ECE 598 – Advanced Operating Systems Lecture 5

Vince Weaver

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

vincent.weaver@maine.edu

6 February 2018

Announcements

- HW#2 was posted
- How is HW#2 going? Don't wait until the last minute!

HW#1 Review

- Put your name on the assignment!
- Short answers OK, but please explain things at least a little (more than a single word).
- Answers
  - Benefits of OS: HW abstraction. Multi-tasking?
  - Downsides of OS: Overhead, Timing, Nontransparency
  - Why C?: Low-level, historical, faster
  - Why not C?: Security risk. Lack of features?
  - OS: ReactOS (windows clone), TempleOS, BeOS,

BareMetal, TI-RTOS, RCA TSOS, Fuchsia, Solaris, ThreadX, iRMX, OS/2, MenuetOS Lots of people BeOS

Cross Compilers

- This year I think people are mostly doing OK, but some notes that might be helpful.
- Just be glad you aren't trying to build a gcc cross compiler from scratch.
- On x86 Linux can install. Either grab the `.tar.gz` and setup, or if on Ubuntu might be as simple as:
  ```
  apt-get install gcc-arm-none-eabi
  ```
  On Debian you can add add
  ```
  deb http://www.emdebian.org/debian unstable main
  ```

to your `/etc/apt/sources.list` first.
If you are installing a .tar.gz on a 64-bit machine, and it's a 32-bit compiled toolchain, you might need to install some helper 32-bit libraries before it will work (in this case you'd get unhelpful errors about the executable not being found even though it quite obviously is there).

- On OSX people seem to mostly successfully getting Yagato working following the instructions given.
- Windows is tricky. Directions inside a .bz2 file and involves installing MINGWIN first plus then compiling a bunch of files.

- Updating your PATH is an issue. Can get around this by just hardcoding the path in your Makefile CROSS variable as described in HW handout.

Serial Ports

- What's the easiest way to get some I/O?
- Blinky LEDs not enough. Could have O/S communicate by Morse code on the LED (were patches for Linux to do this at one point)
- "Serial" meaning one-bit at a time

History

- back in the day spent lot of your life configuring serial connections
- The Linux ioctl interface to this is a pain, old legacy
- Hooking old machines together (Apple II, etc)
- Still used a lot on embedded systems

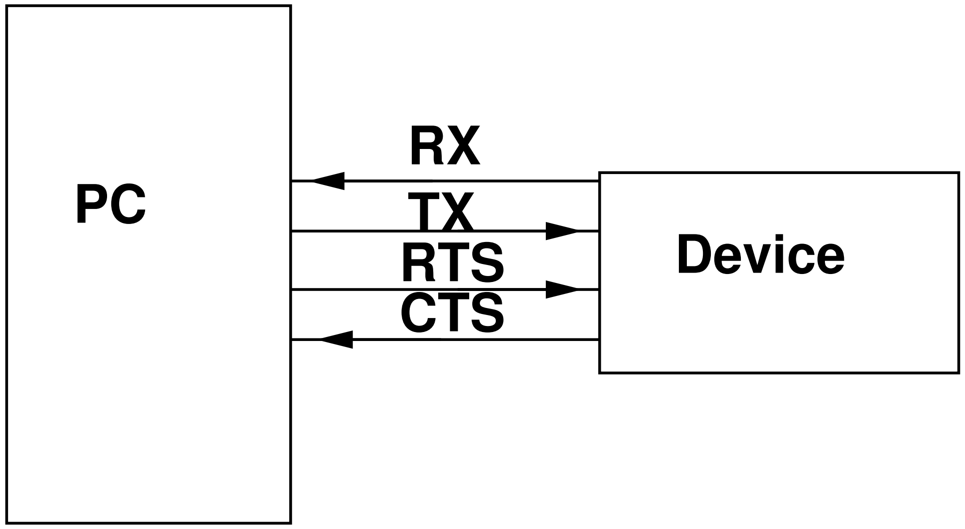
Serial Port Background

- Minimum TX, RX, Ground. Older systems 9pin/25pin
- Used for many things (modems, terminals, mouse, GPS, etc)
- RS-232
- Interface usually set of IO-Address and Interrupt (sometimes DMA too)
- ttyS0, ttyS1 and similar (Linux) COM1, COM2 Windows

Flow Control

- Ability to start/stop without losing bytes
- Often multiple levels of buffering. FIFO, buffer on device, buffer in OS.
- Hardware: RTS (Request to Send) and CTS (Clear to Send) Positive or negative, one to the other. Why in old days needed a crossover cable to connect two PCs together.
- Software: uses the ASCII DC1 (XON) (0x11) (control-q)

and DC3 (XOFF) (0x13) (control-s) device control chars over the line. Why bad? Cannot send binary data easily. Why?



