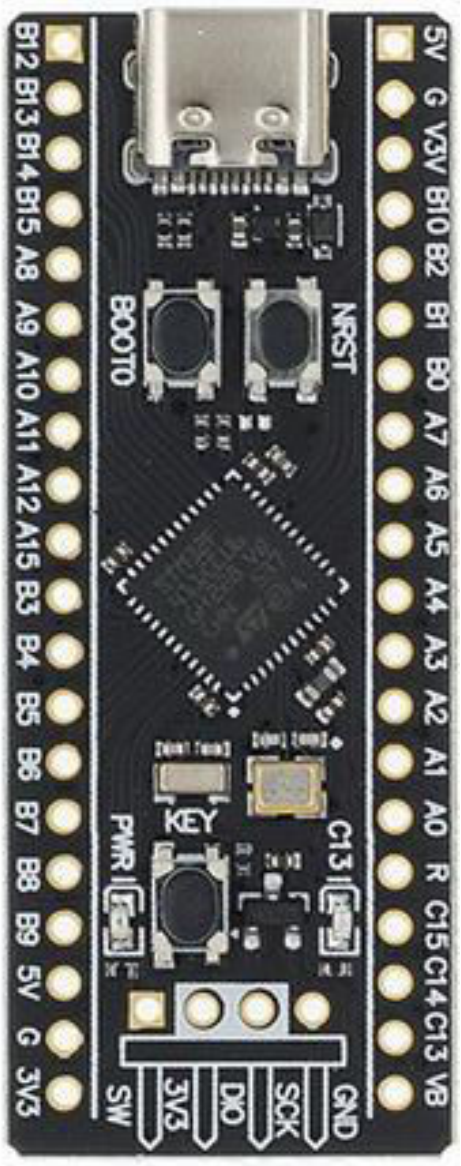
Software Serial on the STM32F411CEU6



Marijn Verschuren

23-06-2023

1. Table of Contents

2. Table of Figures

3. Introduction

The purpose of this project is to create a software serial library on the STM32F411CEU6 that can be customized with a wide range of settings for flexibility. The goal is to create code that can handle as high of a baud rate as possible and stay reliable. As an extra a terminal app is built on top of this library as a demo, this will not be show in this document but code for it can be seen in the repository (https://github.com/MarijnVerschuren/Software_Serial).

4. UART

UART (Universal Asynchronous Receiver Transmitter) is a simple full duplex protocol which is used in almost all embedded applications. As evident by the name the protocol is asynchronous, this means that both parties have to do their own time keeping. This makes UART a little bit less reliable and robust it does make it very simple to connect because UART uses only one wire (if transmission only goes one way) and a common ground of course. The only issue is that you cant just plug into a UART bus without knowing what baud rate is used because it has to be the same on both devices so that the bits are read or sent at the right time.

UART uses data frames to send its data (see Figure 1). These frames can take on different forms depending on the settings like: the amount of data/stop bits or the presence of a parity bit. This parity bit can be used to verify if the data is received correctly.

Figure 1. Data frame

analog.com/-/media/images/analog-dialogue/en/volume-54/number-4/articles/uart-a-hardware-communication-protocol/335962-fig-10.svg?w=900&imgver=1. (n.d.). Retrieved June 22, 2023

These frames are then sent bit for bit at the baud rate this is shown in Figure 2. The formula for the time between bits (in seconds) is the following: 1/(baud + 1). Note that TX is connected to RX (and vice versa).

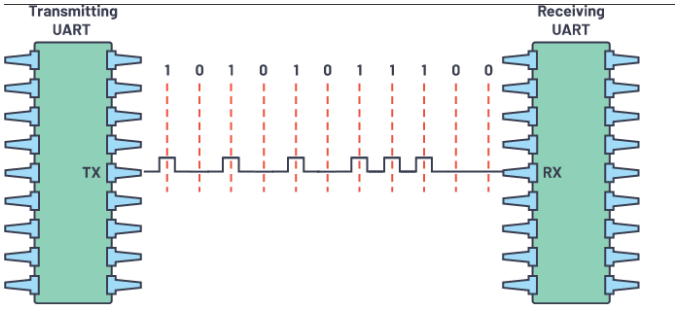


Figure 2. Transmission example

analog.com/-/media/images/analog-dialogue/en/volume-54/number-4/articles/uart-a-hardware-communication-protocol/335962-fig-10.svg?w=900&imgver=1. (n.d.). Retrieved June 22, 2023

