C dla mikrokontrolerów 8051

część 5

W tej części omówimy zagadnienia związane z komunikacją pomiędzy mikrokontrolerem i otoczeniem, za pomocą interfejsu RS232. Sposób obsługi interfejsów SPI i I^2C przedstawimy za miesiąc.

Obsługa RS232

Rzadko zdarza się, aby mikrokontroler, którego zamierzamy użyć, oferował wszystkie potrzebne nam układy peryferyjne. Jeżeli jednak już tak jest, to albo ma również inne, zupełnie niepotrzebne, albo też przeraża jego cena.

Wówczas można użyć taniego mikrokontrolera i dołączyć do niego możliwie jak najtaniej jak najtańsze układy peryferyjne. Tu jednak pojawia się pewien problem - jak dołączyć układy zewnętrzne.

Jego rozwiązanie jest możliwe za pomocą różnych języków programowania. Absolutnie najprostszy w użyciu jest pod tym względem Bascom, który oferuje biblioteki gotowych procedur komunikacyjnych. Inaczej jest w przypadku C. Tu musimy o wszystko zadbać sami. No, może o prawie wszystko, ponieważ w większości kompilatorów obsługa sprzętowego portu UART (RS232) jest dostępna. Jest to zgodne ze specyfikacją ANSI dla języka C, w której przyjęto, że instrukcje *printf*, *getchar*, *putchar* wysyłają znak do (lub pobierają z) standardowego urządzenia wyjściowego (wejściowego). W przypadku komputera PC jest to monitor (i klawiatura).

Trudno jednak wyobrazić sobie prosty sterownik zbudowany z użyciem mikrokontrolera podłączony do monitora. Oczywiście jest to możliwe, ale nieopłacalne. W związku z tym standardowym urządzeniem wejścia/wyjścia dla mikrokontrolera jest port UART. Od niego też zaczniemy opis implementacji interfejsów.

UART - funkcje *stdio.h*

W związku ze specyfiką podawanych w tym opisie informacji, będą one dotyczyć pakietu Raisonance. Instrukcje *printf*, *getchar* i *putchar* będą zapewne działać identycznie w programach skompilowanych za pomocą kompilatorów pochodzących od różnych producentów, ale nastawy dotyczące szybkości przesyłanych danych mogą być przeprowadzone inaczej i jeśli ktoś używa na przykład Keil, to musi sięgnąć do dokumentacji tego pakietu.

List. 1. Tak można wysyłać znaki, używając funkcji putchar().

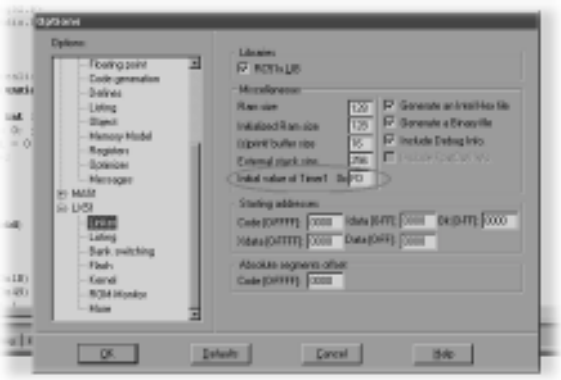
```
/*****************************************
wysyłanie kodów ASCII przez UART
mikrokontroler AT89S8252, kwarc 11,0592 MHz
Raisonance RC-51
*****************************************/
#include <reg52.h>              //definicje rejestrów
#include <stdio.h>              //dołączenie funkcji wejścia - wyjścia
#pragma DEFJ(TIM1_INIT=0xFD)    //ustalenie szybkości transmisji

//funkcja realizuje opóźnienie około k*1ms dla rezonatora f=11.0592 MHz
void delay (unsigned int k)
{
   unsigned int i,j;
   for ( j = 0; j < k; j++)
       for (i = 0; i <= 84;) i++;
}

//program główny, znaki o kodach od 0x20 do 0xFF wysyłane są kolejno przez UART
//co około 300 milisekund
void main(void)
{
   char i;

   for (i = 0x20; i <= 0xFF; i++)  //pętla wykonywana, gdy i<=255
   {
       putchar(i);          //przesłanie bajtu
       delay (300);         //opóźnienie 0,3 sekundy
   }
```