Signals

- TX,RX,GND (minimum)
- RTS,CTS (HW flow control)
- DSR,DTR (Data set ready, data terminal ready, other flow control)
- DCD (data carrier detect – modem there)
- RI (ring indicator)

- +12V for 0, -12V for 1 (inverted on transmit/receive, regular on other pins)
  (-15 to -3 or so, 15 to 3 or so).
  Some are -5/5V tolerant.
  Modern TTL 5/0 exist but not back compatible
  3.3V I/O like on Pi troublesome
  (why awkward on small embedded boards)

- DCE (data communication equipment) to DTE (data terminal equipment) straight through. Need loopback if DTE to DTE (swap RTS and CTS)

Transmitting a Byte

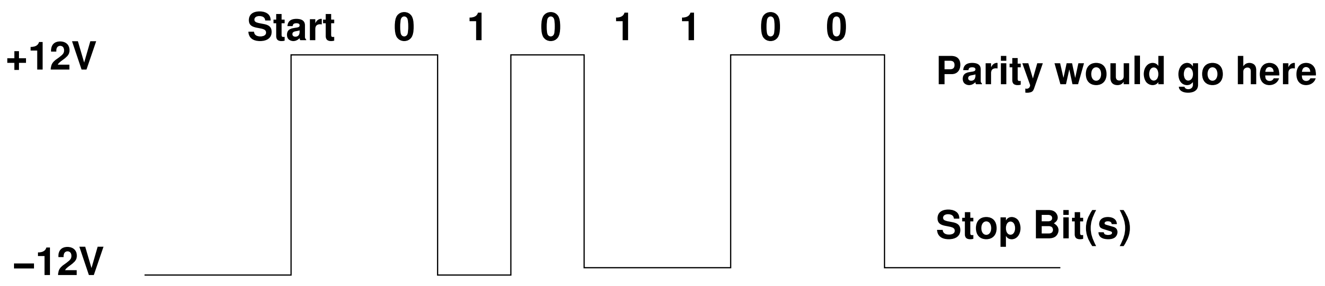
- TX Held -12V when idle (1)
- Jumps to 12V for start bit (0)
- Bits transmitted. Low bit first. Stays at 12V if 0, -12 if 1
- If parity bit desired, sends that (even, odd, stick, or none)

  If odd, then parity bit is included that makes the number of 1s (including parity) odd.

  If even, then parity bit included that makes number of

1s even.
Stick parity (mark = always 1, space = always 0)

- Then down to -12V for stop bit(s)
- Since no clock, no way to tell difference between consecutive bits of same value. Starts a counter counting at the start bit, samples values from there.



Value sent is 001 1010

UART

- Universal Asynchronous Receiver Transmitter
- Convert parallel value to serial
- Asynchronous. Why? No clock signal wire.

UART FIFO

- Internal First-in-First-Out structures
- 1 byte (older) to 16 bytes
- Can generate Interrupt. Have to handle in time or else data lost
- Why timeout? Send 1-byte typing, stall if not 15 more.
- Why FIFOs small? flow control. A byte saying to stall sent, if large FIFO a long time before it actually gets

received

- Interrupts: when FIFO reaches a certain size, or if there's a delay. (so if someone typing slowly at the keyboard) also when transmit FIFO empty

Speed (flow rate)

- Clock crystal, programmable divider
- Default old max speed was 115,200; many modern can go much faster, even up to 4MBps. Cables often not up to it as standard did not specify twisted pair
- Common speeds 1 (115.2k), 2 (57.6k), 3 (38.4k), 6 (19.2k), 12 (9.6k), 24 (4.8k), 48 (2.4k), 96 (1.2k) 300, 110 (war games acoustic coupler)

Programs

- `learn.adafruit.com/adafruits-raspberry-pi-lesson-5-using-a-console-cable/test-and-configure`
- putty is a decent one for Windows
- I use minicom for Linux. A bit of a pain. Have to install it (not by default?) Control-A Control-Z for help. Has similar keybindings to an old DOS program Telix that I used for years.
- OSX? Also Linux?

```
sudo screen /dev/ttyUSB0 115200
```

