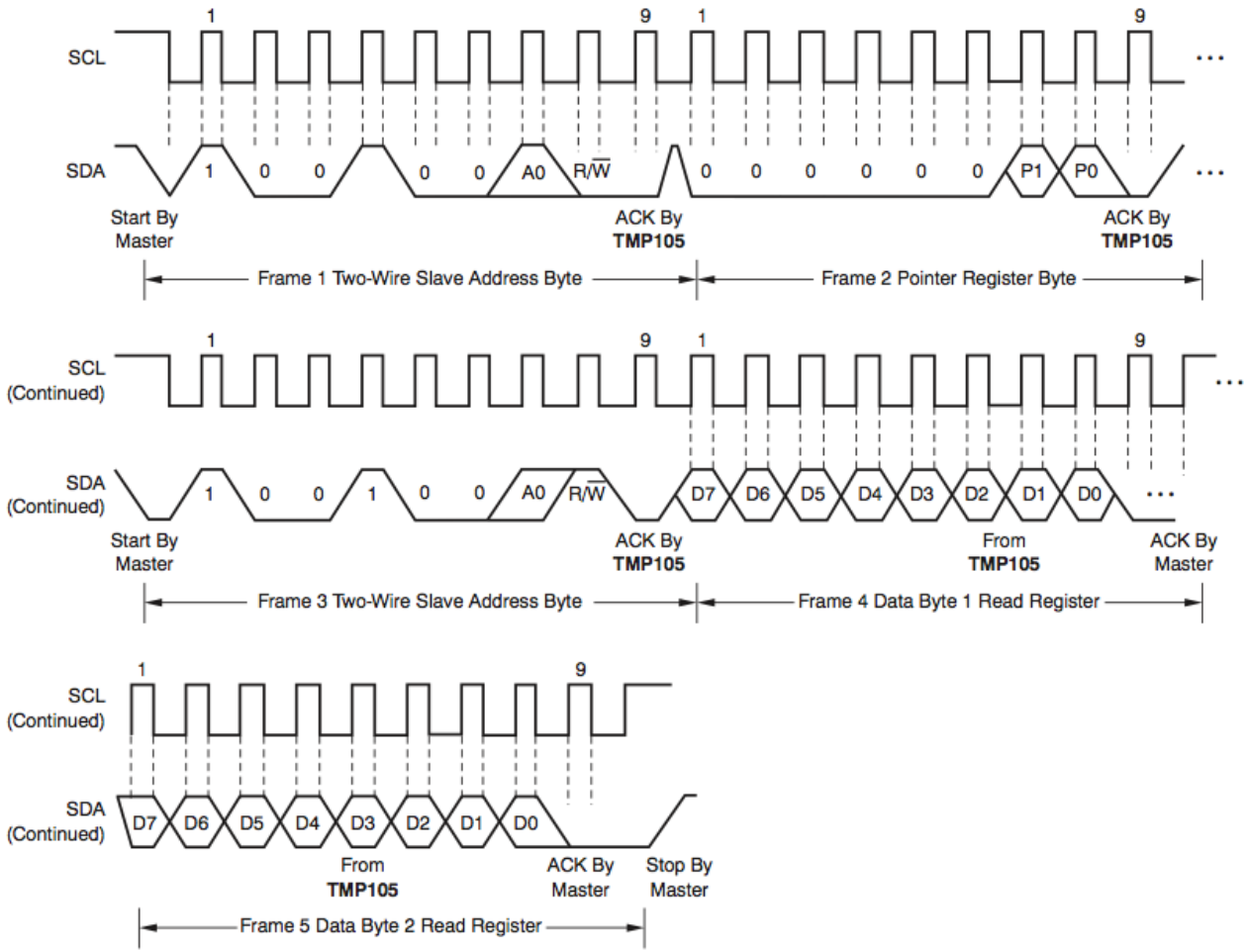
Figure 2. Internal Register Structure of the TMP105



http://www.envirotech-online.com/news/environmental-laboratory/7/sensirion/new_low-cost_humidity_and_temperature_sensor_for_high_production_volumes/20127/

