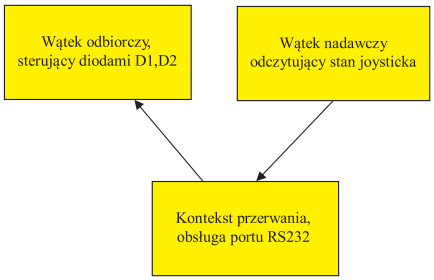
ISIX-RTOS
Obsługa portu szeregowego w STM32

W artykule prezentujemy sposób przygotowania uniwersalnego sterownika zapewniającego obsługę portu szeregowego mikrokontrolera STM32. Przyda się z pewnością w większości aplikacji, chociażby do tworzenia komunikatów diagnostycznych na etapie uruchamiania projektu.

Sterownik dla portu szeregowego przygotujemy z wykorzystaniem systemu przerwań. W przypadku urządzeń znakowych najbardziej odpowiednim będzie użycie kolejek FIFO, jednej nadawczej oraz drugiej odbiorczej. Ponieważ w kontekście przerwań nie możemy wywoływać funkcji blokujących, należy użyć specjalnych metod z przyrostkiem *_isr* (*push_isr(), pop_isr()*) klasy *fifo*.

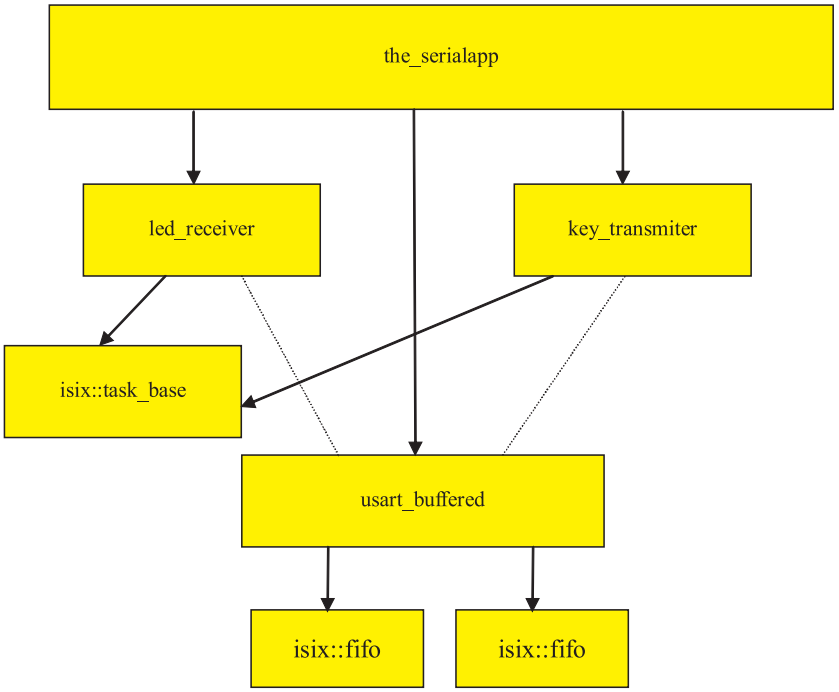


Rysunek 1. Sposób działania aplikacji opisanej w artykule z podziałem na wątki

Aby pokazać możliwości pracy wielowątkowej, w przykładzie utworzymy dwa wątki:

– odbiorczy, służący do odbioru danych z portu szeregowego, który w zależności od odebranego znaku będzie sterował pracą diod LED: D1 i D2 zamontowanych na płytce STM-32Butterfly,
– nadawczy, którego działanie sprowadzać się będzie do odczytania stanu joysticka oraz transmisję poprze UART informacji tekstowej o jego aktualnej pozycji.

