C dla mikrokontrolerów 8051

część 6

Programowa obsługa synchronicznych interfejsów szeregowych: SPI, I²C i 1-Wire z pewnością zainteresuje wielu projektantów systemów mikroprocesorowych. Jak przygotować procedury obsługi dla tych interfejsów w języku C, dowiecie się z artykułu.

Programowa obsługa SPI w języku C

Popularność synchronicznego interfejsu szeregowego SPI (*Serial Peripherial Interface*) rośnie. Pozwala on na szybką komunikację pomiędzy układami bez angażowania dużej liczby wyprowadzeń, co jest bardzo istotne w nowoczesnych aplikacjach mikrokontrolerów.

W tym interfejsie dane są wysyłane szeregowo przez wyjście MOSI i równocześnie odbierane przez wejście MISO (również szeregowo). Sygnał SCK synchronizuje transmisję, ustawianie stanu wyjścia i próbkowanie stanu wejścia. Sygnał selekcyjny *slave SS* umożliwia wybór układu, z którym następuje wymiana danych. Linie MISO i MOSI są jednokierunkowe. Inaczej jest z linią SCK. Jej stan powinien być badany przed wysłaniem danych. W momencie tego „badania" musi być ona ustawiona jako wejściowa. Następnie jest ustawiana jako wyjściowa, ponieważ układ master musi wysyłać sygnał zegarowy synchronizujący transmisję. Stosując konfigurację z pojedynczym układem *master*, także wyprowadzenie SCK może być zwykłym wyjściem.

Programowanie interfejsu SPI zaprezentujemy na przykładzie układu procesora AT89S8252 z rodziny MCS51. Maksymalna szybkość transmisji jego interfejsu wynosi 1,5 Mbd. Istnieje możliwość jej zmiany przez ustawienie odpowiednich bitów rejestrów kontrolnych określających nastawy wewnętrznego preskalera. Do ustawiania trybów pracy oraz kontroli statusu interfejsu SPI służą trzy ośmiobitowe rejestry. Funkcje i znaczenie poszczególnych bitów są identyczne jak w mikrokontrolerach z rodziny AVR. Są to rejestry:

- rejestr kontrolny SPCR (dostępny w obszarze rejestru funkcji specjalnych pod adresem D5H),
- rejestr statusu SPSR (jak wyżej, pod adresem AAH),
- rejestr danych SPDR (jak wyżej, pod adresem 86H).

Po ustawieniu bitu SPE (*SPI Enable*) w rejestrze SPCR, wyprowadzenia portu P1 mikrokontrolera pełnią rolę linii interfejsu SPI i podłączane są odpowiednio: P1.4 jako SS, P1.5 - MOSI, P1.6 - MISO, P1.7 - SCK.

Znaczenie poszczególnych bitów w rejestrze kontrolnym **SPCR** jest następujące:

SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

SPIE - ustawienie tego bitu oraz bitu ES w rejestrze IE powoduje zezwolenie na generowanie przerwań przez interfejs SP. Generowany jest wektor przerwania o numerze 5 (23H).

SPE - ustawienie bitu SPE powoduje załączenie interfejsu SPI oraz skonfigurowanie odpowiednich wyprowadzeń portu P1 jako linii interfejsu.

DORD - ustawienie bitu DORD powoduje, że jako pierwszy wysłany (przyjęty) zostanie najmniej znaczący bit słowa, natomiast wyzerowanie - najbardziej znaczący.

MSTR - ustawienie bitu powoduje, że interfejs SPI pracuje w trybie master. Jego wyzerowanie powoduje przejście do trybu *slave*.

CPOL - polaryzacja sygnału zegarowego. Wartość tego bitu określa, czy aktywne (rozpoczynające transmisję) zbocze sygnału zegarowego SCK jest narastające (zmiana SCK z „0" na „1" - SCK jest „0", gdy SPI nie pracuje), czy opadające (zmiana SCK z „1" na „0" -

