ECE 471 – Embedded Systems
Lecture 16

Vince Weaver

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

vincent.weaver@maine.edu

9 October 2019

Announcements

- Midterm on 18th
- No class on 16th (Career Fair)
- Don't forget HW#5
- If looking for classes next semester, check out ECE498/ECE598 with NASA scientist Alex Barrie. He's giving a talk Friday October 25th at 2pm in Hill auditorium

HW#5 notes

- Using #defines
- Example `(2<<4)|1`
  Clearer to use `HT16K33_REGISTER_SYSTEM_SETUP | HT16K33_ENABLE_CLOCK`

HW#4 – Notes on Reading GPIOs

- If you add additional files to your submission, make sure they get submitted
- Code supposed to be report only keypress/keyrelease events (level shift)
  How do you tell if a level input has changed?
- Strings are of type char, not integer!
- If it says use GPIO17, then use GPIO17
- Return from read() is bytes read, not the result
  Need to check first byte of buffer for result

- Remember reading string you get ASCII '0' (48) not 0
- Lots of people using polling? Why does it double report presses?
- Use of cut and paste code

HW#4 – Debouncing

- Tricky as we are detecting levels not edges here
- Reading and only reporting if you say have 3 in a row of save val
- Reading, sleeping a bit, then report the value after has settled
- Just sleeping a long time after any change? If a short glitch happens this might misreport.
- Sleep too long, might miss events
- Debounce if using interrupt-driven code

In that case debouncing might be to ignore repeated changes if they happen too close together

Raspberry Pi Booting (pre pi4)

- Unusual
- Small amount of firmware on SoC
- ARM 1176 brought up inactive (in reset)
- Videocore loads first stage from ROM
- This reads `bootcode.bin` from FAT partition on SD card into L2 cache. It's actually a RTOS (real time OS in own right "ThreadX") (50k)
- This runs on videocard, enables SDRAM, then loads `start.elf` (3M)

- This initializes things, the loads and boots Linux `kernel.img`. (also reads some config files there first) (4M)

Pi4 booting

- `https://www.raspberrypi.org/documentation/hardware/raspberrypi/booteeprom.md`
- SPI EEPROM holds equivelent of `bootcode.bin`, no longer read from partition
- Why? SDRAM, PCIe USB, etc are more complex
- No network/USB booting yet, coming soon

More booting

- Most other ARM devices, ARM chip runs first-stage boot loader (often MLO) and second-stage (uboot)
- FAT partition
  Why FAT? (Simple, Low-memory, Works on most machines, In theory no patents despite MS's best attempts (see exfat))
  The boot firmware (burned into the CPU) is smart enough to mount a FAT partition

Trusted Firmware

- Can you trust your firmware to be not-evil?
- Evil Maid problem – what if someone breaks into your hotel room and replaces your firmware – could you tell?
- Best you can do is trust it to be the same firmware released by your vendor (you still have to trust them)
- Use cryptographic signing. Hardware will only run code "signed" by a trusted entity.
- A signed firmware can run a signed bootloader which can run a signed operating system which can run signed

apps

- Downside: no longer general purpose, average person cannot run code they wrote unless they can get it signed
- Code still has to be well written. “jailbreaks” on phones and video game consoles are due to trusted code having bugs and then jumping into unsigned code.

Trusted Firmware

- New for ARMv8: ARM Trusted Firmware (ATF). Two standards, vendors have possibly made a mess of it already.
- Other platforms have it too. DRM to keep you from copying movies or video games.

Boot Methods

Firmware can be quite complex.

- Floppy
- Hard-drive (PATA/SATA/SCSI/RAID)
- CD/DVD
- USB
- Network (PXE/tftp)

- Flash, SD card
- Tape
- Networked tape
- Paper tape? Front-panel switches?

Disk Partitions

- Way to virtually split up disk.
- DOS GPT – old partition type, in MBR. Start/stop sectors, type
- Types: Linux, swap, DOS, etc
- GPT had 4 primary and then more secondary
- Lots of different schemes (each OS has own, Linux supports many). UEFI more flexible, greater than 2TB
- Why partition disks?
  - Different filesystems; bootloader can only read FAT?

- Dual/Triple boot (multiple operating systems)
- Old: filesystems can't handle disk size

Device Detection

- x86, well-known standardized platform. What windows needs to boot. Can auto-discover things like PCI bus, USB. Linux kernel on x86 can boot on most.
- Old ARM, hard-coded. So a rasp-pi kernel only could boot on Rasp-pi. Lots of pound-defined and hard-coded hw info.
- New way, device tree. A blob that describes the hardware. Pass it in with boot loader, and kernel can use

it to determine what hardware is available. So instead of Debian needing to provide 100 kernels, instead just 1 kernel and 100 device tree files that one is chosen at install time.

- Does mean that updating to a new kernel can be a pain.

Detecting Devices

There are many ways to detect devices

- Guessing/Probing – can be bad if you guess wrong and the hardware reacts poorly to having unexpected data sent to it
- Standards – always knowing that, say, VGA is at address 0xa0000. PCs get by with defacto standards
- Enumerable hardware – busses like USB and PCI allow you to query hardware to find out what it is and where

it is located

- Hard-coding – have a separate kernel for each possible board, with the locations of devices hard-coded in. Not very maintainable in the long run.

- Device Trees – see next slide

Devicetree

- Traditional Linux ARM support a bit of a copy-paste and #ifdef mess
- Each new platform was a compile option. No common code; kernel for pandaboard not run on beagleboard not run on gumstix, etc.
- Work underway to be more like x86 (where until recently due to PC standards a kernel would boot on any x86)
- A “devicetree” passes in enough config info to the kernel

to describe all the hardware available. Thus kernel much more generic

- Still working on issues with this.

Booting Linux

- Bootloader jumps into OS entry point
- Set Up Virtual Memory
- Setup Interrupts
- Detect Hardware / Install Device Drivers
- Mount filesystems
- Pass control to userspace / call init

- Run init scripts
- rc boot scripts, /etc/rc.local
  Start servers, or “daemons” as they’re called under Linux.
- fork()/exec(), run login, run shell

How a Program is Loaded on Linux

- Kernel Boots
- `init` started
- `init` calls `fork()`
- child calls `exec()`
- Kernel checks if valid ELF. Passes to loader
- Loader loads it. Clears out BSS. Sets up stack. Jumps

to entry address (specified by executable)

- Program runs until complete.
- Parent process returned to if waiting. Otherwise, init.

Viewing Processes

- You can use `top` to see what processes are currently running
- Also `ps` but that's a bit harder to use.