

Zuzanna Godek 318373

MARM sprawozdanie z laboratorium 3

Celem laboratorium było zapoznanie się z układami sprzętowymi portów szeregowych USART w mikrokontrolerach STM32.

Zadanie 1.1. Obsługa portu szeregowego RS232 w trybie odpytywania

Najpierw należało skonfigurować port szeregowy USART1 w trybie asynchronicznej transmisji szeregowej z następującymi parametrami transmisji:

- Prędkość: 115200 bitów / sekundę
- 8 bitów danych
- 1 bit stopu
- Brak kontroli parzystości

```
void usart1_config() {  
    LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_USART1);  
    LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOB);  
  
    LL_GPIO_SetPinMode(GPIOB, LL_GPIO_PIN_6, LL_GPIO_MODE_ALTERNATE);  
    LL_GPIO_SetPinSpeed(GPIOB, LL_GPIO_PIN_6,  
LL_GPIO_SPEED_FREQ_VERY_HIGH);  
    LL_GPIO_SetPinOutputType(GPIOB, LL_GPIO_PIN_6,  
LL_GPIO_OUTPUT_PUSHPULL);  
    LL_GPIO_SetPinPull(GPIOB, LL_GPIO_PIN_6, LL_GPIO_PULL_UP);  
    LL_GPIO_SetAFPin_0_7(GPIOB, LL_GPIO_PIN_6, LL_GPIO_AF_7);  
  
    LL_GPIO_SetPinMode(GPIOB, LL_GPIO_PIN_7, LL_GPIO_MODE_ALTERNATE);  
    LL_GPIO_SetPinSpeed(GPIOB, LL_GPIO_PIN_7,  
LL_GPIO_SPEED_FREQ_VERY_HIGH);  
    LL_GPIO_SetPinOutputType(GPIOB, LL_GPIO_PIN_7,  
LL_GPIO_OUTPUT_PUSHPULL);  
    LL_GPIO_SetPinPull(GPIOB, LL_GPIO_PIN_7, LL_GPIO_PULL_UP);  
    LL_GPIO_SetAFPin_0_7(GPIOB, LL_GPIO_PIN_7, LL_GPIO_AF_7);  
  
    LL_USART_Disable(USART1);  
  
    LL_USART_InitTypeDef USART_InitStruct;  
  
    LL_USART_StructInit(&USART_InitStruct);  
  
    USART_InitStruct.BaudRate = 115200;  
    USART_InitStruct.DataWidth = LL_USART_DATAWIDTH_8B;  
    USART_InitStruct.StopBits = LL_USART_STOPBITS_1;
```

```

USART_InitStruct.Parity = LL_USART_PARITY_NONE;
USART_InitStruct.TransferDirection = LL_USART_DIRECTION_TX_RX;
USART_InitStruct.HardwareFlowControl = LL_USART_HWCONTROL_NONE;
USART_InitStruct.OverSampling = LL_USART_OVERSAMPLING_16;

LL_USART_Init(USART1, &USART_InitStruct);

LL_USART_Enable(USART1);
}

```

Celem tego zadania było przetestowanie transmisji USART w trybie odpytywania. W związku z tym napisano program który wysyłał dane oraz odczytywał je i wypisywał na konsoli.

```

namespace app {
    void test_thread(void*) {
        isix::wait_ms(100);

        send_string_polling("\r\nTerminal testowy USART1
(Polling).\r\n");
        send_string_polling("Pisz na klawiaturze:\r\n");

        for(;;) {
            // 1. Odbierz znak
            char received = receive_char_polling();

            // 2. Odeślij znak z powrotem
            send_char_polling(received);

            // 3. Specjalna obsługa enteru
            if (received == '\r') {
                send_char_polling('\n');
            }
        }
    }
}

```

Ta funkcja wysyła pojedynczy znak kiedy flaga TXE była ustawiona na 1. Następna służyła do wysyłania całego stringa.

```

void send_char_polling(char c) {
    while (!LL_USART_IsActiveFlag_TXE(USART1)) {
    }
    LL_USART_TransmitData8(USART1, c);
}

```

```
void send_string_polling(const char* str) {
    while (*str) {
        send_char_polling(*str++);
    }
    while (!LL_USART_IsActiveFlag_TC(USART1));
}
```

Ostatnia funkcja zczytywała znak i zwracała go.

```
char receive_char_polling() {
    while (!LL_USART_IsActiveFlag_RXNE(USART1)) {
    }
    return LL_USART_ReceiveData8(USART1);
}
```

