ECGR 6181/8181 - Lab 2

Objective: Bare Metal UART device driver development

Outcomes:

After this lab, you will be to

- Understand the process of writing a device driver
- Write a polling based baremetal driver for UART
- Test the driver

Introduction

Starting with a UART driver specifically has its advantages. UARTs are very common peripherals, they're much simpler than other serial buses such as SPI or I2C, and the UART pretty much corresponds to standard input/output when run in QEMU.

Before writing a peripheral device driver, we need to understand, the following about the device:

- How it performs its function(s). Whether it's a communication device, a signal converter, or anything else, there are going to be many details of how the device operates. In the case of a UART device, some of the things that fall here are, what baud rates does it support? Are there input and output buffers? When does it sample incoming data?
- How it is controlled. Most of the time, the peripheral device will have several registers, writing and reading them is what controls the device. You need to know what the registers do.
- How it integrates with the hardware. When the device is part of a larger system, which could be a system-on-a-chip or a motherboard-based design, it somehow connects to the rest of the system.
- Does the device take an external input clock and, if so, where from? Does enabling the device require some other system conditions to be met? The registers for controlling the device are somehow accessible from the CPU, typically by being mapped to a particular memory address.

Basic UART operation

UART is a fairly simple communications bus. Data is sent out on one wire, and received on another. Two UARTs can thus communicate directly, and there is no clock signal or synchronization of any kind, it's instead expected that both UARTs are configured to use the same baud rate. UART data is transmitted in packets, which always begin with a start bit,

followed by 5 to 9 data bits, then optionally a parity bit, then 1 or 2 stop bits. A packet could look like this:

Start bit	0	1	2	3	4	5	6	7	P	S

A pair of UARTs needs to use the same frame formatting for successful communication. In practice, the most common format is 8 bits of data, no parity bit, and 1 stop bit. This is sometimes written as 8-N-1 in shorthand, and can be written together with the baud rate like 115200/8-N-1 to describe a UART's configuration. On to the specific UART device that we have. The Realview PB hardware series comes with the PrimeCell UART PL011, the reference manual is also available from the ARM website.
https://developer.arm.com/documentation/ddi0183/g/
https://developer.arm.com/documentation/dui0417/d/programmer-s-reference/

Reading through the manual, we can see that the PL011 is a typical UART with programmable baud rate and packet format settings, that it supports infrared transmission, and FIFO buffers both for transmission and reception. Additionally there's Direct Memory Access (DMA) support, and support for hardware flow control. In the context of UART, hardware flow control means making use of two additional physical signals so that one UART can inform another when it's ready to send, or ready to receive, data.

Key PL011 registers
The PL011 UART manual also describes the registers that control the peripheral, and we can identify the most important registers that our driver will need to access in order to make the UART work. We will need to work with:

- Data register (DR). The data received, or to be transmitted, will be accessed through DR
- Receive status / error clear register (RSRECR). Errors are indicated here, and the error flag can also be cleared from here.
- Flag register (FR). Various flags indicating the UART's status are collected in FR.
- Integer baud rate register (IBRD) and Fractional baud rate register (FBRD). Used together to set the baud rate.
- Line control register (LCR_H). Used primarily to program the frame format.
- Control register (CR). Global control of the peripheral, including turning it on and off. In addition, there are registers related to interrupts, but we will begin by using polling, which is inefficient but simpler. We will also not care about the DMA feature.

.

Writing the driver

In higher-level programming, you can usually treat drivers as a black box, if you even give them any consideration. They're there, they do things with hardware, and they only have a few functions you're exposed to. Now that we're writing a driver, we have to consider what it consists of, the things we need to implement. Broadly, we can say that a driver has:

- An initialization function. It starts the device, performing whatever steps are needed. This is usually relatively simple.
- Configuration functions. Most devices can be configured to perform their functions differently. For a UART, programming the baud rate and frame format would fall here. Configuration can be simple or very complex.
- Runtime functions. These are the reason for having the driver in the first place, the interesting stuff happens here. In the case of UART, this means functions to transmit and read data.
- A deinitialization function. It turns the device off, and is quite often omitted.
- Interrupt handlers. Most peripherals have some interrupts, which need to be handled in special functions called interrupt handlers, or interrupt service routines. We won't be covering that in this lab.

Now we have a rough outline of what we need to implement. We will need code to start and configure the UART, and to send and receive data. Let's get on with the implementation.

