Chapter 11

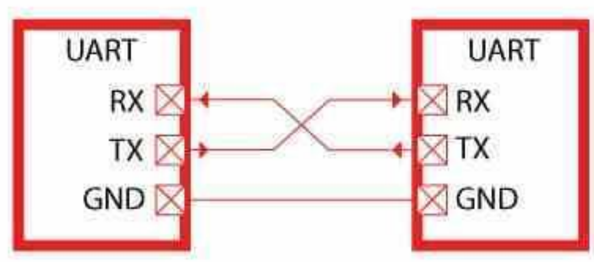
USART

USART

- Another peripheral in our microcontroller
- USART => Universal Synchronous/Asynchronous Receive Transmit
- This peripheral can work with many protocols (e.g. serial communication protocol, etc)
- In last chapter we demonstrated the interaction between debugging host/laptop and serial module
- In this chapter we will use serial communication between MCU and laptop (With the help of USART peripheral and our module)

USART Connections

- Working for this communication we have to make some connections
- We discussed in the last chapter that this protocol (i.e. serial communication protocol) involves 2 data lines (i.e. **Tx** and **Rx**)
- **Tx** => Transmitter , **Rx** => Receiver
- Transmitter and receiver are relative terms

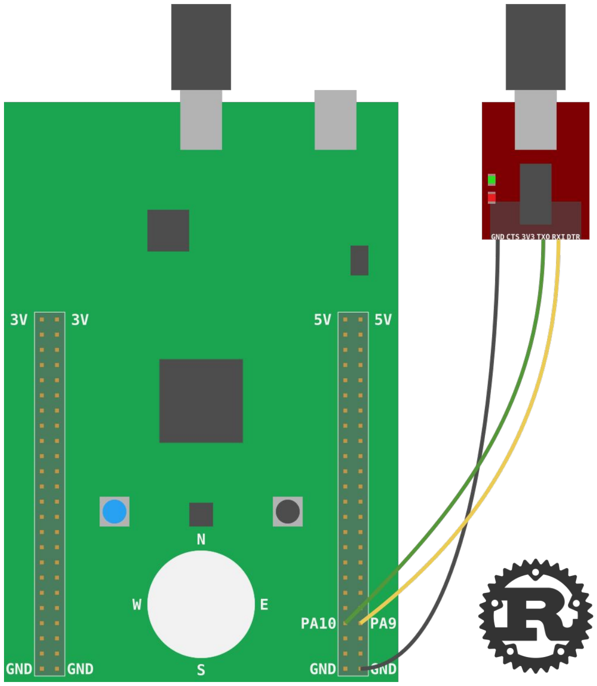


USART Connections

- We’ll be using **PA9** as MCU’s **Tx** line (or data output pin)
- And **PA10** as MCU’s **Rx** line (or data input pin)
- However we can use a different pair of Tx and Rx pin from MCU for communication, for the possibilities we’ll refer to the manual
- Manual tells us we can also use **PA2**, **PA3** and **PA14**,**PA15** as **Tx** line and **Rx** line respectively

USART Connections

- Our serial module also has **Tx** and **Rx** pin
- For making this communication possible we have to **cross these two** devices
- We'll **connect MCU's Tx pin to module's Rx pin**
- And **MCU's Rx pin to module's Tx pin**



USART Connections Steps

Recommended sequence of steps for connections:

- Close OpenOCD and itmdump
- Disconnect the USB cables from the F3 and the serial module.
- Connect one of F3 GND pins to the GND pin of the serial module using a female to male (F/M) wire. Preferably, a black one.
- Connect the PA9 pin on the back of the F3 to the RXI pin of the serial module using a F/M wire.

USART Connections Steps

- Connect the PA10 pin on the back of the F3 to the TXO pin of the serial module using a F/M wire.
- Now connect the USB cable to the F3.
- Finally connect the USB cable to the Serial module.
- Re-launch OpenOCD and itmdump

Everything is done now let's try sending data

Sending A Single Byte

Sending A Single Byte

After successful connections our first task is to send a single test byte from MCU to debugging host/laptop using **serial channel**

- As we know, for this process we are using **USART peripheral** and for convenience it is **initialized already** in the library
- Now we need to focus the core logic and the **registers** that are responsible for this communication

Sending A Single Byte

Here is starter code

```rust
#![deny(unsafe_code)]
#![no_main]
#![no_std]

#[allow(unused_imports)]
use aux11::{entry, iprint, iprintln};

#[entry]
fn main() -> ! {
    let (usart1, mono_timer, itm) = aux11::init();

    // Send a single character
    usart1.tdr.write(|w| w.tdr().bits(u16::from(b'X')));

    loop {}
}
```

Sending A Single Byte

- This program writes to **tdr** register
- Which will cause USART peripheral to send a byte to serial interface
- Upon successfully running this program you'll see on the laptop's minicom console a **letter X** will appear

Sending A String

Sending A String

This time we'll be sending a complete string rather than a byte

```
#![deny(unsafe_code)]
#![no_main]
#![no_std]

#[allow(unused_imports)]
use aux11::{entry, iprint, iprintln};

#[entry]
fn main() -> ! {
    let (usart1, mono_timer, itm) = aux11::init();

    // Send a string
    for byte in b"The quick brown fox jumps over the lazy dog.".iter() {
        usart1.tdr.write(|w| w.tdr().bits(u16::from(*byte)));
    }

    loop {}
}
```

Sending A String

Upon running this program in **debug mode** uninterrupted

```
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Aug 13 2017, 15:25:34.
Port /dev/ttyUSB0, 11:12:40

Press CTRL-A Z for help on special keys

Thequc bon o jms ve helzydg.
```

Sending A String

Upon running this program in **release mode** uninterrupted

```
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Aug 13 2017, 15:25:34.
Port /dev/ttyUSB0, 11:12:40

Press CTRL-A Z for help on special keys

T
```

Buffer Overrun

Buffer Overrun

What’s the problem?

