6.S08: Interconnected embedded systems

Lecture 1
Introduction and microcontrollers

Joel Voldman         Joe Steinmeyer
Jacob White

Sign up for Piazza if you haven’t already

February 7, 2017
Embedded systems are everywhere

- Embedded system: “computer” within a device
- ~70-100 embedded systems in a modern car
What is the Internet of Things?

- Embedded systems + communications + computation
The hype cycle
Introductory EECS classes

• Show how a few different areas in EECS connect to each other
• Provide an introduction to engineering design
• Help you understand
  • Is EECS right for me?
  • What topics (and classes) in EECS might I want to pursue?
• Provide you with some relevant and applicable skills that you can go apply to projects and other work
6. S08 project example
6.S08: what you’ll be able to do by May

• Set up your own IoT system and understand the different parts
• Make tradeoffs in complex engineering design
• Program a microcontroller and peripherals for your own projects, UROP, internship
• Know enough Python & C/C++ for follow-on classes
• Know which classes to take to learn more about microcontrollers, power, crypto, etc.
6.S08: interconnected embedded systems

- An ideal platform for EECS topics
- And for engineering design
Our embedded system

• We will build this up during Feb/Mar
Our wearable compared to FitBit Surge

- Microcontroller
- Display
- GPS
- microphone
- IMU
- Battery Manager
- Power Supply
- WiFi SOC
- Push Buttons

Bluetooth (other side)

iFixit.com
Class architecture

• One lecture/wk: Tue 10-11

• Labs: Tue 90 min
    Thu 3 h

38-530

• Weekly homeworks, which will involve the embedded system

• 1 Midterm Exam

• 1 Project!!

• Grading and other policies are on the class website

iesc-s2.mit.edu/6.S08/S17
6.S08 Bleeding Edge

- Things will go wrong in this course! Be aware!
- There will be frustration and failure and possibly frequent “pivots” in what we do
- This should also be a good learning experience, though
- If this is not what you want let us know soon (now-ish)
Topics in 6.S08

“EE”
- Sensors & signal conditioning
- Energy management

“EECS”
- Communications
- Microcontrollers
- Programming

“CS”
- Algorithms
- Cryptography
Rough schedule

• February/March: we build the system
• March/April: we use the system
• April/May: you extend the system

• Final project
  • Teams will extend their system in an area of interest
  • Chosen by you, approved by us
  • With mentoring and graded milestones
  • Could add new sensors, incorporate new APIs, etc.
Programming

• We will be using two languages simultaneously
  • Embedded system ➔ C/C++
  • Computer/Server ➔ Python

• Most students have had 6.0001 or 6.s080

• We will help you from whatever state you’re in
  • You’ve got to help us help you though: If you see something that doesn’t make sense, ask a question in person or in Piazza!
Developing in both C/C++ and Python

- Seeing differences between languages helps you appreciate their pros and cons
- And see the design choices made in language development
- Multiple environments are common in the real world
The digital abstraction

• Computers represent information in terms of *bits*
• Each bit contains one of two binary symbols: 1 or 0
• Those *bits* physically correspond to *voltages*
  • i.e., 1 might be 3.3 V, and 0 might be 0 V
  • Slight deviations from those values are restored to 3.3 V or 0 V
  • Store those bits using electrons or magnetic dipoles
• Makes our computation highly tolerant to noise
• This *digital* approach is one of the keys to the success of modern computing
Binary numbers

- 8 bits – 256
- 16 bits – 65,536
- etc.
- 8 bits = 1 byte
- 1024 bytes = 1 kB
Binary and hex

• Convenient to wrap 4 bits into a hexadecimal number (base 16, 0..9,A..F)
• We will use the prefix ‘0x’ to denote a hex number
• 0xFF = 1111 1111 = 255
• 155 = 1001 1011 = 0x9B

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>D</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>E</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
<td>15</td>
</tr>
</tbody>
</table>
Microcontrollers

