Obsługa kart pamięci Flash za pomocą mikrokontrolerów, część 7
Karty MultiMedia Card (MMC)

Po ostatniej dawce teorii opisującej tym razem karty MMC (EP8/2004), przyszedł czas na konkrety. W tej części artykułu przedstawię Czytelnikom przykładowe procedury umożliwiające komunikację z kartami MMC przy użyciu mikrokontrolera Atmega 162.

Przedstawione procedury napisano w języku C. Są one przeznaczone do skompilowania za pomocą kompilatora AVR-GCC (WIN-AVR).

Na **list. 1** (plik *mmc.h*) pokazane są deklaracje wszystkich komend dostępnych w trybie SPI, opis bitów potwierdzenia typu R1 oraz deklaracje prototypów funkcji. Ze względu na długość generowanego kodu, przykładowe procedury są dość mocno uproszczone i dlatego wystarczy nam odbiór potwierdzeń typu R1. Znajdujące się pod koniec listingu deklaracje typów u08, u16 i u32 mają na celu ułatwienie pisania programu, bo szybciej jest napisać „*u16*" niż „*unsigned short*", a na dodatek jasno informują, że dany typ jest *unsigned* o długości 16 bitów.

List. 1. Deklaracje komend SPI oraz deklaracje prototypów funkcji

```
// Komendy dostępne w trybie SPI
#define MMC_GO_IDLE_STATE            0
#define MMC_SEND_OP_COND             1
#define MMC_SEND_CSD                 9
#define MMC_SEND_CID                10
#define MMC_SEND_STATUS             13
#define MMC_SET_BLOCKLEN            16
#define MMC_READ_SINGLE_BLOCK       17
#define MMC_WRITE_BLOCK             24
#define MMC_PROGRAM_CSD             27
#define MMC_SET_WRITE_PROT          28
#define MMC_CLR_WRITE_PROT          29
#define MMC_SEND_WRITE_PROT         30
#define MMC_TAG_SECTOR_START        32
#define MMC_TAG_SECTOR_END          33
#define MMC_UNTAG_SECTOR            34
#define MMC_TAG_ERASE_GROUP_START   35
#define MMC_TAG_ERARE_GROUP_END     36
#define MMC_UNTAG_ERASE_GROUP       37
#define MMC_ERASE                   38
#define MMC_CRC_ON_OFF              59

// Odpowiedzi
#define R1_BUSY                    128
#define R1_PARAMETER                64
#define R1_ADDRESS                  32
#define R1_ERASE_SEQ                16
#define R1_COM_CRC                   8
#define R1_ILLEGAL_COM               4
#define R1_ERASE_RESET               2
#define R1_IDLE_STATE                1

#ifndef MMC_ASM
//
// Deklaracje typów (skrótów)
//
typedef unsigned char  u08;
typedef unsigned short u16;
typedef unsigned long  u32;

//
// Prototypy funkcji
//
u08 mmc_reset(void);
u08 mmc_read_sector(u32 sector);
u08 mmc_write_sector(u32 sector);
u32 mmc_capacity(void);
u08 mmc_get_cid(void);
#endif
```

Na **list. 2** pokazano wszystkie niezbędne procedury umożliwiające komunikację z kartą MMC. Najważniejszą z nich jest funkcja *mmc_cmd*, która wysyła komendę do karty oraz odbiera z niej potwierdzenie. Jako parametry funkcji podajemy numer komendy, jej argument w postaci 32-bitowej zmiennej oraz stałą *Finish* lub *Leave*, od której zależy, czy po wykonaniu komendy będą przesyłane jeszcze jakieś dane (*Leave*) i należy pozostawić aktywną linię CS lub że komenda nie wymaga dodatkowych danych i funkcja ma zakończyć transakcje z kartą, czyli dezaktywować linie CS i wysłać 8 impulsów zegarowych na linię CLK. Po wysłaniu bajtu komendy wysyłane są 4 bajty argumentu, a następnie bajt CRC. Jak już wcześniej pisałem, jako bajt CRC komendy wysyłana jest wartość 0x95, która jest prawidłowym CRC wyliczonym dla CMD0 wraz z argumentem o wartości 0. Prawidłowej wartości bajtu CRC potrzebujemy tylko raz, w momencie przełączenia karty z trybu MMC na tryb SPI, czego dokonuje komenda CMD0 (*GO_IDLE_STATE*) podczas inicjalizacji karty. Następnie funkcja próbuje dziesięciokrotnie odebrać potwierdzenie z karty i zwraca je jako wynik działania funkcji. W przypadku nieodebrania potwierdzenia zwracana jest wartość 0xFF, co oznacza, że karta nie odpowiedziała na komendę.

Funkcja *mmc_reset* najpierw odpowiednio konfiguruje linie wejść-wyjść mikrokontrolera wykorzystywane do komunikacji z kartą, inicjuje sprzętowy interfejs SPI, przełącza kartę w tryb komunikacji SPI, a następnie oczekuje na gotowość karty. Dodatkowo ustawia ona

rozmiar bloku na 512 bajtów, co dla większości kart nie jest konieczne, ale nie zaszkodzi.

