Prof. Dr. Marco Zimmerling Max Granzow Jonas Kubicki
https://nes-lab.org/

Networked and Low-Power Embedded Systems

WS 2025/26

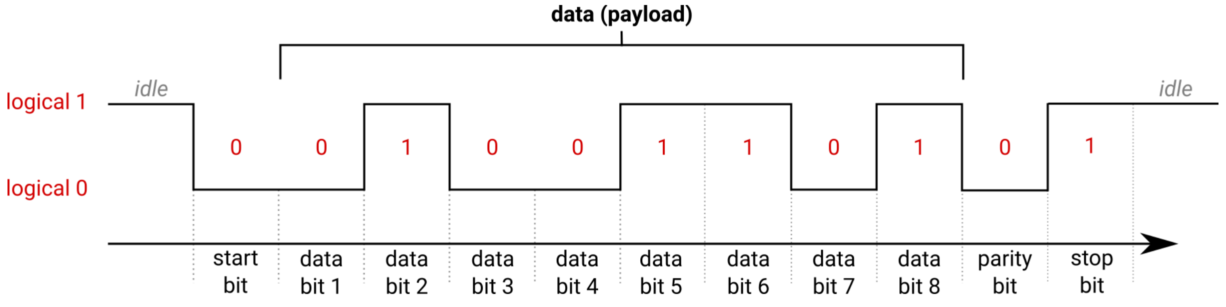
Lab 3: UART

Introduction

In today's lab, we will use the Universal Asynchronous Receiver-Transmitter (UART) peripheral to output information generated on the microcontroller to the terminal on our PC. UART is a widely used communication protocol for serial communication between two devices, such as, microcontrollers, sensors, and peripherals. It enables data transfer by sequentially sending bits over a communication line without a shared clock signal. UART communication is simple and basically involves only two pins: one for transmitting data (TX) and one for receiving data (RX).

Properties of the UART protocol:

- *Asynchronous Communication*: UART is asynchronous, meaning that data is transmitted without a shared clock signal between the sender and receiver. Instead, both devices must agree on specific parameters for successful communication.
- *Data Format*: Each UART frame includes one start bit, a variable number of data bits (typically 8 bits), an optional parity bit for error checking, and one or more stop bits. The start bit indicates the beginning and serves as a time reference, while the stop bit(s) signal the end. In the idle state, the UART signals are at logical level 1.



- *Full-Duplex Communication*: UART enables full-duplex communication, meaning that data can be transmitted and received simultaneously due to the separate TX and RX lines.

1 Configuring the UART Peripheral

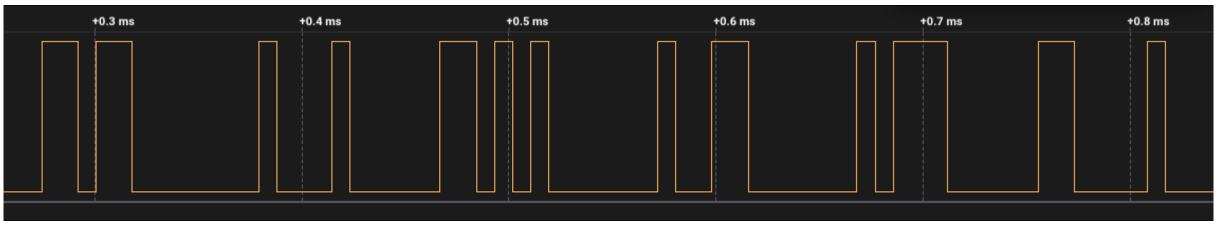
We start by configuring the UART peripheral on the nRF52 chip. Download the source code from Moodle and follow the steps below to complete the code in the file `task1.c`. Remember the steps from lab 1 to compile and flash your program. Compile the source code with `$ make task1`.

1. The UART peripheral is described on page 502 ff. in the datasheet. For now, we only need to look at the registers in Section 6.33.10. In this lab we only want to *transmit* data via UART from the nRF52 chip. Therefore, it is sufficient to configure the `PSEL.TXD` register to establish the connection between our TX signal (i.e. the data we want to transmit) and a GPIO pin. Theoretically, you can use any unused GPIO pin, but in our code example we use pin `P0.06`. The reason for this particular pin will be explained later.
2. Configure the `BAUDRATE` register to a baud rate of 115 200.
3. Specify the UART frame structure in the `CONFIG` register. We can disable hardware flow control. This mechanism is used to automatically coordinate the transmitter and receiver so that the transmitter does not overload the receiver with data, which can happen if the receiver cannot process the data fast enough. Use no parity bit and one stop bit.
4. In the previous steps, we established the connection of the UART TX signal to a specific GPIO pin (`P0.06`). As you have seen in Lab 2, GPIO pins can be used in different ways. To correctly configure the pin for use with the UART peripheral, read the description in Section 6.33.2 of the datasheet. Find out the correct pin direction and output value and configure the GPIO pin. You can also lookup Lab 2 on how to configure the GPIO peripheral.
5. Enable the UART peripheral, see register `ENABLE`.
6. Within the while loop, continuously transmit the prepared data via UART. Read the description in Section 6.33.4 in the datasheet on how transmissions work and pay attention to the comments in the source code.

