The mini UART

Eryn Vineyard

So, after going through a *bit* of difficulty with Mini-UART on the BCM2835, I thought I would document how I got it fully working (or perceptually it fully works).

What is the BCM2835?

For me, the BCM2835 is the Raspberry Pi Zero (or Pi 1)[1]. I'm sure there's other devices that use it and the ideas from here should apply. Taking a cursory glance at the BCM2837 (Pi 2/3)[2] and BCM2711 (Pi 4)[3] data sheets, it could also work there although I've only tested my code against the Raspberry Pi Zero.

What is mini UART?

In the words of the BCM2835 datasheet, "The mini UART is a secondary low throughput UART intended to be used as a console."[4] While the BCM2835 has a full (PL011)[5], it's not necessarily the easiest to use and is a bit overkill for a simple console.

In my experience, the biggest limitation is the 8-byte FIFO[4]. Today, however, we're not going to be limited by this as we're going to use interrupts to ensure the queue never spills.

Prerequisites

I'm going to assume that you already have the following:

1. Booting on the Pi (Andre Leiradella has a good tutorial on this)
2. Some means to use UART (UART-to-USB adapter, Flipper Zero, Digital Logic Analyzer, another Pi, etc)
3. A willingness to trust random numbers that I myself don't understand why they work

Initialization

Initializing Mini-UART can be split into 3 separate parts:

1. GPIO Initialization
2. Auxiliary I/O Initialization
3. Mini-UART Initialization

GPIO Initialization

> GPIO pins should be set up first [sic] before enabling the UART.
>
> —*BCM2835/2.2 p.g. 10*

So first, we need to decide what pins we actually want to use. Below is a table of pin pairs we could use[6].

Pair	Alt Mode
14:15	Alt 5
31:32	Alt 3
36:37	Alt 2

Where does the table come from?. You won't find it in the mini UART section anything about what GPIO pins you need to actually use. Instead, we need to look at the GPIO Alternative Function Assignments[6]. At the end of this section, we'll find that `TXD0` and `RXD0` correspond to "Auxiliary I/O"

After picking a pair of TX/RX pins, we need to configure them for pull-down as a floating state (especially with RX) is not a good idea. Here, the BCM2835 is *almost* clear[7]:

1. Write to `GPPUD` to set the required control signal
2. Wait 150 cycles
3. Write to `GPPUDCLKn` to set the clock
4. Wait 150 cycles
5. Write to `GPPUD` to remove the control signal

(where mini UART is from).

6. Write to `GPPUDCLKn` to clear the clock The purpose of each of these steps doesn't really concern us for the moment. As of now, we just need to resolve what the datasheet means by "cycles." In my own code, I've interpreted this as CPU cycles although this could very well mean GPU cycles, system clock cycles, peripheral clock cycles, etc.

Now the only thing left to do is actually write the code:

```
// --- snip ---

// gpio.zig

fn wait(count: usize) void {
    var c = count;
    while (c != 0) {
        c -= 1;
    }
}

// Table 6-2 - GPIO Pull-up/down Register (GPP
pub const PullMode = enum(u2) {
    Off = 0b00,
    Down = 0b01,
    Up = 0b10,
};

pub fn set_pull(pin: u8, pull: PullMode) void
    GPPUD.write(@intFromEnum(pull));

    wait(150);

    GPPUDCLK.insert(pin, true);

    wait(150);

    GPPUDCLK.remove(pin);
    GPPUD.write(0);
}

// --- snip ---
```

We can save `set_pull` for later.

Auxiliary I/O Initialization

The next step is to initialize the Auxiliary I/O interface. This is probably the simplest part of using the mini UART. The `AUXENB` let's us enable mini UART, although we do need to be careful, given it states "The UART will immediately start receiving data, especially if the UART1_RX line is *low*."[8] We do, however, still need to enable it first before configuring UART given "If clear... [it also] disables any mini UART register access."

