ECE 471 – Embedded Systems
Lecture 21

Vince Weaver

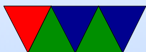
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

23 October 2023

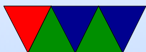
Announcements

- Keep thinking about projects, topic due November 3rd
- HW#7 was posted
- Give back and go over midterm. Average grade was a 91



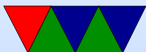
HW#7 – C Floating Point Notes

- You might not have used floating point in C much. Briefly covered in ECE271
- 32-bit integer only lets you have values from -2,147,483,648 to 2,147,483,647
- What if you want bigger? Smaller? Fractional / decimals?



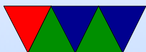
HW#7 – Fixed Point

- With integer math you can have "fixed point" where you arbitrarily put the decimal point at a fixed place in the value.
- 16.16 means 16 bits integer part, 16 bits fractional
- add/subtract work as normal, have to adjust after multiply/divide
- You might use this on an embedded system w/o hardware floating point



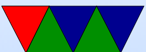
HW#7 – Arbitrary Precision Math

- You can have libraries that dynamically allocate memory for arbitrarily large numbers and fractions
- This is overkill for normal math you might want to do



HW#7 – Floating Point

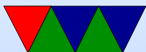
- Also gone over in ECE271
- For a 32 bit value, have one bit for sign, X bits for exponent, X bits for mantissa. How this actually works a bit beyond this class
- Can have a much wider range of values, also things like infinity, NaN, etc.
- Can do this all in hardware. Many low-end embedded systems these days will have support (all the Pis do)



HW#7 – Floating Point in C

- `float` is 32-bit
- `double` is 64-bit
-
- C will auto-cast things so you can do things like this, but be careful as sometimes the rules are a bit obscure. Generally when converting float to integer it will truncate rather than round

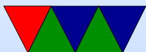
```
int x=5;
double d;
```



```
d=x;
```

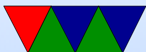
- Printing. First prints a double. Second prints a double with only 2 digits after decimal.

```
printf("%f\n",temp);          // print
printf("%lf\n",temp);         // print
printf("%.2lf\n",temp);       // print
```

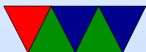


HW#7 – Floating Point Pitfalls

- Sort of like in decimal where only fractions of 1/2 and 1/5 have exact values (because base 10) on binary computers only multiples of 1/2 are exact
- This means unexpected things happen, like if you add 1.5 and 1.5 you might not get 3.0 but something close like 2.9999999999999999943
- That will round normally in say printf() but if you do an implicit cast to an integer you might confusingly get 2 instead of 3

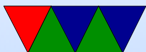


- You can use functions like `round()` and `ceil()` and `floor()` to do proper rounding
- You can force constants to be double/floats by putting a decimal point, so `9.0/5.` will be 1.8 like expected but 9/5 will be truncated to 1 (using integer math)



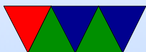
Real Time Wrapup

Some coding tips on how to get the best real time behavior out of your code

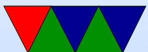


Complications – Interrupts

- Why are interrupts slow?
- Shared lines, have to run all handlers
- On Cortex-A systems have one IRQ line, have to query all to see what caused it. Cortex-M improves this by having dedicated vector for each piece of hardware
- When can they not be pre-empted? IRQ disabled? If a driver really wanted to pause 1ms for hardware to be ready, would often turn off IRQ and spin rather than sleep

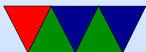


- Higher priority IRQs? FIR on ARM?
- Top Halves / Bottom Halves



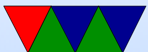
Complications – Threading

- A thread is a unit of executing code with its own program counter and own stack
- It's possible to have one program/process have multiple threads of execution, sharing the same memory space
- Why?
  - Traditionally, to let part of program keep running when another part waiting on I/O (gui keep drawing while waiting for input, sound playing in background during game, etc)



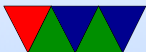
- Lets one program spread work across multiple cores

- This complicates the schedule, and also makes priority more complex

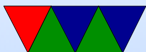


Complications – Locking

- When shared hardware/software and more than one thing might access at once
- Example:
  - thread 1 read temperature, write to temperature variable
  - thread 2 read temperature variable to write to display
  - each digit separate byte
  - Temperature was 79.9, but new is 80.0
  - Thread 1 writing this



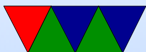
- ◦ What if Thread 2 reads part-way through? Could you get 89.9?
- Is this only a SMP problem? What about interrupts?



Scheduler Complications – Locking

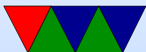
- Previous was example of Race Condition (two threads "racing" to access same memory)
- How do you protect this? With a lock
  - Special data structure, allows only one thread inside the locked area at a time
  - This is called a "critical section"

```
lock(&temp_lock);                    lock(&temp_lock
write_display();                     read_temperatur
unlock(&temp_lock);                  unlock(&temp_lo
```



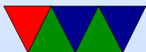
Scheduler Complications – Locking

- Can you have race conditions on a single core?
  - Yes, with interrupts
  - On simple systems you can just disable interrupts during critical section
  - Usually can't do that if have an OS

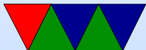


Scheduler Complications – Lock Implementation

- Implemented with special instructions, in assembly language
- Usually you will use a library, like pthreads
- mutex/spinlock
- Atomicity

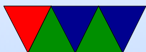


Memory Allocation in Embedded Systems

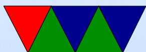


Memory Allocation – Dynamic

- Using `malloc()`/`calloc()` or `new()`
- In C have to make sure you `free()` at end
- Downsides:
  - What to do if fails?
    Can you handle that? What if error code also tries to alloc?
  - Timing overhead? Is it deterministic?
    Especially problem with high-level languages and garbage collection

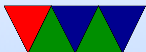


- Fragmentation: when there's plenty of RAM free but it's in small chunks when you need a large chunk



Memory Allocation – Static

- Allocate all memory you need at startup
- Fail early
- This isn't always possible, but avoids issues with failure, overhead, etc.
- Free RTOS (newer) allows static allocation at compile time



Linux Memory Issues

- Even if you statically allocate memory, on system with virtual memory it might swap out to disk
- This can suddenly make your code unexpectedly slower, ruin real-time performance
- Can you prevent this?
  - `mlockall()` syscall can lock memory so it stays in RAM, never goes to disk
  - So at start of program, allocate RAM, touch it (or prefault) to bring it in, then mlock() it