Embedded OSes Must Embrace Distributed Computing

Branden Ghena Jean-Luc Watson Prabal Dutta

brghena@berkeley.edu jeanluc.watson@berkeley.edu prabal@berkeley.edu

Next-Generation Operating Systems for Cyber-Physical Systems 2019

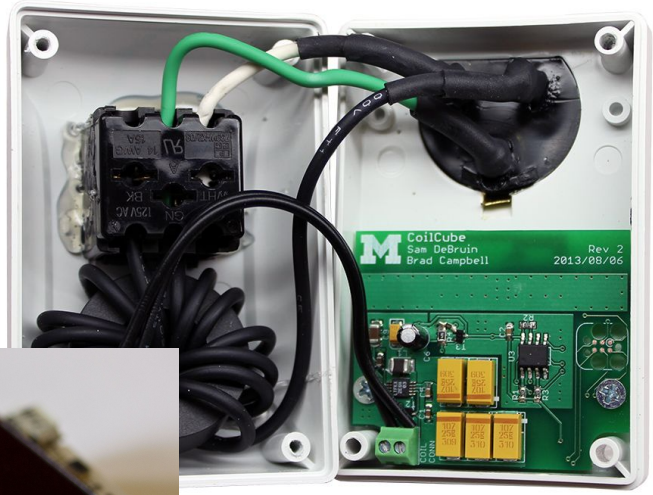
We’re coming from the world of very limited computing

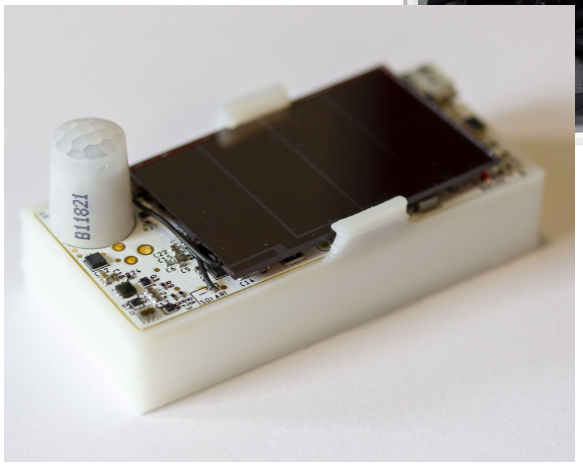
Resource-constrained embedded systems

- Sensor deployments
- Localization systems
- Internet of Things devices

Microcontroller-based systems

- Cortex-M class systems
  - 48 MHz
  - 64 kB RAM
  - 512 kB Flash

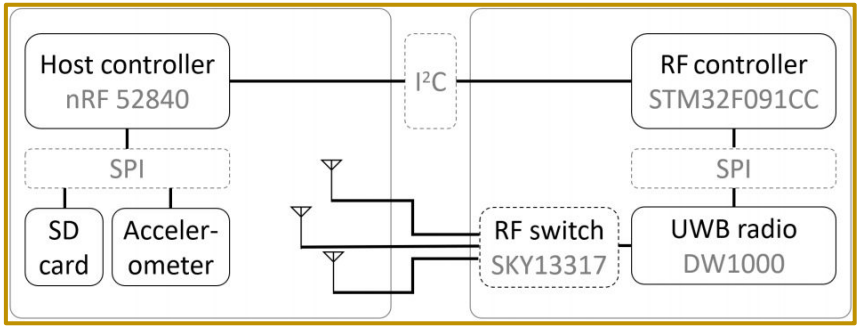




Embedded systems are embracing multi-microcontroller solutions

Design paradigm: separate microcontrollers for each task

- Multiple microcontrollers on a single circuit board
- Sensing and communication tasks
- Avoids complex interactions altogether



