ECE 471 – Embedded Systems
Lecture 7

Vince Weaver

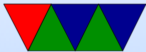
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

14 September 2022

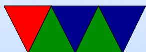
Announcements

- HW#2 due Friday



Homework #1 – Characteristics of Embedded System

- embedded inside – sometimes hard to know. Is a raw pi one? Pi used as desktop? Pi used as retro-pi? Pi controlling a 3D printer?
  Lack of being able to update not necessarily the same
- resource constrained
- dedicated purpose
- lots of I/O
- real-time constraints

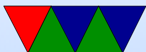


Homework #1 – Identifying an Embedded System

- Be decisive with your answer, and be specific with your reasoning
- iPhone

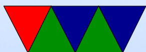
  real time doesn't necessarily mean quick-response, or FLOPS

  updatable not a characteristic
- Toothbrush is actual specs I came across
- Real-Time Confusion: we will discuss this more in future,



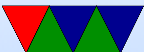
for example Just turning off the motor, and it takes an extra 1/2s is not really considered a real time thing. No one dies, no hardware destroyed, just mild annoyance if noticed at all. Now if somehow it had to keep the waveform to H-bridge exact within 1ms or the motor would overheat and catch on fire, that could be a real-time issue.

- Microwave: having a clock doesn't make it real time. Hopefully the door control has a physical interlock, but you never know. Usually when cooking food second



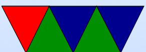
granularity and some jitter not matter much.

- Low-cost is complicated. Something like a desktop might be optimized for cost extremely, while a one-off embedded system might not, and in fact might be over-engineered (like a space probe) because has to operate in tough conditions.
- Low-power, again, this can be part of resource constrained but be sure to explain
- Operating system?
  Can have an OS and still be considered embedded.



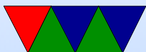
Homework #1 – Bits

- ARM1176 is generally considered 32-bits
- ARMv8 is generally considered 64-bits
- 6502 generally considered 8 bits
- There are people who will have long drawn-out internet arguments about the bitness of old systems



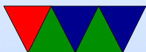
Homework #1 – ASIC vs ucontroller

- cost/power. Depends a lot on numbers made, process, and how well designed it is.
- Could be lower-cost/faster speed, but not necessarily. Why bother then? Cost?
- Extra hardware overhead? ASIC mostly just flip flops and gates. SoC internally a lot more, but these days not much else is needed.
- More secure? Can you reverse engineer an ASIC?



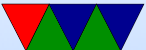
C Review

In past years sometimes the reason a HW assignment didn't work was due to using C poorly rather than misunderstandings of the desired algorithm.



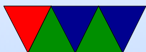
Loops in C

- `for(i=0;i<10;i++) {}`
- `while(i<10) { i++; }`
- `do { i++; } while(i<10);`
  Always runs at least once



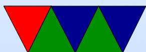
`printf()` in C

- printf()
- Lots of options, see man page
- How print an integer? `printf("%d",i);`
- Character? String? floating point?
  `printf("%c %s %f %x",c,s,f,x);`
- More advanced formatting stuff
  `printf("%0.3f",f);`
- Escape characters like percent, newlines and quotes
  `printf("\t \n \" \%");`



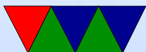
Common C Pitfalls – Static Memory

- Allocating things like arrays (`int a[5]`)
- C doesn't prevent you from accessing past the end
- What happens if you do go outside the boundary?
  - Crash? Memory corruption?
  - Nothing? (you are lucky and it hits something unimportant. Is that best or worst case?)



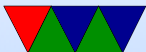
Common C Pitfalls – Dynamic Memory

- Can dynamically allocate memory with `malloc()` and `calloc()`
- Should check returned value against NULL.
  What happens if you de-reference a NULL pointer?
- Need to `free()` memory at end or you can leak memory
  What happens if you free the same memory twice?
- Out of bounds memory access same issue as with static



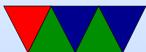
Debugging Memory Access issues

- The **Valgrind** utility can help debug these errors
  Mostly dynamic, not much can be done about static
- Valgrind can also help find memory leaks
  Note not all memory leaks are critical, as at program exit the operating system will close files/free memory



C Pitfalls – Strings

- C strings are just zero-terminated character arrays
- You can end up with all the same problems with memory accesses, especially running off the end
- There are versions of the string routines that take a length (`strncpy()` instead of `strcpy()` but beware those have their own issues

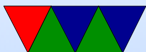


C Pitfalls – Braces

- Missing braces

```
if (a==0)
        b=2;

if (a==0)
        b=2;
        c=3;
```

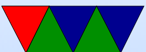


C Pitfalls – equality check

- = vs ==

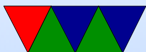
```
if (a=0) do_something_important();
```

- Never ignore warnings from the compiler!
- Some people will use `if (0=a)` to force an error



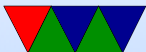
C Pitfalls – Type Issues

- C will happily auto-convert types for you
- Also be careful of signed/unsigned issues



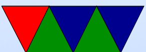
C Pitfalls – Setting Constants

- Floating point constants can be tricky, setting `double x=9/5;` will get you 1, you want 9.5/5.0
- Leading zeros specify Octal (base-8) numbers so something like `int x=010;` might give surprising results.



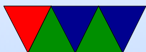
Coding Style

- How should you format your code?
- Does C have rules? Not really.
- International Obfuscated C Code Competition (IOCCC)
- Your company or open-source project might have strict rules
- In this class as long as your code is relatively easy to follow I am fine with it



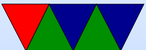
Coding Style – Tabs vs Spaces

- Indent code with a tab character?
  Or 8 spaces (traditional size of a tab)? Or some other amount of spaces?
- How long should lines be? Traditionally was 80 columns (historical size of screens)
- Other spacing, like `if ( x == 5 )` how many of those spaces should be there?



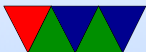
Coding Style – Curly Braces

- `int function() {`?
- Or should it be next line?
- Should int be on its own line too?



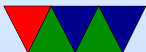
Coding Style – Variable Names

- `count_active_users()`
- `CountActiveUsers()` (camel-case)
- `szName` (Hungarian notation, include type info in name)



Coding Style – Linux kernel stuff

- Use of typedefs to make types shorter? `vpt_a` vs `struct virtual_pointer *a`
- Having only one exit to a function (using goto)
- Restricting the size functions can get
- The `indent` program can reformat your code to match the "proper" style for a project



Debugging – when things go wrong

- Use a debugger like gdb
  - Compile your code with `-g` for debug symbols
  - Run `gdb ./hello`
  - `bt` backtrace, `info regis` gives register, `disassem` disassembles, etc.
- Sprinkle printf calls