Zweryfikowano poprawne działanie programu wywołując terminal szeregowy Picocom:
sudo picocom -b 115200 /dev/ttyUSB0

```
[zgodek@zg-laptop ~]$ sudo picocom -b 115200 /dev/ttyUSB0
[sudo] password for zgodek:
picocom v3.1

port is      : /dev/ttyUSB0
flowcontrol  : none
baudrate is  : 115200
parity is    : none
databits are : 8
stopbits are : 1
escape is    : C-a
local echo is: no
noinit is    : no
noreset is   : no
hangup is   : no
nolock is   : no
send_cmd is  : sz -vv
receive_cmd is: rz -vv -E
imap is      :
omap is      :
emap is      : crlf,delbs,
logfile is   : none
initstring   : none
exit_after is: not set
exit is      : no

Type [C-a] [C-h] to see available commands
Terminal ready

Terminal testowy USART1 (Polling).
Pisz na klawiaturze
123
lalalal
Ala ma kota.
```

Zadanie 1.2. Interaktywny terminal tekstowy

Korzystając z wcześniejszych opracowanych funkcji powstał program interaktywnego terminala (shell), który za pomocą zestawu prostych poleceń tekstowych wydawanych w programie terminalowym Putty.

Funkcja konfiguracyjna USART1 pozostała taka sama, natomiast dodano konfiguracje ledów oraz przycisku USER:

```
auto input_output_config() -> void {
    LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOD |
LL_AHB1_GRP1_PERIPH_GPIOA);

    LL_GPIO_SetPinMode(GPIOD, LL_GPIO_PIN_12, LL_GPIO_MODE_OUTPUT);
    LL_GPIO_SetPinMode(GPIOD, LL_GPIO_PIN_13, LL_GPIO_MODE_OUTPUT);
    LL_GPIO_SetPinMode(GPIOD, LL_GPIO_PIN_14, LL_GPIO_MODE_OUTPUT);
    LL_GPIO_SetPinMode(GPIOD, LL_GPIO_PIN_15, LL_GPIO_MODE_OUTPUT);

    LL_GPIO_SetPinOutputType(GPIOD, LL_GPIO_PIN_12,
LL_GPIO_OUTPUT_PUSHPULL);
    LL_GPIO_SetPinOutputType(GPIOD, LL_GPIO_PIN_13,
LL_GPIO_OUTPUT_PUSHPULL);
    LL_GPIO_SetPinOutputType(GPIOD, LL_GPIO_PIN_14,
LL_GPIO_OUTPUT_PUSHPULL);
    LL_GPIO_SetPinOutputType(GPIOD, LL_GPIO_PIN_15,
LL_GPIO_OUTPUT_PUSHPULL);

    LL_GPIO_SetPinMode(GPIOA, LL_GPIO_PIN_0, LL_GPIO_MODE_INPUT);
    LL_GPIO_SetPinPull(GPIOA, LL_GPIO_PIN_0, LL_GPIO_PULL_DOWN);
}
```

Do programu z poprzedniego zadania dopisano te 4 funkcje, realizujące kolejno włączanie lub wyłączanie wybranej diody, odczytanie stanu klawisza USER, odczytanie ilości wolnej pamięci na stercie systemu ISIX oraz odczytanie bieżące użycie procesora w procentach.

```
void control_led(int led_num, bool state) {
    uint32_t pin = 0;
    switch(led_num) {
        case 3: pin = LL_GPIO_PIN_13; break;
        case 4: pin = LL_GPIO_PIN_12; break;
        case 5: pin = LL_GPIO_PIN_14; break;
        case 6: pin = LL_GPIO_PIN_15; break;
        default: return;
    }
```

```

    if (state) LL_GPIO_SetOutputPin(GPIOB, pin);
    else        LL_GPIO_ResetOutputPin(GPIOB, pin);
}

bool get_button_state() {
    return LL_GPIO_IsInputPinSet(GPIOA, LL_GPIO_PIN_0);
}

auto get_free_heap_bytes() -> std::size_t {
    isix::memory_stat mem_info;

    isix::heap_stats(mem_info);

    return mem_info.free;
}

auto get_cpu_usage() -> int {
    return isix::cpuload();
}