List. 4. Wysłanie bajtu danych, SPI w trybie *master*. Flaga kolizji nie jest badana

```
#include <reg8252.h>
unsigned char spistatus;        //zmienna pomocnicza
sbit   SS = P1^4;               //linia portu sygnału SS

//obsługa przerwania interfejsu SPI (23H)
void SPI_Irq(void) interrupt 5
{
   spistatus = 0;
}

//inicjowanie trybu master interfejsu SPI
void SPI_Master(void)
{
   while (SS != 1);             //oczekiwanie na zwolnienie linii SS
   SPCR = 0xF8;                 //ustalenie trybu pracy SPI
}

//wysłanie bajtu przez interfejs SPI
void SPI_Send(unsigned char x)
{
   SPIE = SPE = 0;              //wyłączenie przyjmowania przerwań
   SPI_Master();                //ustalenie trybu master dla SPI
   SS = 0;                      //zerowanie linii wyboru slave SS
   spistatus = 1;               //znak nie został wysłany, spistatus=1
   SPIE = SPE = 1;              //zezwolenie na przyjmowanie przerwań
   SPDR = x;                    //zapis bajtu do rejestru danych SPI
   while (spistatus == 1);      //oczekiwanie na zakończenie wysyłania bajtu danych
}
```

List. 5. Przykład procedury odbierania znaków przez interfejs SPI w trybie *slave*

```
#include <reg8252.h>
sbit SS = P1^4;                 //deklaracja linii SS
unsigned char counter;          //licznik odebranych bajtów
unsigned char buferror;         //błąd przepełnienia bufora SPI
unsigned char idata bufor[20]; //bufor odebranych znaków (20 bajtów)
unsigned char ptr = *bufor;     //wskaźnik do bufora

//procedura obsługi przerwania interfejsu SPI (23H) napisana tylko jako
//przykład programowania!
void SPI_Irq(void) interrupt 5
{
   ptr* = SPDR;                 //zapis odebranego znaku do bufora
   ptr++;                       //następna pozycja
                                //błąd, gdy odebrano zbyt dużo znaków
   if (counter++ >20) buferror = 1;
}

//ustawienie trybu slave dla SPI
void SPI_Slave(void);
{
   SPCR = 0xE8;                 //bit MSTR = 0
}

//odbiór znaków
void SPI_Receive(void)
{
   SPIE = SPE = 0;              //wyłączenie przyjmowania przerwań
   buferror = counter = 0;      //zerowanie licznika oraz flagi błędu
   ptr = &bufor;                //przypisanie wskazań na początek
                                //bufora odbioru znaków
   SS = 1;                      //zwolnienie SS (na wszelki wypadek)
   SPI_Slave();                 //załączenie trybu slave
   SPIE = SPE = 1;              //zezwolenie na przerwania SPI
   ...
}
```

SCK jest „1“, gdy SPI nie pracuje), jak zestawiono poniżej.

Stan bitu CPOL	Zbocze sygnału SCK inicjujące transmisję	Zbocze sygnału SCK kończące transmisję
CPOL = 0	narastające	opadające
CPOL = 1	opadające	narastające

CPHA - faza sygnału zegarowego. Wartość tego bitu określa moment próbkowania i wyprowadzania informacji poprzez interfejs SPI, jak pokazano poniżej.

Wartość bitu CPHA	Pierwsze zbocze SCK (odpowiednio do CPOL)	Drugie zbocze SCK (odpowiednio do CPOL)
CPHA = 0	próbkowanie MISO	ustawianie MOSI
CPHA = 1	ustawianie MOSI	próbkowanie MISO

SPR1, SPR0 - bity preskalera sygnału zegarowego służące do ustawiania prędkości transmisji interfejsu SPI. Nastawy bitów dotyczą tylko pracy interfejsu w trybie *master*. Częstotliwość sygnału SCK = F_{OSC}/n, przy czym ma wartości podane poniżej.

SPR1	SPR0	Współczynnik podziału n
0	0	4
0	1	16
1	0	64
1	1	128

Uwaga: maksymalna częstotliwość sygnału SCK dla AT89S8252 wynosi 1,5 MHz.

Znaczenie poszczególnych bitów rejestru statusu **SPSR** (AAh) jest następujące:

SPIF WCOL - - - - - -

SPIF - ustawienie SPIF oznacza skompletowanie słowa odbieranego/nadawanego poprzez interfejs SPI i wygenerowanie przerwania. Bit ten jest zerowany podczas odczytu rejestru statusu oraz dostępu do rejestru danych SPI.

WCOL - bit jest ustawiany, jeśli nastąpiła próba zapisu danych podczas trwającej transmisji SPI. Bit jest zerowany wraz z bitem SPIF.

Obsługa sprzętowego interfejsu SPI jest bardzo łatwa. Można ją zrealizować co najmniej dwoma sposobami: wykorzystując przerwanie generowane przez interfejs SPI lub testując bit SPIF podczas transmisji.

Prezentowane poniżej przykłady programów napisanych w języku C wykorzystują mechanizm przerwań. Jest to bardzo korzystne, ponieważ wygenerowanie przerwania upewnia nas, że dane zostały wysłane lub odebrane. Nie ma więc potrzeby wykonywania dodatkowych testów.