Embedded OSes will need to support this paradigm with new primitives

- Inter-microcontroller communication
- Task migration
- Platform management

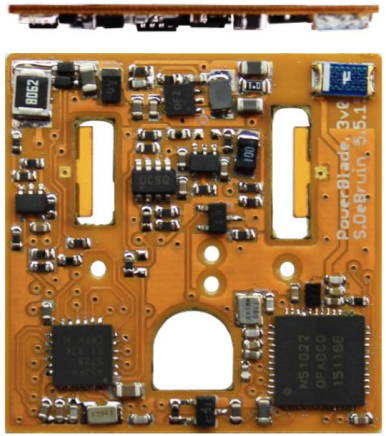
Outline

- System design story
- Necessary OS primitives
- Where we are today

The design of a multi-microcontroller system

Powerblade

- Plug-load power meter
- 1” x 1” x 1/16” form factor (US plug size)
- Measures current and voltage in real time
- Transmits data over BLE
- Limited power budget



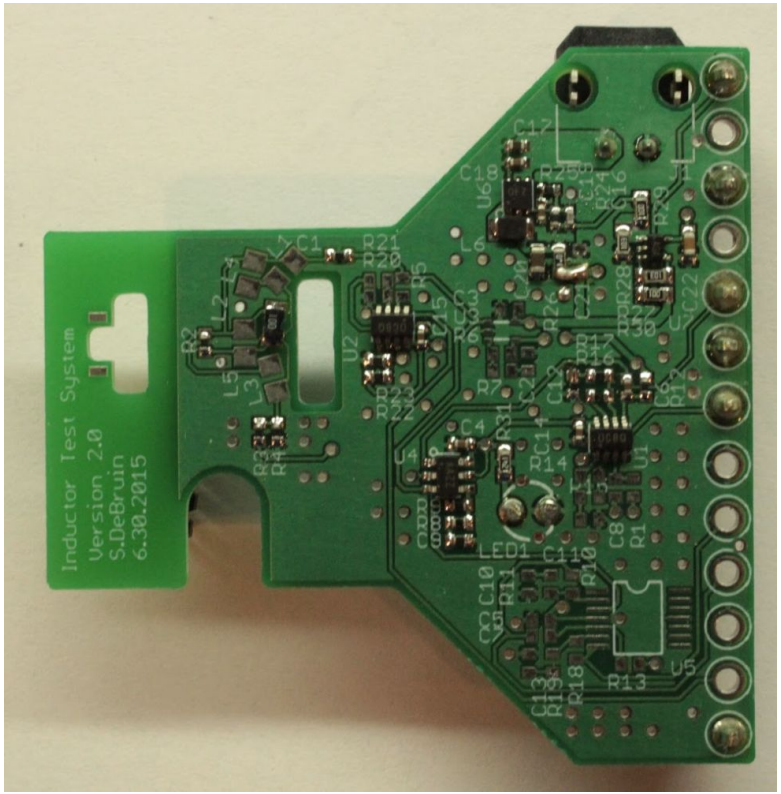
Two microcontrollers

- MSP430 - Analog sampling
- nRF51822 - BLE communications

Step one: a proof of concept

Does it work at all?

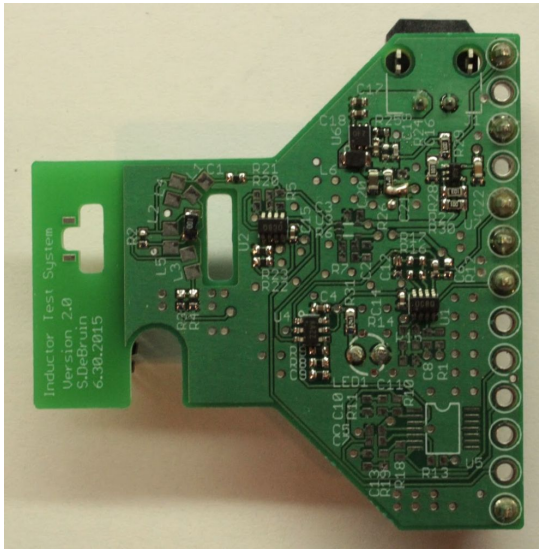
Just throw an MSP430 on there because we're familiar with it.

