

## Vaja 8: Serijski vmesnik SCI – prekinitve

Received Data				
1	5	10	15	20
0	:	Hello	printf()	world!\n
1	:	Hello	printf()	world!\n
2	:	Hello	printf()	world!\n
3	:	Hello	printf()	world!\n
4	:	Hello	printf()	world!\n
5	:	Hello	printf()	world!\n
6	:	Hello	printf()	world!\n
7	:	Hello	printf()	world!\n
8	:	Hello	printf()	world!\n
9	:	Hello	printf()	world!\n
10	:	Hello	printf()	world!\n

## "echo" test




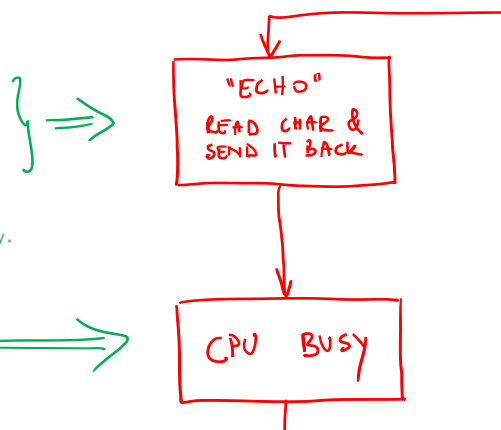
Diagram illustrating the "echo" test setup. A Raspberry Pi 3 is connected via USB to a laptop. The Pi has a black display and a red case. The laptop screen shows a blue background with a white percentage symbol. A red arrow points from the Pi to the laptop with the text "Hello world!". A green arrow points from the laptop back to the Pi with the text "Helo wrld!".

Received Data					
1	5	10	15	20	25
Hello world!\n					
Hello world!\n					
ello world!\n					
Helloworld!\n					
Hello orld!\n					
Hllo world!\n					
Helo world!\n					
Hello wrld!\n					
Hello word!\n					

```
// In an endless loop,
while(1)
{
    // poll for a new received character.
    if ( SCI_read_char(&c) == SCI_NO_ERROR )
        SCI_send_char(c); // and send it back immediately (i.e. echo).

    // After that, simulate the "microcontroller busy" situation by
    // calculating the geometric sum of N_MAX elements.
    // Vary the N_MAX parameter and see the effect on the echo functional.
    // Also try commenting this section.

    a_n = a_0;
    for(uint32_t n=0; n<N_MAX; n++)
    {
        sum = sum + a_n;
        a_n = a_n * a_0;
    }
}
```



*Slabost tehnike poizvedovanja ("polling")* je očitno ta, da moramo nekako poskrbeti, da poizvedovanje izvajamo dovolj pogosto, sicer lahko **pričnemo izgubljati prihajajoče podatke!**

Obstaja pa tudi *"prikrita slabost"* pri izvedbi funkcije za pošiljanje podatkov. Funkcija sicer deluje dobro v tem smislu, da pošlje vse podatke brez izgube podatkov. Slaba pa je v tem smislu, da imamo funkcijo implementirano tako, da pošlje vse podatke bajt za bajtom in da *moramo znotraj te funkcije počakati, da se pošljejo vsi podatki!* Poglejte npr. implementacijo `SCI_send_bytes()` spodaj.

```
// Function SCI_send_bytes() sends several bytes from a given location.
// The input parameters provide the data location and the size of data
// to be sent.
void SCI_send_bytes(uint8_t *data, uint32_t size)
{
    for(uint32_t i=0; i < size ; i++ )
    {
        SCI_send_byte( data[i] );
    }
}
```

*Slika 3 - slabost tako implementirane funkcije za pošiljanje podatkov je ta, da se funkcija izvaja toliko časa, dokler niso poslani vsi podatki. To pa pomeni, da taka funkcija ustavi izvajanje preostalega programa. In če je dolžina podatkov velika, je taka zaustavitev očitno lahko problematična.*

Tako implementirana funkcija pravzaprav **ustavi izvajanje preostalega programa** za toliko časa, dokler se preko serijskega vmesnika ne pošljejo vsi podatki. Takim funkcijam v programerskem žargonu s tujko pravimo **"blocking function"** (tudi "blocking process"). Poglejte izsek spodaj iz dokumentacije HAL knjižnice.