- Actually time taken by bytes transmission is more than the time processor takes for execution of the program
- Let's do some calculations
- With a common configuration of **1-start** bit, **8-data** bits and **1-stop** bit and a **baud rate of 115200** bps
- For calculating baud rate in frames/S ;
  115200/10-bits => **11520 frames/S** (11.52K frames/S)

Buffer Overrun

If we have 23,040 bytes to send then time taken to send them is:

- 23,040 bytes / (11,520 frames/S) => **2S**
- We have 45 bytes so ; 45 bytes/(11,520 frames/S) => 0.00390625S or (3,906 uS)
- So ideally 3,906uS is the time required to send our string
- Now let's see how much time our program take to execute

Buffer Overrun

We will change our starter code to

```rust
#![deny(unsafe_code)]
#![no_main]
#![no_std]

#[allow(unused_imports)]
use aux11::{entry, iprint, iprintln};

#[entry]
fn main() -> ! {
    let (usart1, mono_timer, mut itm) = aux11::init();

    let instant = mono_timer.now();
    // Send a string
    for byte in b"The quick brown fox jumps over the lazy dog.".iter() {
        usart1.tdr.write(|w| w.tdr().bits(u16::from(*byte)));
    }
    let elapsed = instant.elapsed(); // in ticks

    iprintln!(
        &mut itm.stim[0],
        "`for` loop took {} ticks ({} us)",
        elapsed,
        elapsed as f32 / mono_timer.frequency().0 as f32 * 1e6
    );

    loop {}
}
```

Buffer Overrun

Above program will calculate the time taken by processor to execute it

In debug mode the output on itm console is:

```
rajabraza@EliteBook-Folio-9470m:/tmp$ touch itm.txt && itmdump -F -f itm.txt
`for` loop took 18909 ticks (2363.625 us)
```

In release mode the output on itm console is:

```
rajabraza@EliteBook-Folio-9470m:/tmp$ touch itm.txt && itmdump -F -f itm.txt
`for` loop took 91 ticks (11.375 us)
```

Buffer Overrun Solution

Buffer Overrun Solution

Solution to this problem is

- We will be using register ISR
- ISR is a status register in **USART peripheral** like TDR register
- ISR has a **flag TXE** that indicates whether it is safe to write data on TDR register or not
- We will use this flag to actually **halt our program** unless it is safe to write
- We will modify the existing code like

Buffer Overrun Solution

```rust
#![deny(unsafe_code)]
#![no_main]
#![no_std]

#[allow(unused_imports)]
use aux11::{entry, iprint, iprintln};

#[entry]
fn main() -> ! {
    let (usart1, mono_timer, mut itm) = aux11::init();

    let instant = mono_timer.now();
    // Send a string
    for byte in b"The quick brown fox jumps over the lazy dog.".iter() {
        // wait until it's safe to write to TDR
        while usart1.isr.read().txe().bit_is_clear() {} // <- NEW!

        usart1.tdr.write(|w| w.tdr().bits(u16::from(*byte)));
    }
    let elapsed = instant.elapsed(); // in ticks

    iprintln!(
        &mut itm.stim[0],
        "`for` loop took {} ticks ({} us)",
        elapsed,
        elapsed as f32 / mono_timer.frequency().0 as f32 * 1e6
    );

    loop {}
}
```

Buffer Overrun Solution

Now after this modification, when we ran program in debug mode

```
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Aug 13 2017, 15:25:34.
Port /dev/ttyUSB0, 15:50:42

Press CTRL-A Z for help on special keys

The quick brown fox jumps over the lazy dog.
```

And on itm console

```
rajabraza@EliteBook-Folio-9470m:/tmp$ touch itm.txt && itmdump -F -f itm.txt
`for` loop took 30384 ticks (3798 us)
```

Buffer Overrun Solution

Now after this modification, when we ran program in release mode

```
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Aug 13 2017, 15:25:34.
Port /dev/ttyUSB0, 15:50:42

Press CTRL-A Z for help on special keys

The quick brown fox jumps over the lazy dog.
```

And on itm console

```
rajabraza@EliteBook-Folio-9470m:/tmp$ touch itm.txt && itmdump -F -f itm.txt
`for` loop took 29690 ticks (3711.25 us)
```

Receive A Single Byte

Receive A Single Byte

So far we have sent data from the MCU to laptop. Now let's try receiving data from laptop.

- This time we will be using RDR
- RDR register receives data from the senders and filled up the Rx line
- Once the Rx line is filled by the other side of the channel we will retrieve data from the board
- Here is again ISR register comes and tell us whenever some new data sent by any connected device

Receive A Single Byte

- ISR register has another flag RXNE, which keeps a check on and whenever some new data comes in it intimates

```rust
#![deny(unsafe_code)]
#![no_main]
#![no_std]

#[allow(unused_imports)]
use aux11::{entry, iprint, iprintln};

#[entry]
fn main() -> ! {
    let (usart1, mono_timer, itm) = aux11::init();

    loop {
        // Wait until there's data available
        while usart1.isr.read().rxne().bit_is_clear() {}

        // Retrieve the data
        let _byte = usart1.rdr.read().rdr().bits() as u8;

        aux11::bkpt();
    }
}
```

Summary