http://www.ti.com/lit/ds/symlink/tmp105.pdf

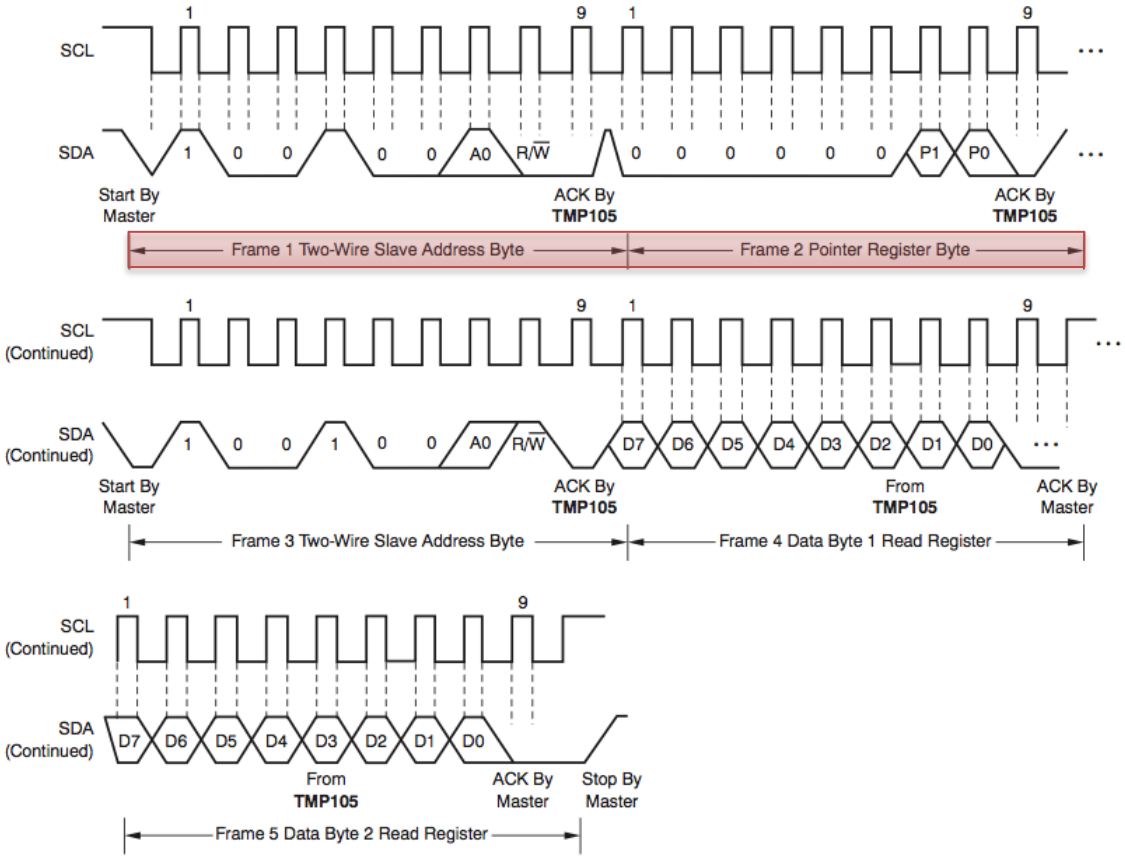
Protocol to Read a Register



http://www.ti.com/lit/ds/symlink/tmp105.pdf

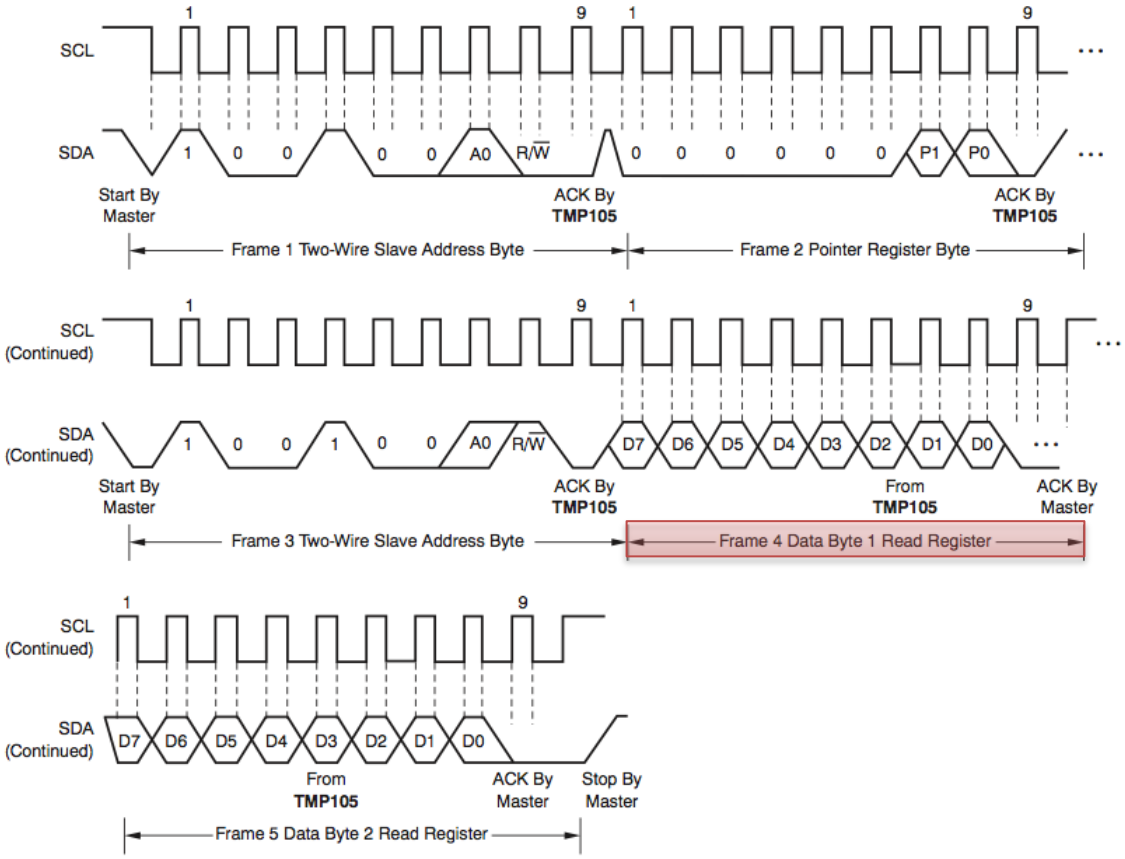
Early Abstraction

1) Write byte over bus to select register from which to read



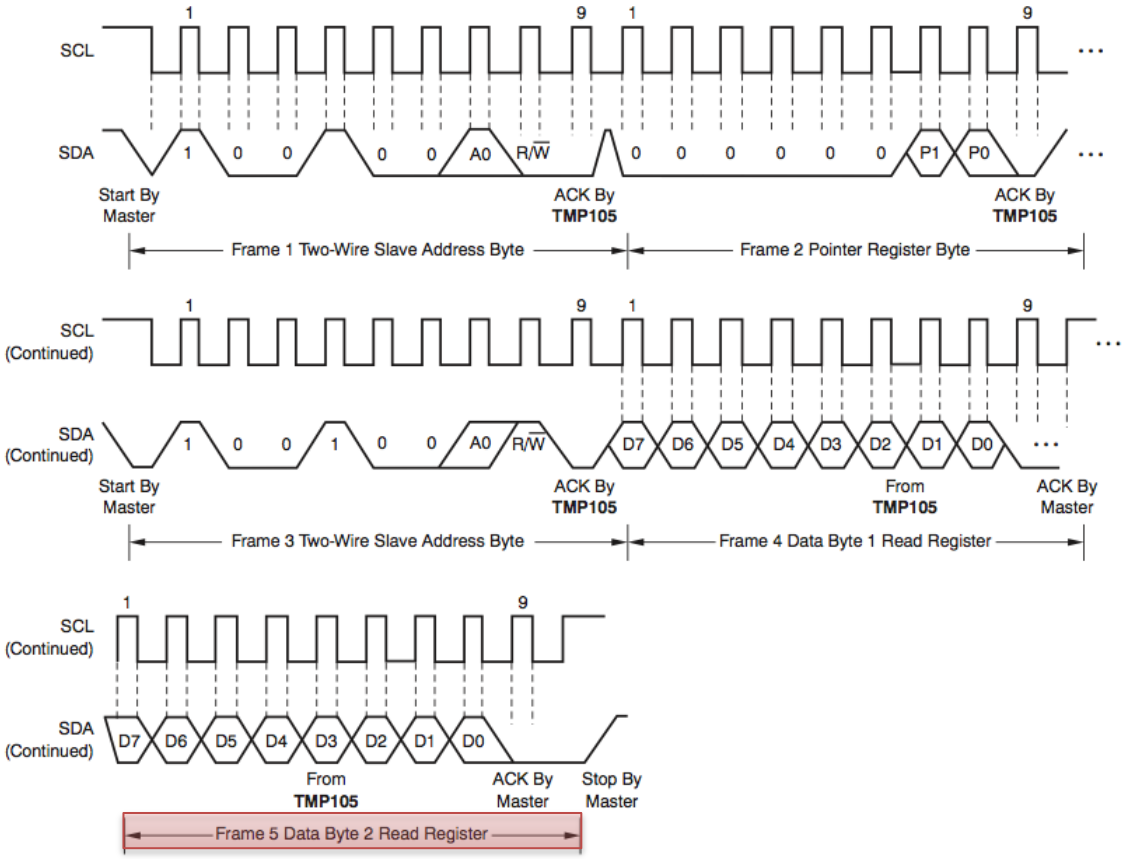
Early Abstraction

1) Write byte over bus to select register from which to read
2) First, read most significant byte



