Lesson UART

- Clock Systems and Timing
- UART
- Queues
- Lab: UART

Clock Systems and Timing

Clock System & Timing

A crystal oscillator is typically used to drive a processor's clock. You will find that many external crystal oscillator clocks are 20Mhz or less. A processor will utilize a "phased-lock-loop" or "PLL" to generate a faster clock than the crystal. So, you could have a 4Mhz crystal, and the PLL can be used to internally multiply the clock to provide 96Mhz to the processor. The same 96Mhz is then fed to microcontroller peripherals. Many of the peripherals have a register that can divide this higher clock to slower peripherals that may not require a high clock rate.

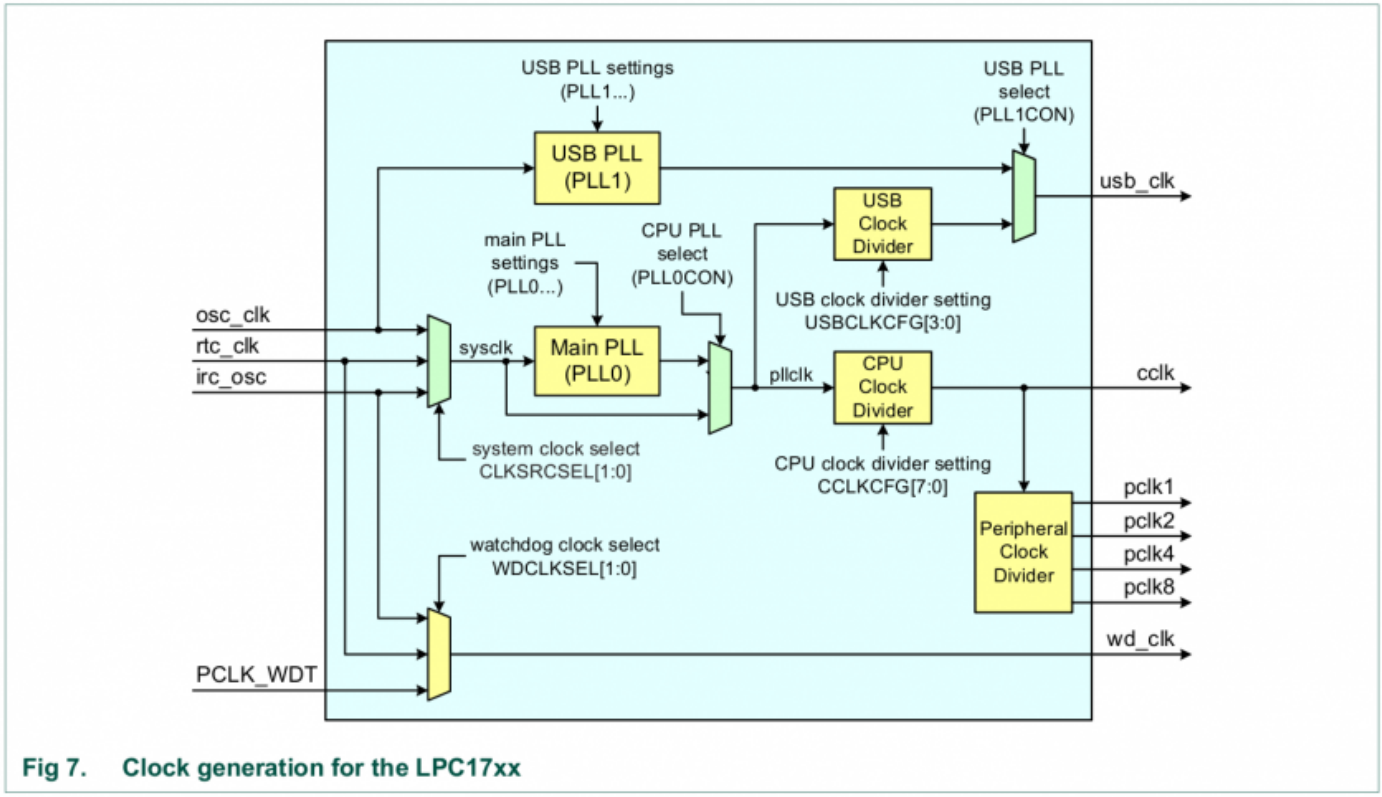
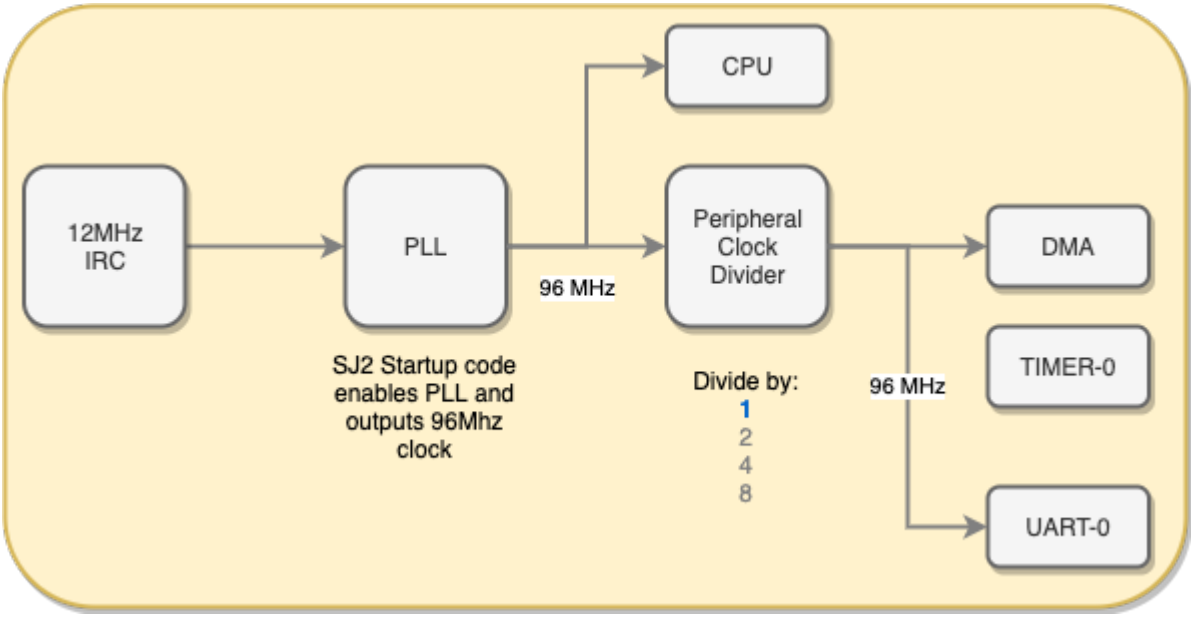


