AVR–GCC: kompilator C dla mikrokontrolerów AVR, część 12
Obsługa interfejsu USART

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

Po modyfikacjach przedstawionych w poprzedniej części artykułu, główny plik projektu *main.c* wygląda teraz następująco:

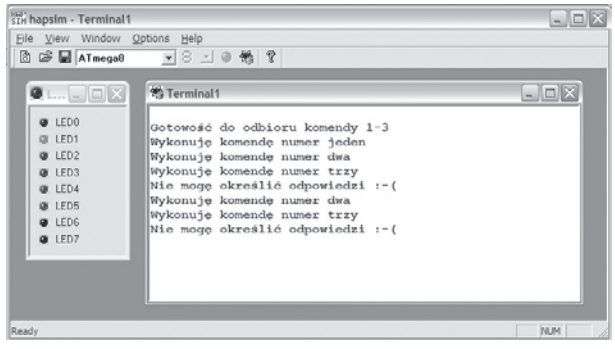
```
// główny moduł projektu
#define _MAIN_MOD_ 1
// pliki dołączone (include):
#include „projdat.h”
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/eeprom.h>
#include <avr/signal.h>

#define MS100_DELAY 5

// dane:
static char Ms100_counter;
static volatile uchar LedState = 1;

// funkcje:


//====================
// funkcja main()
int main(void)
{
 // inicjalizacja

 OSCCAL=eeprom_read_byte((uchar*)E-
2END);
 // zapis kalibracji w ostatniej komór-
ce eeprom
 DDRB=0xff;

 InitT2();
 InitUsart();
 sei();

 SendPrompt();

 // pętla główna
 while (1)
 {
  // obsługa systemowego „zegarka”
100ms
  if (MS100_FLAG)
   {
    MS100_FLAG = false;
    if (++Ms100_counter == MS100_DELAY)
     {
      Ms100_counter = 0;
      // nasza okresowa akcja (przełą-
czenie wyjścia) uruchamiana
      // zegarem systemowym co 100ms *
MS100_DELAY (0,5s)
      PORTB=LedState;
      if(LedState==128) LedState=1;
else LedState = LedState<<1;
     }
   }

  // obsługa komend przesłanych przez
usart
  if (NEW_COMMAND)
   {
    NEW_COMMAND = false;
    SendAnswer(RxBuffer[0] - 0x30);
   }
 }
}
```

Jak widać obsługa naszej prostej komunikacji to w module głównym tylko kilka dodatkowych linijek. Działa to wszystko w następujący sposób:

– przy starcie programu inicjalizujemy USART: przerwania odbiornika są włączone i oczekują na komendy z PC, nadajnik jest włączony ale przerwania nadajnika pozostają zablokowane;
– komunikacja jest zorganizowana w trybie tekstowym, co umożliwia jej wypróbowanie przy pomocy dowolnego terminala: odbiornik reaguje na komendy jednoznakowe, a w odpowiedzi mikrokontroler odsyła krótkie powiadomienia;
– w obsłudze przerwania odbiornika wykonujemy (zgodnie z wcześniejszymi zaleceniami) tylko podstawowe czynności: przepisanie znaku z rejestru UDR do bufora i ustawienie flagi otrzymania nowej komendy *NEW_COMMAND*;
– flaga ta informuje program główny o konieczności wykonania przewidzianej akcji: w naszym przypadku jest to wywołanie funkcji *SendAnswer* uruchamiającej wysyłanie odpowiedzi, parametrem funkcji jest wartość liczbowa komendy (przeliczona z kodu ASCII cyfry 1...3 przysłanej tekstowo z terminala);
– funkcja akceptuje wartości 1...3, w innych przypadkach argument zostaje zastąpiony zerem; w taki sposób argument może bezpośrednio posłużyć jako indeks tablicy *AnswerTable*, w której ulokowane są wskaźniki na poszczególne teksty odpowiedzi;
– uruchomienie wysyłania odpowiedzi sprowadza się do ustawienia wskaźnika *TxPtr* na początek właściwego tekstu i włączenie przerwania nadajnika (makro *TX_ON*);
– całością wysyłania zajmuje się teraz *handler* przerwania nadajnika: pobiera kolejne znaki komunikatu i ładuje je do rejestru UDR, natomiast w momencie natrafienia na znak *null* (o wartości zerowej, który zawsze w standardzie C kończy łańcuch tekstowy) zatrzymuje nadawanie wyłączając przerwanie (makro *TX_OFF*);
– na czas nadawania dodatkowo ustawiamy flagę *TX_BUSY*, która blokuje reakcję na dalsze komendy (co mogłoby w trakcie wysyłania odpowiedzi przestawić wskaźnik i narobić zamieszania);
– oddzielna funkcja *SendPrompt* wysyła jednorazowy komunikat startowy po inicjalizacji.