Po podłączeniu zestawu STM32Butterfly do interfejsu RS232 komputera, w celu przetestowania działania aplikacji należy uruchomić program terminalowy (np. *Minicom*, *Hyperterminal* itp.) oraz skonfigurować wybrany port szeregowy z następującymi parametrami transmisji: prędkość – 115200 b/s, liczba bitów danych: 8, 1 bit stopu, brak kontroli parzystości i kontroli przepływu. Po zaprogramowaniu mikrokontrolera, w oknie terminala powinien pojawić się komunikat informujący o uruchomieniu programu. Po wciśnięciu na klawiaturze PC klawisza A mamy możliwość włączenia diody LED D1 i jej wyłączenia za pomocą klawisza B. W podobny sposób możemy sterować pracą diody LED D2 – służą do tego celu klawisze C i D. Przechylenie joysticka powoduje wyświetlenie informacji tekstowej o jego położeniu. Sposób działania aplikacji z podziałem na wątki przedstawiono na **rysunku 1**.



Rysunek 2. Hierarchia klas aplikacji w przykładowym projekcie

Dodatkowe informacje:
Artykuł poświęcony systemowi operacyjnemu ISIX-RTOS opublikowaliśmy w EP 3/2010.

Dodatkowe materiały na CD i FTP:
ftp://ep.com.pl, user: 10765, pass: 4t4q4glg

Listing 1. Funkcja główna *main*
```
//App main entry point
int main()
{
        //The application object
        static app::the_serialapp app;
        //Start the isix scheduler
        isix::isix_start_scheduler();
}
```

Aplikacja składa się z dwóch wątków, które używają jednego portu szeregowego. Jeden wątek jest odpowiedzialny za odczyt danych z portu szeregowego oraz włączanie i wyłączanie diod LED D1 i D2. Drugi wątek jest odpowiedzialny za cykliczny odczyt stanu joysticka oraz – w przypadku wykrycia odchylenia od położenia standardowego – wysłania informacji o kierunku wychylenia jego osi. Hierarchię klas aplikacji przedstawiono na **rysunku 2**.

Podobnie jak we wszystkich prezentowanych przykładach, klasa *the_serialapp* jest klasą aplikacji przechowującą wszystkie obiekty. Statyczny obiekt tej klasy jest tworzony w funkcji głównej *main()* (**listing 1**).

Deklaracje klasy obiektu aplikacji przedstawiono na **listingu 2**.

Listing 2. Deklaracja klasy serialapp
```
//The application class
class the_serialapp
{
public:
        //App Constructor
        the_serialapp(): usart(USART2),ledrcv(usart),keytran(usart)
        {}
private:
        //Serial device
        dev::usart_buffered usart;

        //The blinker class
        led_receiver ledrcv;

        //The key transmitter class
        key_transmitter keytran;


};
```

Klasa *the_serialapp*, zawiera obiekt *usart* klasy *led_receiver*, która stanowi obiekt portu szeregowego RS232. Obiekt *ledrcv* klasy *led_receiver* odpowiedzialny jest za odbiór znaków z portu szeregowego oraz sterowanie pracą LED w zależności od odebranego znaku. Obiekt *keytran* klasy *key_transmitter* odpowiedzialny jest za odczyt stanu styków joysticka oraz wysyłanie informacji do portu. Oba obiekty przyjmują referencję do wspólnego obiektu klasy *usart_buffered* oraz dziedziczą z klasy *isix::task_base*, więc stanowią odrębne wątki. Transmisja z wykorzystaniem portu szeregowego RS232 jest dupleksowa. Ponieważ jeden wątek tylko odczytuje dane z portu, natomiast drugi tylko zapisuje dane do tego portu, pracują one zupełnie niezależnie i nie wymagają wzajemnej synchronizacji za pomocą semafora, jak jest w przypadku obsługi magistrali I²C, która jest simpleksowa. Klasa *usart_buffered* jest uniwersalną klasą sterownika portu szeregowego RS232 wykorzystującą sprzętowy port USART mikrokontrolera rodziny STM32. Klasa została napisana tak aby była możliwość użycia dowolnego portu szeregowego dostępnego w mikrokontrolerze. Deklaracja klasy znajduje się w pliku *i2c_host.cpp* (**listing 3**).