Figure 1. Clock system of LPC17xx

On the SJ2 board, the 12 Mhz is fed through "PLL" to multiply the frequency to 96 Mhz, which is routed to the core CPU, and to the peripheral clock divider. This single peripheral clock divider then gives you the capability to divide the clock before it goes out to the rest of the peripherals such as I2C, SPI, UART, ADC, PWM etc.



Phase Locked Loop (PLL)

What is a PLL?

A PLL is a control system that takes in a reference signal at a particular frequency and creates a higher frequency signal. Technically, they can also be used to make lower frequency signals, but a simple frequency divider could easily accomplish this and a frequency divider is simple hardware.

How do PLLs work?

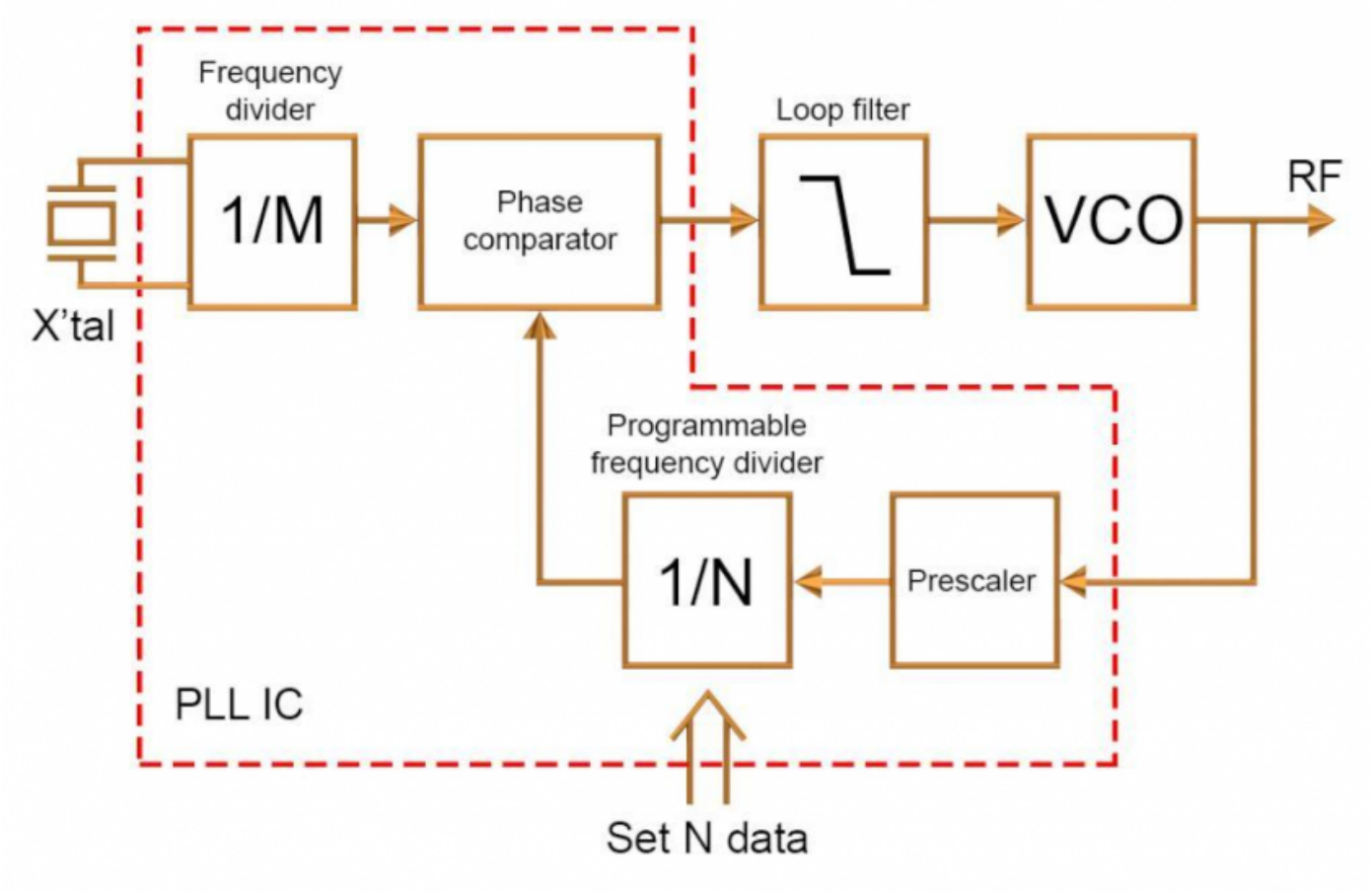


Figure 2. PLL System Diagram

Frequency Divider

If the input is a square wave, this device will reduce the number of edges per second proportional to **M**.

VCO (Voltage Controlled Output)

This a voltage to frequency converter. The higher the input voltage, the higher the output frequency will be.

Phase Comparator

This is used to check if the two frequencies match each other. This device checks for matches by taking two signals and comparing their phase's.

Loop Filter

This converts the pulse output from the Phase comparator to a DC voltage.

Programmable Divider

This is frequency divider that divide the frequency of the input signal by the number it is set to.

How does everything work together?

The Phase Comparator and loop filter will drive the voltage input of VCO up until it begins to see that both signals are synchronized (or locked) with each other. At this point, the PLL has created an output signal with the same frequency as the input reference signal.

By dividing the frequency that the Phase Comparator is trying to reach, lets say by 2, it will output a voltage twice the input reference signal, creating a higher frequency clock signal.

This is how we are able to use a 12 MHz clock and create a 48 MHz signal clock with it, by multiplying it by 4, or in this case, dividing the feedback frequency by 4.

Why use a PLL?

Crystals are extremely consistent frequency sources. PLLs are not very stable and need a control loop to keep them on track. So if a circuit to use crystals is very simple, why not simply just use a high frequency crystal oscillator to generate 100 MHz or more clock signals? There are a few reasons:

- High frequency crystals above 100 Mhz are not common and are hard to find.
- High frequency signals will be distorted due to:
  - Series inductance of board traces
  - Parallel capacitance due to board copper areas and fiber glass
  - Interference from other external signals like power signals and switching signals
- Signal distortion may cause the MCU to malfunction.

Once the signal is within the chip, the environment is a bit more controlled and higher frequencies can be achieved by using a PLL with a crystal as a reference.