Programowa obsługa I²C w języku C

Interfejs I²C pozwala na komunikację wielu różnych układów poprzez dwuprzewodową magistralę. Dwie dwukierunkowe linie, SDA i SCL, służą do transmisji danych pomiędzy układami. Linia SDA nazywana jest linią danych, natomiast linia SCL linią zegara. Wszystkie układy dołączane są do nich bezpośrednio - nie są wymagane żadne obwody sprzęgające (ewentualnie wyprowadzenia odpowiednich linii mogą być dołączone przez rezystor). Układy wyposażone w interfejs I²C mogą pracować w jednym z dwóch trybów:

- ***master***, w którym układ przejmuje kontrolę nad magistralą i przebiegiem transmisji danych, wysyła sygnały *start* (ujemne zbocze na SDA przy wysokim poziomie na SCI) i *stop* (dodatnie zbocze na SDA przy wysokim poziomie na SCI), generuje także sygnał zegarowy,
- ***slave***, w którym funkcje układu są kontrolowane przez urządzenie *master*.

System zbudowany z wykorzystaniem magistrali I²C posiada jeden mikrokontroler pracujący w trybie *master* oraz wiele układów *slave*. Każdy z układów *slave* musi mieć własny unikatowy adres. Mikrokontroler odczytując lub zapisując dane, inicjuje transmisję, posługując się adresem konkretnego układu.

Niektóre z układów dołączonych do magistrali I²C mogą nie tylko odczytywać, lecz również

Tab. 2. Zestawienie trybów pracy interfejsu SPI i przyporządkowanie im wartości bitów CPOL i CPHA

Stan bitów CPOL i CPHA	Pierwsze zbocze SCK	Drugie zbocze SCK	Tryb pracy SPI
CPOL = 0, CPHA = 0	próbkowanie MISO	ustawianie MOSI	0
CPOL = 0, CPHA = 1	ustawianie MOSI	próbkowanie MISO	1
CPOL = 1, CPHA = 0	próbkowanie MISO	ustawianie MOSI	2
CPOL = 1, CPHA = 1	ustawianie MOSI	ustawianie MOSI	3

wysyłać dane. Pracują więc jako nadajniki albo jako odbiorniki danych. Należą do nich na przykład układy pamięci lub przetworników analogowo-cyfrowych. Możliwość wysyłania danych nie oznacza, że dany układ pracuje jako *master*. Układ przesyłający nie musi bowiem przejmować kontroli nad transmisją - korzysta z sygnałów sterujących występujących na magistrali I²C. Układem *master* jest każdorazowo ten, który inicjuje transmisję, wysyłając sygnał *start*, a przerywa ją, wysyłając sygnał *stop* oraz generuje sygnał zegara.

Transmisja jest synchronizowana sygnałem SCK, który może mieć częstotliwość z zakresu 0...100/400/3400 kHz. Nie są określone minimalne wartości szybkości transmisji, a więc i częstotliwości sygnału zegarowego. Inaczej jest dla wartości maksymalnej. Ta jest określona jako 100 kbd w trybie *standard*, 400 kbd w trybie szybkim oraz 3,4 Mbd w trybie *high speed*.

Chociaż typową jest konfiguracja z jednym układem *master*, to możliwe jest również dołączenie wielu układów pracujących w tym trybie do jednej magistrali. Stosowana jest wówczas odpowiednia procedura arbitrażu w celu rozstrzygnięcia, który z układów *master* przejmie w danej chwili kontrolę nad magistralą, a który musi czekać. Zwykle arbitraż nie jest potrzebny, ponieważ transmisja danych może być zainicjowana tylko wtedy, gdy magistrala jest wolna. Jednak może zdarzyć się tak, że dwa układy *master* dołączone do tych samych linii interfejsu nieomal jednocześnie wygenerują sygnał *startu*. Wówczas arbitraż staje się niezbędny.

Na **list. 6** pokazano, jak można napisać procedurę w języku C obsługi pojedynczego układu *master*. Jest to przykład procedury, która może przydać się wówczas, gdy do magistrali I²C podłączone są na przykład układy: pamięć EEPROM, przetwornik analogowo-cyfrowy, układ zegara czasu rzeczywistego i pojedynczy mikrokontroler korzystający z ich danych.