Early Abstraction

1) Write byte over bus to select register from which to read
2) First, read most significant byte
3) Optionally, read another byte …



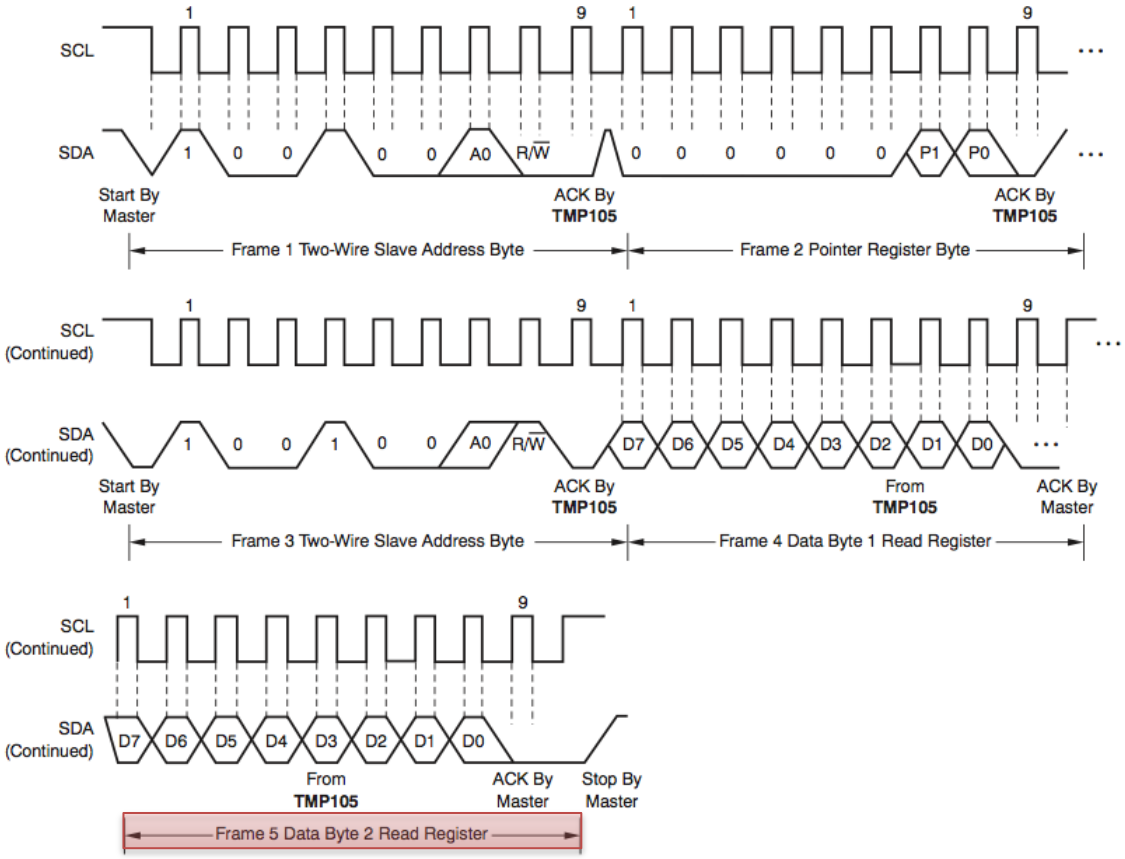
Early Abstraction

Protocol Constraints

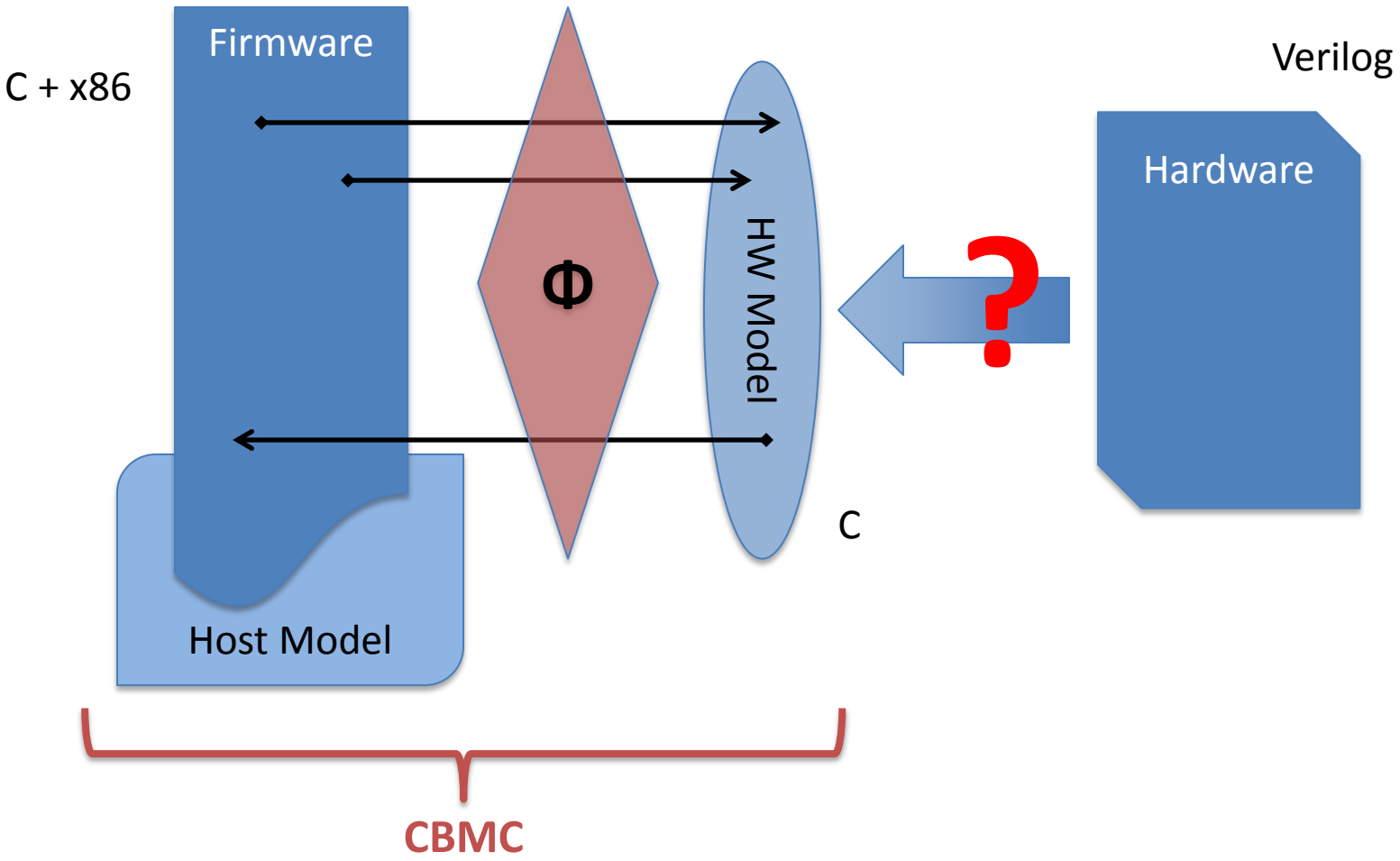
What registers are readable (step 1).