Clock Frequency and Power Consumption

As you increase the clock of a microprocessors the power consumption of the processor will increase following this formula:

$$P = CV^2f$$

- **C** is the capacitance of the CPU (typical MOSFET gate transistors capacitance)
- **V** CPU core voltage
- **f** is the frequency used to drive the CPU and its peripherals

Given this information, we can figure out which options will decrease the power consumption of our CPU. We have a few options.

- **Reduce the capacitance of the CPU**
  - Which basically means purchasing a CPU or microcontroller with this characteristic. Typically such CPUs will be marked for lower power.
  - If you cannot change your CPU this is not feasible.
- **Reduce the CPU core voltage or supply voltage.**
  - Most micro-controllers perform the best at a particular supply voltage and lowering it, even towards the what the datasheet says is its minimum could be problematic.
- **Reduce the CPU and Perpheral frequency**
  - In most cases, this is the most practical option.
  - You can reduce the system clock frequency you use by manipulating the PLL's clock divider.
  - You can reduce the power consumption of peripherals by using a lower frequency peripheral clock.

Underclocking Advantages

- Reduced heat generation, which is exactly proportional to the power consumption.
- Longer hardware lifespan.
- Increased system stability.
- Increased battery life.

UART

Objective

The objective of this lesson is to understand UART, and use two boards and setup UART communication between them.

UART

UART stands for Unive

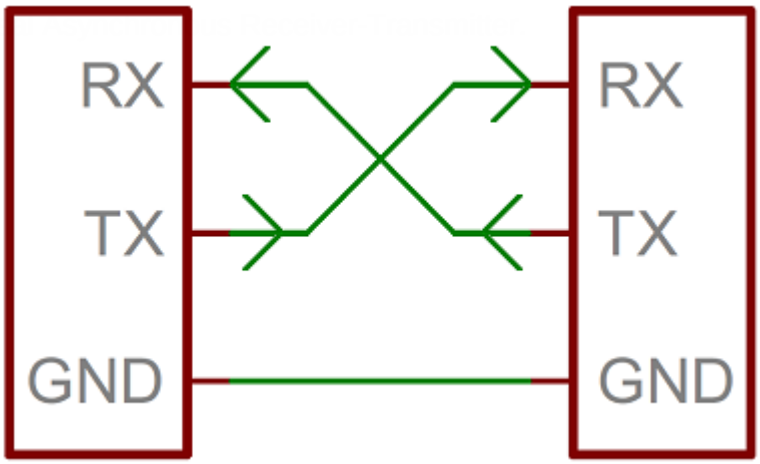


Figure 1. UART connection between two devices.

For **U**niversal **A**synchronous **R**eceiver **T**ransmitter. There is one wire for transmitting data (**TX**), and one wire to receive data (**RX**). It is asynchronous because there is no clock line between two UART hosts.

BAUD Rate

A common parameter is the baud rate known as "bps" which stands for **b**its **p**er **s**econd. If a transmitter is configured with 9600bps, then the receiver must be listening on the other end at the same speed.

Using the 9600bps example, each bit time is `1 / 9600` = `104uS`. That means that if a transmitter wants to transmit a byte, it must do so by latching one bit on the wire, and then waiting 104uS before another

bit is latched on the wire.

If you were to take a GPIO, and emulate UART at 9600 to send out a byte of data, it would look like this:

```
// Assumes GPIO is a memory that can set level of a Port/Pin (psuedocode)
void uart_send_at_9600bps(const char byte) {
  // 9600bps means each bit lasts on the wire for 104uS (approximately)
  GPIO = 0; delay_us(104); // Start bit is LOW
  // Check if bit0 is 1, then set the GPIO to HIGH, otherwise set it to LOW
  GPIO = (byte & (1 << 0)) ? 1 : 0; delay_us(104); // Use conditional statement
  GPIO = (bool) (byte & (1 << 1)); delay_us(104);  // Case to bool
  GPIO = (byte & (1 << 2)); delay_us(104);
  GPIO = (byte & (1 << 3)); delay_us(104);

  GPIO = (byte & (1 << 4)); delay_us(104);
  GPIO = (byte & (1 << 5)); delay_us(104);
  GPIO = (byte & (1 << 6)); delay_us(104);
  GPIO = (byte & (1 << 7)); delay_us(104);
  GPIO = 1; delay_us(104); // STOP bit is HIGH}
```