W asynchroniczny port UART, spełniający wymogi standardu RS232, wyposażony jest prawie każdy mikrokontroler. Oczywiście podłączenie UART do linii transmisyjnej wymaga układu dopasowującego zbudowanego z elementów dyskretnych lub układów scalonych, np. typu MAX232. Zgodnie z normą tego interfejsu poziomy napięć powinny zawierać się w przedziałach:
- -12...-5 V dla logicznej jedynki,
- 5...12 V dla logicznego zera.

Wymaga to zasilania układów dopasowujących z symetrycznego źródła napięcia, czyli najczęściej zastosowania przetwornicy. Wspomniany układ MAX232 zawiera wbudowane pompy ładunkowe wytwarzające z jednego napięcia zasilającego wymagane napięcia dodatnie i ujemne. Uwalnia nas tym samym od konieczności stosowania symetrycznego zasilacza.

Podobnie jak w przypadku rozwiązań innych problemów, mamy co najmniej dwie możliwości poprawnego wykorzystania układu UART. Możemy na przykład skorzystać z systemu przerwań oferowanego przez mikrokontroler. Wówczas UART pracuje w tle i dopiero skompletowanie słowa danych spowoduje, że zgłoszone zostanie przerwanie - podczas jego obsługi możemy opróżnić bufor, odebrać dane itp.

Możemy także oczekiwać na odbiór bajtu w pętli z instrukcją *getchar()*. Wówczas przypisanie *znak= getchar()* rozwiązuje problem odbioru bajtu. Wykorzystanie procesora nie jest jednak w tym przypadku



Zdjęcie 1. Korzystając z okienka Options, możemy wpisać wartość bajtu TH1

optymalne. Może on sporą część czasu tracić bezproduktywnie na oczekiwanie znaku. Wykorzystanie przerwań pozwala mu zająć się w przerwach między odbieranymi danymi innymi zadaniami.

Zacznijmy opis obsługi UART-u od prostszej metody, tej, w której nie wykorzystuje się przerwań. Funkcje wysyłania i odbioru znaków zdefiniowane są w bibliotece *stdio.h*. Aby ich użyć, musimy tę bibliotekę dołączyć dyrektywą *#include*. UART wykorzystuje Timer 1 do ustalenia szybkości transmisji. Timer pracuje w trybie 2, czyli jako ośmiobitowy z automatycznym odświeżaniem zawartości przy przepełnieniu. Szybkość pracy UART można więc ustalić wartością bajtu ładowanego do rejestru TH1. Poniżej przytaczam wzór zaczerpnięty z instrukcji programowania mikrokontrolera 80C51 pozwalający wyliczyć wartość TH1 odpowiednią do danej szybkości transmisji:

TH1 = 256 - (k x częstotliwość kwarcu/ /(384 x szybkość transmisji)),

gdzie „k“ to mnożnik prędkości transmisji - dla bitu SMOD równego 0 wynosi on 1, natomiast dla SMOD ustawionego na 1 wynosi on 2.

Przykładowo obliczymy wartość TH1 dla kwarcu 11,0592 MHz, bitu SMOD = 0 oraz prędkości transmisji 9600 bodów:

TH1 = 256 - (1x11059200/ /(384x9600)) = 253 (0xFD)

Kolejne pytanie. Jak przekazać wartość bajtu TH1 do procedur transmisji danych tak, aby funkcje zawarte w *stdio.h* mogły poprawnie ją odczytywać i interpretować?