This is a timing diagram of a UART transmission with even parity and two stop bits. Note that when sending the character ‘W‘ (0x57) 0xEA is sent this because all bits are flipped when sending data over UART (the first bit ends up at bit position 0 etc...).

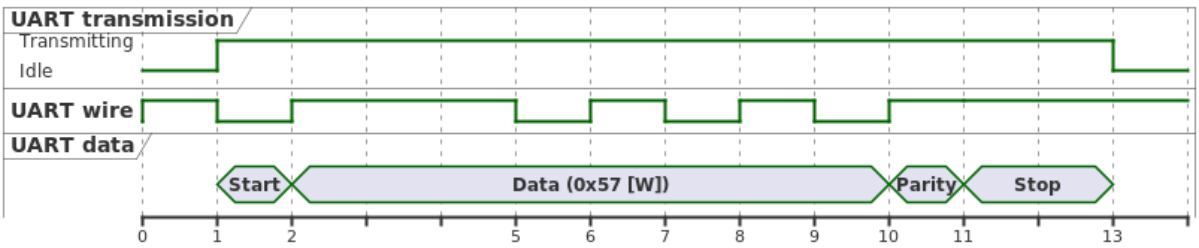


Figure 3. Example of a transmission with 8 data bits 1 parity bit and 2 stop bits.

5. Implementation

The RX code will follow the behavior seen in Figure 4. The only difference is that when an error is detected the code will simply continue whilst setting an error flag instead of ending early because there may be a possibility to repair the data (this is up to the user).

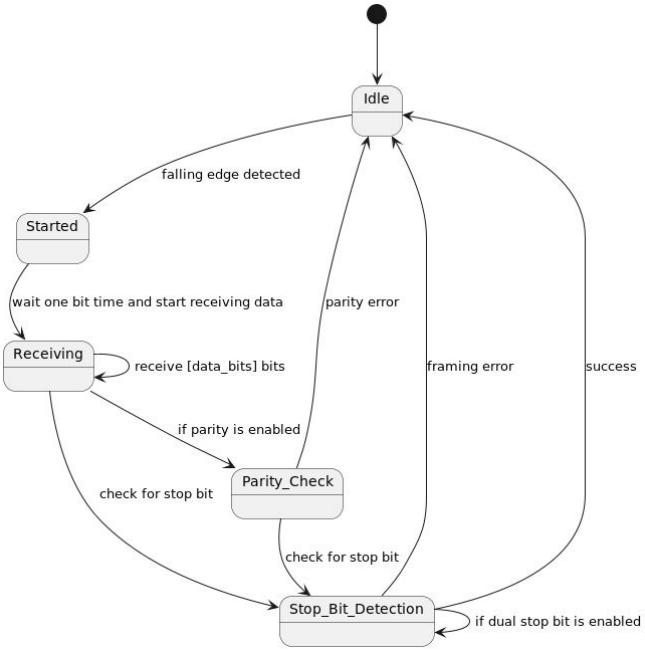


Figure 4. RX state diagram

This is the TX state diagram which basically does the same as RX.