How many bytes can be read (step 2, 3).

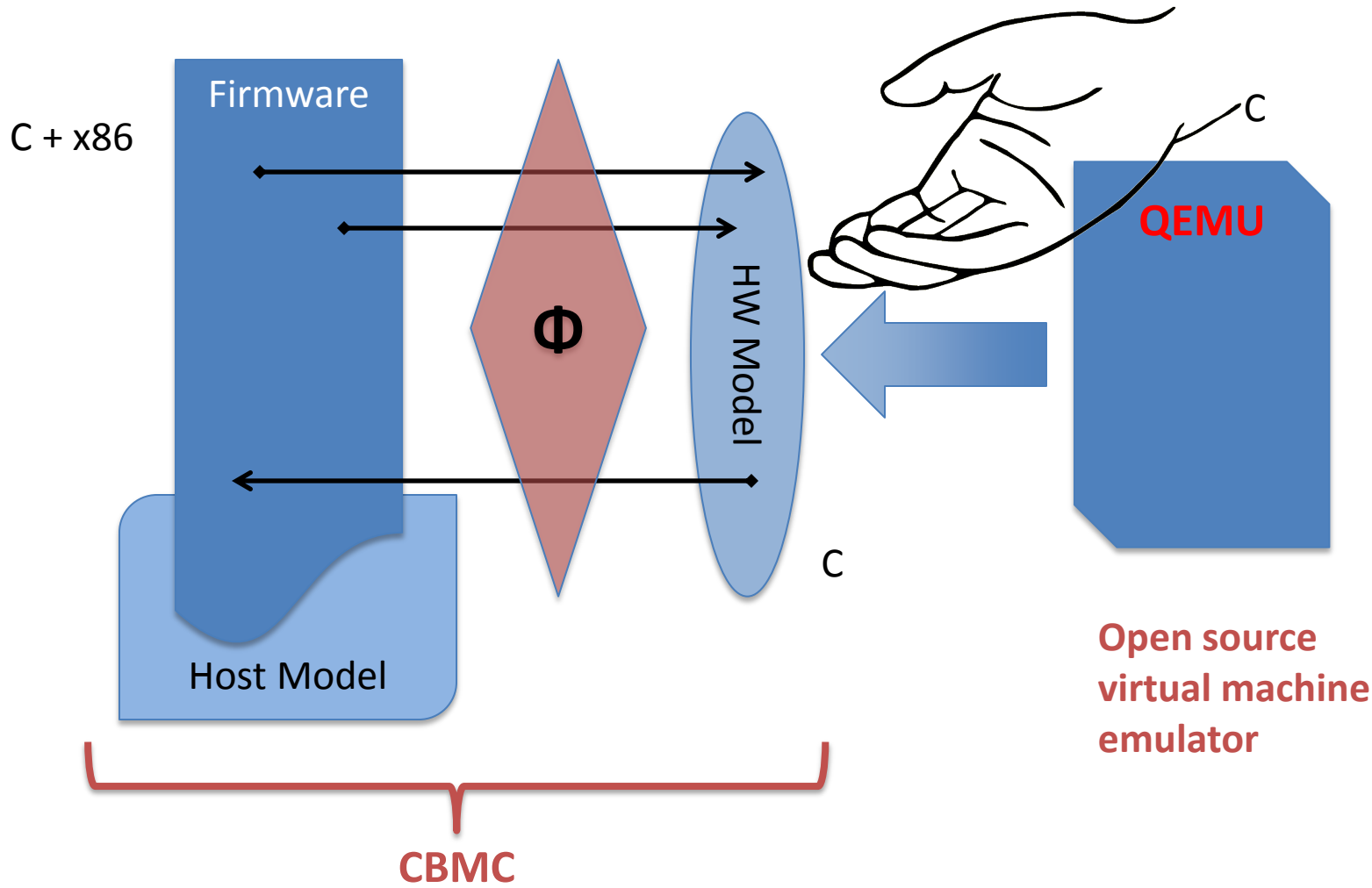
Conditions under which read can be done.



Project Approach



QEMU Virtual Prototype



TMP105 in QEMU

QEMU code of the TMP105 hardware model

=

***language* to express interface properties**

- At most two bytes can be read or written of the temperature threshold registers.
- Exactly one configuration byte can be read or written.
- Each read of the temperature register is preceded by a write of a 1 to the MSB of the configuration register if and only if its LBS is a 1 (i.e. shutdown mode).
- But when the MSB of the configuration register is read, it is 0 regardless of any previous writes to it.