In the header file uart_pl011.h (see Lab2 folder) we first define a struct of PL011 registers

```
typedef volatile struct __attribute__((packed)) {
  uint32_t DR;                      /* 0x0 Data Register */
  uint32_t RSRECR; /* 0x4 Receive status / error clear register */
  uint32_t _reserved0[4];                /* 0x8 - 0x14 reserved */
  const uint32_t FR;                     /* 0x18 Flag register */
  uint32_t _reserved1;                   /* 0x1C reserved */
  uint32_t ILPR;                /* 0x20 Low-power counter register
*/
  uint32_t IBRD;                /* 0x24 Integer baudrate register */
  uint32_t FBRD;             /* 0x28 Fractional baudrate register */
  uint32_t LCRH;                     /* 0x2C Line control register */
  uint32_t CR;                           /* 0x30 Control register */
} uart_registers;
```

`__attribute__((packed))` is a GCC attribute (also recognized by some other compilers like clang) that tells the compiler to use the struct as it is written, using the least amount of memory possible to represent it. Forcing structs to be packed is generally not a great idea when working with “normal” data, but it’s very good practice for structs that represent registers.

We define an enumeration of error codes

```
typedef enum {
        UART_OK = 0,
        UART_INVALID_ARGUMENT_BAUDRATE,
        UART_INVALID_ARGUMENT_WORDSIZE,
        UART_INVALID_ARGUMENT_STOP_BITS,
        UART_RECEIVE_ERROR,
        UART_NO_DATA
} uart_error;
```

Configuration parameters such as baud rate, word size etc. are defined by a struct uart_config

```
typedef struct {
    uint8_t     data_bits;
    uint8_t     stop_bits;
    bool        parity;
    uint32_t    baudrate;
} uart_config;
```

Bit masks to access individual bits in the register are defined to make the code more readable. For example,

```
#define DR_DATA_MASK    (0xFFu)
#define FR_BUSY         (1 << 3u)
```

We wrap up the header file with a declaration of the interface methods implemented by the driver.

```
uart_error uart_configure(uart_config* config);
void uart_putchar(char c);
void uart_write(const char* data);
uart_error uart_getchar(char* c);
```

In the driver implementation (uart_pl011.c), we first perform some validation of the configuration, returning the appropriate error code if some parameter is outside the acceptable range.

With validation done, the rest of the function essentially follows the PL011 UART's manual for how to configure it. First the UART needs to be disabled, allowed to finish an ongoing transmission, if any, and its transmit FIFO should be flushed. Here's the code -

```
 /* Disable the UART */
 uart0->CR &= ~CR_UARTEN;
 /* Finish any current transmission, and flush the FIFO */
 while (uart0->FR & FR_BUSY);
 uart0->LCRH &= ~LCRH_FEN;
```

Next we configure the UART's baudrate. This is another operation that translates to fairly simple code, but requires a careful reading of the manual. To obtain a certain baudrate, we need to divide the (input) reference clock with a certain divisor value. The divisor value is stored in two registers, IBRD for the integer part and FBRD for the fractional part. According to the manual, baudrate divisor = reference clock / (16 * baudrate) . The integer part of that result is used directly, and the fractional part needs to be converted to a 6-bit number m , where m = integer((fractional part * 64)+ 0.5) . We can translate that into C code as follows:

```
 /* Set baudrate */
 double intpart, fractpart;
 double baudrate_divisor = (double)refclock / (16u *
config->baudrate);
 fractpart = modf(baudrate_divisor, &intpart);

 uart0->IBRD = (uint16_t)intpart;
 uart0->FBRD = (uint8_t)((fractpart * 64u) + 0.5);
```

Since our reference clock is 24 MHz, as we established before, the refclock variable is 24000000u. Assuming that we want to set a baudrate of 9600 , first the baudrate_divisor will be calculated as 24000000 / (16 * 9600) , giving 156.25 . The modf function will helpfully set intpart to 156 and fractpart to 0.25 . Following the manual's instructions, we directly write the 156 to IBRD , and convert 0.25 to a 6-bit number. 0.25 * 64 + 0.5 is 16.5 , we only take the integer part of that, so 16 goes into FBRD . Note that 16 makes sense as a representation of 0.25 if you consider that the largest 6-bit number is 63 .