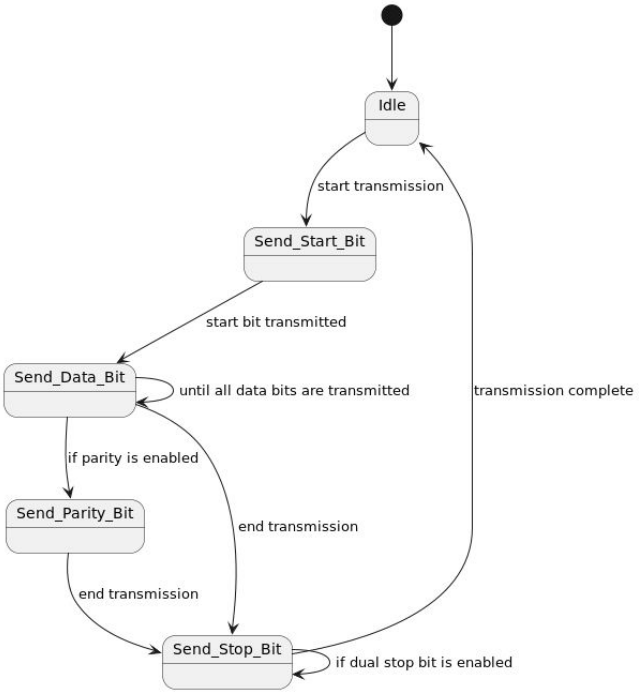


Figure 5. TX state diagram

5.1. Code

The implementation has a TX and RX portion. The TX code is a bit simpler than the RX code and this is because TX is made to be blocking whereas RX is completely interrupt based, this means that for RX a state has to be kept this is done in the struct seen below (figure 6).

```
struct {
    io_buffer_t* buffer;
    // state
    volatile uint32_t data;  // GCC wide-char is 4 bytes
    // transmission state
    volatile uint16_t cnt              : 6;
    volatile uint16_t started          : 1;
    volatile uint16_t rec_parity       : 1;
    volatile uint16_t rec_dual_stop    : 1;
    // frame state
    volatile uint16_t parity           : 1;
    volatile uint16_t framing_error    : 1;
    volatile uint16_t parity_error     : 1;
    volatile uint16_t transfer_complete : 1;
    uint16_t _                         : 3;
} state;
```

Figure 6. RX state struct

All settings for the data frames are stored in the struct seen below.

```
struct {
    uint16_t parity         : 1;    // even (0), odd (1)
    uint16_t parity_enable  : 1;    // send out parity bit if set
    uint16_t dual_stop_bit  : 1;    // enable second stop bit
    uint16_t data_bits      : 5;    // (normally: 5 - 9) (1 is added afterwards)
    uint16_t frame_size     : 8;    // total frame size
} settings;
```

Figure 7. Settings struct

The bit timing is done using timers the initialization of which can be seen in figure 8. The RX timer runs twice as fast as the TX timer but this is because it has to generate an interrupt and this is only done when it has ticked twice so the code effectively runs on the same timing.

```
// TIM
tx_psc = (APB2_clock_frequency / (BAUD + 1));
rx_psc = (APB2_clock_frequency / (BAUD * 2 + 1));
config_TIM(TIM1, tx_psc, 0xffff);
start_TIM(TIM1);  // start TX timer
config_TIM(TIM10, rx_psc, 1);  // RX timer (started when SUART_start_receive is called)
start_TIM_update_irq(TIM10);  // TIM1_UP_TIM10_IRQHandler
```

Figure 8. Timer configuration

This function starts the RX timer and enables the falling edge interrupt on the RX pin, it also sets the buffer and resets the state. The falling edge interrupt is used to detect when a start bit is sent after which the state is unlocked.

```
void SUART_start_receive(io_buffer_t* buffer) {
    start_EXTI(RX_PIN);
    start_TIM(TIM10);
    state.buffer = buffer;
    state.started = 0;
    state.cnt = 0;
}
```

Figure 9. Start RX function

This is the falling edge interrupt which unlocks the state by setting ‘started’

```
extern void EXTI2_IRQHandler() {
    EXTI->PR = EXTI_PR_PR2;
    state.started = 1;
    state.transfer_complete = 0;
}
```