• A computer on a chip
  • Microprocessor: to perform computation
  • Memory: to store programs and data
  • Peripherals: to connect to the world

TMS1000
The first microcontroller
1974
4-bits, 400 kHz
1kB ROM, 256 bits RAM
$3 in volume ($15 in 2017 dollars)

http://www.datamath.org/Story/Intel.htm
http://smithsonianchips.si.edu/augarten/p38.htm

SR-16 calculator
Microcontrollers

• The Arduino family
• Use an Atmel 8-bit AVR microcontroller
• Developed by **Alf Egil Boger** and **Vegard Wollan** and is a **RISC** processor
• Developed in Italy
• A board + software environment + useful libraries + add-ons
Microcontrollers (used in 6.S08)

- Teensy 3.2 development board
- Contains an ARM Cortex M4 microcontroller
  - Manufactured by Freescale
- ARM develops chip design and architecture but doesn’t make anything
  - Licenses design to others
  - ARM cores are widely used
- “Teensy” refers to the entire board
Teensy Development Board

- ARM Cortex M4
- 3.3V Regulated Switching Power Supply
- ARM M0 (for programming)
- 16 MHz Crystal Oscillator
- USB Micro Port
- Program Button
- On-board LED
- ARM Cortex M4
- 3.3V Regulated Switching Power Supply
Microcontrollers

- Cortex M-4
- 32-bit
  - Size of registers used to store and process data
- 96 MHz
  - Speed
  - How long each clock cycle takes
  - $1/96$ MHz $\sim 10$ ns
Microcontroller memory

• Non-volatile memory
  • Information will stick around after turning off and on
  • EEPROM (2 kb)
    • Electrically erasable programmable read-only memory
    • This can be used to store data that should be persistent
  • Flash (256 kb)
    • Same as in USB memory sticks
    • This holds the programs that we write

• Volatile memory
  • Loses values during restarts
  • SRAM (64 kb)
    • This holds data variables
Microcontroller memory model

• 32-bit micro can address up to ~4 Billion addresses

• Each piece of data (aka variable) and each instruction in the program has an address associated with it
How do you control a Microcontroller

• A microcontroller is a machine that is hardwired to respond to a set of instructions known as its Instruction Set
ARM Cortex-M Instruction Set Architecture

- **Cortex-M4**
  - Floating Point
  - DSP (SIMD, Fast MAC)
- **Cortex-M3**
  - Advanced data processing
  - Bit field manipulations
- **Cortex-M0/M0+**
  - General data processing
  - I/O control tasks

http://www.anandtech.com/show/8400/arms-cortex-m-even-smaller-and-lower-power-cpu-cores
Reduced Instruction Set Computing

- The “Reduced” part of RISC has nothing to do with the number of instructions (~200 instructions for our ARM Cortex M4)!
- Means the individual instructions each do less and are lower-level
- Do things like:
  - Add, Subtract, Multiply, Divide
  - Logical Operations
  - “Saves” and “ Loads”, etc...
  - “Jumps”, “Branches”
Sum 0 to 9 in ARM Assembly Using for Loop

MOV r0, #0 ; set 0 in r0
MOV r1, #0 ; set 0 in r1

L1 CMP r1, #10 ; r1–10
BGE L2

ADD r0, r0, r1
ADD r1, r1, #1
B L1

L2 B L2
Sum 0 to 9 in ARM Assembly Using 
for Loop

MOV r0, #0 ; set 0 in r0
MOV r1, #0 ; set 0 in r1
L1 CMP r1, #10 ; r1-10
BGE L2
ADD r0, r0, r1
ADD r1, r1, #1
B L1
L2 B L2
Going Deeper...

- Inside the microcontroller is the microprocessor which is hardwired (fabricated) to respond to these 1/0 bits (which in turn are really voltages)

```
1110 001 0100 0 0001 001 0000 0000 0001
```
Going Deeper...