### 3.12.3 HAL I/O operation process

The HAL functions with internal data processing like transmit, receive, write and read are generally provided with three data processing modes as follows:

- Polling mode
- Interrupt mode
- DMA mode

#### 3.12.3.1 Polling mode

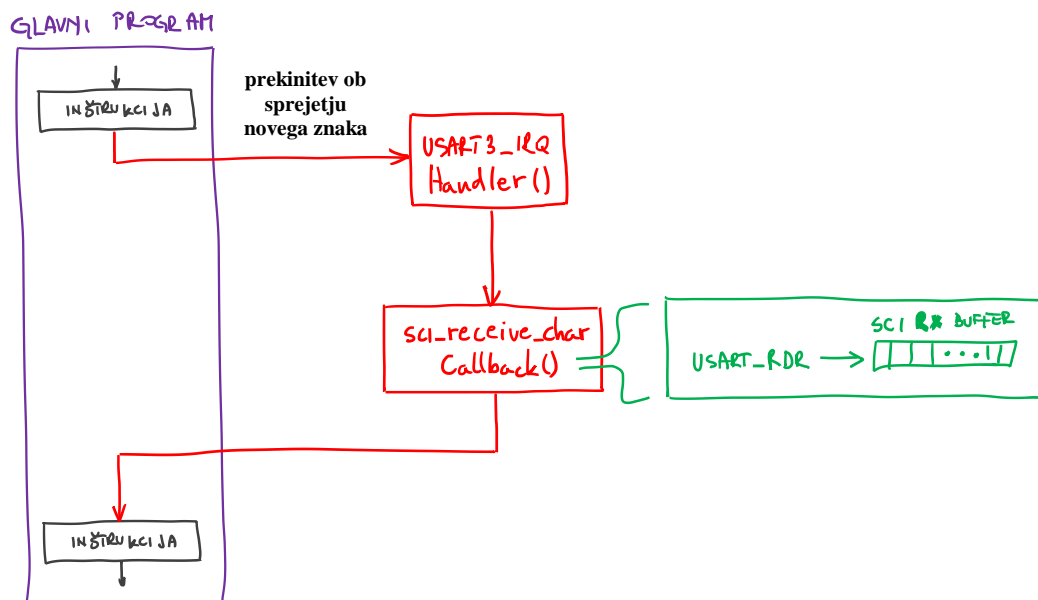
In Polling mode, the HAL functions return the process status when the data processing in blocking mode is complete. The operation is considered complete when the function returns the HAL\_OK status, otherwise an error status is returned. The user can get more information through the `HAL_PPP_GetState()` function. The data processing is handled internally in a loop. A timeout (expressed in ms) is used to prevent process hanging.

*Slika 4 - besedna zveza "blocking mode" namiguje, da so funkcije za obdelavo podatkov (npr. pošiljanje) implementirane tako, da ustavijo izvajanje preostalega programa, dokler podatki niso obdelani.*