As this is just setting 1 bit, we can do it as simply as:

```
pub fn enable_mini_uart() void {
    AUXENB.insert(0, true);
}
```

Mini UART Initialization

Now this is where we need to actually think of what we want our UART to represent. If you don't care about any sort of explanation, here's the configuration for an 8N1 UART:

```
pub fn configure_uart_8n1(baud: u32) void {
    // Disable interrupts
    // Page 12: AUX_MU_IER_REG Register
    // When bit 0/1 is 0, interrupts are disab
    AUX_MU_IER_REG.write(0);

    // Disable RX/TX during configuration
    // Page 17: AUX_MU_CNTL_REG Register
    // When bit 0/1 is 0, UART receiver/transm
    AUX_MU_CNTL_REG.write(0);

    // Set data size to 8 bits
    // Page 14: AUX_MU_LCR_REG Register
    // When bit 0 is 1, UART is in 8-bit mode,
    // Errata: "bit 1 must be set for 8 bit mo
    AUX_MU_LCR_REG.write(0b11);

    // Put RTS high (indicate request to send)
    // Page 14: AUX_MU_MCR_REG Register
    // When bit 1 is 0, RTS line is high, else
    AUX_MU_MCR_REG.write(0);

    // Clear FIFO
    // Page 13: AUX_MU_IER_REG Register
    // When bit 1/2 is 1, receive/transmit wil
    AUX_MU_IIR_REG.write(0b110);

    // Set baud rate
    // Page 11: 2.2.1 Mini UART Implementation
    // The baudrate formula is given as:
    //   (system_clock_freq)/(8 * (baudrate_re
    AUX_MU_BAUD_REG.write(CLOCK_FREQ / (8 * ba

    // Enable RX/TX again
    // Page 17: AUX_MU_CNTL_REG Register
```

```
    // When bit 0/1 is 1, UART receiver/transm
    AUX_MU_CNTL_REG).write(0b11);
}
```

Now if you want something other than 8N1, we need to venture deeper.

> **Warning:** I haven't tested anything other than 8N1 so this is my personal interpretation of the BCM2835 and the BCM2835 datasheet errata.

Data Size

For 7 bits: clear bit 0 of `AUX_MU_LCR_REG`[9].

For 8 bits, we find the first piece of errata:

> LCR register, bit 1 must be set for 8 bit mode, like a 16550
> write a 3 to get 8-bit mode
>
> *—elinux BCM2835 errata/p14*

In short, set bits 0 and 1 of `AUX_MU_LCR_REG`.

RTS/CTS

We can get the mini UART to either manually or automatically set our request-to-send and automatically control clear-to-send. Before we can use RTS/CTS, we need to configure GPIO pins. As before, we can look towards the GPIO Alternative Functions[6] table to see we can use the following pins:

Pair	Alt Mode
16:17	Alt 3
30:31	Alt 3
38:39	Alt 2

And just as before, we configure them for pull-down.

Manual Flow Control

For manual control of RTS, we can look towards `AUX_MU_MCR_REG`[10], which tells us to set bit 1 to set RTS low (and clear to set RTS high).

Automatic Flow Control

For automatic control, we can look towards `AUX_MU_CNTL_REG`[11].

For `CTS`, set bit 3 of `AUX_MU_CNTL_REG` which automatically stops transmission if the CTS pin is de-asserted

For `RTS`, here we have some configuration, depending on when to de-assert `RTS` in relation to the RX FIFO. Set bits 4 and 5 to:

- `0b00` - de-assert with 3 slots left
- `0b01` - de-assert with 2 slots left
- `0b10` - de-assert with 1 slot left
- `0b11` - de-assert with 4 slots left

We can also choose to invert `CTS` and `RTS` (such that assertion is when the pin is low) by configuring bits 6 (for `RTS`) and 7 (for `CTS`).

Break Condition

While the mini UART can't detect a break, it can send out a break. Setting bit 6 of `AUX_MU_LCR_REG`[9] for 12 bits times (time it takes to send out 12 bits) will indicate a break condition.

Baud Rate

Finally, we can configure the baud rate for our mini UART through the `AUX_MU_BAUD`[12]. The first 16 bits of this register specify our baudrate, but first we need to calculate the baudrate.