Można to zrobić kilkoma sposobami. Można samodzielnie napisać procedurę inicjalizacji. Można również w parametrach kompilatora wstawić potrzebną wartość. Można też zmienić ją za pomocą dyrektywy *defj* umożliwiającej modyfikację stałych systemowych. Jeśli zdecydowaliśmy się na zmianę ustawienia stałej systemowej bez przygotowywania własnej procedury inicjalizacji, zdecydowanie nie zalecam korzystania z okienka Options (**rys. 1**). Może się bowiem zdarzyć, że wartość ustawiona dla jednego programu nie będzie odpowiednią dla innego, natomiast system zapamięta ją jako domyślną. Jeśli zapomnimy o okienku opcji, nowy program po skompilowaniu nie będzie działał prawidłowo. Będziemy szukać błędu, który jest tym trudniejszy do lokalizacji, że nie znajduje się w kodzie źródłowym programu.

Ten sam efekt, jak przez zmianę opcji kompilatora, można uzyskać używając dyrektywy defj. Jej użycie jest następujące: *#pragma DEFJ(TIM1_INIT=wartość)*, czyli dla przykładu: *#pragma DEFJ(TIM1_INIT=0xFD)*. Zdecydowanie zalecam ten właśnie sposób, jeśli nie chce się pisać procedur do inicjalizacji UART.

Na **list. 1** zamieszczono fragment programu powodującego wysyłanie znaków do urządzenia dołączonego do UART. Na początku dołączane są zbiory biblioteczne oraz ustalana jest wartość TH1 za pomocą *defj*. Znaki (bajty) wysyłane są przez funkcję *putchar()*. Domyślnie bit SMOD ma wartość „0“.

Podobnie jest z odbiorem. Jednak zanim przedstawię przykład programu odbioru danych, kilka słów wyjaśnienia. Typowo, do odbioru danych ze standardowego

List. 2. Fragment programu do obsługi programatora szeregowego.

```
#pragma DEFJ(TIM1_INIT=0xFE)    //timer 1 ustala prędkość transmisji
                                //tutaj 19200 bodów (SMOD będzie równy "1")
#pragma SMALL                   //wybór modelu pamięci programu
#include <stdio.h>              //funkcje wejścia - wyjścia
#include <reg51.h>              //definicje rejestrów

//program główny
void main ()
{
   char temp, temp1, cmd1, cmd2, cmd3;

   set_reset();          //wystawienie sygnału reset dla programowanego uK
                         //faza reset jest zależna od stanu linii resettype
                         //1=AT90, 0=AT89
   PCON |= 0x80;         //ustawienie bitu SMOD na "1"
   EI |= 0x81;           //włączenie przerwań i zezwolenie na int0

   clr_reset();          //zwolnienie reset z uwagami jak dla set_reset

   while (1)
   {
       while ((temp = _getkey()) == 0x1B);
       switch (temp)
       {
           case 'T':     //'T' typ urządzenia
               device = _getkey();
               put_ret();
               break;

           case 'S':     //'S' rodzaj podłączonego programatora
               putchar('A');     //wysłanie napisu "AVR ISP"
               putchar('V');
               putchar('R');
               putchar(' ');
               putchar('I');
               putchar('S');
               putchar('P');

/* lub inaczej - znacznie prościej: printf("AVR ISP"); instrukcja printf
wykorzystuje funkcję putchar(). Dodatkową korzyścią jest możliwość wyprowadzania
sformatowanych wydruków np. printf("%#bx",165); spowoduje wyświetlenie liczby
165 w zapisie szesnastkowym 0xA5 */
               break;

           case 'V':             //'V' wersja programu
               putchar('1');     //wysłanie napisu "10"
               putchar('0');
               break;

           case 'v':             //'v' wersja urządzenia
               putchar('1');     //wysłanie napisu "10"
               putchar('0');
               break;

.............
```