UART Frame

UART is a serial communication, so bits must travel on a single wire. If you wish to send a 8-bit byte `(uint8_t)` over UART, the byte is enclosed within a **start** and a **stop** bit. Therefore, to transmit a byte, it would require 2-bits of overhead; this 10-bit of information is called a **UART frame**. Let's take a look at how the character `'A'` is sent over UART. In ASCII table, the character `'A'` has the value of `65`, which in binary is: `0100_0001`. If you inform your UART hardware that you wish to send this data at 9600bps, here is how the frame would appear on an oscilloscope :

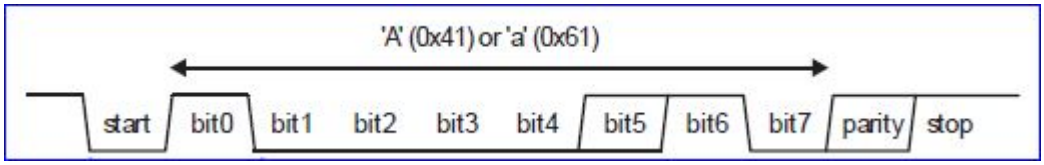


Figure 2. UART Frame sending letter `'A'`

UART Ports

It would normally not make sense to use the main processor (such as NXP LPC40xx) to send data on a wire one bit at a time, thus there are peripherals, or UART co-processor whose job is to solely send and receive data on UART pins without having to tax the main processor.

A micrcontroller can have multiple UART peripherals. Typically, the UART0 peripheral is interfaced to with a USB to serial port converter which allows users to communicate between the computer and microcontroller. This port is used to program your microcontroller.

Benefits

- Hardware complexity is low.
- No clock signal needed
- Has a parity bit to allow for error checking
- As this is one to one connection between two devices, device addressing is not required.

Drawbacks

- The size of the data frame is limited to a maximum of 8 bits (some micros may support non-standard data bits)
- Doesn’t support multiple slave or multiple master systems
- The baud rates of each UART must be within ~3% (or lower, depending on device tolerance) of each other

