C dla mikrokontrolerów 8051

część 13

Pisząc program w języku C, czasami zadajemy sobie pytanie: czy naprawdę muszę tworzyć funkcję dokonującą konwersji wartości dziesiętnej na szesnastkową? Przecież chociażby biblioteka o nazwie STDIO zawiera w sobie możliwość formatowania zarówno danych wejściowych, jak i wyjściowych. Czy nie ma możliwości skorzystania z czyjejś pracy i zaoszczędzenia własnego czasu?

printf() - jak korzystać z tej funkcji?

Kilka słów o STDIO.H

Biblioteka o nazwie STDIO.H (*STanDard Input-Output*) zawiera szereg funkcji umożliwiających odczyt i wyprowadzanie znaków do (z) standardowego urządzenia wejścia/wyjścia. W „dużym" komputerze role tych urządzeń spełniają klawiatura i monitor. W przypadku mikrokontrolera przyjęto, że funkcje STDIO wykorzystują interfejs szeregowy UART (po konwersji poziomów napięć wyjściowych - RS232), traktując go jako standardowe urządzenie do komunikacji z użytkownikiem.

W przypadku kompilatora RC-51 (*www.raisonance.com*) nastawy UART dokonywane są tuż po uruchomieniu napisanej dla mikrokontrolera aplikacji. Zajmuje się tym funkcja _C_INIT_IO. Ustawia ona TIMER1 w mikrokontrolerze ë51 jako generator sterujący transmisją, nadając jego rejestrowi TH1 predefiniowaną wartość początkową. Domyślnie jest to 0xE8, co odpowiada prędkości transmisji 1200 bps przy częstotliwości zegara 11,0592 MHz. Wartość tę można zmienić, używając polecenia *#pragma* (na przykład polecenie *#pragma DEFJ(TIM1_INIT=0xFD)* przy tej samej częstotliwości generatora zegarowego, ustawi prędkość transmisji na wartość 19200 bps). Ale jeśli byłyby to tylko i wyłącznie instrukcje wysyłania oraz odbioru znaków, nie warto by było poruszać tego tematu. Istnieje bowiem mnóstwo dobrych opracowań na temat bibliotek wykorzystywanych przy programowaniu w języku C.

Funkcje STDIO.H predefiniowane przez Raisonance, producenta pakietu RC-51

```
extern int _getkey(void);
extern int getchar (void);
extern char ungetchar (char c) reentrant;
extern char *gets (char *s) reentrant;
extern int putchar (const int c|);
extern int puts (const char *s ) reentrant;
extern int printf(const char *format,...) reentrant;
extern int sprintf(char *buffer, const char *format,...) reentrant;
extern int scanf(const char *format,...) reentrant;
extern int sscanf(const char *buffer, const char *format,...) reentrant;
```

printf() = formatowane wyjście

Każdy, kto kiedykolwiek wykorzystywał funkcje predefiniowane w STDIO.H wie, że umożliwiają one formatowanie danych. Zgodnie ze specyfikacją standardu ANSI, szereg z nich dokonuje przekształceń wewnętrznych wartości na znaki lub odwrotnie. W tym odcinku kursu szczególną uwagę poświęcimy funkcji *printf()* dającej programiście nie tylko szereg możliwości wykorzystania, lecz również pozwalającej na redukcję czasu koniecznego do stworzenia aplikacji. Wyjściowa funkcja *printf()* tłumaczy wewnętrzne wartości na znaki. Jednym słowem - bajty danych zamieniane są na postać zrozumiałą przez człowieka:

```
int printf(char *wzorzec,
argument_1, argument_2... )
```

Przekształcenie odbywa się według i pod nadzorem wzorca zapisanego we „wzorzec". Funkcja przekształca, formatuje i wypisuje swoje argumenty do standardowego wyjścia. Jak wspomniałem wcześniej, w przypadku mikrokontrolera 8051 jest to interfejs UART. Wzorzec zawiera obiekty dwojakiego rodzaju: zwykłe znaki, które są przesyłane do wyjścia oraz specyfikacje przekształceń. Każda z nich wskazuje na sposób, w jaki zostanie przekształcony i wypisany dany argument. Specyfikację przekształcenia rozpoczyna znak %, a kończy znak dla niego charakterystyczny. Między znakiem % i znakiem przekształcenia mogą - według następującej kolejności - wystąpić:

- znak „-" (minus) polecający dosunięcie przekształconego argumentu do lewego krańca jego pola,
- liczba określająca rozmiar pola (argument zostanie wypisany w postaci o rozmiarze co najmniej pola, a jeśli będzie taka potrzeba, zostanie uzupełniony znakami odstępu z prawej lub lewej strony w zależności od żądania dosunięcia znaków w lewo),
- znak „." (kropka) oddzielający rozmiar pola argumentu od jego precyzji,
- liczba określająca precyzję, to jest

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

Znak formatujący	Typ przekształcanego argumentu	Opis przekształcenia Przekształcenie do postaci:
d lub i	int	liczba dziesiętna ze znakiem
o	int	liczba ósemkowa bez znaku i bez wiodącego zera
x lub X	int	liczba szesnastkowa bez znaku i bez wiodącego zera z użyciem małych liter dla wzorca **0x** i dużych dla **0X**
u	int	liczba dziesiętna bez znaku
c	int	pojedynczy znak po przekształceniu do typu *unsigned char*
s	char*	tekst wypisywany do napotkania znaku końca łańcucha /0 lub osiągnięcia zadanej precyzji
f	double	liczba dziesiętna ze znakiem w postaci [-]xxx.yyy, gdzie liczba cyfr po kropce (yyy) określona jest przez precyzję
e lub E	double	liczba dziesiętna ze znakiem w tzw. notacji inżynierskiej (na przykład 3.45234e-10); podobnie jak wyżej, liczba cyfr po kropce określana jest przez precyzję
g lub G	double	jeśli wykładnik potęgi jest mniejszy od -4 lub >= precyzji, to przyjmuje się specyfikację identyczną z wzorcem *e* (*E*); inaczej stosowana jest specyfikacja *f*
p	void*	wskaźnik – reprezentacja zależy od konkretnej implementacji
n	int*	liczbę znaków wypisanych w TYM wywołaniu printf zapisuje się do odpowiedniego argumentu; nie są wykonywane żadne przekształcenia
%		nie ma przekształcenia (%%); zostanie wypisany znak %

List. 1. Przykłady użycia funkcji *printf()*

```
const char* TEKST = „Tekst przykladowy”;
printf(“:%s:”,TEKST);       :Tekst przykladowy:
printf(“:%10s:”,TEKST);     :Tekst przykladowy:
printf(“:%.10s:”,TEKST);    :Tekst przy:
printf(“:%25.s:”,TEKST);    :         Tekst przykladowy:
printf(“:%-25.s:”,TEKST);   :Tekst przykladowy        :
printf(“:%025.10s:”,TEKST); :Tekst przy               :

int X = 123;

printf(“%s %04X %s”, “123 Dec. =”, X, “Hex”);    123 Dec. = 007B Hex
printf(“%s %o %s”, “123 Dec. =”, X, “Oct”);      123 Dec. = 173 Oct
```

maksymalną liczbę znaków dla tekstu, liczbę cyfr po kropce dziesiętnej dla liczb zmiennopozycyjnych, minimalną liczbę cyfr dla wartości całkowitych,
- litera „h“, jeśli argument całkowity należy wyprowadzić w postaci *short*, lub „l“ (litera „l“) jeśli argument należy wyprowadzić jako *long*.

W **tab. 1** zestawiono podstawowe znaki przekształcenia dla funkcji *printf()*. Szerokość pola lub precyzję można w specyfikacji zastąpić znakiem „*“ (gwiazdki), co oznacza, że żądany argument należy wyprowadzić i przekształcić, korzystając z kolejnego argumentu funkcji (uwaga - musi on być typu *int*!). Na przykład, wypisanie co najwyżej *max* znaków z *S* wygląda następująco: *printf(„%.*s“, max, S);*