Luckily the datasheet gives us a formula in section 2.2.1[13]:

$$\text{baudrate} = \frac{\text{system clock freq}}{8 \cdot (\text{baudrate reg} + 1)}$$

Now we have two variables to plug into this, system clock frequency and our baudrate (e.g. `9600`).

According to the errata[14], for the Pi we have a clock frequency of 250MHz, so we can just hardcode it in as `250_000_000`.

> ▶ System Clock Frequency

Finally, after calculating our baudrate, we can truncate it into just 16 bits before writing it into `AUX_MU_BAUD`. According to the implementation details[13], there isn't actually any maximum baudrate, with the only limit being however fast the CPU can handle (although I wouldn't recommend anything above `921600` when using interrupts and `115200` when using polling).

Putting it all together

Finally, we can put everything together, with just a few extra details:

1. We should probably disable interrupts (at least temporarily) to ensure we're not getting interrupted for `0`s.
2. We should probably disable the transmitter and receiver to ensure we're not transmitting or filling the FIFO with `0`s.
3. We should probably clear the FIFO as it probably has junk
4. We should enable the transmitter and receiver again after configuration.

We can accomplish each of these with:

1. `AUX_MU_IER_REG`[19], clearing bits 0 and 1 to disable interrupts.
2. `AUX_MU_CNTL_REG`[11], clearing bits 0 and 1 to disable the receiver and the transmitter.
3. `AUX_MU_IIR_REG`[20], setting bits 1 and 2 to clear the RX/TX queues.
4. `AUX_MU_CNTL_REG`, except now setting bits 0 and 1 to enable the receiver and the transmitter.

Putting it all together, we get:

```
pub const DataSize = enum {
    .@"7" = 0b00,
    .@"8" = 0b11,
};

pub const RtsControl = enum(u3) {
    .@"4" = 0b011,
    .@"3" = 0b000,
    .@"2" = 0b001,
    .@"1" = 0b010,
    Manual = 0b100,
};

pub const CtsControl = enum {
    Auto,
    Off,
}

pub fn configure_uart(
    baud: u32,
    data_size: DataSize,
    rts: RtsControl,
    cts: CtsControl,
) {
    AUX_MU_IER_REG.write(0);
    AUX_MU_CNTL_REG.write(0);

    AUX_MU_LCR_REG.write(@intFromEnum(data_siz

    if (rts == .Disabled) {
        AUX_MU_MCR_REG.write(0);
    } else {
        AUX_MU_CNTL_REG.write(@as(u2, @truncat
    }
```

```
    if (cts == .Auto) {
        AUX_MU_CNTL_REG.write(1 << 3);
    }

    AUX_MU_IIR_REG.write(0b110);

    AUX_MU_BAUD_REG.write(CLOCK_FREQ / (8 * ba

    AUX_MU_CNTL_REG.bor(0b11);
}
```

Tada! An initialized mini UART!

Polling

First, we'll write some simple interfaces that just poll for availability. They might not be the most efficient but hey- at least they work.

Transmitting

If you don't care for any explanation:

```
pub fn write_byte(byte: u8) void {
    while (AUX_MU_STAT_REG.get() & 0b10) == 0)
    AUX_MU_IO_REG.write(@as(u32, byte) & 0xFF)
}
```

If you do care for explanation, transmitting a byte is composed of two main parts:

1. Checking if we have enough space in the TX queue
2. Putting the byte into the TX queue

We can accomplish each of these with:

1. `AUX_MU_STAT_REG`[21], bit 1 indicates if the TX queue "can accept at least one more [byte]."

2. `AUX_MU_IO_REG`[22], writing bits 0 to 7 puts data into the TX queue "(provided it is not full)."

And now we can transmit data! Wasn't that simple?