Figure 10. Falling edge interrupt

This is the RX timer interrupt which only runs when ‘started’ is set. It samples the bit on the RX pin and updates the state accordingly (see Figure 4 for the RX state diagram). At the end ‘SUART_transmission_complete’ is called, this function simply looks in the state struct and copies the data into the buffer passed when initializing the Software Serial RX.

```
extern void TIM1_UP_TIM10_IRQHandler() {
    TIM10->SR &= ~TIM_SR_UIF;
    if (!state.started) { return; }
    if (state.cnt <= (settings.data_bits + 1)) {
        if (!state.cnt) { state.parity = settings.parity; }  // reset parity before starting
        state.cnt++; state.data >>= 1;
        uint8_t d = GPIO_read(RX_PORT, RX_PIN);
        state.data |= d << settings.data_bits;
        state.parity ^= d;
    } else if (settings.parity_enable && !state.rec_parity) {
        state.parity_error = state.parity != GPIO_read(RX_PORT, RX_PIN);
        state.rec_parity = 1;
    } else if (settings.dual_stop_bit && !state.rec_dual_stop) {
        state.framing_error = !GPIO_read(RX_PORT, RX_PIN);
        state.rec_dual_stop = 1;
    } else {
        // reset transmission state variables
        state.framing_error |= !GPIO_read(RX_PORT, RX_PIN);
        state.started = state.cnt = 0;
        state.rec_parity = state.rec_dual_stop = 0;
        state.transfer_complete = 1;
        SUART_transfer_complete();
    }
}
```

Figure 11. RX timer interrupt

This is the TX code which is described in the TX state diagram (Figure 5). TX mainly differs from RX because it is blocking and thus does not have the need to keep track of its state because it just runs until all frames are sent.

```
void SUART_write(const uint32_t* buffer, uint32_t size) {
    uint16_t uart_tick = TIM1->CNT; uint8_t j, d, p;
    for (uint32_t i = 0; i < size; i++) {
        p = settings.parity;  // reset parity
        while (uart_tick == TIM1->CNT); uart_tick = TIM1->CNT;
        GPIO_write(TX_PORT, TX_PIN, 0);  // start bit
        for (j = 0; j < settings.data_bits + 1; j++) {
            while (uart_tick == TIM1->CNT); uart_tick = TIM1->CNT;
            d = (buffer[i] >> j) & 0b1u; p ^= d;
            GPIO_write(TX_PORT, TX_PIN, d & 0b1u);
        }
        if (settings.parity_enable) {
            while (uart_tick == TIM1->CNT); uart_tick = TIM1->CNT;
            GPIO_write(TX_PORT, TX_PIN, p);  // parity
        }
        while (uart_tick == TIM1->CNT); uart_tick = TIM1->CNT;
        GPIO_write(TX_PORT, TX_PIN, 1);  // stop bit
        if (settings.dual_stop_bit) {
            while (uart_tick == TIM1->CNT); uart_tick = TIM1->CNT;
            GPIO_write(TX_PORT, TX_PIN, 1);  // dual stop bit
        }
        while (TIM1->CNT - uart_tick < 3); uart_tick = TIM1->CNT;
    }
}
```

Figure 12. TX code

6. Result

To test the code a setup with two micro-controllers was made, one sending “Hello World!” and the other reading and then repeating it. Tests both TX and RX at the same time was only possible due to RX being interrupt based.

In figure 13 a standard transmission can be seen (8 data bits, even parity, 2 stop bits).
In figure 14 a custom setting testing timing drift can be seen (32 data bits, odd parity, 2 data bits). This setting would not be feasible if any significant amount of timing drift is present.
Both these tests were ran at the maximum stable speed of 460.8K baud (this is made possible by the 100MHz clock speed of the STM32F411).

