ECE 598 – Advanced Operating Systems
Lecture 6

Vince Weaver

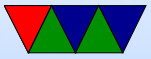
http://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

3 February 2015

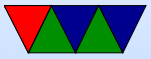
Announcements

- Homework #2 will be released shortly
- Raspberry Pi 2???

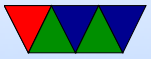


BCM2835 UART on the Pi

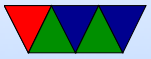
- Section 13 of the Peripheral Manual
- Two UARTS. Mini (pc reg layout compat) and ARM PL011. We use the latter.
- No IrDA or DMA support, no 1.5 stop bits.
- Separate 16x8 transmit and 16x12 receive FIFO memory. Why 12? 4 bits of error info on receive. overrun (FIFO overflowed), break (data held low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

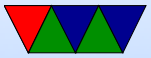


- Fully-programmable serial interface characteristics: data can be 5, 6, 7, or 8 bits
- even, odd, stick, or no-parity bit generation and detection
- 1 or 2 stop bit generation
- baud rate generation, dc up to UARTCLK/16
- 1/8, 1/4, 1/2, 3/4, and 7/8 FIFO interrupts



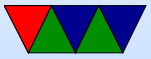
BCM2835 UART

- Can map to GPIO14/15 (ALT0), GPIO36/37 (ALT2), GPIO32/33 (ALT3)
- Base ni 0x20201000, 18 registers

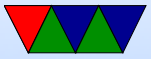


Hooking up Cable to Pi

- 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-`
- Adapter will provide 5V to your board, won't even need to USB-micro cable.
- Hookup:



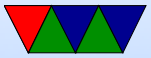
Red (5V) to pin 2,
Black (GND) to pin 6
White (TXD) to pin 8 (GPIO14)
Green (RXD) to pin 10 (GPIO15)



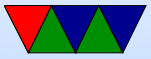
Inline Assembly

- 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 %[count], %[count], #1; "
                     "bne __delay_%=\n"
                : : [count]"r"(count) : "cc");
}
```

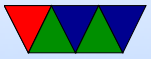


- : output operands
  = means write-only, + is read/write `r`=general reg
- : input operandss
- : clobbers – list of registers that have been changed memory is possible, as is cc for status flags
- can use `%[X]` to refer to reg X that can then use `[X]"r"(x)` to map to C variable



MMIO

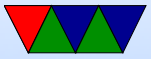
- Memory mapped I/O
- As opposed to separate I/O space (as found on x86 and some other processors)
- For HW#2 instead of using array for MMIO access, we will use inline assembly



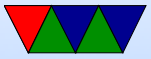
UART Interrupts

- 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)

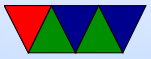


4. UARTEINTR (error in reception)
   - UARTOEINTR (overrun error)
   - UARTBEINTR (break in reception)
   - UARTPEINTR (parity error)
   - UARTFEINTR (framing error)

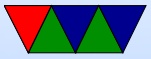


UART Init Code

```
/* Disable UART */
mmio_write(UART0_CR, 0x0);

/* Setup GPIO pins 14 and 15 */

/* Disable the pull up/down on pins 14 and 15 */
/* See the Peripheral Manual for more info */
/* Configure to disable pull up/down and delay for 150 cycles */
mmio_write(GPIO_GPPUD, GPIO_GPPUD_DISABLE);
delay(150);

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

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

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

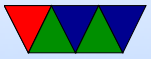


```
/* Set integer & fractional part of baud rate. */
/* Divider = UART_CLOCK/(16 * Baud)            */
/* Fraction part register = (Fractional part * 64) + 0.5 */
/* UART_CLOCK = 3000000; Baud = 115200.        */
/* Enable FIFO */
/* And 8N1 (8 bits of data, no parity, 1 stop bit */
mmio_write(UART0_LCRH, UART0_LCRH_FEN | UART0_LCRH_WLEN_8BIT);

/* Mask all interrupts. */
mmio_write(UART0_IMSC, UART0_IMSC_CTS | UART0_IMSC_RX |
                       UART0_IMSC_TX | UART0_IMSC_RT |
                       UART0_IMSC_FE | UART0_IMSC_PE |
                       UART0_IMSC_BE | UART0_IMSC_OE);

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



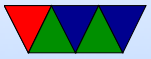
UART Send byte

```
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 */
        mmio_write(UART0_DR, byte);
}
```



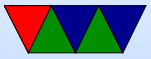
UART Receive byte

```
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);
}
```



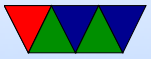
Escape Codes

- 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

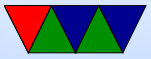


Carriage Return vs Linefeed

- Typewriters
- Carriage return (
  r), go to beginning of line
- Linefeed (
  n), move down a row
- DOS uses both CRLF
- UNIX uses just LF

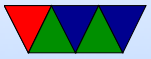


- MAC uses just CR
- Which does your terminal program use?



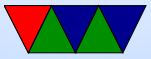
Writing header files

- Including with " " versus <>



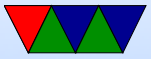
Writing printk

```
int printk(char *string,...) {

        va_list ap;
        va_start(ap, string);

        while(1) {
                if (*string==0) break;

                if (*string=='%') {
                        string++;
                        if (*string=='d') {
                                string++;
                                x=va_arg(ap, int);
```

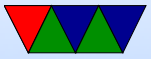


Scanning ATAGS

- List of variables passed by bootloader. A standard.
- We mostly care about getting memory size.
- Size, Tag-type, additional
- Located traditionally at 0x800 but you should really check r3 for addr.
- Starts with ATAG_CORE



- Ends with ATAG_NONE
- We wants ATAG_MEM and maybe ATAG_CMDLINE on Raspberry Pi.



Parsing Command Line

- Read into buffer
- When CR or LF happens, pass buffer to string that handles things.
- Complications: backspace? UNIX challenge
- Writing a parser, how? W/o strlen, strtok, etc?