File Transfer

- Kermit
- Zmodem/Xmodem/Ymodem/etc
- sz / rz on Linux

Things you will need to set

- 115200 8N1 Software Flow
- 115200 Baud
- 8 data bits (7 or 8)
- no parity (even, odd, none)
- 1 stop bit (1, 1.5, or 2)

USB Serial Converters

- PL2303 / FTDI most common
- Often counterfeited, in the news for that recently (and how the companies tried to kill the counterfeits)
- Takes serial in, presents as a serial port to the OS. ttyUSB0 on Linux, COM something really high on windows

BCM2835 UART on the Pi

- Section 13 of the Peripheral Manual
- Two UARTS. Mini (pc reg layout compat) and ARM PL011. We use the latter.
- No IrDA or DMA support, no 1.5 stop bits.
- Separate 16x8 transmit and 16x12 receive FIFO memory. Why 12? 4 bits of error info on receive. overrun (FIFO overflowed), break (data held low over full time), parity, frame (missing stop bit).

- Programmable baud rate generator.
- start, stop and parity. These are added prior to transmission and removed on reception.
- False start bit detection.
- Line break generation and detection.
- Support of the modem control functions CTS and RTS. However DCD, DSR, DTR, and RI are not supported.
- Programmable hardware flow control.

- Fully-programmable serial interface characteristics: data can be 5, 6, 7, or 8 bits
- even, odd, stick, or no-parity bit generation and detection
- 1 or 2 stop bit generation
- baud rate generation, dc up to UARTCLK/16
- 1/8, 1/4, 1/2, 3/4, and 7/8 FIFO interrupts

BCM2835 UART

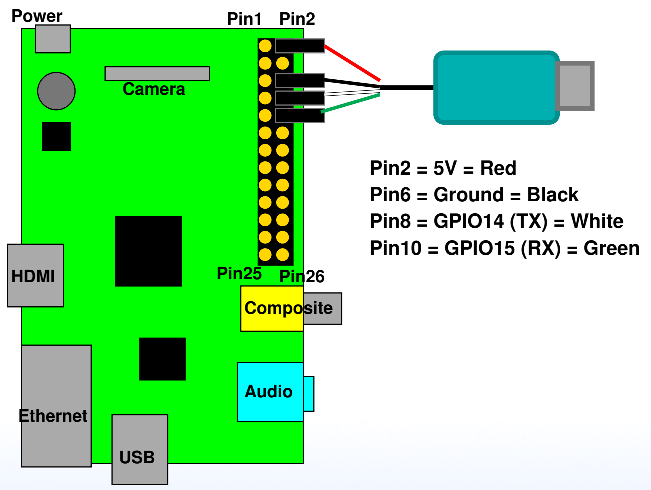
- Can map to GPIO14/15 (ALT0), GPIO36/37 (ALT2), GPIO32/33 (ALT3)
- Default mapping has RX/TX on GPIO14/15. It is possible to configure RTS/CTS pins for HW flow control, but our adapter doesn't support them anyway.
- Base address `0x20201000`, 18 registers

Hooking up Cable to Pi

- Linux should come with a driver. May need to download PL2303 OSX or Windows driver.
- Some useful documentation:
  `http://www.adafruit.com/products/954`
  `https://learn.adafruit.com/adafruits-raspberry-pi-lesson-5-using-a-console-cable`
- Can provide 5V to your board with the red wire so you don't need USB-micro cable. This might be dangerous however as you are bypassing the power conditioning. If you are leaving USB micro hooked up, then don't

connect the red wire.

- Hookup:
  Red (5V) to pin 2,
  Black (GND) to pin 6
  White (TXD) to pin 8 (GPIO14)
  Green (RXD) to pin 10 (GPIO15)



Inline Assembly

- Can write assembly code from within C
- gcc inline assembly is famously hard to understand/write
- volatile keyword tells compiler to not try to optimize the code within

```
static inline void delay(int32_t count) {
        asm volatile("__delay_%=: subs %[co
                            "bne __delay_%=\n"
                    : : [count]"r"(count) : "c
}
```

- : output operands
  = means write-only, + is read/write `r`=general reg
- : input operands
- : clobbers – list of registers that have been changed
  memory is possible, as is cc for status flags
- can use `%[X]` to refer to reg X that can then use `[X]"r"(x)` to map to C variable