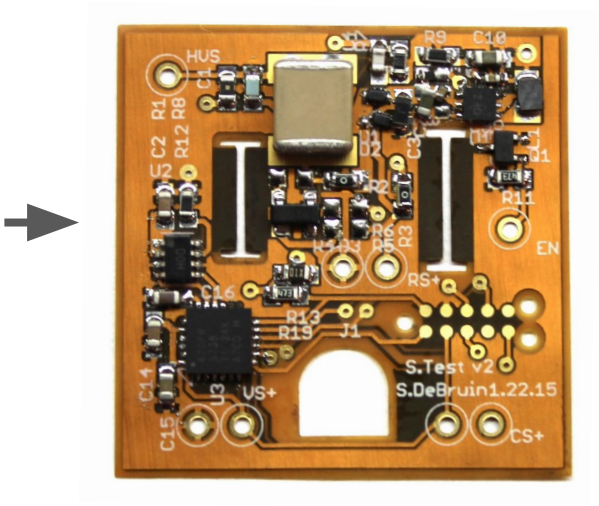


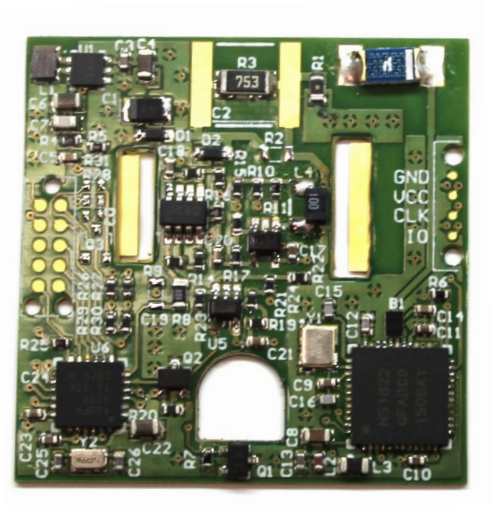
Step two: make a deployable system

Make it the right size

Make it wireless







Adding wireless leads to a second MCU

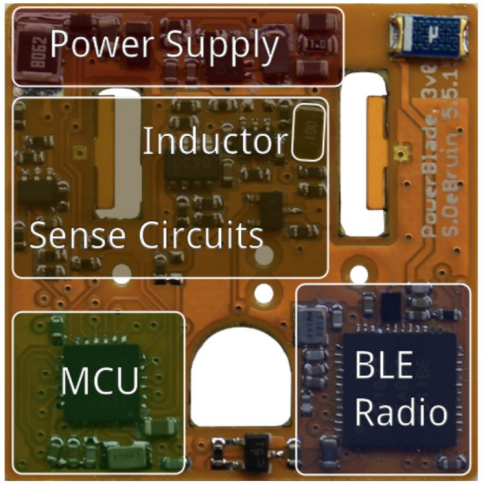
MSP430 is already a maximum capacity

- Radio peripheral could mess up sampling timing
- Radio ICs are a lot harder to use than modules anyways

Could rewrite sampling code on new microcontroller

- Probably powerful enough to do both
- Requires a lot of engineering to port
    - Especially to make it reliable

And now a second microcontroller gets added



Inter-microcontroller communication gets complicated

Need to shuttle messages back and forth

- Sensor readings
- Get version number
- Clear cumulative energy value

Make a state machine on both sides

- Pull apart bytes
- Decide actions
- Perform response

```
void on_receive_message(uint8_t* buf, uint16_t len) {
    if (len > 0) {
        // switch on add data type
        uint8_t data_type = buf[0];
        switch (data_type) {
            case UART_NAK:
                status_code = STATUS_GOT_NAK;
                simple_ble_notify_char(&config_status_char);
                nak_state = NAK_RESEND;
                break;

            case START_LOCALC:
                // write status to characteristic
                calibration_control = 1;
                simple_ble_notify_char(&calibration_control_char);

                // MSP has begun interal calibration
                if (calibration_state != CALIB_STOP) {
                    calibration_state = CALIB_CONTINUE;
                }
                break;

            case CONT_LOCALC:
                // MSP is still running internal calibration
                if (calibration_state != CALIB_STOP) {
                    calibration_state = CALIB_CONTINUE;
                }
                break;
```

Step three: add more features

Automatic device configuration

- Time to revise both state machines

Over-the-air firmware updates

- There are libraries to do this for the nRF microcontroller
- But how do we get updates to the MSP430?

