ECE 471 – Embedded Systems
Lecture 12

Vince Weaver

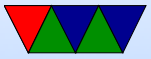
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

1 October 2018

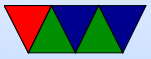
Announcements

- HW#4 was posted.
- Permissions!
  Unless your user is configured to have gpio permissions you'll have to run as root or use sudo. raspbian there's a "gpio" group which has permissions `sudo addgroup vince gpio`
  `udev` is responsible for updating permissions as the files are created and it can take a fraction of a second to detect and update.



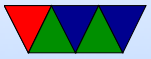
This might not work if you have an older version of Raspbian

- What should your code do if permission is denied? Not crash, certainly.



Homework 2

- Comment your code!
- check argc before accessing argv



Homework 3

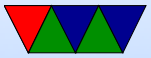
- Be sure to put your name in the README!
- Should do HW, even if you only do the short-answer part. Good practice for midterm
- Comment code!

  Make sure comments makes sense

  Also high level comments are best:

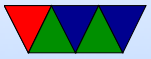
  `add r1,r2,#0x30 @ add 48 to r2 and store in r1`

  vs `@ convert integer result to ASCII`
- `print_ number()` code



- No conversion to binary, number is in binary in register.
- The divide by 10 code is almost more interesting.
- Good to be able to look at code and see what doing. Reverse engineering, but also debugging code you don't have the source to.

```
print_number:
        push    {r10,LR}        // Save registers
        ldr     r10,=buffer     // what does = mean?  where is buffer?
        add     r10,r10,#10     // why 10 bytes?

divide:
        bl      divide_by_10    // why no div instruction?
        add     r8,r8,#0x30     // why add 0x30?
        strb    r8,[r10],#-1    // why moving backwards?
        adds    r0,r7,#0        //
        bne     divide          //

write_out:
        add     r1,r10,#1       // why adjust pointer?
```



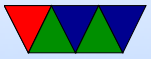
```
        bl      print_string    //

        pop     {r10,LR}        //

        mov     pc,lr           //
```

how would you convert to hex? Why 10 chars reserved?

```
divide_by_10:
        ldr     r4,=429496730           @ 1/10 * 2^32
        sub     r5,r0,r0,lsr #30
        umull   r8,r7,r4,r5             @ {r8,r7}=r4*r5
        mov     r4,#10                  @ calculate remainder
        mul     r8,r7,r4
        sub     r8,r0,r8
        mov     pc,lr
```

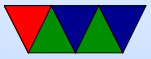
- strlen code example, many ways to do this

```
mov     r2,#0
```



```
print_loop:
        ldrb    r0,[r1,r2]
        add     r2,r2,#1
        cmp     r0,#0
        bne     print_loop
```

- Can see why THUMB2 is nicer than THUMB (assembler does most of work)
- THUMB code should have been less.
  You need to run `strip` on this to see it. Why?
  Debug info, including extra thumb debug as well as the longer filename.

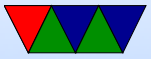


You can use `readelf -a` and `readelf -s` to see the space the various segments take up.

Look at executables, *not* the C source code.

arch	unstripped	stripped
arm32	1424	620
thumb	1444	600
thumb2	1420	596
C	8156	5608
C/thumb2	8144	5612
C static	569288	484620

You would think THUMB2 would be much smaller, but the assembler makes some poor decisions about



wide/narrow instructions.

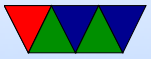
Reference my LL work

C code is larger, but also remember to include the C library:

```
ls -lart /lib/arm-linux-gnueabihf/libc-2.24.so
-rwxr-xr-x 1 root root 1234700 Jan 14  2018 /lib/arm-linux-gnueabihf/libc-2.24.so
```

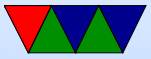
There are embedded C libraries, musl, newlib, uclibc, which are much smaller and often used in embedded systems.

- Illegal instruction error usually because there are *two*



calls to print string, need to make sure both are blx

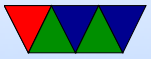
- Comments: meaningful! Not just add 10 to r10 or add 48 to r3
- Not sure why STDIN something cool not working for people
- cal. Missing days. Julian to Gregorian calendar. People sad who paid weekly but paid rent monthly.
  Be careful using Google.



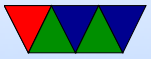
Coding Directly for the Hardware

One way of developing embedded systems is coding to the raw hardware, as you did with the STM Discovery Boards in ECE271.

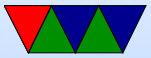
- Compile code
- Prepare for upload (hexbin?)
- Upload into FLASH
- Boots to offset



- Setup, flat memory (usually), stack at top, code near bottom, IRQ vectors
- Handle Interrupts
- Must do I/O directly (no drivers)
  Although if lucky, can find existing code.
- **Code is specific to the hardware you are on**

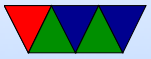


Instead, one can use an Operating System

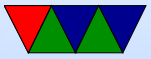


Why Use an Operating System?

- Provides Layers of Abstraction
  - Abstract hardware: hide hardware differences. same hardware interface for classes of hardware (things like video cameras, disks, keyboards, etc) despite differing implementation details
  - Abstract software: with VM get linear address space, same system calls on all systems
- Other benefits:
  - Multi-tasking / Multi-user

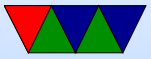


  - Security, permissions (Linus dial out onto /dev/hda)
  - Common code in kernel and libraries, no need to re-invent
  - Handle complex low-level tasks (interrupts, DMA, task-switching)
- Abstraction has a cost
  - Higher overhead (speed)
  - Higher overhead (memory)
  - Unknown timing



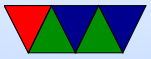
What's included with an OS

- kernel / drivers – Linux definition
- also system libraries – Solaris definition
- low-level utils / software / GUI – Windows definition
  Web Browser included?
- Linux usually makes distinction between the OS Kernel and distribution. OSX/Windows usually doesn't.



Bypassing Linux to hit hardware directly

- Linux does not support things like pullups, but people have written code that will poke the relevant bits directly.



Bypassing Linux for speed

http://codeandlife.com/2012/07/03/benchmarking-raspberry-pi-gpio-speed/

Trying to generate fastest GPIO square wave.

shell	gpio util	40Hz
shell	sysfs	2.8kHz
Python	WiringPi	28kHz
Python	RPi.GPIO	70kHz
C	WiringPi	4.6MHz
C	libbcm2835	5.4MHz
C	Rpi “Native”	22MHz