Hardware Design

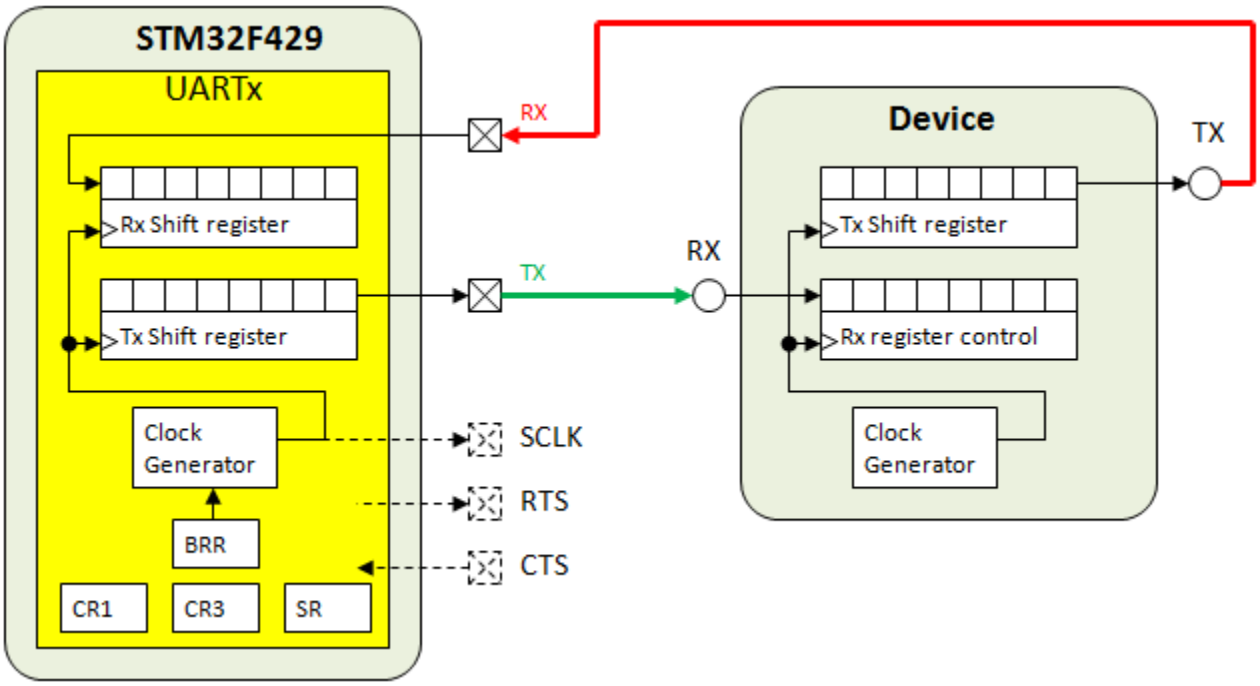


Figure 3. Simplified UART peripheral design for the *STM32F429*. SCLK is used for USART.

> WARNING: The above is missing a common ground connection

Software Driver

The UART chapter on LPC40xx has a really good summary page on how to write a UART driver.
Read the register description of each UART register to understand how to write a driver.

Memory Shadowing in UART driver?

Table 270. UART0/2/3 Register Map

Generic Name	Description	Access	Reset value[1]	UARTn Register Name & Address
RBR (DLAB =0)	Receiver Buffer Register. Contains the next received character to be read.	RO	NA	U0RBR - 0x4000 C000 U2RBR - 0x4009 8000 U3RBR - 0x4009 C000
THR (DLAB =0)	Transmit Holding Register. The next character to be transmitted is written here.	WO	NA	U0THR - 0x4000 C000 U2THR - 0x4009 8000 U3THR - 0x4009 C000
DLL (DLAB =1)	Divisor Latch LSB. Least significant byte of the baud rate divisor value. The full divisor is used to generate a baud rate from the fractional rate divider.	R/W	0x01	U0DLL - 0x4000 C000 U2DLL - 0x4009 8000 U3DLL - 0x4009 C000
DLM (DLAB =1)	Divisor Latch MSB. Most significant byte of the baud rate divisor value. The full divisor is used to generate a baud rate from the fractional rate divider.	R/W	0x00	U0DLM - 0x4000 C004 U2DLM - 0x4009 8004 U3DLM - 0x4009 C004
IER (DLAB =0)	Interrupt Enable Register. Contains individual interrupt enable bits for the 7 potential UART interrupts.	R/W	0x00	U0IER - 0x4000 C004 U2IER - 0x4009 8004 U3IER - 0x4009 C004

Figure 4. Memory Shadowing using DLAB Bit Register

In figure 4, you will see that registers **RBR**/**THR** and **DLL** have the same address **0x4000C000**. These registers are shadowed using the DLAB control bit. Setting DLAB bit to 1 allows the user to manipulate **DLL** and **DLM**, and clearing DLAB to 0 will allow you to manipulate the **THR** and **RBR** registers.

The reason that the DLL register shares the same memory address as the RBR/THR may be historic. My guess is that it was intentionally hidden such that a user cannot accidentally modify the DLL register. Even if this case is not very significant present day, the manufacturer is probably using the same UART verilog code from many decades ago.

Control Space Divergence (CSD) in UART driver

In figure 4, you will see that register **RBR** and **THR** have the same address **0x4000C000**. But also notice that access to each respective register is only from read or write operations. For example, if you read from memory location **0x4000C000,** you will get the information from receive buffer and if you write to memory location **0x4000C000**, you will write to a separate register which the transmit holding register. We call this ***Control Space Divergence*** since access of two separate registers or devices is done on a single address using the read/write control signal is used to multiplex between them. That address is considered to be **C*ontrol Space Divergent***. Typically, the control space aligns with its respective memory or io space.

> Note that ***Control Space Divergence*** does not have a name outside of this course. It is Khalil Estell's phrase for this phenomenon.

BAUD Rate Formula

$$UARTn_{baudrate} = \frac{PCLK}{16 \times (256 \times UnDLM + UnDLL) \times \left(1 + \frac{DivAddVal}{MulVal}\right)}$$

Figure 5. Baud rate formula

To set the baud rate you will need to manipulate the **DLM** and **DLL** registers. Notice the **256*UnDLM** in the equation. That is merely another way to write the following (DLM << 8). Shifting a number is akin to multiplying it by 2 to the power of the number of shifts. DLM and DLL are the lower and higher 8-bits of a 16 bit number that divides the UART baudrate clock. DivAddVal and MulVal are used to fine tune the BAUD rate, but for this class, you can simply get "close enough" and ignore these values. Take these into consideration when you need an extremely close baudrate.