```

Główna logika odczytu znaków i parsowania ich wyglądała następująco:

```

namespace app {
    static char cmd_buffer[64];
    static int cmd_idx = 0;

    void parse_command() {
        send_string_polling("\r\n");

        char reply[64];

        if (strcmp(cmd_buffer, "help") == 0) {
            send_string_polling("Dostepne komendy:\r\n");
            send_string_polling("led [3-6] on   - Wlacz diode\r\n");
            send_string_polling("led [3-6] off  - Wyłącz diode\r\n");
            send_string_polling("btn           - Stan przycisku
USER\r\n");
            send_string_polling("mem          - Wolna pamiec
RAM\r\n");
            send_string_polling("cpu          - Uzycie
procesora\r\n");
        }
        else if (strcmp(cmd_buffer, "btn") == 0) {

```

```

        if (get_button_state()) send_string_polling("USER Button:
WCISNIETY\r\n");
        else                      send_string_polling("USER Button:
PUSZCZONY\r\n");
    }
    else if (strcmp(cmd_buffer, "mem") == 0) {
        std::size_t free = get_free_heap_bytes();
        std::snprintf(reply, sizeof(reply), "Free Heap: %u
bytes\r\n", (unsigned int)free);
        send_string_polling(reply);
    }
    else if (strcmp(cmd_buffer, "cpu") == 0) {
        int load = get_cpu_usage();
        std::snprintf(reply, sizeof(reply), "CPU Load: %d.%d%\r\n",
load / 10, load % 10);
        send_string_polling(reply);
    }
    else if (strncmp(cmd_buffer, "led", 3) == 0) {
        int led_num = cmd_buffer[4] - '0';

        if (led_num >= 3 && led_num <= 6) {
            if (strstr(cmd_buffer, " on")) {
                control_led(led_num, true);
                send_string_polling("OK. LED ON.\r\n");
            }
            else if (strstr(cmd_buffer, " off")) {
                control_led(led_num, false);
                send_string_polling("OK. LED OFF.\r\n");
            } else {
                send_string_polling("Blad: uzyj 'on' lub
'off'\r\n");
            }
        } else {
            send_string_polling("Blad: zly numer diody (3-6)\r\n");
        }
    }
    else {
        if(strlen(cmd_buffer) > 0)
            send_string_polling("Nieznana komenda. Wpisz
'help'.\r\n");
    }

    send_string_polling("> ");

```

```

cmd_idx = 0;
memset(cmd_buffer, 0, sizeof(cmd_buffer));
}

void test_thread(void*) {
    isix::wait_ms(100);
    send_string_polling("\r\n== STM32 ISIX SHELL ==\r\n");
    send_string_polling("Wpisz 'help' aby zobaczyć liste
komend.\r\n> ");

    memset(cmd_buffer, 0, sizeof(cmd_buffer));

    for(;;) {
        char c = receive_char_polling();

        if (c == '\r') {
            parse_command();
        }
        else if (c == 127 || c == '\b') {
            if (cmd_idx > 0) {
                cmd_idx--;
                cmd_buffer[cmd_idx] = 0;
                send_string_polling("\b \b");
            }
        }
        else {
            if (cmd_idx < (int)sizeof(cmd_buffer) - 1) {
                cmd_buffer[cmd_idx++] = c;
                send_char_polling(c);
            }
        }
    }
}
}

```

A tak uruchamiano program Putty:
putty -serial /dev/ttyUSB0 -sercfg 115200,8,n,1,N

Następnie przetestowano wszystkie możliwości interaktywnego programu:

1. Włączanie/wyłączanie diod

```
/dev/ttyUSB0 - PuTTY

==== STM32 ISIX SHELL ====
Wpisz 'help' aby zobaczyć liste komend.
> help
Dostępne komendy:
led [3-6] on    - Włącz diode
led [3-6] off   - Wyłącz diode
btn            - Stan przycisku USER
mem            - Wolna pamięć RAM
cpu            - Użycie procesora
> led 3 on
OK. LED ON.
> led 4 on
OK. LED ON.
> led 5 on
OK. LED ON.
> led 6 on
OK. LED ON.
> led 3 off
OK. LED OFF.
> led 4 off
OK. LED OFF.
> led 5 off
OK. LED OFF.
> led 6 off
OK. LED OFF.
> █
```

2. Odczyt stanu klawisza USER (najpierw odczytano przed wciśnięciem a potem w trakcie i na końcu po puszczeniu)

```
> led 6 off
OK. LED OFF.
> btn
USER Button: PUSZCZONY
> btn
USER Button: WCISNIETY
> btn
USER Button: PUSZCZONY
> █
```