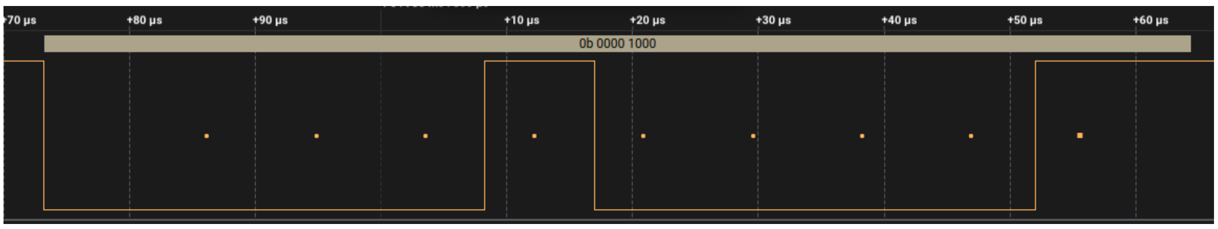
2 Analyzing the UART Data Stream

We will now see if everything has been configured correctly by inspected the TX pin with the LA.

1. Connect the LA to `GND` and `P0.06/TX` (see the labels on the development kit).
2. Start the `Logic 2` software, wait until the LA is detected, and start recording. You should now see something like this:



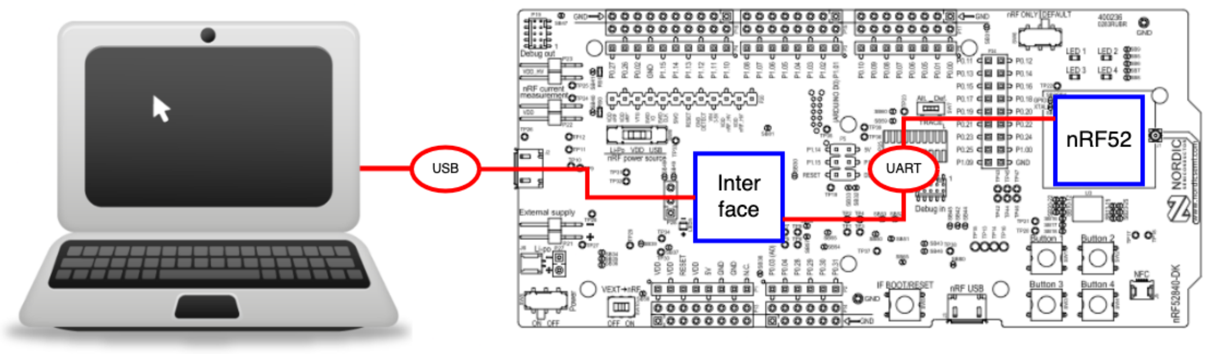
3. `Logic 2` has a feature to decode the UART stream. Check the menu on the right and click “Analyzers”. Select “Async Serial” and make sure the configuration matches our UART configuration (for most settings, the standard values are already correct). Correctly configured, the analyzer view should look like this:



   The long bar at the top visualizes the beginning (the falling edge indicates the start bit) and the end (stop bit) of the UART frame. The transmitted data is displayed within the bar. You can switch between different representations of this data by right-clicking on the bar and choosing between binary, decimal, hexadecimal, and ASCII representation. The sampling points are displayed with small circles and the parity bit at the end with a small square. Switch to the binary representation and compare the transmitted data (within the long bar) with the actual signal. What do you notice? Switch to the ASCII representation, zoom out a little and check if you can find the alphabet.

3 Displaying the Microcontroller’s Output on the PC

When you write C programs on your PC, you typically use `printf` from C’s standard IO library to print information on your terminal. Internally, `printf` writes the output data by default to the stdout file stream, which is displayed in your terminal. In our bare metal embedded microcontroller world, there is neither a stdout file stream nor a file system, so the output from `printf` does not go anywhere. Your task is to change the default behavior of `printf` and send the output data to the PC via UART.



To do this, we need to forward the UART stream to the PC via the USB connection. Fortunately, the interface chip on the development kit already takes care of the USB communication for us and forwards all received UART data. The interface chip is configured to listen for UART input on specific pins. This is the reason why we use `P0.06` as the GPIO pin for the TX signal on the nRF52 in task 1. Complete the steps below and compile the source code with `$ make task3`.

1. The TIMER and UART peripheral are already setup in the code. The timer triggers an interrupt every second and calls the interrupt service routine `TIMER0_IRQHandler()`. You have to output some data via `printf` in this ISR.

2. Part of our source code is a tiny `printf` implementation (`printf.c`), which is especially useful for resource-constrained embedded devices. To redirect the output stream of our `printf` function, we need to implement the function `_putchar(char)`, which is called internally by `printf` for each character of the output data. Implement `_putchar(char)` and transmit each character via UART. Pay attention to the comments in the source code.

3. First connect the LA and check that the correct data is output every second.

4. To retrieve the data via USB in your terminal, we need a program to monitor the serial port at which the devlopment kit registered on your PC. You can use any program that can read the serial output. The `pyserial` package provides such a functionality. Optionally activate a virtual environment and type the following.

   Linux:

   ```
   # Create virtual environment in the `venv` folder
   $ python3 -m venv venv
   # Activate the virtual environment
   $ source venv/bin/activate
   # Install pyserial
   (venv) $ python3 -m pip install pyserial
   # List the ports of connected devices, for example:
   (venv) $ python3 -m serial.tools.list_ports
   /dev/ttyACM0
   # Connect to the serial port.
   (venv) $ python3 -m serial.tools.miniterm /dev/ttyACM0 115200
   ```

   Windows:

   ```
   # Install pyserial
   $ python3 -m pip install pyserial
   # List the ports of connected devices, for example:
   $ python3 -m serial.tools.list_ports
   COM3
   # Connect to the serial port.
   $ python3 -m serial.tools.miniterm COM3 115200
   ```

   You should now see the output of your device in your terminal. To close the program type “`Ctrl + ]`”.

   In case you are missing access rights on Linux (e.g. because your user is not part of the dialout group), use sudo.

> If you cannot close the miniterm program with “`Ctrl + ]`”, use “`Ctrl + T`” followed by “Q”, or set the exit character to “Esc” by adding the `--exit-char 27` flag to the command.