ECE 598 – Advanced Operating Systems
Lecture 6

Vince Weaver

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

vincent.weaver@maine.edu

4 February 2016

Announcements

- Homework #2 was due

- Homework #3 will be released shortly

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

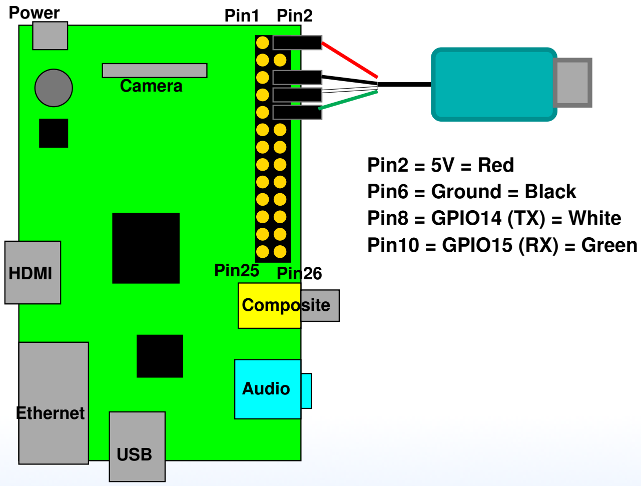
- Linux should come with a driver. May need to download PL2303 OSX or Windows driver.
- Some useful documentation:

  `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)



Pin2 = 5V = Red
Pin6 = Ground = Black
Pin8 = GPIO14 (TX) = White
Pin10 = GPIO15 (RX) = Green

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 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

MMIO

- Memory mapped I/O
- 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
- `mmio_write()`, `mmio_read()`
-
  ```
  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));
  }
  ```

Writing a Device Driver

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

UART Init Code

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

Set up the GPIO Pins

```
/* 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);
```

Disable Interrupts

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

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

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)

Set UART speed

- Calculate for 14.4kb/s
- Divider $= \frac{BaseFrequency}{16 \times Desired}$
- Divider $= \frac{3000000}{16 \times 14400} = 13.020$
- IBRD register = Integer part = 13.
  FBRD register $= (.020 \times 64) + 0.5 = 1.78$ so 1 or 2.
- `mmio_write(UART0_IBRD, 13);`
  `mmio_write(UART0_FBRD, 1);`

Set UART 8N1

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

Enable the UART

```
/* 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
- 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);
        }
}
```

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);
```