3. Potem odczytano wolną pamięć na stercie systemu ISIX

```
USER Button: PUSZCZONY
> mem
Free Heap: 124952 bytes
> █
```

4. Na końcu sprawdzono CPU load w kilku różnych momentach

```
Free Heap: 124952 bytes
> cpu
CPU Load: 0.0%
> █
```

Ostatnia funkcjonalność nie działała tak jak powinna, ale to było spowodowane tym, że program działał poprzez odpytywanie. Nie usypiał się w żadnym momencie, przez co ‘zagładzał’ inne procesy w tym proces liczący obciążenie procesora. W związku z tym wartość była ciągle 0.

Zadanie 1.3 Obsługa portu szeregowego RS232 z wykorzystaniem przerwań

Żeby zapobiec marnotrawstwu zasobów procesora należy korzystać z przerwań. Aby to osiągnąć należało dodać kilka zmiennych globalnych, jedną do przechowywania znaku odebranego oraz także zastosowano bufor znaków do przechowywania stringów przeznaczonych do wysłania. Globalnie przechowywano takie indeksy początku (następny znak do wysyłania) i końca bufora (ostatni znak do wysłania).

```
namespace {
    volatile char g_rx_char = 0;
    isix::semaphore g_rx_sem(0, 1);
    constexpr int TX_BUF_SIZE = 256;
    volatile char g_tx_buffer[TX_BUF_SIZE];
    volatile int g_tx_head = 0;
    volatile int g_tx_tail = 0;
}
```

Do funkcji konfigurującej USART1 dodano te 3 linijki, żeby włączyć przerwania dla USART1:

```
NVIC_SetPriority(USART1_IRQn, 1);
NVIC_EnableIRQ(USART1_IRQn);
LL_USART_EnableIT_RXNE(USART1);
```

Zmodyfikowano także funkcje wysyłające i odbierające dane. Funkcja wysyłająca znaki dodawała je na koniec bufora oraz inkrementowała indeks. Na końcu włączała przerwania dla

rejestru nadawczego USART1.

```
void send_char_interrupts(char c) {
    g_tx_buffer[g_tx_head] = c;
    g_tx_head = (g_tx_head + 1) % TX_BUF_SIZE;

    LL_USART_EnableIT_TXE(USART1);
}

void send_string_interrupts(const char* str) {
    while (*str) {
        send_char_interrupts(*str++);
    }
}
```

Funkcja odbierająca znak opierała się na działaniu semafora, ponieważ *wait* semafora usypia program i nie marnuje zasobów procesora.

```
char receive_char_interrupts() {
    int ret = g_rx_sem.wait(ISIX_TIME_INFINITE);

    if (ret != ISIX_EOK) {
        return 0;
    }

    return g_rx_char;
}
```

Obsługa przerwań wyglądała następująco:

```
extern "C" {
    void usart1_isr_vector(void) {
        if (LL_USART_IsActiveFlag_RXNE(USART1)) {
            g_rx_char = LL_USART_ReceiveData8(USART1);
            g_rx_sem.signal_isr();
        }

        if (LL_USART_IsEnabledIT_TXE(USART1) &&
LL_USART_IsActiveFlag_TXE(USART1)) {
            if (g_tx_head != g_tx_tail) {
                LL_USART_TransmitData8(USART1, g_tx_buffer[g_tx_tail]);

                g_tx_tail = (g_tx_tail + 1) % TX_BUF_SIZE;
            }
        }
    }
}
```

```
        else {
            LL_USART_DisableIT_TXE(USART1);
        }
    }
}
```

W przypadku jeśli przerwanie zgłosił rejestr odbiorczy, odczytywano odebrany znak i budzono semafor, który kontynuował działanie funkcji *receive_char_interrupts*.

Dla przerwań zgłoszonych przez rejestr nadawczy sprawdzano czy indeksy bufora są sobie równe (nie ma nowych znaków do wysyłania) i jeśli nie były to ‘wyrychały’ kolejny znak do wysyłania i inkrementowały indeks znaku do wysyłania. Jeśli indeksy były sobie równe to wyłączały przerwania, co oznaczało koniec transmisji.

Reszta programu pozostała bez zmian.