Watchdogs

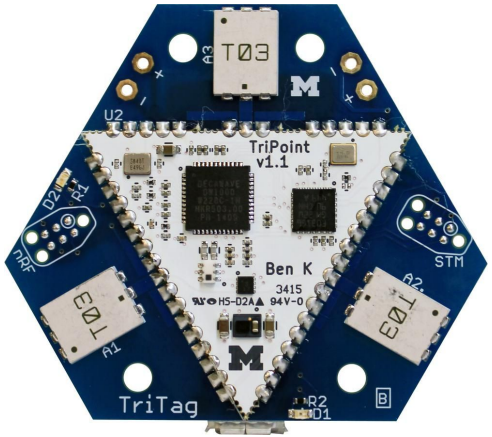
- How should the MSP430 tell if the nRF has crashed?

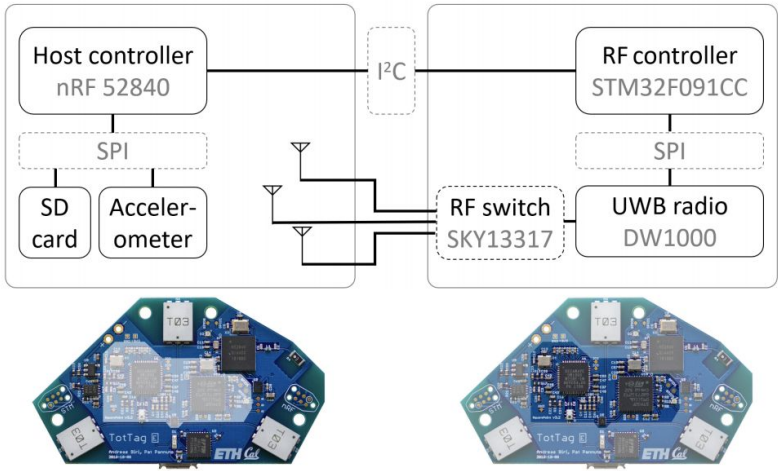
Each of these end up being custom software

There are many twists to this same story

Polypoint/TotTag

- Localization hardware
- One microcontroller for wireless communications (BLE)
- One microcontroller for ultra wideband localization