Listing 3. Deklaracja klasy sterownika portu szeregowego
```
class usart_buffered
{
        friend void usart1_isr_vector(void);
        friend void usart2_isr_vector(void);

public:

        enum parity                         //Baud enumeration
        {
                parity_none,
                parity_odd,
                parity_even
        };

        //Constructor
        explicit usart_buffered(
                USART_TypeDef *_usart, unsigned cbaudrate = 115200,
                std::size_t queue_size=192, parity cpar=parity_none
        );

        //Set baudrate
        void set_baudrate(unsigned new_baudrate);

        //Set parity
        void set_parity(parity new_parity);

        //Putchar
        int putchar(unsigned char c, int timeout=isix::ISIX_TIME_INFINITE)
        {
                int result = tx_queue.push( c, timeout );
                start_tx();
                return result;
        }

        //Getchar
        int getchar(unsigned char &c, int timeout=isix::ISIX_TIME_INFINITE)
        {
                return rx_queue.pop(c, timeout );
        }

        //Put string into the uart
        int puts(const char *str);

        //Get string into the uart
        int gets(char *str, std::size_t max_len, int timeout=isix::ISIX_TIME_
INFINITE);

private:
        static const unsigned IRQ_PRIO = 1;
        static const unsigned IRQ_SUB = 7;

private:
        void start_tx();
        void isr();
        void irq_mask() { ::irq_mask(IRQ_PRIO, IRQ_SUB); }
        void irq_umask() { ::irq_umask(); }
        void periphcfg_usart1(bool is_alternate);
        void periphcfg_usart2(bool is_alternate);
private:
        USART_TypeDef *usart;
        isix::fifo<unsigned char> tx_queue;
        isix::fifo<unsigned char> rx_queue;
        volatile bool tx_en;

private:        //Noncopyable
        usart_buffered(usart_buffered &);
        usart_buffered& operator=(const usart_buffered&);
};
```

Klasa została zaprzyjaźniona z funkcjami obsługi przerwań portów szeregowych, które zostały wcześniej zadeklarowane z linkowaniem typu C, co powoduje wyłączenie manglowania nazw. Funkcje obsługi przerwań są wywoływane przez kontroler sprzętowy w momencie wystąpienia przerwania bez dodatkowych parametrów, co wymusza istnienie dostępu do instancji klasy obsługującej port szeregowy, poprzez wskaźnik lub referencję globalną. Wskaźniki dostępu do poszczególnych instancji klas przypisanych do portów szeregowych zostały umieszczone w nienazwanej przestrzeni nazw w pliku implementacji (*usart_buffered.cpp*), przez co dostęp do wskaźników jest możliwy tylko w obrębie danego modułu.

Zadeklarowanie przyjaźni funkcji z klasą umożliwia wywołanie dowolnych metod z funkcji zaprzyjaźnionej, co zostało wykorzystane do wywołania metody *isr()* stanowiącej wektor obsługi przerwania. Klasa obsługi portu szeregowego zawiera dwa obiekty *tx_queue*, *rx_queue* (**listing 4**) klasy *isix::fifo*, które są wykorzystywane jako bu-

fory nadajnika oraz odbiornika. Konstruktor klasy przyjmuje cztery parametry: adres wybranego kontrolera portu szeregowego (np. *USART1*, *USART2*), prędkość transmisji z ustawionym argumentem domyślnym na 115200, wielkość kolejek FIFO ustawionych domyślnie na 192 bajty oraz tryb kontroli parzystości z domyślnym argumentem ustawionym na *parity_none*.

Konstruktor odpowiedzialny jest za inicjalizację wybranego układu USART zgodnie z zadanymi parametrami. Na liście inicjalizacyjnej konstruktora tworzone są obiekty kolejek FIFO o zadanej wielkości. Następnie w zależności od numeru portu szeregowego inicjalizowane są linie GPIO tak, aby pełniły funkcję obsługi układu peryferyjnego, oraz włączany jest wybrany USART, co realizowane jest przez metody *periphcfg_usart1()*, oraz *periphcfg_usart2()*. Następnie włączany jest port szeregowy oraz jest konfigurowany podzielnik układu tak, aby pracował z zadaną prędkością poprzez wywołanie metody *set_baudrate()*. W następnej kolejności wywoływana jest metoda *set_parity()*, której zadaniem jest odpowiednie skonfigurowanie bitu parzystości. W zależności od wykorzystanego układu USART, do wskaźników obiektów przypisanych do przerwania przypisywane są adresy obiektu, oraz w kontrolerze NVIC włączane są przerwania. Sterownik posiada dwie podstawowe metody interfejsu umożliwiające wysłanie oraz odbieranie znaków, które mogą być wykorzystane przez inne klasy.