Funkcja *Delay()* występująca w różnych miejscach procedury wstrzymuje pracę mikrokontrolera na około 5..6 µs. W przypadku mikrokontrolera z rodziny 8051 pracującego z zegarem 10...12 MHz, wystarczy 6 instrukcji NOP.

List. 6. Przykład realizacji interfejsu I²C dla pojedynczego układu *master*

```
/*********************************
pojedynczy układ master I2C
RC-51
*********************************/
#include <reg51.h>
sbit   SDA P1^0;             //definiowanie połączeń I2C
sbit   SCL P1^1;

//wysłanie sekwencji START, bez badania stanu zajętości I2C
void I2C_Start(void)
{
   SDA = SCL = 1;    Delay();
   SDA = 0;   Delay();
   SCL = 0;
}

/* wysłanie sekwencji STOP, funkcja zwraca stan linii SDA po zakończeniu;
 jeśli układ slave wymusza stan niski SDA - błąd transmisji */
bit I2C_Stop(void)
{
   SDA = 0;   Delay();
   SCL = 1;   Delay();
   SDA = 1;   Delay();
   return (~SDA);
}

/* odczyt bajtu z magistrali I2C; jako parametr wywołania podawany jest bit ACK,
  ponieważ niektóre bajty odczytywane są z NACK */
unsigned char I2C_Read(bit ack)
{
   unsigned char bitCount = 8, temp;

   SDA = 1;
   do
   {
       Delay();
       SCL = 1;
       Delay();
       temp <<= 1;           //wprowadzenie 0
       if (SDA) temp++;      //ustawienie bitu, gdy SDA jest wysoki
       SCL = 0;
   } while(--bitCount);

   SDA = ack;                //wysłanie bitu ACK
   Delay();
   SCL = 1;
   Delay();
   SCL = 0;
   return (temp);
}

/* wysłanie słowa na magistralę I2C, funkcja zwraca stan bitu ACK */
bit I2C_Send(unsigned char byte)
{
   unsigned char bitCount = 9;
   bit temp;

   do
   {
       SCL = 0;
       SDA = byte & 0x80;    // stan SDA jako wynik iloczynu bitowego
       byte = (byte << 1) + 1;
       Delay();
       SCL = 1;
       Delay();
   } while(--bitCount);

   temp = SDA;
   SCL = 0;
   return (temp);            // ACK = 0, NACK = 1
}
```

Programowa obsługa interfejsu 1-Wire w języku C

Interfejs *1-Wire* (pamiętam swoją reakcję na pierwsze artykuły w EP na ten temat) zyskuje coraz więcej zwolenników. Transmisja danych z jego użyciem nie jest może oszałamiająco szybka, lecz możliwość dołączenia czujników temperatury czy układów sterowania światłem, za pomocą tylko jednego przewodu dostarczającego sygnał i zasilanie (oczywiście konieczna jest również masa, ale może być zrealizowana na wiele różnych sposobów), może być bardzo atrakcyjna w wielu zastosowaniach.