Przetestowano wszystkie funkcjonalności programu i działały poprawnie. Tym razem CPU Load 0.0% był poprawną wartością, ponieważ program opierał się na przerwaniach i w przypadku pisania na klawiaturze pojawiają się na tyle rzadko, że obciążenie było bliskie零.

```
==== STM32 ISIX SHELL ====
Wpisz 'help' aby zobaczyć liste komend.
> help
Dostępne komendy:
led [3-6] on      - Włącz diode
led [3-6] off     - Wyłącz diode
btn                - Stan przycisku USER
mem                - Wolna pamięć RAM
cpu                - Użycie procesora
> led 3 on
OK. LED ON.
> led 3 off
OK. LED OFF.
> btn
USER Button: PUSZCZONY
> btn
USER Button: WCISNIETY
> btn
USER Button: PUSZCZONY
> mem
Free Heap: 124648 bytes
> cpu
CPU Load: 0.0%
> █
```

Zadanie 1.4. Pomiar

Do pomiarów podczas laboratorium wykorzystano prostszy program, który wysyłał tylko "AAA" co 0,5 s. Program działał w trybie odpytywania:

```
void send_string_polling(const char* str) {
    while (*str) {
        while (!LL_USART_IsActiveFlag_TXE(USART1)) {}

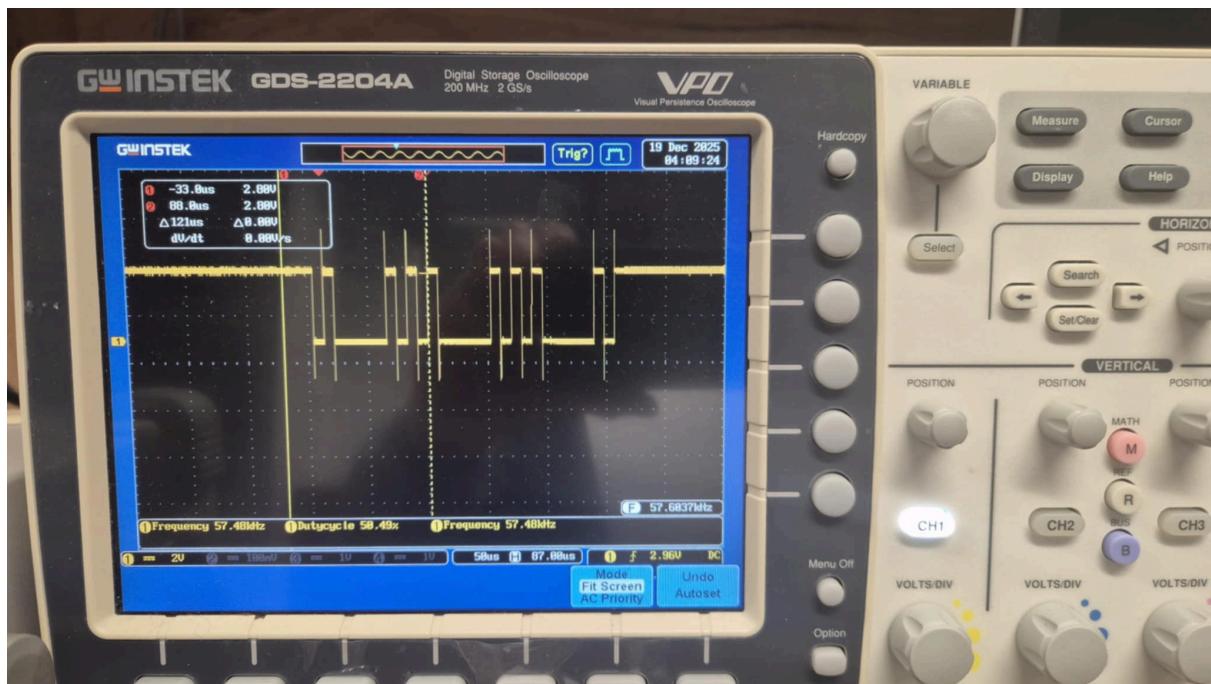
        LL_USART_TransmitData8(USART1, *str++);
    }

    while (!LL_USART_IsActiveFlag_TC(USART1));
}

namespace app {
    void test_thread(void*) {
        isix::wait_ms(100);

        for(;;) {
            send_string_polling("AAA");
            isix::wait_ms(500);
        }
    }
}
```