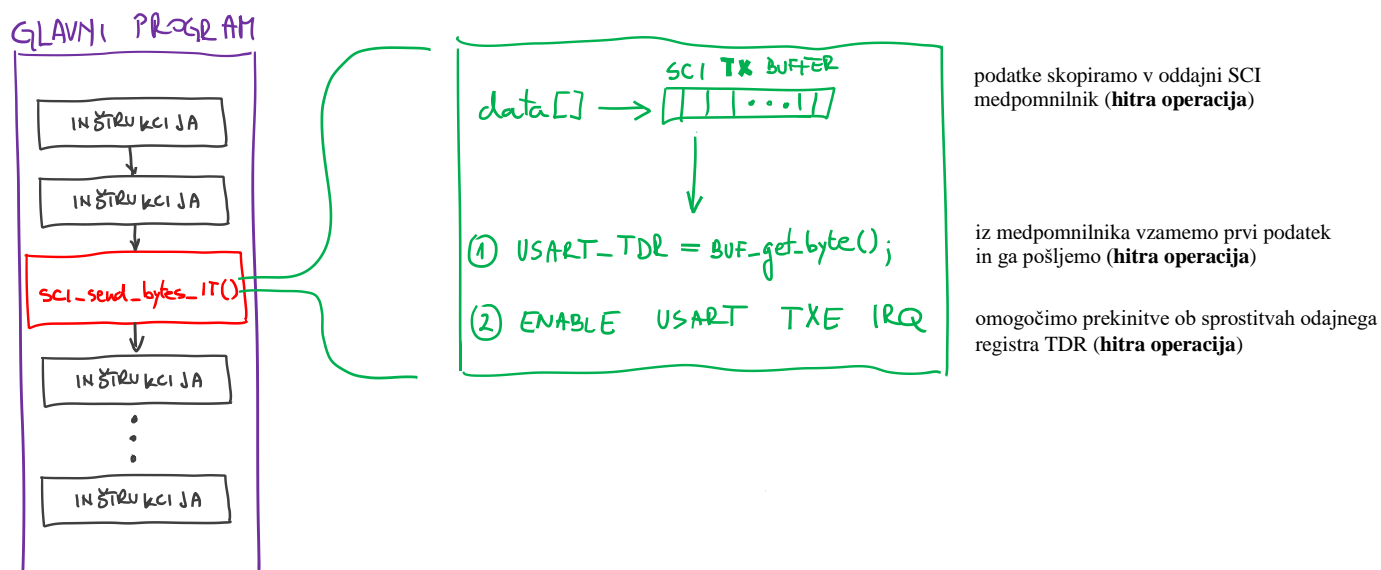
Namen te vaje je, da s pomočjo prekinitvev in medpomnilnikov odpravimo obe slabosti, ki smo jih izpostavili zgoraj. Poglejmo kako.

## Izboljšave v implementaciji serijskega vmesnika

**Slabost tehnike poizvedovanja**, kjer lahko pride do izgube sprejetega podatka, bomo odpravili tako, da bomo novo-sprejete znake shranjevali v *sprejemni* medpomnilnik s pomočjo prekinitvev. Poglejte idejo spodaj.



**Slabost "blocking" funkcije za pošiljanje podatkov** pa bomo tudi rešili s pomočjo prekinitvev in uporabe *oddajnega* medpomnilnika. Funkcija za pošiljanje podatkov `SCI_send_bytes_IT()` bo implementirana tako, da po potrebi pošlje le prvi bajt podatkov, pošiljanje preostalih podatkov pa prepusti prekinitveni rutini, ko bo naslednji podatek sploh mogoče poslati. Na ta način funkcija `SCI_send_bytes_IT()` postane hitra in ne predstavlja več kritične zaustavitve izvajanja preostalega programa. Poglejte idejo spodaj. Tako implementirani funkciji pravimo v žargonu tudi "**non-blocking function**".



## Naloge vaje

Izvedite nadgradnjo implementacije serijskega vmesnika tako, da bo za sprejemanje podatkov in za pošiljanje podatkov izkoristil funkcionalnost prekinitvev in sistemskih medpomnilnikov.

