AVR–GCC: kompilator C dla mikrokontrolerów AVR, część 11
Obsługa interfejsu USART

Jako uzupełnienie odcinków o przerwaniach przedstawimy kilka przykładów ich praktycznego zastosowania. Jednym z najpopularniejszych przykładów jest obsługa interfejsu komunikacji szeregowej USART.

Zazwyczaj pierwsze próby uruchomienia komunikacji szeregowej opierają się na wykorzystaniu pętli oczekujących na wystąpienie odpowiednich stanów portu USART:
- opróżnienia rejestru UDR w przypadku nadawania – wtedy możemy wpisać kolejny bajt do wysłania,
- pojawienia się nowego (jeszcze nie odczytanego) znaku (bajtu) w rejestrze UDR odbiornika (dla przypomnienia: rejestr pod tą samą nazwą obsługuje dwie różne funkcje, nadawanie przy zapisie oraz odbiór przy odczycie).

Jest to zrealizowane poprzez *polling* (cykliczne kontrolowanie) wartości flag odpowiadających tym stanom. Podczas nadawania przed każdym wpisem znaku do UDR sprawdzamy czy zapalona jest flaga wskazująca na jego opróżnienie, np. w taki sposób:

```
void Wyslij_znak (uchar Znak)
{
 while ((UCSRA & _BV(UDRE)) == 0);
 // czekaj do ustawienia flagi UDRE
 UDR = Znak;
 // wpisz znak do rejestru
}
```

Dla odebrania znaku również oczekujemy na zapalenie odpowiedniej flagi, np. w taki sposób:

```
uchar Czytaj_znak (void)
{
 while ((UCSRA & _BV(RXC)) == 0);
 // czekaj do ustawienia flagi RXC
 return UDR;
 // odczytaj odebrany znak z rejestru
}
```

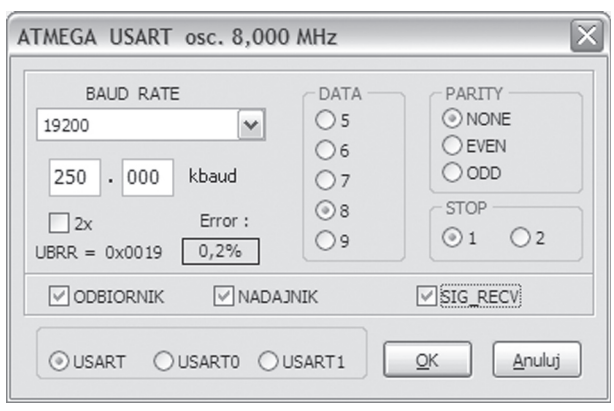
Nadawanie, oczywiście, będzie działać bez problemów, ale kosztem znacznego marnowania czasu procesora. Na przykład przesłanie bloku 64 bajtów z szybkością 9600 baud (przy 8 bitach danych, 1 bicie stopu i bez bitu parzystości, czyli przy 10 bitach na znak) zajmuje 640/9600=ok. 67 ms. To bardzo niewiele w skali operatora terminala, jednak np. mikrokontroler ATmega pracujący z częstotliwością 8 MHz (czyli z czasem trwania cyklu 0,125 μs) traci bezproduktywnie 67000/0,125=536000 cykli (!) tylko na oczekiwanie, aż powolny interfejs szeregowy wykona swoją pracę. W praktyce będzie to czas krótszy, gdyż zwykle przy pierwszym znaku rejestr jest zwolniony, poza tym na ogół obecnie stosujemy znacznie większe szybkości przesyłu, jednak i tak dla mikrokontrolera jest to bardzo dużo. Oczywiście nie ma sprawy jeśli komunikacja jest czynnością nadrzędną i możemy sobie w programie na takie czekanie pozwolić – gorzej gdy zabiera ono „moc obliczeniową" jakimś innym, być może krytycznym czasowo, zadaniom.

Z odbieraniem jest zupełnie źle. Wchodząc w pętlę oczekiwania na znak uzależniamy działanie naszego programu od czynnika zewnętrznego – oddalonego nadajnika, który ten znak może przysłać prędzej, później albo wcale (co skutecznie "obezwładni" nasze urządzenie). Takie rozwiązanie ma sens jedynie przy nawiązywaniu terminalowej komunikacji z operatorem. Łatwo zauważyć, że jest to przeniesienie "żywcem" zasad programowania znanych z konsolowych aplikacji PC: program zgłasza komunikat, czeka cierpliwie na wprowadzenie danych lub komendy w klawiatury, wykonuje zlecone zadanie i wypisuje wynik. Ten schemat przeważnie zupełnie nie pasuje do większości zastosowań mikrokontrolerów ukierunkowanych na działanie samodzielne – po raz kolejny zauważamy, że w świecie małych kostek obowiązują nieco inne reguły.