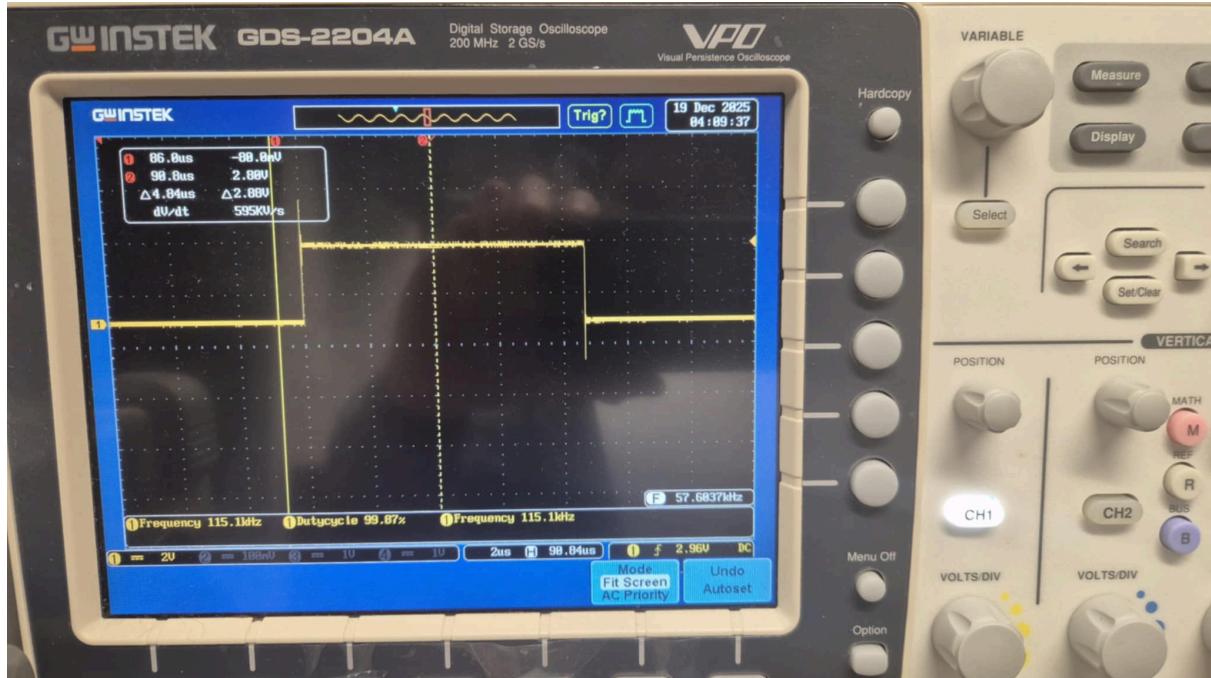
Oscyloskopem uchwycono tę transmisję:



Jeden znak ‘A’ w zapisie binarnym to 0100 0001. Natomiast USART wysyła w kolejności LSB, więc 1000 0010. Do tego należy dodać start bit (logiczne 0), potem są dane a na końcu stop bit (logiczne 1).

Na zdjęciu po starcie (0) widać 1, potem długą przerwę 0 (jest ich 5) i na końcu 1, 0 i stop bit (1). Potem znowu start bit i ta sama sekwencja jeszcze dwa razy. Czyli zostały poprawnie wysłane trzy znaki ‘A’.

Zrobiono także zbliżenie na pojedynczy bit wysłany przy użyciu odpytywania:



Ze zdjęcia można odczytać, że jeden bit trwał około 8,8 us, czyli na sekundę dało się wysłać $1/(8,8 \cdot 10^{-6}) \approx 113636$ bitów. W takim razie błąd procentowy wynosił:
 $(115200 - 113636)/115200 * 100\% \approx 1,36\%$

Nie wystarczyło czasu podczas laboratorium, żeby przeprowadzić pomiar średniego czasu odstępu między znakami przy nadawaniu w trybie z odpytywaniem oraz w trybie z przerwaniami.

Transmisja szeregową jest jednym z częściej wykorzystywanych protokołów transmisji w najtańszych mikrokontrolerach oraz w celach diagnostycznych ze względu na swoją prostotę (są tylko dwie linie RX i TX i kilka rejestrów). Asynchroniczność też upraszcza transmisję, ponieważ niepotrzebna jest synchronizacja sygnału zegara. Standardowe 115200 b/s nie obciąża zbytnio procesora w najtańszych mikrokontrolerach, gdzie są duże ograniczenia sprzętowe, a transmisja nie musi być bardzo szybka, szczególnie do celów diagnostycznych.