Pracę programu możemy obejrzeć bez fizycznego sprzętu posługując się opisanym już *HAPSIM'em* . Dodajemy okno terminala, ustawiamy jego typ na *usart* i wyłączamy lokalne echo. Po uruchomieniu AvrStudio wysyłamy z terminala jednoznakowe komendy otrzymując z symulowanej kostki odpowiedzi (**rys. 28**).

Koniecznie trzeba zdawać sobie sprawę, że zadziałanie powyższej symulacji wcale nie jest równoznaczne z poprawną pracą rzeczywistego zmontowanego układu. Do-



Rys. 28. Terminal znakowy symulatora HAPSIM

chodzi tutaj cały wachlarz dodatkowych możliwych problemów:
- poprawność połączenia kablowego ze współpracującym portem szeregowym;
- właściwe wlutowanie i praca kostki konwertera TTL<–>RS 232 (np. typowy MAX232);
- zgodność ustawionych parametrów transmisji w obu urządzeniach komunikacyjnych (zauważmy, że HAPSIM w ogóle o to nie dba);
- utrzymanie szybkości transmisji na odpowiednim poziomie (czyli stabilna i pewna praca oscylatora);
- sprawdzona i działająca konfiguracja współpracującego portu (w PC wiele zależy od zastosowanego oprogramowania, np. czasem można się spotkać z koniecznością odpowiedniego skrosowania linii RTS, CTS, DSR, DTR we wtyku).

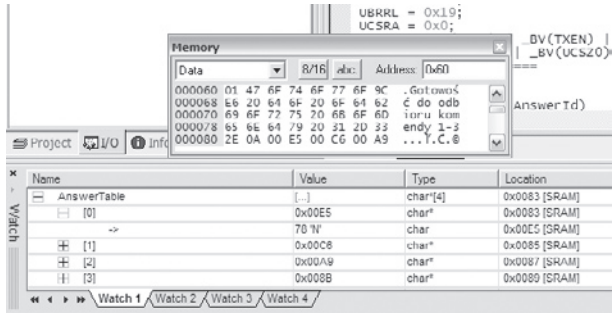
Często pojawia się pytanie, czy wbudowany w AVR oscylator (*calibrated internal RC oscillator*) jest wystarczająco „pewny" dla używania w celach transmisyjnych. Na ogół możemy przyjąć, że w zastosowaniach domowych (stała pokojowa temperatura, przebieg transmisji pod bezpośrednią kontrolą – np. w przyborach warsztatowych, przystawkach do PC itp.) nie napotkamy na problemy (chociaż czasem wymagane jest niewielkie dostrojenie fabrycznej wartości bajtu kalibracyjnego). Natomiast w urządzeniu pracującym autonomicznie w zmiennych warunkach termicznych znacznie bezpieczniej będzie przewidzieć klasyczny układ oscylatora kwarcowego.