Wszystkich powyższych niedogodności unikniemy stosując przerwania. Każdy przychodzący znak wywołuje przerwanie *SIG_UART_RECV*, w którego obsłudze wykonujemy potrzebną czynność (czyli przede wszystkim przepisanie znaku z UDR do bufora), a poza przerwaniem program cały czas normalnie pracuje sprawdzając jedynie okresowo czy przyszły jakieś nowe komunikaty/komendy/dane (to oczywiście zawsze będzie zależeć od konkretnego sposobu w jaki zorganizowaliśmy sobie protokół komunikacji).

Z kolei w celu wysłania pakietu danych ładujemy bufor nadawania potrzebną zawartością i po prostu uruchamiamy przerwanie nadajnika – w jego obsłudze przepisujemy kolejne znaki z bufora do rejestru UDR a po ostatnim znaku wyłączamy przerwanie (tutaj także szczegółowe rozwiązanie zależy od przyjętego protokołu). Pętla główna przygotowuje dane i rozpoczyna transmisję – później już nie musi się przebiegiem procesu nadawania w ogóle zajmować.

Jeśli zajmowaliśmy się programowaniem mikrokontrolerów rodziny '51 zwróćmy uwagę na zasad-



Rys. 27. Okno konfiguratora ustawień portu USART

niczą różnicę w działaniu przerwania nadajnika:
– w '51 jest ono wywoływane dopiero **po** wysłaniu kolejnego znaku; aby zapoczątkować pracę nadajnika ustawialiśmy samodzielnie w programie flagę TI;
– w AVR jest odwrotnie: przerwanie (*SIG_UART_DATA*) jest aktywne zawsze gdy rejestr UDR jest pusty i gotowy na przyjęcie następnego znaku; oznacza to, że będzie wywołane przed wysłaniem znaku, natychmiast po odblokowaniu przerwania.

Błędem będzie więc w AVR odblokowanie na stałe *SIG_UART_DATA*, a dopiero później wysyłanie znaków, przerwanie bowiem zacznie pracować od razu wywołując na ogół nieoczekiwane efekty.

Zauważmy jednak, że nie dotyczy to drugiego wbudowanego w AVR przerwania: *SIG_UART_TRANS*, które zachowuje się odwrotnie, jest bowiem uaktywniane po kompletnym wysłaniu całego znaku (łącznie z bitem stopu) z rejestru przesuwnego nadajnika na pin *TxD*, pod warunkiem, że w tym momencie rejestr UDR jest pusty (a więc był to ostatni wysyłany znak). Takie działanie (wykrycie końca przekazywania ostatniego znaku w pakiecie do linii transmisyjnej) jest przewidziane specjalnie dla przypadków łączności *half–duplex* (np. przy jednoparowym RS–485) w celu prawidłowego przełączenia interfejsu linii z powrotem w tryb odbioru. W zwykłej transmisji RS–232 zazwyczaj używamy *SIG_UART_DATA* , jednak zastosowanie zamiennie *SIG_UART_TRANS* (w sposób podobny jak w 51) jest oczywiście również możliwe.

Jak zwykle najlepiej obejrzeć to wszystko na przykładzie. Nowy projekt będzie kontynuacją poprzedniego. W tym celu w dowolnym managerze plików tworzymy folder *...\Kurs\Przyklad–05* i kopiujemy do niego pliki źródłowe (*main.c*, *timers.c*, *projdat.h*) z *...\Kurs\Przyklad–04*. Teraz w AvrSide otwieramy nowy projekt, ładujemy do niego pliki źródłowe poleceniem menu *Projekt–>Importuj z folderu* (przechodzimy w oknie wyboru do nowego subfolderu *...\Przyklad–05* i zaznaczamy wszystkie skopiowane tam przed chwilą pliki z kodem), ustawiamy potrzebne opcje (ścieżka do własnych plików nagłówkowych) i zapisujemy projekt jako *...\Przyklad–05\test05.gcp*. W ten sposób utworzyliśmy kopię poprzedniego projektu, którą teraz możemy rozwijać pozostawiając wcześniejszy przykład w stanie nie naruszonym.