Receiving

Receiving is similarly simple. If you don't care for any explanation:

```
pub fn read_byte(byte: u8) void {
    while (AUX_MU_LSR_REG.get() & 0b1 != 0) {}
    return @truncate(AUX_MU_IO_REG.get());
}
```

If you do care for explanation, receiving a byte is similar to transmitting:

1. Checking if there's a byte available in the RX queue
2. Reading the byte from the RX queue

We can accomplish each of these with:

1. `AUX_MU_LSR_REG`[23], bit 0 indicates if "the [RX queue] holds at least 1 [byte]."
2. `AUX_MU_IO_REG`[22], reading bits 0 to 7 gets data from the RX queue "(provided it is not empty)."

And now we can read data!

Some extras

You might've noticed that I've been using Zig in all of these example code blocks. With Zig, we can implement `Io.Reader`[24] and `Io.Writer`[25] and treat our UART just like any other Io stream.

Writer

Implementing Zig's `Io.Writer` is pretty simple. We only need 1 function, `drain`, although today we're going to also implement `flush`.

`drain`'s signature is a bit interesting:

```
*const fn (
    w: *Writer,
    data: []const []const u8,
    splat: usize,
) Error!usize
```

We get a 2 dimensional array for our data and a splat parameter. I'm not actually sure what's the purpose of the splat parameter is (my guess is vectored writing) or any additional dimensions of `data`. For me, ignoring `splat` and looking at only `data[0]` always worked.

```
fn writer_drain(
    io_w: *std.Io.Writer,
    data: []const []const u8,
    splat: usize,
) !usize {
    _ = io_w;
    _ = splat;

    for (data[0]) |byte| {
        write_byte(byte);
    };

    return data[0].len;
}
```

Next, we have the `flush` implementation. Here we look at `AUX_MU_STAT_REG`[21] once again, with bit 9 representing "the transmitter is idle and the transmitter FIFO is empty." Basically, we just loop until bit 9 is set.

```
fn writer_flush(io_w: *std.Io.Writer) !void {
    _ = io_w;

    while (AUX_MU_STAT_REG.get() & 0x200 != 0)
}
```

Finally, we just need to implement the VTable:

```
const writer_vtable: std.Io.Writer.VTable = .{
    .drain = writer_drain,
    .flush = writer_flush
};
pub var writer = std.Io.Writer{
    .buffer = undefined,
    .vtable = &writer_vtable
};
```

Tada! We now have `Io.Writer` implemented! With this, we can now print out formatted strings to our mini UART:

```
try writer.print("Hello {s}!", .{"world"});
```

Reader

Implementing `Io.Reader` is about as simple as `Io.Writer`. Here we need to implement `stream` which has us putting data into a writer until some limit. In short, no new code.

```
fn stream(
    io_r: *std.Io.Reader,
    io_w: *std.Io.Writer,
    limit: std.Io.Limit,
) !usize {
    _ = io_r;

    if (limit.toInt()) |max| {
        for (0..max) |_| {
            try io_w.writeByte(read_byte());
        }

        return max;
    } else {
        return std.Io.Reader.StreamError.ReadF
    }
}
```

Then, we just need to implement the VTable:

```
const reader_vtable: std.Io.Reader.VTable = .{
    .stream = stream
};

pub var reader = std.Io.Reader{
    .buffer = undefined,
    .seek = 0,
    .end = 0,
    .vtable = &reader_vtable
};
```

And just like that, we now can treat our mini UART as any other stream, opening our possibilities!

Interrupts

Now it's time for the fun part, interrupts. I won't explain how to setup the exception vector (that's left as an exercise for the reader). First, we need to actually configure the mini UART to send interrupts.

First, however, we should figure out what the interrupts will tell us:

> [Enable transmit interrupt]: If this bit is set the interrupt line is asserted whenever the transmit FIFO is empty.
>
> [Enable receive interrupt]: If this bit is set the interrupt line is asserted whenever the receive FIFO holds at least 1 byte.
>
> *—BCM2835/2.2.2 p.g. 12*

In short, we want receive interrupts to always be enabled. For transmit interrupts, however, we only want to enable them when we have data and disable them as soon as we've sent all our data.

Interrupt Setup

Before we can think about using interrupts, we need somewhere to put our data. For this, a ring buffer is probably one of the simplest data structures for this purpose. Implementing a ring buffer is left as an exercise to the reader.