List. 3. Przykład obsługi transmisji szeregowej w oparciu o przerwanie generowane przez UART.

```
/****************************************
Obsługa transmisji szeregowej przez UART
z wykorzystaniem przerwań.
****************************************/
#include <reg51.h>

#define ROZM_BUFORA_TX 32
#define ROZM_BUFORA_RX 32
#define OSCYLATOR 11059200

unsigned char buf_wysylki[ROZM_BUFORA_TX];
unsigned char buf_odbioru[ROZM_BUFORA_RX];
unsigned char do_wysylki, wyslano;
unsigned char wysylka_wylaczona;
unsigned char do_odbioru, odebrano;

//funkcja obsługująca przerwanie UART; using 2 oznacza, że używany jest
//bank rejestrów R0..R7 numer 2
void UART_irq (void) interrupt 4 using 2
{
   if (RI != 0)  //fragment wykonywany, gdy do_odbioru znak
   {
       RI = 0;    //zerowanie flagi "do_odbioru"
       if ((do_odbioru+1) != odebrano) buf_odbioru[do_odbioru++] = SBUF;
                  //pobranie znaku do bufora odbioru, gdy jego
   }              //rozmiar jest wystarczający

   if (TI != 0)      //fragment wykonywany, gdy znak do wysłania
   {
       TI = 0;                              //zerowanie flagi "do wysyłki"
       if (do_wysylki != wyslano)
           SBUF = buf_wysylki[wyslano++];  //jeśli indeksy ilości znaków
                                            //i ilości znaków do wysłania
           else wysylka_wylaczona = 1;      //są różne, pobierz i wyślij
                                            //znak
   }
}

//obliczenie rozmiaru wolnego miejsca w buforze odbioru
unsigned char rozm_bufora_odbioru (void)
{
   return (do_odbioru - odebrano);
}

//obliczenie ilości znaków pozostających do wysyłki
unsigned char rozm_bufora_wysylki (void)
{
   return (do_wysylki - wyslano);
}

//ustawienie prędkości transmisji, inicjacja Timera 1
void UART_baudrate (unsigned char baudrate)
{
   EA = 0;                      //wyłączenie przerwań
   TI = 0;                      //kasowanie flagi przerwania od UART
   do_wysylki = wyslano = 0;    //nie wysłano i nie odebrano danych
   wysylka_wylaczona = 1;       //wyłączenie funkcji nadawania
   TR1 = 0;                     //zatrzymanie timera 1
   ET1 = 0;                     //wyłączenie przerwań timera 1
   PCON |= 0x80;                //SMOD = 1, mnożnik dla kwarcu x2
   TMOD &= ~0xF0;               //ustawienie trybu pracy timera 1
   TMOD |= 0x20;
                                //wyliczenie wartości dla TH1
   TH1 = (unsigned char)(256-(OSCYLATOR/(16L*12L*baudrate)));
   TR1 = 1;                     //uruchomienie timera 1
   EA = 1;                      //zezwolenie na przerwania
}
```

urządzenia wejścia - wyjścia (w naszym przypadku jest to UART) służy funkcja *getchar()*. Tkwi w niej pewna „pułapka". Zgodnie ze specyfikacją ANSI funkcja ta odsyła odebrany bajt. Można powiedzieć, że występuje efekt echa. Mimo iż jest to zgodne z normą języka, to najczęściej zupełnie niepotrzebne. Są oczywiście sytuacje, w których jest to bardzo wygodna metoda kontroli tego, co zostało wysłane. Częściej trzeba po prostu odebrać bajt i echo odsyłane do urządzenia nadającego bardzo w tym przeszkadza. Z moich doświadczeń wynika, że dla większości kompilatorów obok funkcji *getchar()* definiowana jest funkcja *getkey()*, która odbiera znak i nie wysyła echa. Tak jest np. w przypadku kompilatora RC-51. Wśród funkcji biblioteki *stdio.h* znajdziemy również *_getkey()*. Na **list. 2** pokazano fragment programu programatora sterowanego przez port szeregowy.

Słowo kluczowe while *((temp = _getkey()) == 0x1B)* inicjuje pętlę, w której wykonywane są dwie instrukcje. Jedna to przypisanie zmiennej temp wartości bajtu odebranego przez UART. Druga to porównanie tego bajtu z kodem ESC (0x1B) i zakończenia działania pętli, jeśli odebrany znak będzie różny od ESC. Zwróćmy uwagę na różnice w składni instrukcji przypisania (*zmienna = wartość*) i porównania (*zmienna == wartość*).

Opisane przykłady są bardzo proste. W programie obsługi transmisji należy dołączyć biblioteki *stdio.h*, ustawić odpowiednią prędkości transmisji i wywołać odpowiednią do potrzeb funkcji.

