ECE 531 – Advanced Operating Systems
Lecture 8

Vince Weaver

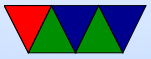
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

19 September 2025

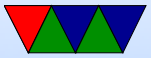
Announcements

- Homework #2 due today
- Homework #3 was posted already
  - I will try to cover everything you need to know to start in this class
  - Some stuff might not get covered until Monday
- Don't forget grad seminar (with pizza) immediately after class



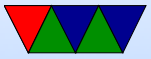
UARTs on the Pi

- Two UARTs (six on pi4?)
  - ARM PL011 UART
  - 16550-based mini-UART
- Default changed from PL011 to mini on Pi3/Pi4 because bluetooth was hooked up to PL011

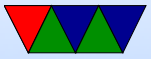


BCM2835 PL011 UART

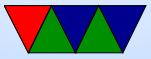
- /dev/ttyAMA0 on Linux
- Section 13 of the Peripheral Manual
- Separate 16x8 transmit and 16x12 receive FIFO memory. Why 12? 4 bits of error info on receive. overrun (FIFO overflowed), break (data held low over full time), parity, frame (missing stop bit).
- Programmable baud rate generator.
- start, 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

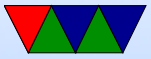


- No IrDA or DMA support, no 1.5 stop bits.



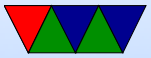
mini-UART

- /dev/ttyS0 on Linux
- Section 2.2 of BCM2835 document
- UART Clock scales with CPU frequency
  If cpu scaling then you are out of luck (for Linux you can specify boot to have fixed clock, or to swap back in the good UART)
- Registers mixed in with SPI registers
- Roughly register compatible with popular 16550 UART
- Set up GPIOs first, if you don't it will see RX as zero



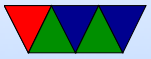
and start receiving 0x0 bytes (it ignores stop bits) and FIFO will fill in 2.5usec

- 7 or 8 bit, parity not supported, RTS/CTS possible, 8-byte FIFO



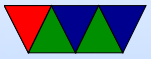
BCM2835 PL0111

- We'll be using this one
- Can map to GPIO14/15 (ALT0), GPIO36/37 (ALT2), GPIO32/33 (ALT3) (hooked to Bluetooth on Pi3)
- 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 `IO_BASE+0x201000`, 18 registers



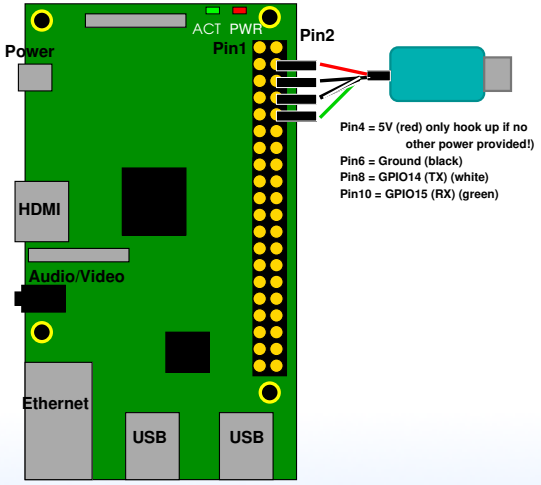
Hooking up Cable to Pi

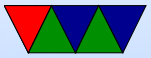
- I handed out adapters with CP2102 chips in them
- Linux should come with a driver.
- MacOS you will need to download and install driver, Windows I don't know
- Some useful documentation:
  `http://www.adafruit.com/products/954`
  with a link to getting driver and setting up terminal programs
- Don't hook up the red wire if you are also powering it



externally through the power USB port!