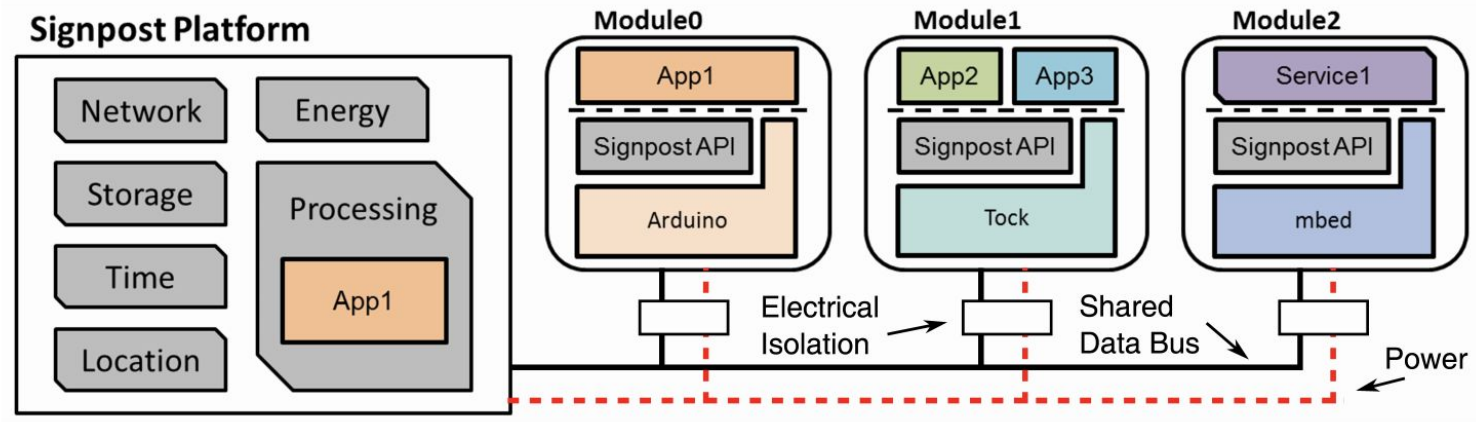




There are many twists to this same story

Signpost

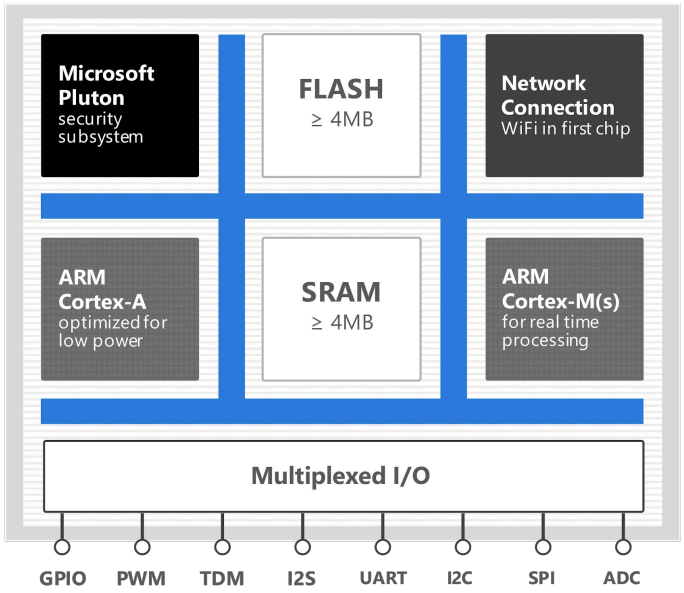
- City-scale sensing platform
- One microcontroller for each application
- Additional microcontrollers for services



There are many twists to this same story

Azure Sphere

- Secure, reliable Internet of Things
- Microcontroller for managing security
- Microcontroller for networking
- Two application processors
  - General purpose
  - Hard real-time needs
- Most may have access to shared memory, which makes it more traditional



Embedded OSes should support these designs

What are the primitives that can do so?

1) Enable microcontrollers to communicate

Communication has different requirements from normal distributed case

- Topology of the system is set at design time
- Reliability of the channel and processors is very high (but never perfect)
- Energy cost for communication is important

What is the right abstraction?

- Shared memory won't work here
- A common interface is `send_msg()`
  - But doesn't solve state machine problem
- Single microcontroller systems just use function calls
  - Has implications for OS runtime
  - Results in callable run-to-completion tasks, possibly in parallel with a main thread

2) Enable migration of tasks

Some embedded tasks are tightly coupled with their microcontroller

- Real-time access to external signals
- Energy-constrained operations need microcontrollers to sleep

But there are other classes of tasks

- Data processing and filtering
- Control algorithms
- Machine learning

Even compile-time migration would be valuable

- But at run time allows adaptation to deployment conditions

3) Provide system management tools

Firmware updates are hugely important

- But often only one microcontroller is connected to the network
- Need to ship updates to other microcontrollers on the system
  - Another task for the inter-microcontroller communication bus

Watchdog functionality too

- Multiple microcontrollers mean multiple points of failure

Danger: distributed systems aren’t known for their simplicity

Need to ensure we don’t make challenges harder

- Messages have latency and priority that should be obvious
- Task migration is complicated by hardware peripherals
- System management could mean one compromised chip takes down the whole system