Do pliku *projdat.h* dopisujemy kilka nowych definicji, zmiennych (klasyfikator *volatile dla zmiennych używanych w przerwaniach !) oraz deklaracji funkcji:*

```
#define RXSIZE 4
// rozmiar bufora odbiornika

volatile Flags UsartFlags;
// flagi stanu portu szeregowego

volatile char RxBuffer[RXSIZE];
// definicja bufora odbiornika

#define NEW_COMMAND UsartFlags.Bits.Flag1
#define TX_BUSY UsartFlags.Bits.Flag2
// wygodne nazwanie poszczególnych flag
stanu portu

extern void InitUsart(void);
extern void SendAnswer(int AnswerId);
extern void SendPrompt(void);
// deklaracje funkcji obsługi USART
```

Dodajemy nowy moduł usart.c o następującej treści:

```
// obsługa USART
#include „projdat.h"
#include <avr/io.h>
#include <avr/signal.h>

#define TX_ON (UCSRB |= _BV(UDRIE))
#define TX_OFF (UCSRB &= ~_BV(UDRIE))
// włączanie i wyłączanie przerwań na-
dajnika

volatile static char *TxPtr;
// wskaźnik na znak wysyłany

static char Prompt[] = „Gotowość do od-
bioru komendy 1–3.\n";
// tekst zgłoszenia

static char Odp0[] = „Nie mogę określić
komendy:–(!\n";
static char Odp1[] = „Wykonuję komendę
numer jeden.\n";
static char Odp2[] = „Wykonuję komendę
numer dwa.\n";
static char Odp3[] = „Wykonuję komendę
numer trzy.\n";
// teksty odpowiedzi na komendy

static char *AnswerTable[] = {Odp0,Od-
p1,Odp2,Odp3};
// tablica wskaźników na komunikaty od-
powiedzi

void InitUsart(void)
{
 // ==== single usart configuration ====
 // 19200 baud with 8000 kHz osc./erro-
r=0,2%
 // data 8/stop 1/parity NONE
 // receiver ON/transmitter ON/recv in-
terrupt enabled
 UBRRH = 0x00;
 UBRRL = 0x19;
 UCSRA = 0x0;
 UCSRB = _BV(RXEN) | _BV(TXEN) |
_BV(RXCIE);
 UCSRC = _BV(URSEL) | _BV(UCSZ0) |
_BV(UCSZ1);
 // ==== end usart ====
}

void SendAnswer(int AnswerId)
{
 if((AnswerId < 1) || (AnswerId > 3))
AnswerId = 0;
 TxPtr = AnswerTable[AnswerId];
 TX_BUSY = true;
 TX_ON;
}

void SendPrompt(void)
{
 TxPtr = Prompt;
 TX_BUSY = true;
 TX_ON;
}

SIGNAL (SIG_UART_DATA)
{
 char Znak;
 Znak=*(TxPtr++);
 if(Znak) UDR = Znak; else
  {
   TX_OFF;
   TX_BUSY = false;
   }
}

SIGNAL (SIG_UART_RECV)
{
 RxBuffer[0] = UDR;
 if (! TX_BUSY) NEW_COMMAND = true;
}
```

Szablony handlerów przerwań tworzymy korzystając z opisanego wcześniej okienka autokompletacji kodu. Natomiast dla ustawienia parametrów USART użyjemy wspomagającego konfiguratora. Polecenie *Narzędzia –> Kreator kodu –> Atmega usart* otwiera okienko pokazane na **rys. 27**. Wybieramy według potrzeb:
– Długość słowa danych, liczbę bitów stopu i rodzaj parzystości.
– Szybkość transmisji. Do dyspozycji mamy konwencjonalny szereg szybkości RS 232 oraz wartość dowolną (USER) przydatną przy mniej typowych rozwiązaniach. Pole *Error* pokazuje nam na bieżąco procentową odchyłkę szybkości rzeczywiście możliwej do uzyskania (przy stosowanej w projekcie częstotliwości oscylatora) od pożądanego ideału. Łatwo zauważymy, że wbudowany generator 8 MHz dopuszcza tylko kilka wartości z typowego szeregu. Doskonale natomiast nadaje się do nawiązania komunikacji USB z użyciem kostki FT8U232BM i szybkościami 125 kbaud lub 250 kbaud (do sprawy używania wewnętrznego generatora jeszcze za chwilę powrócimy).
– Włączenie nadajnika, odbiornika oraz przerwań odbiornika (nie ma tu oczywiście, zgodnie z poprzednimi uwagami, możliwości włączenia przerwań nadajnika).
– Numer portu (USART dla kostek z jednym portem, USART0 lub USART1 dla kostek dwuportowych).

Zatwierdzenie dialogu (OK) powoduje wstawienie bloku odpowiedniego kodu w miejscu ustawienia kursora tekstowego (karetki). W naszym przykładzie kod ten lokujemy wewnątrz funkcji *InitUsart (void)* inicjalizującej port.

Jerzy Szczesiul, EP
jerzy.szczesiul@ep.com.pl

UWAGA!
Środowisko IDE dla AVR–GCC opracowane przez autora artykułu można pobrać ze strony http://avrside.ep.com.pl.