?

Advanced Design

If you used 9600bps, and sent 1000 characters, your processor would basically enter a "busy-wait" loop and spend 1040ms to send 1000 bytes of data. You can enhance this behavior by allowing your uart send function to enter data to a queue, and return immediately, and you can use the **THRE** or "Transmitter Holding Register Empty" interrupt indicator to remove your busy-wait loop while you wait for a character to be sent.

Queues

Moved to here

Lab: UART

Objective

- To learn how to communicate between two devices using UART.
- **Reinforce** interrupts by setting up an interrupt on receive (Part 2)
- **Reinforce** RTOS queues
- It is required to **finish** `Part 0` **and** `Part 1` prior to your lab's start time

Assignment

This assignment will require a partner except for Part 0 and Part 1 . The overall idea is to interface two boards using your UART driver. It may be best to test a single UART driver using loopback (tie your own RX and TX wires together) in order to ensure that your driver is functional before trying to put two boards together.

Part 0: Implement UART driver

- Before you start the assignment, please read **Chapter 18: UART0/2/3** in your LPC User manual (`UM10562.pdf`). You can skip the sections related to FIFO, interrupts or the DMA.
  - From LPC User manual to understand the different registers involved in writing a UART driver
  - Refer `Table 84` from LPC User manual for pin configuration
- From the schematics pdf(`Schematics-RevE.2.pdf`), identify the pins numbers which can do UART2 (U2_TXD/U2_RXD) and UART3(U3_TXD/U3_RXD) and make a note of it because you will be needing them for pin function configuration

Implement `uart_lab.h` and `uart_lab.c`:

```
#pragma once
#include <stdint.h>
#include <stdbool.h>
typedef enum {
  UART_2,
  UART_3,
} uart_number_e;
void uart_lab__init(uart_number_e uart, uint32_t peripheral_clock, uint32_t baud_rate) {
  // Refer to LPC User manual and setup the register bits correctly
  // The first page of the UART chapter has good instructions
  // a) Power on Peripheral
  // b) Setup DLL, DLM, FDR, LCR registers
}
// Read the byte from RBR and actually save it to the pointer
bool uart_lab__polled_get(uart_number_e uart, char *input_byte) {
  // a) Check LSR for Receive Data Ready
  // b) Copy data from RBR register to input_byte
}
bool uart_lab__polled_put(uart_number_e uart, char output_byte) {
  // a) Check LSR for Transmit Hold Register Empty
  // b) Copy output_byte to THR register}
```

The divider equation in the LPC user manual is a bit confusing, please reference the code below to figure out how to set the dividers to achieve your baud rate.

```
    /* Baud rate equation from LPC user manual:
     * Baud = PCLK / (16 * (DLM*256 + DLL) * (1 + DIVADD/DIVMUL))
     *
     * What if we eliminate some unknowns to simplify?
     * Baud = PCLK / (16 * (DLM*256 + DLL) * (1 + 0/1))
     * Baud = PCLK / (16 * (DLM*256 + DLL)
     *
     * | DLM | DLL | is nothing but 16-bit number
```

```
     * DLM multiplied by 256 is just (DLM << 8)
     *
     * The equation is actually:
     * Baud = PCLK / 16 * (divider_16_bit)
     */
    const uint16_t divider_16_bit = 96*1000*1000 / (16 * baud_rate);
    LPC_UART2->DLM = (divider_16_bit >> 8) & 0xFF;
    LPC_UART2->DLL = (divider_16_bit >> 0) & 0xFF;
```

Part 1: Loopback test of **UART** driver

1. Choose either UART2 or UART3. Connect TX with RX pin.
2. Implement two tasks which reads whatever you just write and test if everything works successfully.
3. Note that you will use the "polled" version of the driver that will consume the CPU while waiting for data to be received. Once you get this working, the next part will utilize interrupt driven approach.

```
#include "FreeRTOS.h"
#include "task.h"
#include "uart_lab.h"
void uart_read_task(void *p) {
  while (1) {
    //TODO: Use uart_lab__polled_get() function and printf the received value
    vTaskDelay(500);
  }
}
void uart_write_task(void *p) {
  while (1) {
    //TODO: Use uart_lab__polled_put() function and send a value
    vTaskDelay(500);
  }
}
void main(void) {
  //TODO: Use uart_lab__init() function and initialize UART2 or UART3 (your choice)
  //TODO: Pin Configure IO pins to perform UART2/UART3 function
```

```
  xTaskCreate(uart_read_task, ...);
  xTaskCreate(uart_write_task, ...);
  vTaskStartScheduler();}
```