The goal was to make application design *easier*

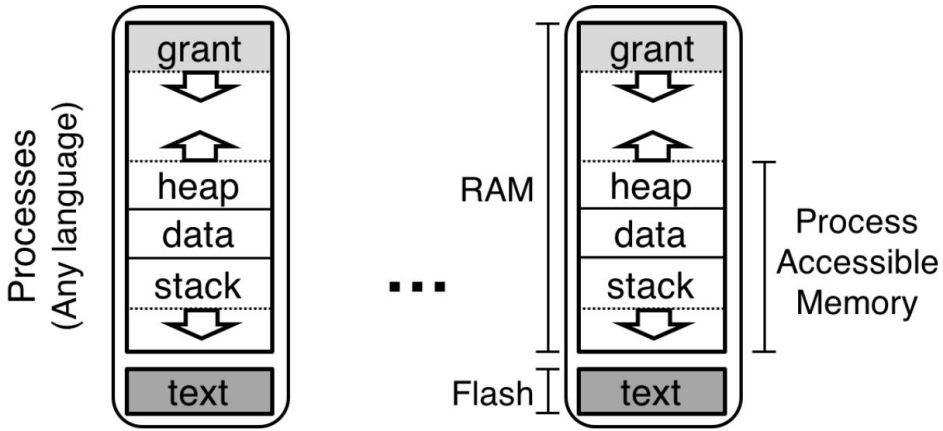
- But distributed challenges could be more difficult than monolithic ones

OS support today is limited

Tock enables compile-time task independence

Tock

- Allows for kernel-agnostic applications
- The same code can be reused on many different platforms
- Calls across kernel boundaries could be a natural point where external communication occurs



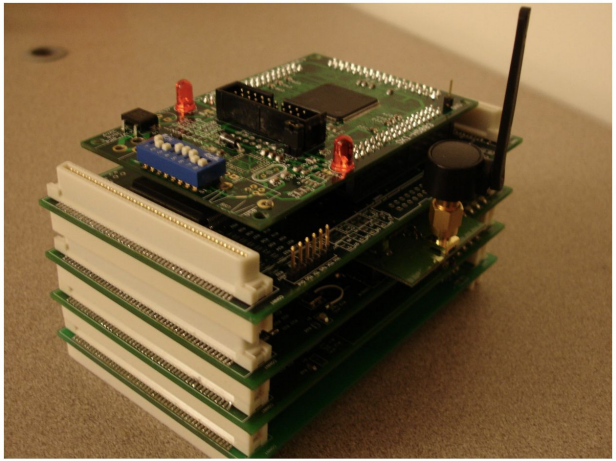
CoMOS explores runtime task migration

CoMOS and mPlatform

- Microcontrollers: 4x MSP430, 1x ARM7
- Sound source localization application

Moves an FFT task between microcontrollers depending on latency requirements

- Improves energy efficiency



Are these OS primitives or libraries?

Libraries could (and do) implement some of these needs

- Firmware updates
- Communication protocols
- Serialization and RPC libraries

OS support would allow better abstractions

- The inter-microcontroller bus is a resource to be managed
  - With latencies and priorities
- Making some tasks hardware independent is also an OS job
- Libraries would be a good start though!

Conclusion: embedded OSes need to support multi-microcontroller systems

- Real-world embedded systems are adopting multi-microcontroller designs to reduce complex interactions

- Today leads to increased development needs to handle interactions

- We see a need for embedded OSes to provide better support

Embedded OSes Must Embrace Distributed Computing

Branden Ghena brghena@berkeley.edu
Jean-Luc Watson jeanluc.watson@berkeley.edu
Prabal Dutta prabal@berkeley.edu

Lab11.eecs.berkeley.edu --- github.com/lab11

Next-Generation Operating Systems for Cyber-Physical Systems 2019

Embedded OS support in general is lacking

Embedded, resource-constrained OSes are still very limited

- TinyOS, Contiki, FreeRTOS, etc.
- Lots of bare metal programming

Not so much beyond POSIX as haven’t reached it yet...

Embedded systems abstractions need to be different

General distributed systems are often about capability

- Make multiple computer looks like one more powerful computer
- Abstractions make the boundaries disappear whenever possible

Embedded systems need *opaque* abstractions

- Make the challenging parts appear seamless
- Keep the ramifications of actions apparent to the programmer

Sounds great, but how do we do this?

Questions to lead the audience with

- Implemented by the OS or a library?
- When do distributed systems become harder challenges than monolithic ones?
-