List. 7. Przykład programowej realizacji interfejsu 1-Wire

```
/*   Biblioteka procedur obsługi magistrali 1W Raisonance RC-51   */
#include <reg51.h>
#define MATCH_ROM       0x55       //definicje komend 1W
#define SKIP_ROM        0xCC
#define SEARCH_ROM      0xF0
#define SEARCH_FIRST    0xFF
#define PRESENCE_ERR    0xFF
#define DATA_ERR        0xFF
#define LAST_DEVICE     0x00       //0x00: znaleziono urządzenie
                                   //0x01...0x40: kontynuacja poszukiwania
#define   XTAL   110592            //definicja rezonatora (8..25MHz!)
#define   nop()  ACC++             //opóźnienie, 1 cykl maszynowy

sbit  one_wire_IO = P1^0;          //definicja podłączenia linii portu 1W

//wysłanie polecenia "1W Reset"
bit one_wire_reset(void)
{
   unsigned char delay;
   bit err;
                                   //pętla opóźnienia 480 < t < 960 cykli
   delay = (unsigned char)(XTAL/12e6*480/4);
   do                              //pętle opóźniające służą do wytworzenia
   {                               //tzw. time slots, których dokładny opis
       one_wire_IO = 0;            //można znaleźć w opisie standardu 1W
       nop();
   } while(--delay);
                                   //pętla opóźnienia 60 < t < 75 cm.
   delay = (unsigned char)( XTAL / 12e6 * 66 / 2 );
   do
   {
       one_wire_IO = 1;
   } while(--delay);
   err = one_wire_IO;              //stan niski oznacza, że urządzenie(a)
                                   //1W jest(są) podłączone
                                   //opóźnienie 480 < t
   delay = (unsigned char)(XTAL/12e6*(480-66)/4);
   do
   {
       nop();  nop();
   } while(--delay);
   err = one_wire_IO;
   return !err;                    //stan niski linii portu 1W oznacza błąd
}

//przesłanie lub odczyt bitu przez linię 1W
bit one_wire_bit_io(bit b)
{
   unsigned char delay;

   delay = (unsigned char)(XTAL/12e6*15/2-2);//15 > t
   one_wire_IO = 0;          //1
   one_wire_IO = b;          //3
   while(--delay);           //3 + delay * 2
   b = one_wire_IO;
   delay = (unsigned char)(XTAL/12e6*45/2);  //60 < t
   while(--delay);
   one_wire_IO = 1;
   return b;
}

//przesłanie bajtu przez linię 1W
unsigned char one_wire_byte_write(unsigned char b)
{
   unsigned char bit_counter = 8;

   do
   {
       b = b >> 1 | (one_wire_bit_io( b & 1 ) ? 0x80: 0);
   } while(--bit_counter);
   return b;
}

//odczyt bajtu z linii 1W
unsigned char one_wire_byte_read( void )
{
 return one_wire_byte_write(0xFF);
}

//wysłanie komendy (komend) do urządzenia 1W; lista 8 bajtów do wysłania
//wskazywana przez ptr wypełnienie tablicy wskazywanej przez ptr oznacza,
//że komendy dotyczą konkretnego urządzenia podłączonego do 1W; wówczas to
//tablica powinna zawierać jego identyfikator
void one_wire_send_command(unsigned char command, unsigned char *ptr)
{
   unsigned char byte_count = 8;

   one_wire_reset();
   if(ptr)
   {
       one_wire_byte_write(MATCH_ROM); //komendy przesyłane do urządzenia
       do
       {
           one_wire_byte_write(*ptr);
           ptr++;
       } while(--byte_count);
   }
   else
       one_wire_byte_write(SKIP_ROM);  //ptr = null, komenda dla wszystkich
                       //urządzeń
   one_wire_byte_write(command);
}
```

Pierwotnie interfejs był przeznaczony do komunikacji na bardzo małe odległości. Służył do podłączenia na przykład układu zewnętrznej pamięci do mikrokontrolera z użyciem tylko jednego wyprowadzenia. Wkrótce jednak projektanci zaczęli domagać się rozwiązań umożliwiających podłączenie układów znajdujących się w znacznie większej odległości. Wówczas to opracowano zupełnie nowe protokoły transmisji danych uwzględniające większe długości połączeń, pracę w sieci i mechanizmy kontroli przesyłanych danych.

Podobnie jak dla większości interfejsów szeregowych, również i dla *1-Wire* transmisja przebiega w konfiguracji *master-slave*. Układ *master* wyszukuje i adresuje układ *slave* oraz steruje przesyłaniem danych. Dane przesyłane są synchronicznie z prędkością do 16,3 kbd (w trybie *overdrive* do 115 kbd). W dużym uproszczeniu można powiedzieć, że każde opadające zbocze sygnału inicjuje i synchronizuje przesyłanie bitu. Podobnie jak przy transmisji I^2C, nie są określone dolne granice wartości częstotliwości sygnału synchronizującego. Pozwala to na łatwą implementację programów obsługi transmisji w standardzie *1-Wire*. Nie jest bowiem wymagany dokładny pomiar czasu impulsów, jak przy programowej obsłudze interfejsu UART.

Na **list. 7** zastosowano następującą instrukcję:

```
b = b >> 1 | (one_wire_bit_io( b &
1 ) ? 0x80: 0);
```

Użyto w niej operatora warunkowego. Działa on w ten sposób, że najpierw wyznaczana jest wartość pierwszego wyrażenia *one_wire_bit_io(b & 1)*. Jeśli wartość tego wyrażenia jest różna od 0, to wynikiem jest wartość wyrażenia drugiego - 0x80, w przeciwnym wypadku trzeciego - 0.

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

Dodatkowe informacje

Wszystkie programy prezentowane w artykule kompilowano za pomocą ewaluacyjnej wersji pakietu RIDE firmy Raisonance (www.raisonance.com).

Krajowym dystrybutorem tej firmy jest Eurodis Microdis, www.microdis.net, tel. (71) 301-04-00.

Ewaluacyjną wersję pakietu firmy Raisonance dla mikrokontrolerów '51 zamieściliśmy na CD-EP8/2002B.