ECE 471 – Embedded Systems
Lecture 30

Vince Weaver

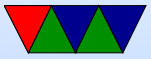
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

15 November 2021

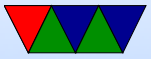
Announcements

- Don't forget project status reports due Monday (22nd)
- Don't forget HW#9
- Midterm on Friday, will be review on Wednesday

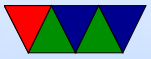


HW#9 Notes – Building Separate Files

- In previous homeworks we put everyting in one C file
- This isn't really practical for large projects
- By splitting things up into smaller files you can have some benefits
  - Easier to organize/find code
  - Can re-use code easier
  - Less chance of merge conflicts when multiple people working on project
  - Can take common code and make libraries

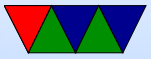


- For example in the homework, we could put temperature read code into its own file with a `double get_temperature(void)` interface
- For other C files to see this, you need to export the definition. Usually this is done by putting the advance definition `double get_temperature(void);` in a .h header file and then including it in the other files
- Note: don't put full C functions in header files. I know this is a C++ thing but it's usually frowned upon when programming in C
- Each file does not need a `main()` function, you only



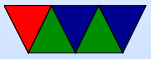
need one per combined program.

- To link the various .o files together involves the "linker". However it's easier to just let gcc do it (gcc knows how to run the linker for you) `gcc -o display_temp display.o temperature.o`
- The linker merges the .o files into one big executable, and makes sure the placeholders to functions/variables in all of the files get the right addresses/pointers to where things live in the finished executable.
- How do you make sure when you change one C file that everything that uses it is also rebuilt? A well-crafted

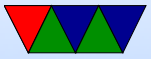


Makefile will have all these dependencies in place and will rebuild everything properly.

- Static vs Dynamic linking redux (we did discuss that earlier in the semester)

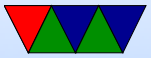


Other I/O You'll find on Embedded Boards

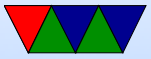


Digital Audio

- How can you generate audio (which is analog waves) with a digital computer?
- One way is PCM, Pulse Code Modulation, i.e. use a DAC.
  - Sample the sound at a frequency (say 44.1kHz), and take amplitude (16-bit audio, 64k possible values)
  - Why 44.1kHz? Nyquist theorem. Twice sample rate to reproduce properly. 22kHz roughly high end of human hearing.

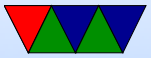


- A WAV file is basically this, has the samples (8 or 16-bit, stereo or mono) sampled at a regular frequency (often 44.1kHz) to play back, write the values to a DAC at the sample rate.



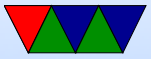
What if no DAC? (Pi has none)

- Can do PWM, Pulse-Width Modulation
- By varying the width of pulses can have the average value equal to an intermediate analog value. For example with duty cycle of 50% average value is 1/2 of Vdd
- Can be "converted" to analog either by a circuit, or just by the inertia of the coil in a speaker.



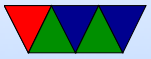
Saving space

- Can be tens of megabytes per song.
- Music can be compressed
- Lossy: MP3, ogg vorbis
- lossless AAC, FLAC



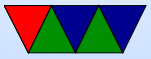
PWM GPIO on Pi

- You can't get good timings w/o real-time OS
- Available on GPIO18 (pin 12)
- Can get 1us timing with PWM in Hardware
  Software: 100us Wiring Pi, less with GPIO interface.
- Which would you want for hard vs soft realtime?
- Other things can do? Beaglebone black as full programmable real-time unit (PRU)
  200MHz 32-bit processor, own instruction set, can control pins and memory, etc.

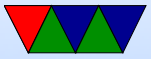


Linux Audio

- In the old days audio used to be just open `/dev/dsp` or `/dev/audio`, then `ioctl()`, `read()`, `write()`
- These days there's ALSA (Advanced Linux Sound Architecture)
  The interface assumes you're using the ALSA library, which is a bit more complicated.
  - Handles things like software mixing (if you want to play two sounds at once)
  - Other issues, like playing sound in background

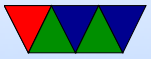


- On top of that is often another abstraction layer, pulse-audio
- A mixer interface to pick volumes, muting
- For quick hack can use `system()` to run a command-line audio player like `aplay`
- Better idea might be to use a library such as SDL-mixer which handles all of this in a portable way with a nice library interface.



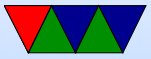
Pi Limitations

- Pi interface is just a filter on two of the PWM GPIO outputs
- Also can get audio out over HDMI.
- If you want better, can get i2s hat
- Pi lacks a microphone input, so if want audio in on your pi probably need a USB adapter.



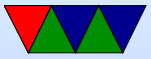
i2s

- PWM audio not that great
- i2s lets you send packets of PWM data directly to a DAC
- At least 3 lines. bit clock, word clock (high=right/low=left stereo), data
- Pi support i2s on header 5



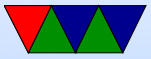
SD/MMC

- MultiMediaCard (MMC) 1997
- Secure Digital (SD) is an extension (1999)
- SDSC (standard capacity), SDHC (high capacity), SDXC (extended capacity), SDIO (I/O)
- Standard/Mini/Micro sizes
- SDHC up to 32GB, SDCX up to 2TB
- Support different amounts of sustained I/O. Class rating 2, 4, 6, 10 (MB/s)
- Patents. Need license for making.



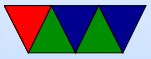
SD/MMC Hardware Interface

- 9 pins (8 pins on micro)
- Starts in 3.3V, can switch to 1.8V
- Write protect notch. Ignored on pi?



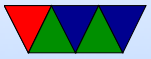
SD/MMC Software Interface

- SPI bus mode
- One bit mode – separate command and data channels
- Four-bit mode
- Initially communicate over 1-bit interface to report sizes, config, etc.
- DRM built in, on some boards up to 10% of space to handle digital rights



SDIO

- SDIO – can have I/O like GPS, wireless, camera
- Can actually fit full Linux ARM server on a wireless SDIO card



eMMC

- eMMC = like SD card, but soldered onto board