Part 2: Receive with Interrupts

Instead of polling for data to be received, you will extend your UART driver to trigger an interrupt when there is data available to be read.

```
// file: uart_lab.c
// TODO: Implement the header file for exposing public functions (non static)
// The idea is that you will use interrupts to input data to FreeRTOS queue
// Then, instead of polling, your tasks can sleep on data that we can read from the queue
#include "FreeRTOS.h"
#include "queue.h"
// Private queue handle of our uart_lab.c
static QueueHandle_t your_uart_rx_queue;
// Private function of our uart_lab.c
static void your_receive_interrupt(void) {
  // TODO: Read the IIR register to figure out why you got interrupted

  // TODO: Based on IIR status, read the LSR register to confirm if there is data to be read

  // TODO: Based on LSR status, read the RBR register and input the data to the RX Queue
  const char byte = UART->RBR;
  xQueueSendFromISR(your_uart_rx_queue, &byte, NULL);
}
// Public function to enable UART interrupt
// TODO Declare this at the header file
void uart__enable_receive_interrupt(uart_number_e uart_number) {
  // TODO: Use lpc_peripherals.h to attach your interrupt
```

```
  lpc_peripheral__enable_interrupt(..., your_receive_interrupt);
  // TODO: Enable UART receive interrupt by reading the LPC User manual
  // Hint: Read about the IER register

  // TODO: Create your RX queue
  your_uart_rx_queue = xQueueCreate(...);
}
// Public function to get a char from the queue (this function should work without modification)
// TODO: Declare this at the header file
bool uart_lab__get_char_from_queue(char *input_byte, uint32_t timeout) {
  return xQueueReceive(your_uart_rx_queue, input_byte, timeout);}
```

Part 3: Interface two SJ2 boards with UART

- After all the above parts are completed and successfully tested, you are now ready to establish communication between 2 SJ2 boards.
- Assign one board as **Sender** and another as **Receiver**. Connect Tx pin of Sender to Rx pin of Receiver and Rx pin of Sender to Tx pin of Receiver. Do not forget the common ground.
- Have one board send a random number, one char at a time over UART
  - See reference code below; most of the code is setup for you at `board_1_sender_task()`
- Have the other board re-assemble this number
  - You will have to figure out some of the logic yourself

```
#include <stdlib.h>

// This task is done for you, but you should understand what this code is doing
void board_1_sender_task(void *p) {
  char number_as_string[16] = { 0 };

   while (true) {
     const int number = rand();
     sprintf(number_as_string, "%i", number);
```

```
    // Send one char at a time to the other board including terminating NULL char
    for (int i = 0; i <= strlen(number_as_string); i++) {
      uart_lab__polled_put(number_as_string[i]);
      printf("Sent: %c\n", number_as_string[i]);
    }

    printf("Sent: %i over UART to the other board\n", number);
    vTaskDelay(3000);
  }
}
void board_2_receiver_task(void *p) {
  char number_as_string[16] = { 0 };
  int counter = 0;
  while (true) {
    char byte = 0;
    uart_lab__get_char_from_queue(&byte, portMAX_DELAY);
    printf("Received: %c\n", byte);

    // This is the last char, so print the number
    if ('\0' == byte) {
      number_as_string[counter] = '\0';
      counter = 0;
      printf("Received this number from the other board: %s\n", number_as_string);
    }
    // We have not yet received the NULL '\0' char, so buffer the data
    else {
      // TODO: Store data to number_as_string[] array one char at a time
      // Hint: Use counter as an index, and increment it as long as we do not reach max value of 16
    }
  }}
```

? Make sure to connect the ground pins of both the boards together. Otherwise you will see scrambled characters.

Conclusion

- Submit all relevant files and files used (`uart_lab.h` and `uart_lab.c`)
  - Do not submit dead or commented code
  - Do not submit code you did not write
- Turn in any the screenshots of terminal output
- Logic Analyzer Screenshots: Waveform of UART transmission (or receive) between two boards
  - Make sure your logic analyzer is configured to decode UART protocol
  - Whole window screenshot with the **Decoded Protocols** (lower right hand side of window) clearly legible.