We continue now by setting up the rest of the configuration - data bits, parity and the stop bit.

```
uint32_t lcrh = 0u;

  /* Set data word size */
  switch (config->data_bits)
  {
  case 5:
      lcrh |= LCRH_WLEN_5BITS;
      break;
  case 6:
      lcrh |= LCRH_WLEN_6BITS;
      break;
  case 7:
      lcrh |= LCRH_WLEN_7BITS;
      break;
  case 8:
      lcrh |= LCRH_WLEN_8BITS;
      break;
  }

  /* Set parity. If enabled, use even parity */
  if (config->parity) {
      lcrh |= LCRH_PEN;
      lcrh |= LCRH_EPS;
      lcrh |= LCRH_SPS;
  } else {
      lcrh &= ~LCRH_PEN;
      lcrh &= ~LCRH_EPS;
      lcrh &= ~LCRH_SPS;
  }

  /* Set stop bits */
  if (config->stop_bits == 1u) {
      lcrh &= ~LCRH_STP2;
  } else if (config->stop_bits == 2u) {
      lcrh |= LCRH_STP2;
  }

  /* Enable FIFOs */
  lcrh |= LCRH_FEN;
```

```
    uart0->LCRH = lcrh;
```

Here we just pick the correct bits to set or clear depending on the provided configuration. One thing to note is the use of the temporary lcrh variable where the value is built, before actually writing it to the LCRH register at the end. It is sometimes necessary to make sure an entire register is written at once, in which case this is the technique to use. In the case of this particular device, LCRH can be written bit-by-bit, but writing to it also triggers updates of IBRD and FBRD , so we might as well avoid doing that many times. At the end of the above snippet, we enable FIFOs for potentially better performance, and write the LCRH as discussed. After that, everything is configured, and all that remains is to actually turn the UART on:

```
uart0->CR |= CR_UARTEN;
```

Read and Write functions

We can start the UART with our preferred configuration now, so it's a good time to implement functions that actually perform useful work - that is, send and receive data. Code for sending is very straightforward:

```
void uart_putchar(char c) {
    while (uart0->FR & FR_TXFF);
    uart0->DR = c;
}

void uart_write(const char* data) {
    while (*data) {
        uart_putchar(*data++);
    }
}
```

Given any string, we just output it one character at a time. The TXFF bit in FR that uart_putchar() waits for indicates a full transmit queue - we just wait until that's no longer the case.

And the data reception code -

```
uart_error uart_getchar(char* c) {
    if (uart0->FR & FR_RXFE) {
        return UART_NO_DATA;
    }
```

```
    *c = uart0->DR & DR_DATA_MASK;
    if (uart0->RSRECR & RSRECR_ERR_MASK) {
        /* The character had an error */
        uart0->RSRECR &= RSRECR_ERR_MASK;
        return UART_RECEIVE_ERROR;
    }
    return UART_OK;
}
```

First it checks if the receive FIFO is empty, using the RXFE bit in FR . Returning UART_NO_DATA in that case tells the user of this code not to expect any character. Otherwise, if data is present, the function first reads it from the data register DR , and then checks the corresponding error status - it has to be done in this order, once again according to the all-knowing manual. The PL011 UART can distinguish between several kinds of errors (framing, parity, break, overrun) but here we treat them all the same, using RSRECR_ERR_MASK as a bitmask to check if any error is present. In that case, a write to the RSRECR register is performed to reset the error flags.

Testing the driver (uart_test.c)

The main function asks the UART driver to configure it for 9600/8-N-1 , a commonly used mode, and then outputs some text to the screen. The while loop constantly polls the UART for incoming data and appends any characters read to a buffer, and prints that same character back to the screen. Then, when a carriage return (\r) is read, it calls parse_cmd(). That's a very basic method of waiting for something to be input and reacting on the Enter key. parse_cmd() is a simple function that has responses in case the input line was help or uname.

We use the same startup.s file as Lab 1, changing the initial function to main. The linker script stays the same as Lab 1.

```
$ arm-none-eabi-as -mcpu=cortex-a8 -o startup.o startup.s
$ arm-none-eabi-gcc -c -mcpu=cortex-a8 -o uart_pl011.o uart_pl011.c
$ arm-none-eabi-gcc -c -mcpu=cortex-a8 -o uart_test.o uart_test.c
$ arm-none-eabi-gcc -nostartfiles -T uart_test.ld startup.o uart_pl011.o uart_test.o -o uart
$ arm-none-eabi-objcopy -O binary uart uart.bin

$ qemu-system-arm -M realview-pb-a8 -nographic -kernel uart.bin
```

Type in help and uname to check if you are getting the messages specified in the test file.

Ctrl-A x to terminate QEMU

To do -

1. Modify putchar() function so that it prints uppercase characters.
2. Implement a simple bare metal calculator that perform addition, subtraction, multiplication and division. User interaction should look like the following -

   user input: calc 3 + 4

   output: 7

   In the initial version demonstrate functionality, assuming no errors. In the second version, incorporate basic error checking.

The material in this lab is adapted from the book "Bare-metal programming for ARM" by Daniels Umanovskis. The book and the code is available for free at https://github.com/umanovskis/baremetal-arm
Please consult the book for additional material on bare-metal programming, especially the chapters on bootloaders, and interrupt based device drivers.