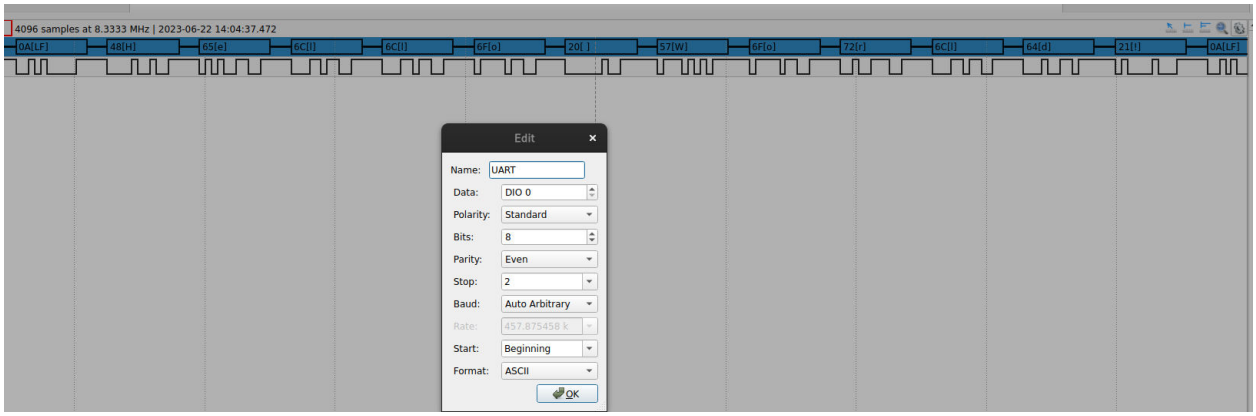


Figure 13. Standard transmission result

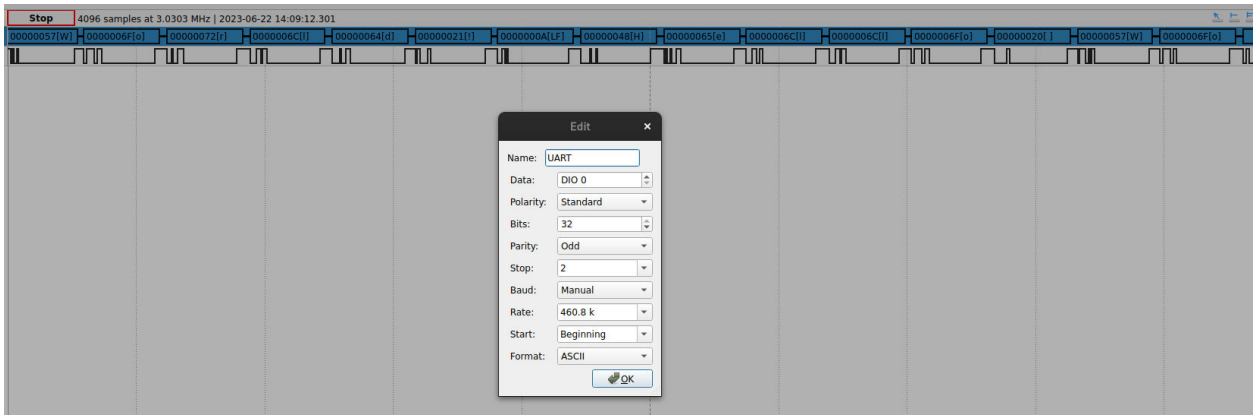


Figure 14. Custom transmission result

7. Reflection

Throughout the process of creating this software serial project, I have learned every thing there is to know about the UART protocol like the various settings or possible baud rates. Furthermore it taught me the limits off my micro-controller when pushing the baud rate to the highest possible setting. I also practiced debugging serial communication using an analog discovery in combination with the STM debugger.

Overall, this project has improved my knowledge of the UART protocol, micro-controller limitations and most importantly the debugging tools available to me.

8. Bibliography

analog.com/-/media/images/analog-dialogue/en/volume-54/number-4/articles/uart-a-hardware-communication-protocol/335962-fig-03.svg?w=900&imgver=2. (n.d.). Retrieved June 22, 2023, from https://www.analog.com/-/media/images/analog-dialogue/en/volume-54/number-4/articles/uart-a-hardware-communication-protocol/335962-fig-03.svg?w=900&imgver=2

analog.com/-/media/images/analog-dialogue/en/volume-54/number-4/articles/uart-a-hardware-communication-protocol/335962-fig-10.svg?w=900&imgver=1. (n.d.). Retrieved June 22, 2023, from https://www.analog.com/-/media/images/analog-dialogue/en/volume-54/number-4/articles/uart-a-hardware-communication-protocol/335962-fig-10.svg?w=900&imgver=1