Nadgradnjo boste izvedli tako, da boste *dopolnili programsko kodo*, ki ste jo v sklopu priprave že umestili v datoteke vašega projekta. Pri dopolnitvi implementacije vas bodo *vodili komentarji v programski kodi*, zato na tem mestu le povzemimo ključne korake nadgradnje:

1. **sistemskemu serijskemu vmesniku SCI dodajte sprejemni RX medpomnilnik in oddajni TX medpomnilnik:**
  - a) v SCI modul vključite ustrezno knjižnico za podporo za delo z medpomnilniki,
  - b) definirajte podatkovne strukture obeh medpomnilnikov,
    - dolžina obeh medpomnilnikov naj bo 512 bajtov,
  - c) poskrbite za inicializacijo obeh medpomnilnikov.
2. **Omogočite prekinitve USART enote ob sprejemu novega podatka znotraj `SCI_init()` funkcije.**
3. **Dopolnite implementacijo funkcije `SCI_receive_char_Callback()`**, ki jo bomo uporabili ob ustrezni prekinitvi in bo poskrbela, da se novo-sprejeti podatek shrani v *sprejemni RX medpomnilnik* SCI vmesnika.
4. **Dopolnite implementacijo "non-blocking" funkcij za pošiljanje podatkov** preko serijskega vmesnika:
  - `SCI_send_string_IT()` za pošiljanje znakovnega niza,
  - `SCI_send_bytes_IT()` za pošiljanje zaporedja binarnih podatkov.
5. **Dopolnite implementacijo funkcije `SCI_send_char_Callback()`**, ki jo bomo uporabili ob ustrezni prekinitvi in bo poskrbela, da se preko serijskega vmesnika pošlje naslednji podatek iz *oddajnega TX medpomnilnika* SCI vmesnika.
6. **Modulu za implementacijo prekinitvenih rutin `stm32g4xx_it.c` :**
  - a) **dodajte podporo za delo z SCI vmesnikom** (tj. vključite ustrezno zglavno .h datoteko)

Kot zanimivost: sedaj se prvič srečate s situacijo, ko *nižje-nivojski strojni nivo* (tj. USART enota) *potrebuje storitve višjega systemskega nivoja* (tj. medpomnilnikov SCI vmesnika, "callback" funkcij za implementacijo prekinitvene rutine).
  - b) **Dopolnite implementacijo splošne prekinitvene rutine** `USART3_IRQHandler()`, kjer poskrbite, da se ob smiselnih prekinitvenih dogodkih kličejo ustrezne "callback" funkcije za pošiljanje oziroma sprejem podatka.

**7. Dopolnite implementacijo sledečih testnih funkcij:**

- a) `SCI_demo_receive_with_interrupts()` – testna funkcija, s katero demonstriramo *sprejemanje podatkov* s pomočjo prekinitvev in sprejemnega medpomnilnika,
- b) `SCI_demo_transmit_with_interrupts()` – testna funkcija, s katero demonstriramo *"non-blocking" pošiljanje podatkov* s pomočjo prekinitvev in oddajnega medpomnilnika,
- c) `SCI_demo_echo_with_interrupts()` – testna funkcija, s katero demonstriramo *"echo" funkcionalnost* na podlagi sprejema in pošiljanja podatkov s pomočjo prekinitvev in medpomnilnikov.

## Dodatna pojasnila

### Vloga modula `stm32g4xx_it.c`

Poglejte izsek iz HAL dokumentacije spodaj.

#### 3.1.2 User-application files

The minimum files required to build an application using the HAL are listed in the table below:

**Table 3. User-application files**

File	Description
<code>stm32g4xx_hal_conf.h</code>	This file allows the user to customize the HAL drivers for a specific application. It is not mandatory to modify this configuration. The application can use the default configuration without any modification.
<code>stm32g4xx_it.c/h</code>	This file contains the exceptions handler and peripherals interrupt service routine, and calls <code>HAL_IncTick()</code> at regular time intervals to increment a local variable (declared in <code>stm32g4xx_hal.c</code> ) used as HAL timebase. By default, this function is called each 1ms in SysTick ISR. . The <code>PPP_IRQHandler()</code> routine must call <code>HAL_PPP_IRQHandler()</code> if an interrupt based process is used within the application.
<code>main.c/h</code>	This file contains the main program routine, mainly: <ul style="list-style-type: none"><li>• Call to <code>HAL_Init()</code></li><li>• <code>assert_failed()</code> implementation</li><li>• system clock configuration</li><li>• peripheral HAL initialization and user application code.</li></ul>