One of annoying issues I experienced was not realizing that a race condition was possible here. As such, we also need to implement a critical section handler (or just manually disable and enable interrupts) so that we don't get interrupted when interacting with our ring buffers.

Implementing a critical section is, too, left as an exercise for the reader.

For now, assume have have the following declarations:

```
var tx_list: StackRingBuffer(u8, 128) = .init(
var rx_list: StackRingBuffer(u8, 128) = .init(
```

Configuring Interrupts

Before we can configure mini UART interrupts, we need to enable Auxiliary I/O interrupts (I'll assume that interrupts are already globally enabled). For this, we look towards the peripherals interrupt table[26]. We see that auxiliary interrupts are numbered #29 and with this, we know to set bit 29 in `ENABLE_IRQS_1`[27].

```
fn enable_aux_interrupts() void {
    ENABLE_IRQS.insert(AUX_INTERRUPT_BIT, true
}
```

Now that we have Auxiliary I/O interrupts enabled, we can configure mini UART interrupts.

So as you've might've guessed, we'll be using `AUX_MU_CNTL_REG`[11]. One might think to enable interrupts, we just set bits 0 and 1 to 1 but that isn't actually correct. If we open up the errata,

> Bits 3:2 are marked as don't care, but are actually required in order to receive interrupts.
>
> Bits 1:0 are swaped. bit 0 is receive interrupt and bit 1 is transmit. we find two different things to consider
>
> *—elinux BCM2835 errata/p12*

As we have transmit interrupts constantly being flipped on and off, I decided to just write a function to encapsulate all possible states of interrupts:

```
fn interrupt_state(tx: bool, rx: bool) void {
    if (tx and rx) {
        AUX_MU_IER_REG.write(0b1111);
    } else if (tx and !rx) {
        AUX_MU_IER_REG.write(0b1110);
    } else if (!tx and rx) {
        AUX_MU_IER_REG.write(0b0101);
    } else if (!tx and !rx) {
        AUX_MU_IER_REG.write(0b0000);
    }
}
```

I'm not sure why any of these numbers work except for when `tx` and `rx` are both false/true. But they do work!

Interrupt Handler

Now we need to configure the interrupt handler. For this, we need to:

1. Check that an auxiliary interrupt is pending
2. A mini UART interrupt is pending
3. If there's data in the receiver, read it
4. If there's data for the transmitter, send it
5. If there isn't data for the transmitter, disable transmission interrupts

We can accomplish #1 by reading bit 29 of `IRQ_PENDING_1`[28].

For numbers 2-4, we can look towards `AUX_MU_IIR_REG`[20], where if bit 0 is clear, an interrupt is pending and bits 1 and 2 telling us if the interrupt is from the transmitter or receiver.

Now to implement this:

```
fn interrupt_handler()
    callconv(.{ .arm_interrupt = .{ .type = .i
    // Always a good idea to use barriers on i
    mem.barrier(.Write);
    defer mem.barrier(.Write);

    if (!IRQ_PENDING_1.get(AUX_INTERRUPT_BIT))

    const IIR = AUX_MU_IIR_REG.get();

    if (IIR & 0b1 == 1) return;

    // RX has byte
    if (IIR & 0b110 == 0b100) {
        rx_list.push(read_byte());
    }

    // TX has space
    if (IIR & 0b110 == 0b010) {
        if (tx_list.pop()) |byte| {
            write_byte(byte);
        } else {
            interrupt_state(false, true);
        }
    }
}
```

Receiving

Receiving data is pretty simple:

1. Enter a critical section
2. Pop data from rx_list
3. Exit critical section

```
pub fn read_byte_async() ?u8 {
    const cs = mem.enter_critical_section();
    defer cs.exit();

    return rx_list.pop();
}
```

Transmitting

Transmitting data is also pretty simple:

1. Enter a critical section
2. Insert data into tx_list
3. Enable TX interrupts
4. Exit critical section

```
pub fn write_byte_async(byte: u8) void {
    const cs = mem.enter_critical_section();
    defer cs.exit();

    tx_list.push(byte);

    interrupt_state(true, true);
}
```