Metoda *putchar()* (**listing 5**), odpowiedzialna za wysyłanie znaku do portu szeregowego, przyjmuje dwa parametry: znak do wysłania, oraz maksymalny dopuszczalny czas oczekiwania, na miejsce w kolejce FIFO. Działanie tej metody jest bardzo proste i sprowadza się do próby zapisania danych do kolejki, a następnie wywołanie metody *start_tx()*, której zadaniem jest rozpoczęcie nadawania znaków. Pozostała część jest realizowana przez procedurę obsługi przerwania.

Metoda *getchar()* (**listing 6**) umożliwia odbieranie znaków z portu szeregowego. Przyjmuje dwa argumenty: referencję do znaku oraz maksymalny czas oczekiwania na ten znak. Działanie tej metody sprowadza się jedynie do wywołania metody *pop()* kolejki odbiorczej. Jeżeli w buforze jest jakiś znak umieszczony przez procedurę obsługi przerwania, wówczas następuje jego odczytanie. Jeżeli w buforze nie ma ani jednego znaku następuje zablokowanie aktualnego wątku do momentu odebrania znaku.

Cała praca realizowana jest głównie przez procedury obsługi przerwania, które są wywoływane w momencie, gdy na wybranym porcie szeregowym jest miejsce w buforze nadawczym lub został odebrany jakiś znak. Zgłoszenie przerwania od danego portu szeregowego powoduje rozpoczęcie wykonania funkcji *usart1_isr_vector()* lub *usart2_isr_vector()*.

W przypadku, gdy do wskaźnika danego portu szeregowego został przypisany jakiś obiekt, wówczas wywoływana jest metoda *isr()* – **listing 7** – odpowiedzialna za realizację procedury obsługi przerwania.

Działanie procedury obsługi przerwania jest bardzo proste: sprowadza się do odczytania statusu kontrolera USART oraz

Listing 4. Implementacja konstruktora klasy portu szeregowego

```
/*-----------------------------------------------------------*/
//! Constructor called for usart buffered
usart_buffered::usart_buffered(USART_TypeDef *_usart, unsigned cbaudrate,
                std::size_t queue_size ,parity cpar
) : usart(_usart), tx_queue(queue_size),
   rx_queue(queue_size) , tx_en( false )
{
        if(_usart == USART1)
        {
                periphcfg_usart1(false);
        }
        else if(_usart == USART2)
        {
                periphcfg_usart2(true);
        }
        //Enable UART
        usart->CR1 = CR1_UE_SET;
        //Setup default baudrate
        set_baudrate( cbaudrate );
        set_parity( cpar );

        //One stop bit
        usart->CR2 = USART_StopBits_1;

        //Enable receiver and transmitter and anable related interrupts
        usart->CR1 |=  USART_Mode_Rx |USART_RXNEIE | USART_Mode_Tx ;

        if( _usart == USART1 )
        {
                usart1_obj = this;
                //Enable usart IRQ with lower priority
                nvic_set_priority( USART1_IRQn,IRQ_PRIO, IRQ_SUB );
                nvic_irq_enable( USART1_IRQn, true );
        }
        else if( _usart == USART2 )
        {
                usart2_obj = this;
                //Enable usart IRQ with lower priority
                nvic_set_priority( USART2_IRQn,IRQ_PRIO, IRQ_SUB );
                nvic_irq_enable(  USART2_IRQn, true );
        }
}
```

Listing 5. Definicja metody wysłania znaku do portu szeregowego

```
int usart_buffered::putchar(unsigned char c, int timeout=isix::ISIX_TIME_
INFINITE)
{
        int result = tx_queue.push( c, timeout );
        start_tx();
        return result;
}
```

Listing 6. Definicja metody odebrania znaku z portu szeregowego

```
//Getchar
int getchar(unsigned char &c, int timeout=isix::ISIX_TIME_INFINITE)
{
        return rx_queue.pop(c, timeout );
}
```