Stosując funkcję *printf()*, należy pamiętać, że wykorzystuje ona swój pierwszy argument do określenia typu, rozmiarów i liczby pozostałych argumentów. Jeśli programista poda zły wzorzec przekształceń, to mimo opisywanej wcześniej filozofii języka C (zaufaj programiście, on wie co robi), funkcja będzie „zdezorientowana“ i na wyjściu wyprowadzone zostaną błędne rezultaty jej pracy. Programista powinien mieć świadomość, że efekt wywołań funkcji *printf()* w postaci *printf(s)* oraz *printf(„%s“,s)* może być zupełnie odmienny, aczkolwiek kompilator języka C dopuszcza stosowanie jednej i drugiej postaci. Jeśli jednak nie podamy wzorca wyprowadzanego łańcucha, to może się okazać, że gdy w zmiennej *s* wystąpią znaki specjalne (%, *), łańcuch, który zamierzamy wyprowadzić, zostanie potraktowany jako wzorzec. Na koniec tej krótkiej prezentacji warto również wspomnieć o funkcji *sprintf()*, będącej odmianą *printf()*, lecz z tą różnicą, że nie wyprowadza ona danych, tylko zapisuje je w pamięci.

List. 2. Przykład programu zmieniającego definicję funkcji *putchar()*

```
// zamiana funkcji putchar()
// oryginalnie funkcja PUTCHAR
// wykorzystuje tylko rejestr R7
// i akumulator
// jesli ponizsza uzywa czegos wiecej
// - moze nie funkcjonowac
// nalezy uwaznie przygladac sie
// rejestrom

#include <reg51.h>
#include <stdio.h>
#include <lcd4b.h>

// zmiana definicji putchar(),
// metoda 1, mniej bezpieczna
int putchar (const int c)
{
   LcdWrite(c);
   return (0);
}

void main (void)
{
   int x = 241;
   // inicjalizacja LCD w trybie 4 bity
   LcdInitialize();
   LcdClrScr();
   // zamiana liczby x na wartosc
   // szesnastkowa
   printf(„%d %s %02x %s”, x, „dec =”,
       x, „hex”);
   // koniec programu
   while (1);
}
```

Dla praktyków - obsługa wyświetlacza LCD z wykorzystaniem funkcji *printf()*

Teraz dotarliśmy wreszcie do meritum tego artykułu. Oczywiście, chciałem w krótki sposób zaprezentować funkcje STDIO.H, jednak celem tego artykułu jest nie tyle ich prezentacja, ile wytłumaczenie metody, dzięki której można zaprząc je do pracy. Z doświadczenia wiem, że 80% tworzonych przeze mnie aplikacji nie korzysta z interfejsu UART i nic nie stoi na przeszkodzie w wykorzystaniu STDIO.H dla innych potrzeb.

Funkcja *printf()* jest zaimplementowana od szczegółu do ogółu. Co to oznacza? U podstaw jej działania leży funkcja *putchar()* wysyłająca pojedynczy znak przez UART. Funkcja *printf()* nie wie, gdzie i za pomocą jakiego interfejsu wyprowadzane są dane. Zajmuje się tym *putchar()* i to ją właśnie należy zmienić, aby znaki wysyłane były nie przez UART, ale na przykład na wyświetlacz LCD. Oczywiście, o ile UART i jego obsługa są pewnym standardem w obrębie rodziny mikrokontrolerów 8051, o tyle implementacja obsługi wyświetlacza zależy od konkretnego środowiska, w którym pracuje mikrokontroler.

W przykładzie programu pokazanym na **list. 2** dokonałem zmiany definicji *putchar()* w taki sposób, że znaki wysyłane są na wyświetlacz LCD, a nie przez UART. Wykorzystałem tu bibliotekę funkcji obsługi LCD z jednego z poprzednich odcinków kursu.

Jak widać na podstawie przykładu programu, redefinicja *putchar()* nie jest zbyt trudna do wykonania. Nagłówek funkcji musi być zgodny ze zdefiniowanym wcześniej przez producenta pakietu. Można zobaczyć jego pożądany wygląd, otwierając właściwy zbiór nagłówkowy o rozszerzeniu „H“ (np. STDIO.H). Ciało może być zestawem dowolnych instrukcji.