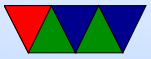
- Hookup:
  Red (5V) to pin 2 or 4
  Black (GND) to pin 6
  White (TXD) to pin 8 (GPIO14)
  Green (RXD) to pin 10 (GPIO15)





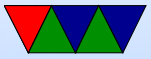
Low-level Programming

- In the previous homework we used `volatile` to trick the compiler into letting us access hardware directly
- Is there a better way to do it?
- Some things would be better if we could directly specify the code we want to run at the assembly level



Inline Assembly

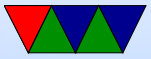
- Can write assembly code from within C
- gcc inline assembly is famously hard to understand/write
- You'll still see the volatile keyword telling the compiler to not try to optimize the code within



Delay Inline Assembly Example

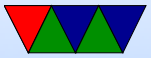
```
static inline void delay(int32_t count) {
  asm volatile("__delay_%=: subs %[count], %[count], #1; "
               "bne __delay_%=\n"
               : : [count]"r"(count) : "cc");
}
```

- : output operands (none in this example)
  = 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



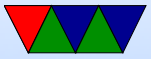
Memory Mapped I/O (MMIO)

- As opposed to separate I/O space (as found on x86 and some other processors)
- For HW#3 instead of using array for MMIO access, we will use inline assembly
- technically, to be correct, we need memory barriers (See BCM2835 Document 1.3)
  - The AXI bus can return reads out of orders if talking to different devices
  - When switching from one to another write barrier



before first write and read barrier after last read.

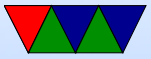
- Also modern out-of-order processors might let loads bypass loads or stores by pass stores. Fine for memory, not so much I/O
- ARM has a less strict memory model than x86



mmio_read() / mmio_write())

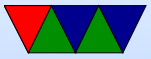
```
static inline void mmio_write(uint32_t address,
                              uint32_t data) {
        uint32_t *ptr = (uint32_t *)address;
        asm volatile("str %[data], [%[address]]" :
                : [address]"r"(ptr), [data]"r"(data));
}
```

NOTE: In the homeworks I provide a version of this, bcm2835_read() / bcm2835_write() that handle adjusting for the differing iobase on the different pi models.



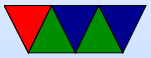
Writing a Device Driver

- Code to initialize the device
- Set of methods for interacting with device (read/write/ioctl/etc)
- Code to run if device is removed
- Interrupt handling



Device Initialization

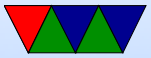
- Ideally the documentation will tell you how to do this
- The default values we'd like to use
  - 115200 8N1 Software Flow
  - 115200 Baud
  - 8 data bits (7 or 8)
  - no parity (even, odd, none)
  - 1 stop bit (1, 1.5, or 2)



PL011 UART Init – First Disable

- Disable UART by writing 0 to Command Register

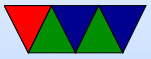
```
/* Disable UART -- Command Register */
bcm2835_write(UART0_CR, 0x0);
```



PL011 UART – Set up GPIO Pins

- Disable the pullup/pulldowns on GPIO Pins

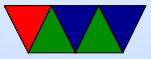
```
/* Disable the pull up/down on GPIO14 and 15 */
/* See the Peripheral Manual for more info */
/* Disable pull up/down and delay for 150 cycles */
bcm2835_write(GPIO_GPPUD, GPIO_GPPUD_DISABLE);
delay(150);

/* Pass the disable clock to GPIO pins 14 and 15 and delay
bcm2835_write(GPIO_GPPUDCLK0, (1 << 14) | (1 << 15));
delay(150);

/* Write 0 to GPPUDCLK0 to make it take effect */
bcm2835_write(GPIO_GPPUDCLK0, 0x0);
```



PL011 UART – Disable Interrupts

- You don't want interrupts happening while configuring
- For this HW we won't be using interrupts anyway

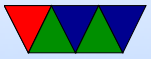
```
/* Mask all interrupts. */
bcm2835_write(UART0_IMSC, 0);

/* Clear pending interrupts. */
bcm2835_write(UART0_ICR, 0x7FF);
```



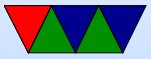
UART Interrupts You Can Get

- Supports one interrupt (UARTRXINTR), which is signaled on the OR of the following interrupts:

1. UARTTXINTR – if FIFO less than threshold or (if FIFO disabled) no data present
2. UARTRTINTR – if receive FIFO crosses threshold or (if FIFO disabled) data is received
3. UARTMSINTR which can be caused by
   - UARTCTSINTR (change in nUARTCTS)
   - UARTDSRINTR (change in the nUARTDSR)



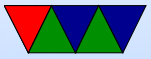
4. UARTEINTR (error in reception)
   - UARTOEINTR (overrun error)
   - UARTBEINTR (break in reception)
   - UARTPEINTR (parity error)
   - UARTFEINTR (framing error)



PL011 UART – Set speed

- Note: At some point around the Pi4 release the Pi people updated the firmware so all Pis use 48MHz base frequency instead of 3MHz
- Example: Calculate for 14.4kb/s
- Divider $= \frac{BaseFrequency}{16 \times Desired}$
- Divider $= \frac{48000000}{16 \times 14400} = 208.333$
- IBRD register = Integer part = 208.
  FBRD register $= (.333 \times 64) + 0.5 = 21.8$ so 22

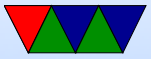
```
bcm2835_write(UART0_IBRD, 208);
bcm2835_write(UART0_FBRD, 22);
```



PL011 UART – Set 8N1

- Enable FIFO and Set 8 bit, no-parity, 1 stop bit in LCRH register

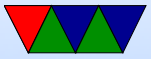
```
/* Enable FIFO */
/* And 8N1 (8 bits of data, no parity, 1 stop bit */
bcm2835_write(UART0_LCRH, UART0_LCRH_FEN |
                          UART0_LCRH_WLEN_8BIT);
```



PL011 UART – Enable

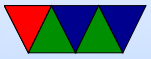
- Enable the UART as well as transmit and receive

```
/* Enable UART0, receive, and transmit */
bcm2835_write(UART0_CR, UART0_CR_UARTEN |
                        UART0_CR_TXE |
                        UART0_CR_RXE);
```



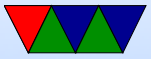
PL011 UART – Send byte

```
void uart_putc(unsigned char byte) {

        /* Check Flags Register */
        /* And wait until FIFO not full */
        while ( mmio_read(UART0_FR) & UART0_FR_TXFF ) {
        }

        /* Write our data byte out to the data register */
        bcm2835_write(UART0_DR, byte);
}
```



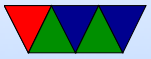
PL011 UART – Receive byte

```
unsigned char uart_getc(void) {

        /* Check Flags Register */
        /* Wait until Receive FIFO is not empty */
        while ( mmio_read(UART0_FR) & UART0_FR_RXFE ) {
        }

        /* Read and return the received data */
        /* Note we are ignoring the top 4 error bits */

        return mmio_read(UART0_DR);
}
```



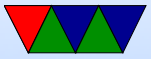
Sending more than just a byte

- We now know enough to send / receive single characters
- While that's fine for sending plain text, can we go fancier?



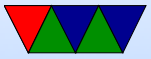
Escape Codes

- VT102/Ansi
- Historical reasons, oldest terminals. Used to be hundreds of types supported (see termcap file)
- Color, cursor movement
- The escape character (ASCII 27) used to specify extra commands



Carriage Return vs Linefeed

- Typewriters
- Carriage return (\r), go to beginning of line
- Linefeed (\n), move down a row
- DOS uses both CRLF
- UNIX uses just LF
- Old MacOS used just CR
- Most com programs want both, so our code should output both



Do other OSes have to handle this CR/LF difference

From `linux/drivers/tty/serial/serial_core.c`

```
void uart_console_write(struct uart_port *port,
                const char *s,unsigned int count,
                void (*putchar)(struct uart_port *, int)) {
        unsigned int i;

        for (i = 0; i < count; i++, s++) {
                if (*s == '\n')
                        putchar(port, '\r');
                putchar(port, *s);
        }
}
```