Listing 7. Implementacja metody obsługi przerwań klasy portu szeregowego

```
void usart_buffered::isr()
{
        uint16_t usart_sr = usart->SR;
        if( usart_sr & USART_RXNE  )
        {
                //Received data interrupt
                unsigned char ch = usart->DR;
                //fifo_put(&hwnd->rx_fifo,ch);
                rx_queue.push_isr(ch);
        }
        if(tx_en && (usart_sr&USART_TXE) )
        {
                unsigned char ch;
                if( tx_queue.pop_isr(ch) == isix::ISIX_EOK )
                {
                        usart->DR = ch;
                }
                else
                {
                        usart->CR1 &= ~USART_TXEIE;
                        tx_en = false;
                }
        }
}
```

Listing 8. Deklaracja klasy *led_receiver*

```
//Serial receiver task class
class led_receiver: public isix::task_base
{
public:
        //Constructor
        led_receiver(dev::usart_buffered &_serial);
protected:
        //Main thread method
        virtual void main();
private:
        //Stack configuration
        static const unsigned STACK_SIZE = 256;
        static const unsigned TASK_PRIO = 3;
        //The usart obj ref
        dev::usart_buffered &serial;
};
```

Listing 9. Implementacja metody *main* klasy *led_receiver*

```
//Main task/thread function
void led_receiver::main()
{
        while(true)
        {
                unsigned char c;
                //Receive data from serial
                if(serial.getchar(c)==isix::ISIX_EOK)
                {
                        //Check for received char
                        switch(c)
                        {
                        //On led 1
                        case 'a':
                        case 'A':
                                io_clr( LED_PORT, LED1_PIN );
                                break;
                        //Off led 1
                        case 'b':
                        case 'B':
                                io_set( LED_PORT, LED1_PIN );
                                break;
                        //On led 2
                        case 'c':
                        case 'C':
                                io_clr( LED_PORT, LED2_PIN );
                                break;
                        //Off led 2
                        case 'd':
                        case 'D':
                                io_set( LED_PORT, LED2_PIN );
                                break;
                        }
                }
        }
}
```

podjęciu odpowiedniej akcji. W przypadku, gdy przerwanie zostało wygenerowane w wyniku odebrania znaku, wówczas jest on odczytywany z rejestru danych, a następnie przekazywany do kolejki. Do wysłania znaku do kolejki FIFO używana jest nieblokująca metoda *push_isr()*, dedykowana procedurom obsługi przerwań. W przypadku, gdy zostało wygenerowane przerwanie przy braku danych w buforze nadawczym, wówczas znak odczytywany jest z kolejki nadawczej za pomocą nieblokującej metody *pop_isr()*, a następnie odczytany znak zapisywany jest do rejestru danych układu USART. W przypadku, gdy nie ma danych w kolejce zerowana jest flaga zgłoszenia przerwania

Klasa *led_receiver* (**listing 8**) odpowiedzialna jest za odbieranie danych z portu szeregowego oraz sterowanie diodami LED w zależności od kodu odebranego znaku. Klasa dziedziczy z klasy bazowej *isix::task_base*.

Klasa zawiera referencję do obiektu sterownika portu szeregowego *usart_buffered*. Realizacja zadania odbywa się w metodzie wirtualnej *main()*, stanowiącą odrębny wątek systemowy – **listing 9**.

Działanie metody sprowadza się do wywołania metody *getchar()* obiektu sterownika portu szeregowego, która blokuje się do momentu odebrania znaku. W przypadku odebrania prawidłowego kodu znaku, odpowiednie diody LED są włączane lub wyłączane.

Na początku, za pomocą metody *puts()* sterownika portu szeregowego, wysyłane są teksty powitalne do portu szeregowego, a następnie program wchodzi do pętli głównej. Pętla główna wykonywana jest cyklicznie z czasem *DELAY_TIME* (25 ms), co umożliwia sprawdzenie stanu joysticka z eliminacją drgań zestyków. W przypadku wykrycia zmiany stanu portów, sprawdzany jest numer klawisza, a następnie za pomocą metody *puts* sterownika portu szeregowego wypisywane są komunikaty informujące o pozycji joysticka.

Lucjan Bryndza, EP
lucck@boff.pl