Tworząc redefinicje, należy zwrócić szczególną uwagę na to, jakie rejestry będą modyfikowane przez nową funkcję. Zgodnie z dokumentacją producenta (a do niej należy każdorazowo odwoływać się tworząc redefinicję) funkcja *printf()* spodziewa się, że *putchar()* modyfikuje wyłącznie zawartość rejestrów UART, R7 i ACC mikrokontrolera oraz przydzielonego na zmienne obszaru pamięci. Jeśli nowo napisana funkcja zmienia zawartość również innych rejestrów, musi być zastosowana inna metoda redefinicji, zaprezentowana w przykładzie na **list. 3**. Przed użyciem *putchar_c()* wewnątrz *putchar()*, wszystkie żywotne rejestry mikrokontrolera są zapamiętywane na stosie i odtwarzane po powrocie z wywołania funkcji. Na listingu pokazano również fragment kodu w asemblerze 8051 wykonywany podczas wywołania *putchar()*.

List. 3. Bezpieczna redefinicja *putchar()* oraz odpowiadający jej listing programu po kompilacji - zauważyć można, że wszystkie ważne rejestry zapamiętywane są na stosie przed wywołaniem putchar (PUSH) i odtwarzane po powrocie (POP)

```
#include <reg51.h>
#include <stdio.h>
#include <lcd4b.h>

// zamiana funkcji putchar(),
// metoda 2, bezpieczna
void putchar_c (const int c) reentrant
{
   LcdWrite(c);
}

int putchar (const int c)
{
   putchar_c(c);
   return(0);
}

void main (void)
{
   int x = 134;
   // inicjalizacja LCD w trybie 4 bity
   LcdInitialize();
   LcdClrScr();
   // zamiana liczby x na wartosc
   // szesnastkowa
   printf(„%d %s %02x %s”,x,”dec.to”,
       x,”hex”);
   //koniec programu
   while (1);
}


 ; FUNCTION _putchar (BEGIN)
                                           ;
SOURCE LINE # 15
0000   C0F0    PUSH   B
0002   C083    PUSH   DPH
0004   C082    PUSH   DPL
0006   C0D0    PUSH   PSW
0008   C000    PUSH   AR0
000A   C001    PUSH   AR1
000C   C002    PUSH   AR2
000E   C003    PUSH   AR3
0010   C004    PUSH   AR4
0012   C005    PUSH   AR5
0014   C006    PUSH   AR6
               ; Register R4R5 is
               ; assigned to parameter c
0016   120000 R LCALL ?putchar_c
0019   D006    POP    AR6
001B   D005    POP    AR5
001D   D004    POP    AR4
001F   D003    POP    AR3
0021   D002    POP    AR2
0023   D001    POP    AR1
0025   D000    POP    AR0
0027   D0D0    POP    PSW
0029   D082    POP    DPL
002B   D083    POP    DPH
002D   D0F0    POP    B
002F   22      RET
```

Słowo kluczowe *reentrant* języka RC-51 informuje kompilator o tym, że funkcja może być wywoływana przez wiele procesów jednocześnie.

Inny przykład redefinicji *putchar* (tu wykorzystano również metodę mniej bezpieczną) pokazano na **list. 4**. Oryginalnie (i zgodnie ze specyfikacją standardu ANSI C) *putchar* wysyła po każdym argumencie o wartości 0x0A znak o kodzie 0x0D. Tworzą one w sumie sekwencję składającą się na znak nowej linii (powrót karetki - CR=0x0D oraz znak nowej linii - LF=0x0A). W niektórych aplikacjach jest to jednak cecha niepożądana, a wręcz przeszkadzająca. Nowa definicja funkcji *putchar* nie posiada już tej właściwości.

Przedstawione tu przykłady tworzenia własnych funkcji zamieniających oryginalne definicje to wierzchołek góry. Istnieje bowiem cały szereg różnych możliwości - począwszy od bibliotek obsługi standardowego wejścia - wyjścia aż po bibliotekę MATH (operacje matematyczne na liczbach zmiennopozycyjnych). Wszystko zależy od inwencji programisty i od faktycznych potrzeb aplikacji. Wykorzystując biblioteki, należy jednak pamiętać o tym, że oferują one szereg różnych możliwości kosztem zajętej pamięci programu mikrokontrolera.

List. 4. Przykład własnej definicji *putchar()*

```
//nowa definicja funkcji putchar
//wysylajaca dane przez UART
int putchar (const int c)
{
   SBUF = c;
   TI = 0;
   while (!TI);
}
```

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