TMP105 in QEMU

✔ Benefits of properties as runtime assertions

- Code more familiar to developers than formulas
- Properties can be also checked by simulation (appeals to existing testing disciplines)
- Unrestricted flexibility

✖ Downsides

- No independent logical semantics
- Extra information may need to be carried in hardware models to check properties

RETURN TO INTEL FIRMWARE TALK

Outline

- What is firmware?
- What makes firmware validation so important?
- What does firmware look like?
- Research needs
- Case studies for researchers

Research: Interface Capture

- We lack a precise language to capture how hardware and firmware request services of each other.
  - IP-XACT* (IEEE 1685*) captures physical aspects
    - Where are the ports (memory addresses, registers, ...).
    - What is their bit-wise layout, and read/write permission.
- Required use patterns are not captured by current languages. E.g., to set the power/frequency of an attached device:
  1. Write a byte (describing a voltage) to some register, then
  2. Write the low-byte of a frequency to another register, then
  3. Write high-byte of the frequency to the same register, then
  4. Write a byte (voltage-domain-id) to a third register

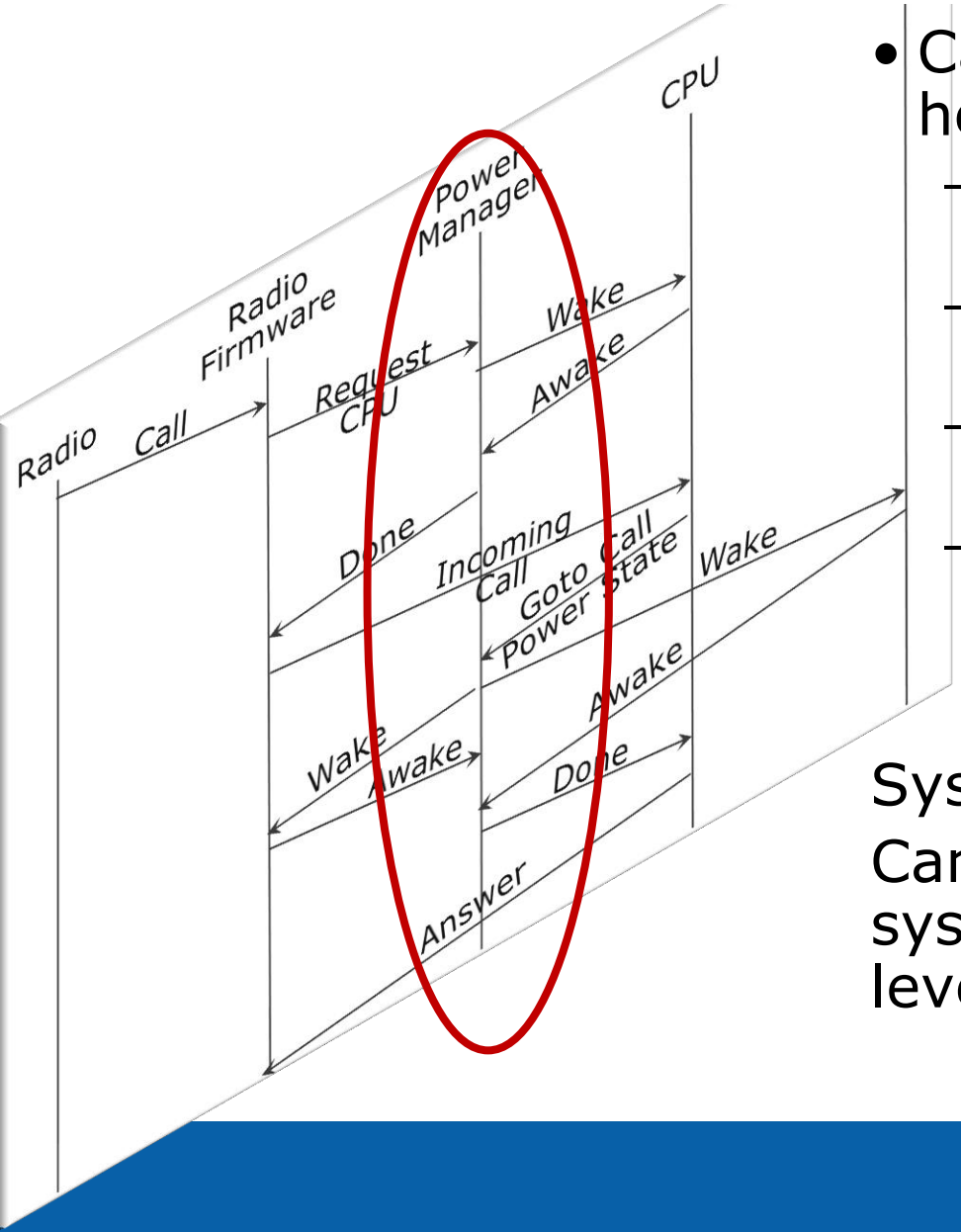
Research: Interface Exploitation

- Can we check firmware complies with the interface?
  - Like driver/OS checking work, but on two temporal levels
    - Sequences of operations to request a service
    - Sequences of service requests
- Can we generate code from the interface to raise the service request abstraction to something like a call?
  - Generate functions for firmware to request a hardware service.
  - Generate polling/interrupt code to call programmer written functions for hardware to request a firmware service.
- Can we generate deep test collateral from interfaces?
  - Tests from IP-XACT exercise only shallow properties.
  - Generating validation collateral from the interface will reduce inconsistency and firmware adaptation later.

Research: Firmware Semantic Tools

- C semantics is rich, but well understood
- But, firmware combines C with inline assembler
  - Tool support is all but absent
- Must be reasoned about with its environment.
  - The environment is hardware.
  - Has a different execution model to (single threaded) C.
- Hardware models exist in challenging languages
  - C++, SystemC, RTL
  - Another reason to capture an abstraction of the interface

Research: System Level Sketches



- Can system-level use cases help validate agent firmware?
  - Infer allowable sequences of messages at each agent.
  - Does agent respond to all legal incoming request sequences?
  - Can agent produce illegal sequences of outgoing requests?
  - Can we generate firmware tests?
    - Long legal prefixes more valuable than random traffic.

System-level validation is hard.

Can system-level use cases find system-level bugs with agent level analysis?

Research Needs: Solutions To "Solved" Problems

Powerful processors, operating systems, and run-times have banished whole classes of bugs from software.

- Virtual memory vs. manual paging.
- Garbage collection vs. manual memory management.
- Pre-emptive scheduling vs. manual scheduling.

Can formal solve resource problems another way?

- Generate or check paging code.
- Check memory allocation and return.
- Generate or check static scheduling of code.