- Elements of the microprocessor are comprised of

```
1110 001 0100 0 0001 0001 0000 0000 0001
```

4-bit ALU
Ours is 32 bit
(waaaay more complicated, but same general design principles)

https://en.wikibooks.org/wiki/Microprocessor_Design/ALU
Large blocks are comprised of logic gates which are simple circuits made of transistors that perform well-defined operations.

http://www.allaboutcircuits.com/worksheets/basic-logic-gates/

http://electroiq.com/chipworks_real_chips_blog/author/insights-from-leading-edge/page/3/

https://www.usenix.org/legacy/events/smartcard99/full_papers/kommerling/kommerling_html/
Sum 0 to 9 in ARM Assembly Using for Loop

    MOV  r0, #0  ; set 0 in r0
    MOV  r1, #0  ; set 0 in r1
    L1   CMP  r1, #10  ; r1 - 10
         BGE  L2
    ADD  r0, r0, r1
    ADD  r1, r1, #1
    B    L1
    L2   B    L2
Assembly...

• Everything done by any classical computational device (microcontrollers, microprocessors in computers, etc...) is done using only these commands

• If you write a program in assembly (the instruction set) you can get very efficient code

• Gets very complex very fast though!

• Can we write at a higher level so it could be easier to think and design?

• In comes C to the rescue!
Why C/C++?

• C was developed early on and much of what you do in C has a 1:1 relationship to Machine instructions
  • By Dennis Ritchie at Bell Labs (~1970)
  • Unix, etc... is written in C
  • Brian Kernighan and Dennis Ritchie wrote the essential book of C (K&R C)

• At the time C was a high-level language (relative) to machine code/assembly

• Nowadays it is “medium-level” since we have languages like Python that are even higher level
  • Common Python is actually written in C (known as CPython)...that’s what we’ll be using in class
Much of what you do in C has a 1:1 relationship to Machine instructions.

**Assembly:**

```
MOV r0, #0
MOV r1, #0
L1: CMP r1, #10
    BGE L2
    ADD r0, r0, r1
    ADD r1, r1, #1
    B L1
L2: B L2
```

**C/C++:**

```
int a = 0;
for (int x=0; x<10; x++) {
    a = a + x;
}
```

*colors indicate approximate matching functionalities and operations*
Much of what you do in C has a 1:1 relationship to Machine instructions

**Assembly:**

```
MOV r0, #0
MOV r1, #0
L1: CMP r1, #10
    BGE L2
    ADD r0, r0, r1
    ADD r1, r1, #1
    B L1
L2: B L2
```

**C/C++:**

```
int a = 0;
for(int x=0; x<10; x++){
    a = a+x;
}
```

**Python:**

```
a=sum(range(0,10))
```

*colors indicate approximate matching functionalities and operations*
Python

• High Level Language gives us lots of power!
• A few lines of Python can do a lot
• One line of Python can *potentially* have a ton of operations in it:

```python
>>> a = sum(range(0,10))
```

• Sums up values from 0 to 9 and stores in variable `a`
• Open up the source file for the `range` method in `rangeobject.c` and it is 1400 lines long and the `sum` method is about 200 lines long
• The list object in PyC is implemented in ~3000 lines of C

• Mapping from Python to machine code can be difficult to do efficiently!
So Why do we care if Mapping from Python to Machine Code isn’t Super Clear?

• As you write Python you may not have efficiency in mind. `sum(range(0,10))` is super fun to use.

• When it comes time to compile and run, it might not be the fastest.

• On a QuadCore Macbook this may not matter, but on a microcontroller...this REALLY CAN since you have much less memory and processing power
  • ARM M4s are awesome, but not invincible!
Future?

• There are some Python-based microcontroller environments out there now! (MicroPython for example).

• However the results you get still tend to not be as good as you can get with working lower-level (C/C++) or lowest level (Machine Instructions)