Powyższy przykład komunikacji tekstowej – chociaż prosty w uruchomieniu i obsłudze – zazwyczaj nie wystarcza w wielu typowych zastosowaniach mikrokontrolerów, gdy przesyłamy bloki danych binarnych, wśród których mogą rzecz jasna pojawić się zera. Uniemożliwia to wykorzystywanie zaprezentowanego sposobu wykrywania końca ramki danych. Spotkamy się z wielką liczbą rozmaitych rozwiązań protokołów komunikacyjnych stosowanych w takich przypadkach – od całkiem prostych do wyrafinowanych i skomplikowanych. Zawsze więc da się wybrać sposób odpowiadający konkretnym potrzebom. Całkiem często wystarcza coś zupełnie zwyczajnego. Na przykład przewidujemy okresowe podłączenie naszego urządzenia do aplikacji PC w celu wprowadzenia nastaw konfiguracyjnych, przeprowadzenia kalibracji itp. Od strony programowej sprawę rozwiązuje nam schemat komunikacji *master<–>slave* (aplikacja przesyła do urządzenia ramkę danych o określonej długości i zawartości, a potem czeka na analogicznie sformatowaną odpowiedź). Wykrycie kompletności bloku danych odbywa się tutaj na podstawie liczby odebranych lub wysłanych bajtów, nie ma także problemów czasowych z obróbką bloku (mamy pewność, że w jej trakcie nie nadejdzie następny blok, co pozwala nam uniknąć dodatkowego buforowania np. przy użyciu typowego bufora kołowego). Jeśli zależy nam na zwiększonej odporności na błędy umieszczamy na końcu ramek danych sumę kontrolną ramki. Może to być zwykła suma modulo 8 lub 16 ale *avr–libc* dostarcza nam gotową funkcję wyliczania CRC, więc użycie tego silnego i niezawodnego sposobu jest całkiem niekłopotliwe. Np. przykład współpraca aplikacji avr–gcc z programem Object Pascal (Delphi) wygląda następująco:

```
void SetTxCrc(void)
{
 uint CrcCalcValue;
 int j;

 CrcCalcValue = 0xffff;
  for (j=0;j<(TX_SIZE-2);j++)
    CrcCalcValue = _crc16_update(CrcCal-
cValue,TxBuffer[j]);
 memcpy((uchar*)&TxBuffer[TX_SIZE-
-2],(uchar*)&CrcCalcValue,2);
}

bool CheckRxCrc(void)
{
 uint CrcCalcValue;
 uint CrcRcvValue;
 int j;

 CrcRcvValue = (uint)(RxBuffer[RX_SI-
ZE-1] <<8) + RxBuffer[RX_SIZE-2];
 CrcCalcValue = 0xffff;
  for (j=0;j<(RX_SIZE-2);j++)
    CrcCalcValue = _crc16_update(CrcCal-
cValue,RxBuffer[j]);
  return(CrcRcvValue == CrcCalcValue);
}
```

Funkcje *avr–gcc* operują bezpośrednio na buforach *TxBuffer* oraz *RxBuffer* o stałych rozmiarach ramki *TX_SIZE* oraz *RX_SIZE*. A tak to samo realizuje Delphi po stronie PC:

```
function TRejTempForm.Crc16(Abuffer:
String): Word;
var
  crc:Word;
  i,bit:Integer;
begin
   crc:=$ffff;
   for i:= 1 to Length(Abuffer) - 2 do
begin
     crc:=crc xor Ord(Abuffer[i]);
     for bit:=0 to 7 do
       if odd(crc) then
       crc:=(crc shr 1) xor $a001
       else
       crc:=crc shr 1;
   end;
   Result:=crc;
end;

function TRejTempForm.GetRxCrc: Boole-
an;
var
  cr16:Word;
begin
   Move(RxBuffer[RX_SIZE - 1],cr16,2);
   Result:= (cr16=Crc16(RxBuffer));
end;

procedure TRejTempForm.SetTxCrc;
var
  cr16:Word;
begin
   cr16:=Crc16(TxBuffer);
   Move(cr16,TxBuffer[TX_SIZE - 1],2);
end;
```



Rys. 29. Rozmieszczenie tekstów komunikatów w pamięci RAM

Zwróćmy uwagę na jeszcze jeden szczegół prezentowanego przykładowego projektu. Otóż łańcuchy znakowe (stringi) poszczególnych odpowiedzi ulokowane są w pamięci RAM mikrokontrolera. Dotyczy to również tablicy zawierającej wskaźniki na te łańcuchy (możemy to obejrzeć w oknach podglądu zmiennych oraz pamięci w AvrStudio – **rys. 29**). W praktyce takie przeznaczone tylko do odczytu zasoby przechowujemy raczej w pamięci programu Flash – aby nie marnować cennej (dużo mniejszej) przestrzeni RAM. Jednak avr–gcc ma dość specyficznie (w porównaniu z komercyjnymi kompilatorami) rozwiązaną obsługę dostępu do obszarów Flash oraz EEPROM. Dlatego wrócimy do tej sprawy w oddzielnym odcinku poświęconym specjalnie pamięciom AVR.

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.