Research: Languages

- C dominates: can we do no better after 40 years?
  - Domain favors static analysis; can a language enables more static analysis?
  - Eco-system (libraries, debuggers, …) and experience too valuable. Keep ABI:
    - Object file format
    - Data sizes, alignments
    - Calling conventions
- C dominates because it fits the domain
  - Procedural/imperative programming
  - Small runtime (essential libraries)
- Opportunity: Reduce the need for inline assembly
  - Raise the abstraction level and avoids bugs
  - Blocks formal analysis at a key point: the hardware/software interface
- Why do people still write assembly?
  - Precise memory accesses: reads, writes, and their order effect hardware
  - Precise bit-level layout: bit-fields give insufficient control of layout and access
  - µ-controller specific IO features (instructions/registers/interrupts)
- Opportunity: Type systems add value without dynamic overhead
  - Correct operation sequencing is key, can types help?
  - Pointer manipulation remains error prone, types can help.
  - Resource uncertainty drives impacts programming and hinder reuse.

Experience: Concolic Firmware Test

- What:
  - Test generation to maximize code coverage by symbolic analysis of concrete traces
- Why:
  - Developers understand testing and coverage
  - Finds bugs without complete specifications
- Can scale to firmware and can find bugs. But…
  - #1 issue is understanding the HW/SW interface
    - Every firmware project has a different HW/SW interface
  - Software models of hardware may exist, but maybe not C
  - Embedded assembly impacts source or IR trace analysis

Outline

- What is firmware?
- What makes firmware validation so important?
- What does firmware look like?
- Research needs
- Case studies for researchers

Research Case Study Selection

- Hard to find case studies to drive firmware research
  - Embody full and realistic firmware systems
  - Are open to academic examination
- Few open source firmware projects
  - Most have a single agent rather than a system of agents
  - Most assume a full embedded Linux stack
- Researchers need to develop own case studies
  - Large enough to include elements of real firmware
  - Small/modular/fun enough to be built by student teams

Open Source Inspiration: RockBox

Open source firmware for personal music players

- Can serve as an exemplar of firmware code

✓ Programmed in C with a little assembler
✓ Assumes relatively constrained resources
✓ Provides its own runtime and scheduler
✓ Abstracts device communication steps into services

✗ Hardware interface models are absent
✗ Application and device interaction combined
✗ Single agent style of firmware

Open Source Inspiration: QEMU

Software model of a PC, including common devices

✓ Can serve as a source of analyzable device models

✓ Firmware/Hardware issues recur at Device/Driver

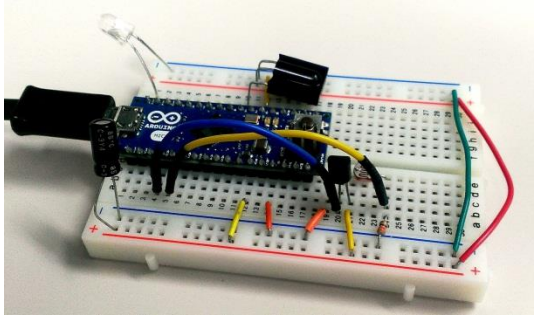
- Driver/Kernel interaction well studied but Device/Driver less

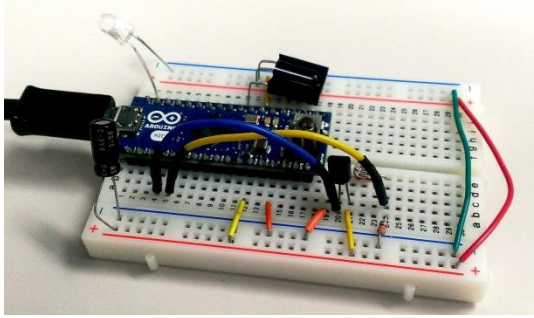
✗ Not all Hardware/Firmware issues recur with Drivers

- ✗ Firmware exists to abstract interfaces for drivers
- ✗ Firmware has finer grained control than exposed to drivers
- ✗ Firmware runs in resource constrained environment
- ✗ Drivers have less need for embedded assembly

✗ Faux OO coding style is atypical of firmware

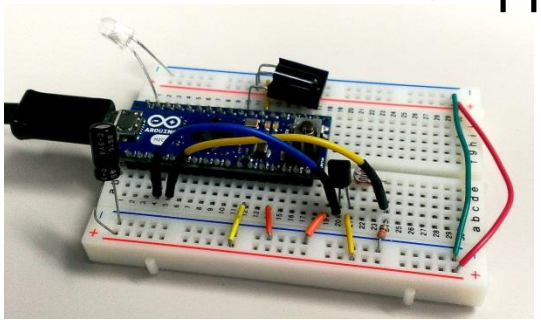
Realistic Firmware Case Study

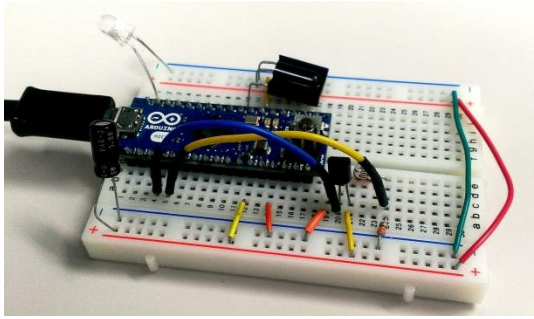




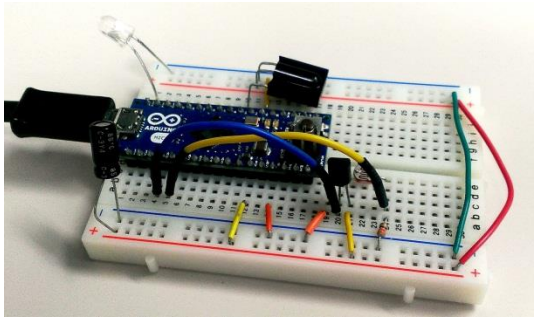
Multiple 8-bit/16-bit µ-controllers

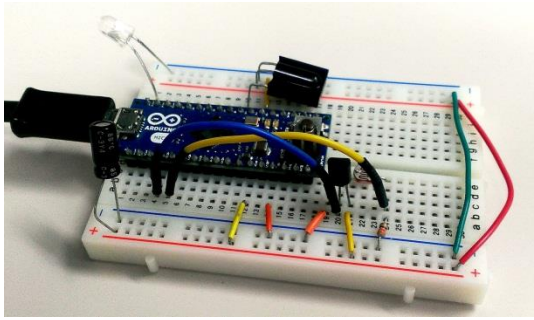
- Attached devices
- All device control local
- FPGA a bonus