Flushing

Since we now have a secondary queue, we need to update our `flush` function to account for this. Essentially, enter critical section, get list length, leave critical section.

```
fn tx_list_length() void {
    const cs = mem.enter_critical_section();
    defer cs.exit();

    return tx_list.length();
}

pub fn writer_flush(io_w: *std.Io.Writer) !voi
    _ = io_w;

    while (
        AUX_MU_STAT_REG.get() & 0x200 != 0 and
        tx_list_length() > 0
    ) {}
}
```

Io Interfaces

Updating `Io` interfaces is left as an exercise to the reader.

Closing Remarks

The code we've written today does about everything you might want from a mini UART. Maybe I'll write a future blog post on some potential optimizations we may might want to make for this (such as directly streaming receive interrupts into a writer). In addition, you may want to add timeouts to `read_byte` and `write_byte` to avoid waiting forever.

> **Warning:** None of the code in this post has been tested as written. Most of this code is the "abridged" version from my own mini UART code. No Generative AI was used in the production of this blog post (research, writing, etc), this post is entirely natural stupidity. All code examples are licensed under the Apache-2.0 License.

Acknowledgements

- The Mini UART
- Sending Files Via the Mini UART
- dwelch67's Raspberry Pi Examples

Footnotes

1. Link: "The BCM2835 is the Broadcom chip used in the Raspberry Pi 1 Models A, A+, B, B+, the Raspberry Pi Zero, the Raspberry Pi Zero W, and the Raspberry Pi Compute Module 1." ↩
2. Link: "The Broadcom chip used in early models of Raspberry Pi 2 Model B." ↩
3. Link: "This is the Broadcom chip used in the Raspberry Pi 4 Model B, Compute Module 4, and Pi 400." ↩
4. BCM2835/2.2 p.g. 10 - Mini UART ↩ ↩²
5. BCM2835/13 p.g. 175 - UART ↩
6. BCM2835/6.2 p.g. 102-104 - GPIO Alternative Functions ↩ ↩² ↩³
7. BCM2835/6.1 p.g. 101 - `GPPUDCLKn` ↩
8. BCM2835/2.1.1 p.g. 9 - `AUXENB` ↩
9. BCM2835/2.2.2 p.g. 14 - `AUX_MU_LCR_REG` ↩ ↩²
10. BCM2835/2.2 p.g. 14 - `AUX_MU_MCR_REG` ↩
11. BCM2835/2.2.2 p.g. 16 - `AUX_MU_CNTL_REG` ↩ ↩² ↩³
12. BCM2835/2.2.2 p.g. 19 - `AUX_MU_BAUD` ↩
13. BCM2835/2.2.1 p.g. 11 - Mini UART implementation details ↩ ↩²
14. elinux BCM2835 errata/the BCM2835 on the RPI ↩
15. Link: "Mailboxes facilitate communication between the ARM and the VideoCore." ↩
16. Link: Mailbox get clock rate ↩
17. Link: Mailbox clocks ↩
18. Link: Accessing Mailboxes ↩
19. BCM2835/2.2.2 p.g. 12 - `AUX_MU_IER_REG` ↩
20. BCM2835/2.2.2 p.g. 13 - `AUX_MU_IIR_REG` ↩ ↩²
21. BCM2835/2.2.2 p.g. 18 - `AUX_MU_STAT_REG` ↩ ↩²
22. BCM2835/2.2.2 p.g. 11 - `AUX_MU_IO_REG` ↩ ↩²
23. BCM2835/2.2.2 p.g. 15 - `AUX_MU_LSR_REG` ↩
24. `std.Io.Reader` ↩

25. `std.Io.Writer` ↩
26. BCM2835/7.5 p.g. 113 - ARM peripherals interrupts table ↩
27. BCM2835/7.5 p.g. 112 - Enable IRQs 1 ↩
28. BCM2835/7.5 p.g. 112 - IRQ pending 1 ↩

Last edited May 12th, 2026