Inaczej (i trudniej) jest w przypadku wykorzystania przerwania. Na **list. 3** pokazano przykład programu do obsługi UART wykorzystującego przerwania.

Wróćmy jeszcze do biblioteki *stdio.h*. Jej opis nie byłby kompletny bez wyjaśnień dotyczących funkcji *ungetchar()*, *printf()* i *scanf()*.

Jak wspomniałem, zgodnie ze specyfikacją ANSI dla języka C funkcja *getchar()* przesyłała do nadajnika echo odebranego znaku. Funkcja *_getkey()* działa prawie identycznie jak *getchar()* ale nie wysyła echa. Do zestawu tych funkcji dołączona jest jeszcze *ungetchar()*, która umieszcza znak odebrany przez *getchar()* lub *_getkey()* z powrotem w buforze odbiornika tak, że następne wywołanie *getchar()* spowoduje odebranie tego samego znaku. Jest ona użyteczna wówczas, gdy kilka różnych procedur korzysta w programie, niezależnie od siebie, ze znaków odebranych przez UART. Można na przykład wyobrazić sobie sytuację, gdy odebrany znak jest kodem sterującym przeznaczonym dla innej procedury niż ta, która go odebrała. Odebranie znaku zeruje flagę RI oznaczającą gotowość bajtu do odbioru - ponowne użycie *getchar()* nie jest możliwe. Wówczas *ungetchar()* przywraca stan taki, jakby znak był właśnie przed chwilą odebrany. Można wtedy przekazać sterowanie do innego fragmentu programu, który odbierze bajt i właściwie go zinterpretuje.

List. 3. - cd.

```
//inicjalizacja trybu transmisji szeregowej
void UART_inicjalizacja (void)
{
   UART_baudrate (19200);       //ustawienie prędkości transmisji
   EA = 0;                      //wyłączenie przerwań
   do_wysylki = wyslano = 0;    //zerowanie indeksów nadawania i odbioru
   wysylka_wylaczona = 1;
   do_odbioru = odebrano = 0;
   SM0 = 0; SM1 = 1;            //ustawienie trybu pracy UART na "mode 1"
   SM2 = 0;
   REN = 1;                     //zezwolenie na pracę odbiornika UART
   TI = RI = 0;                 //kasowanie flag przerwania UART
   ES = 1;                      //zezwolenie na przerwania od UART
   PS = 0;                      //ustawienie niskiego priorytetu
   EA = 1;                      //załączenie przerwań
}

//przykład własnej implementacji funkcji wysyłającej znak przez UART
signed char _putchar(unsigned char c)
{
                                //bufor zbyt mały, błąd
   if ((ROZM_BUFORA_TX-rozm_bufora_wysylki())<=2) return (-1);
   EA = 0;    //wyłączenie przerwań
   buf_wysylki[do_wysylki++] = c;  //wstawienie znaku do bufora nadawania
   if (wysylka_wylaczona)       //jeśli nadawanie jest wyłączone
   {
       wysylka_wylaczona = 0;
       TI = 1;                  //załącz je
   }
   EA = 1;                      //załączenie przerwań
   return (0);                  //jeśli operacja poprawna, zwróć 0
}

//przykład wykonania funkcji odbierającej znak z UART
signed int _getchar (void)
{
   unsigned char c;

   if (rozm_bufora_odbioru() == 0)
       return (-1);             //brak odebranych znaków, błąd
   EA = 0;//wyłączenie przerwań
   c = buf_odbioru[odebrano++];    //pobranie znaku z bufora
   EA = 1;//załączenie przerwań
   return (c);
}
```