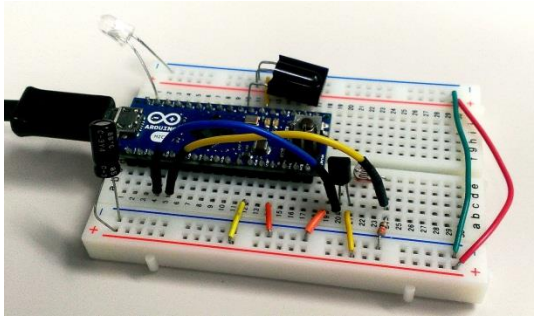


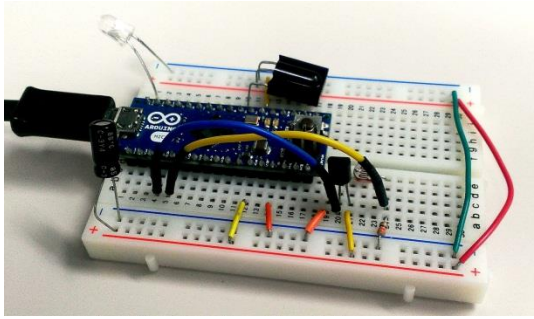
Realistic Firmware Case Study



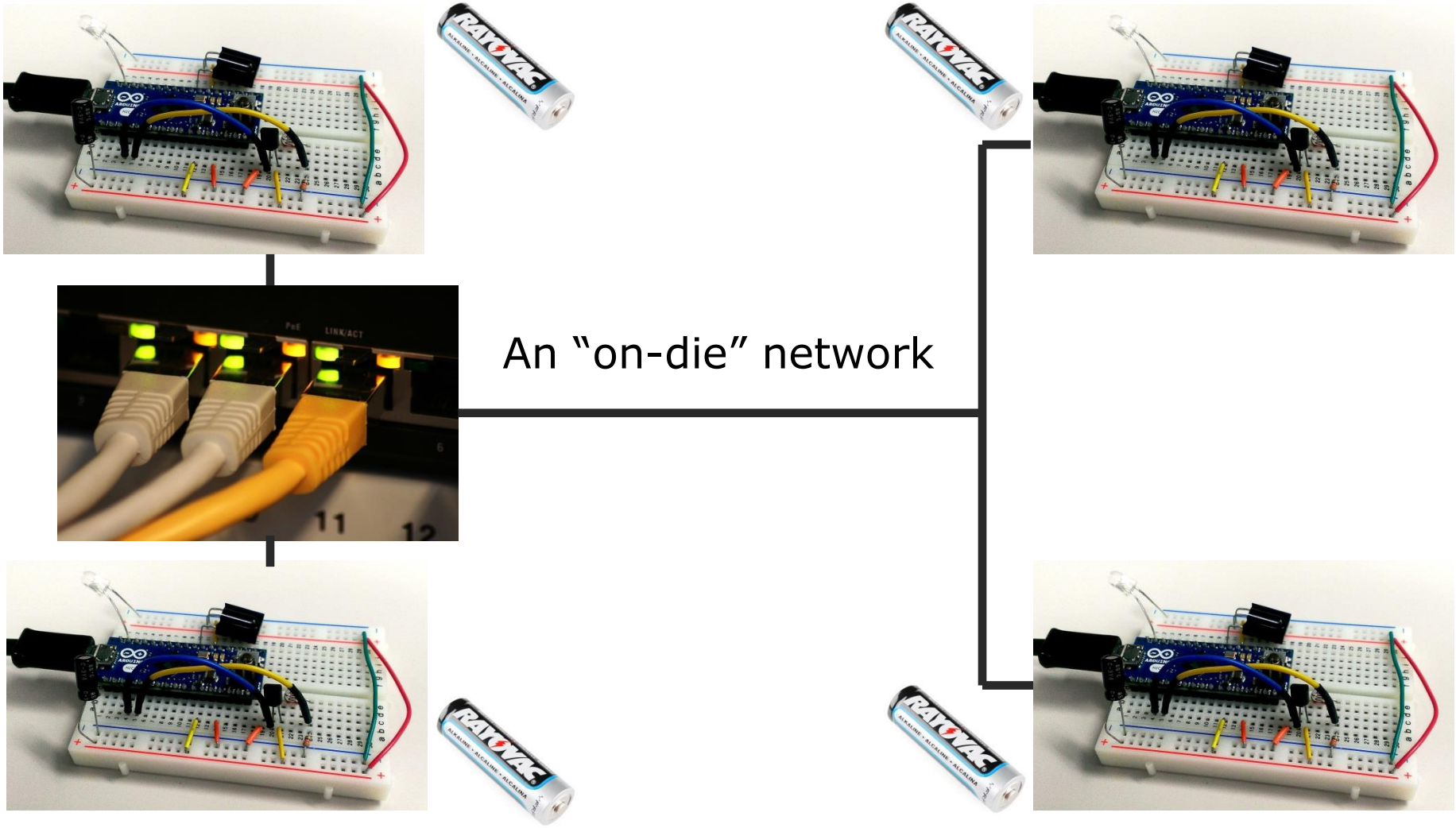


A limited power budget for them

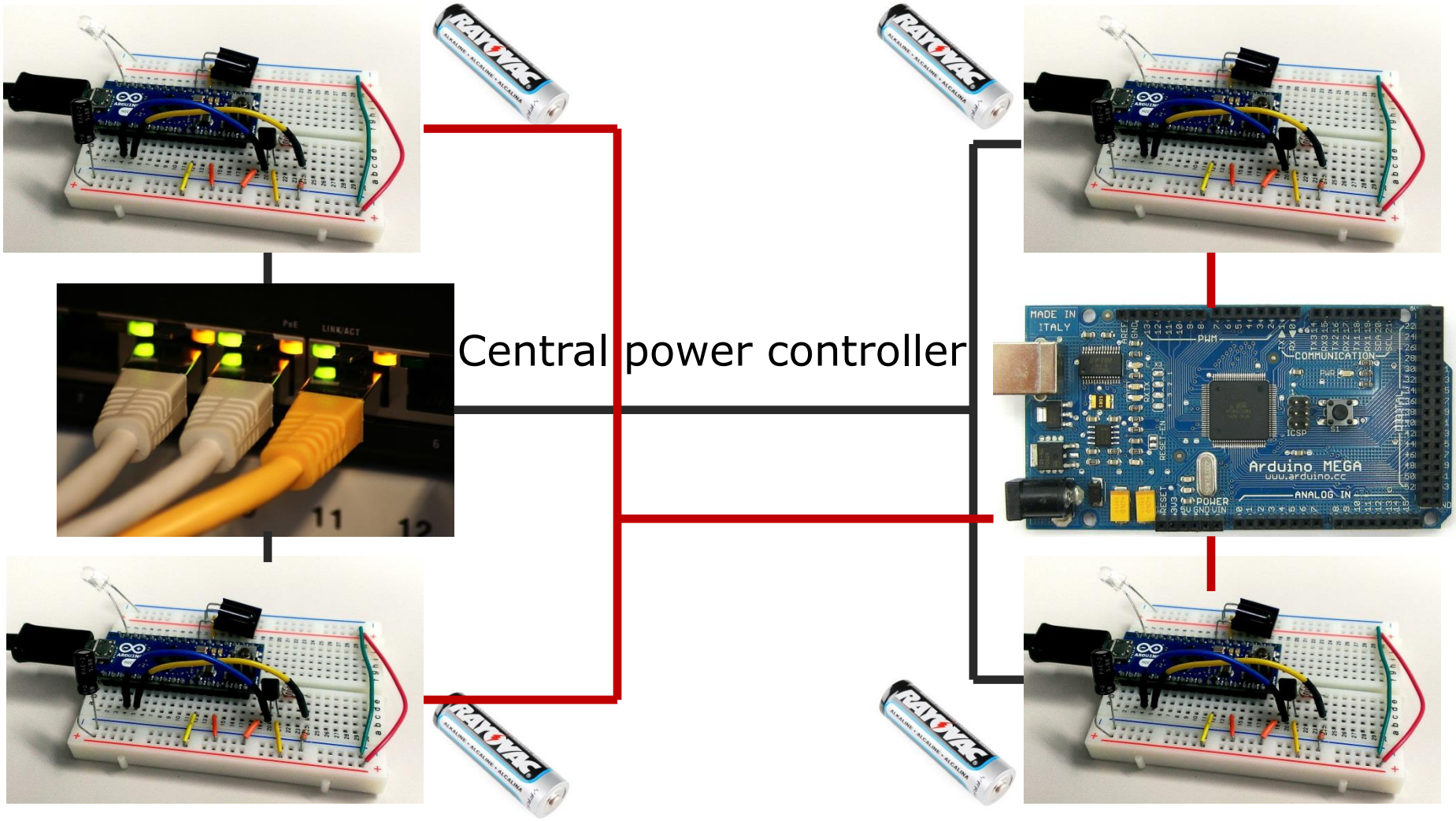




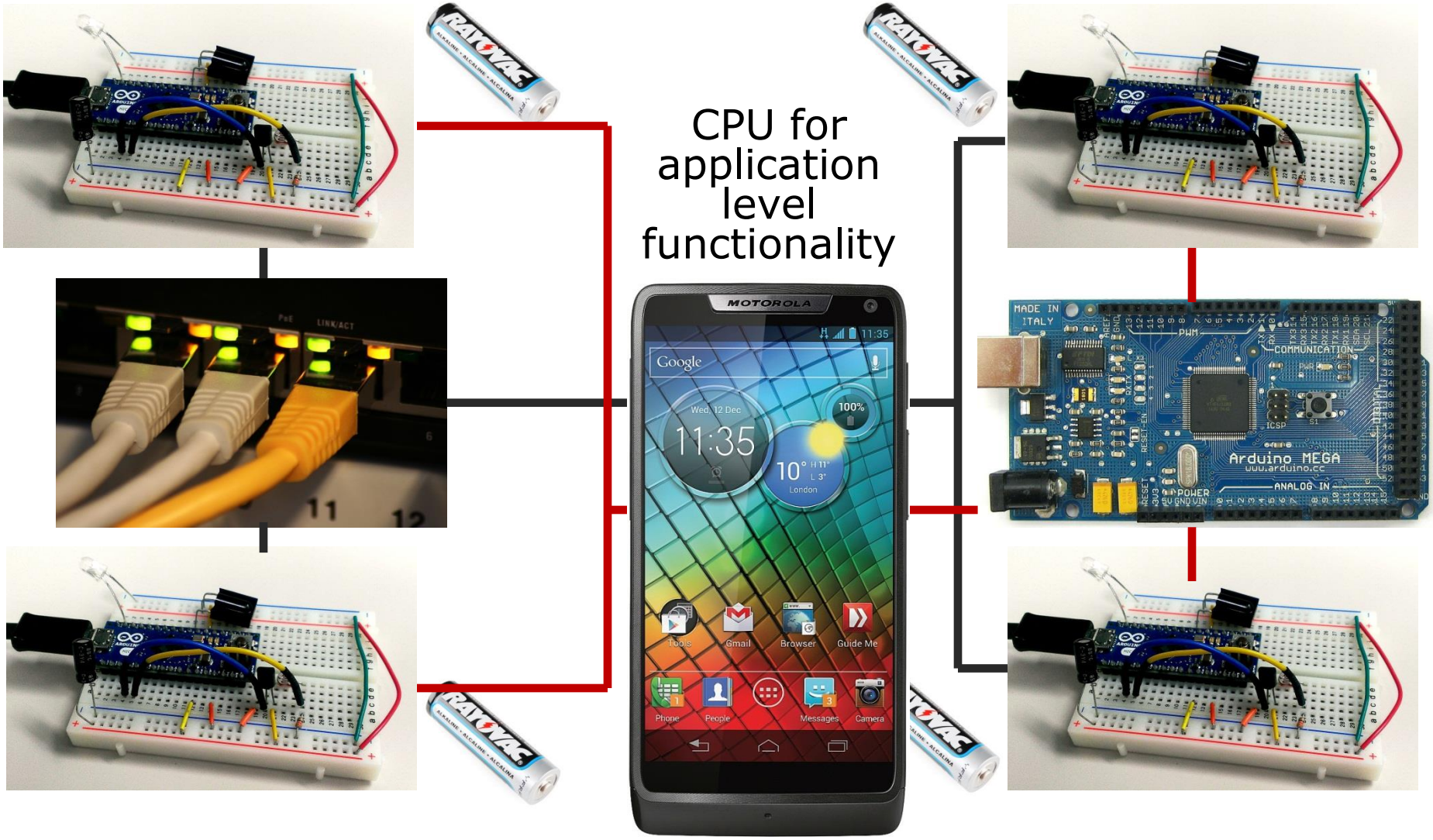
Realistic Firmware Case Study



Realistic Firmware Case Study



Realistic Firmware Case Study



Example: Multi-zone audio service

- Per-room firmware
  - 8-bit/16-bit μ-controller
  - Decode audio streams locally
  - Sensors detect presence of listeners for power savings
  - Basic function (play/pause?) that requests high level help
- Central firmware services
  - Power management
  - Audio storage and streaming
  - Firmware update and distribution
    - Check firmware updates are authentic, consistent, fresh
- Application services
  - User interface

Questions?