Funkcja obsługująca standardowe wyjście danych *printf()* tłumaczy wartości odebranych bajtów na znaki. Sposób jej wywołania jest następujący: *int printf(char *format, arg1 [, arg2,...., arg-n])*. Funkcja *printf()* wykorzystuje polecenie *putchar()* do wysyłania łańcucha znaków powstałego na skutek przekształcenia do wymaganego formatu. Przekształcenie odbywa się zgodnie z wzorcem zawartym w argumencie format. Zawiera on różnego rodzaju obiekty - zwykłe znaki, które są kopiowane wprost do łańcucha wyjściowego oraz specyfikacje różnych przekształceń, z których każda wskazuje na sposób przekształcenia i wysłania kolejnego argumentu printf(). Każdą specyfikację formatu rozpoczyna znak %, a kończy znak charakterystyczny dla danego przekształcenia. Pomiędzy znakiem % a znakiem przekształcenia mogą wystąpić dodatkowe symbole sterujące w kolejności takiej, jak poniżej:

- „-“ przesuwający przekształcony argument do lewej strony,
- liczba określająca minimalny rozmiar pola,
- „.“ oddzielająca rozmiar pola od jego precyzji (części ułamkowej),
- liczba określająca precyzję, to jest maksymalną liczbę znaków dla tekstu, liczbę cyfr po kropce dziesiętnej dla wartości zmiennopozycyjnej lub minimalną liczbę znaków dla wartości stałopozycyjnej,
- „h“ lub „l“ (litera małe L), jeśli argument całkowity należy wyświetlić odpowiednio - w postaci *short* lub w postaci *long*.

W **tab. 1**. zestawiono najważniejsze znaki kontrolujące przekształcenia danych.

Funkcja *scanf()* jest odpowiednikiem *printf()*, lecz działającym w przeciwną stronę. To znaczy wprowadza ona znaki ze standardowego wejścia, interpretuje je zgodnie z informacjami zawartymi w formacie oraz zapamiętuje w miejscach określonych przez pozostałe argumenty. Jej wywołanie ma postać: *int scanf(char *format, *arg1[, *arg2,....])*. Odczyt danych ze standardowego wejścia zakończy się, gdy *scanf()* zinterpretuje wszystkie znaki lub dane nie pasują do specyfikacji przekształcenia. Każdy z argumentów funkcji *scanf()* musi być wskaźnikiem. Do interpretacji wprowadzanego ciągu znaków używane są te same symbole przekształceń co dla *printf()*.

Funkcjami *printf()* i *scanf()*, a zwłaszcza pierwszą z nich, zajmiemy się jeszcze w następnym odcinku kursu. Jak wspomniałem, funkcja *printf()* używa do wysyłania znaków *putchar()*, dlatego zmieniając *putchar()* tak aby znaki były kierowane do wyświetlacza LCD zamiast do portu UART, można wykorzystać ją do formatowania wyświetlanego tekstu. Przyda się to zwłaszcza przy wyświetlaniu liczb zmiennopozycyjnych. Jednak temat związany z użyciem *printf()*, *scanf()* i formatowaniem tekstu jest tak obszerny, że zasługuje na oddzielny artykuł.

Jacek Bogusz, AVT
jacek.bogusz@ep.com.pl

Dodatkowe informacje

Ewaluacyjną wersję pakietu firmy Raisonance prezentowanego w artykule zamieściliśmy na CD-EP8/2002B.

Tab. 1. Podstawowe przekształcenia funkcji *printf()*

Znak	Typ argumentu	Przekształcenie do postaci
d, i	int	liczba dziesiętna ze znakiem
o	int	liczba ósemkowa bez znaku i wiodącego zera
x, X	int	liczba szesnastkowa bez znaku i wiodącego *0x* użycie małej litery x w konsekwencji powoduje przy przekształcaniu stosowanie znaków *abcdef*, natomiast dużego X - *ABCDEF*
u	int	liczba dziesiętna bez znaku
c	int	pojedynczy znak
s	char*	tekst, wypisywany do momentu napotkania znaku końca tekstu \0 lub osiągnięcia rozmiaru (precyzji) pola
f	double	liczba dziesiętna zmiennopozycyjna, gdzie liczbę cyfr po kropce dziesiętnej określa precyzja
e, E	double	liczba dziesiętna w postaci wykładniczej
p	void*	wskaźnik, reprezentacja zależy od implementacji w konkretnej bibliotece stdio.h
%		brak przekształcenia, wypisywany jest znak %