Funkcje *mmc_read_sector* i *mmc_write_sector* umożliwiają odczyt i zapis pojedynczego 512-bajtowego bloku. Jako parametr podajemy numer bloku (sektora), a dane są przenoszone poprzez 512-bajtowy bufor *mmc_sbuf* umieszczony w pamięci RAM mikrokontrolera. Najpierw wysyłana jest odpowiednia komenda, której argumentem jest adres pierwszego bajtu danych do odczytu/zapisu. Jako że do funkcji przekazujemy numer bloku, a argumentem funkcji odczytu i zapisu musi być adres pierwszej komórki danego bloku, to przed przekazaniem jako parametr komendy, ów numer bloku jest mnożony przez 512 (poprzez przesunięcie w lewo o 9 bitów). W przypadku odczytu, po wykonaniu komendy wywoływana jest pomocnicza funkcja czekająca na odebranie bajtu o wartości 0xFE, czyli na *data token* lub na mogący się pojawić *data error token*. Przy zapisie postępujemy odwrotnie, czyli wysyłamy do karty *data token* poprzedzony jednym pustym bajtem – co wynika z zależności czasowych opisanych w poprzednim odcinku kursu. Następnie odbieramy lub wysyłamy 512 bajtów danych, a następnie 2 bajty CRC, których wartość jest ignorowana. Procedura odczytu jest w tym momencie kompletna i można zakończyć transakcję z kartą

List. 2. Listing procedur wykorzystywanych do komunikacji z kartą MMC

```
#include <avr/io.h>
#include „mmc.h”

u08 mmc_sbuf[512];  // bufor sektora

#define MMC_PORT  PORTB
#define MMC_DDR   DDRB
#define MMC_CS    PB0

#define Finish 1
#define Leave 0

// ***********************************************************
//  Procedury pomocnicze
// ***********************************************************

void spi_init(void)                  // inicjalizacja interfejsu SPI
{
   DDRB |= (1<<DDB3) | (1<<DDB5) | (1<<DDB2);
   SPCR = (1<<SPE) | (1<<MSTR);      // SPI master
}


u08 spi_tx_rx(u08 byte)              // wysłanie i odbiór bajtu przez SPI
{
   SPDR = byte;
   loop_until_bit_is_set(SPSR, SPIF);
   return (SPDR);
}


void mmc_finish(void)                // Zakończenie transakcji z kartą
{
   sbi(MMC_PORT, MMC_CS);            // wyłącz sygnał chip select
   spi_tx_rx(0xff);                  // wyślij 8 impulsów zegarowych
}


void flush_mmc(u08 count)            // odbierz i odrzuć „count” bajtów
z karty
{
   while(count--)
      spi_tx_rx(0xff);
}

u08 read_tag(void)                   // oczekiwanie na „data token” czyli
{                                    // bajt startu bloku danych
   u08 tmp;
   while(1)
   {
      tmp = spi_tx_rx(0xff);         // odczyt bajtu MMC
      if(tmp == 0xFE)
         return 0;                   // jeśli to jest data token

      if((tmp & 0xF1) == 1)
      {
         mmc_finish();               // jeśli Data Error Token
         return 1;
      }
   }
}

// ***********************************************************
//  Wysłanie komendy do karty i odbiór potwierdzenia R1
// ***********************************************************

u08 mmc_cmd(u08 cmd, u32 param, u08 state)
{
   u08 i,tmp;
   cbi(MMC_PORT, MMC_CS);        // aktywuj CS
   spi_tx_rx(cmd | 0x40);        // wyślij komendę
   spi_tx_rx(param >> 24);       // wyślij 4 bajty argumentu
   spi_tx_rx(param >> 16);
   spi_tx_rx(param >> 8);
   spi_tx_rx((u08)param);        // LSB
   spi_tx_rx(0x95);              // wyślij poprawna sumę CRC
                                 // dla komendy CMD64
   for(i=0 ; i<10 ; i++)         // czekaj na odpowiedź
   {
      tmp = spi_tx_rx(0xff);     // odbierz odpowiedź
      if((tmp & R1_BUSY) == 0)   // jeśli BUSY == 0
      {
         if(state == Finish)
            mmc_finish();
         return tmp;             // komenda wykonana
      }
   }
   mmc_finish();
   return -1;                    // błąd braku odpowiedzi z karty
}

// ***********************************************************
//  Inicjalizacja interfejsu SPI oraz Reset karty
// ***********************************************************

u08 mmc_reset(void)
{
   sbi(MMC_PORT, MMC_CS);        // CS wysoki
   sbi(MMC_DDR, MMC_CS);         // linia portu CS jako wyjście
   spi_init();                   // inicjalizacja SPI
   flush_mmc(10);                // 80 pustych cykli zegarowych
                                 // wysłanie komendy GO_IDLE_STATE
```

List. 2. cd.

```
if((mmc_cmd(MMC_GO_IDLE_STATE,  0,  Finish)  &  0x85)  !=  R1_IDLE_STATE)
      return 1;

   // wysłanie komendy SEND_OP_COND
   while(mmc_cmd(MMC_SEND_OP_COND, 0, Finish) != 0);

   // ustawienie długości bloku danych na 512 bajtów
   mmc_cmd(MMC_SET_BLOCKLEN, 512, Finish);
   return 0;
}

// ***********************************************************
//  Odczyt 512 bajtowego sektora z karty MMC
// ***********************************************************

u08 mmc_read_sector(u32 sector)
{
   u16 i;
   if(mmc_cmd(MMC_READ_SINGLE_BLOCK, sector << 9, Leave) != 0)
      return(1);

   if(read_tag())             // czekaj na „data token”
      return(1);

   for(i=0 ; i<512 ; i++)     // odczyt 512 bajtów danych
      mmc_sbuf[i] = spi_tx_rx(0xff);

   flush_mmc(2);              // odrzuć CRC
   mmc_finish();              // zakończ transakcję
   return 0;
}

// ***********************************************************
//  Zapis 512 bajtowego sektora do karty MMC
// ***********************************************************

u08 mmc_write_sector(u32 sector)
{
   u16 i;
   if(mmc_cmd(MMC_WRITE_BLOCK, sector << 9, Leave) != 0)
      return(1);

   spi_tx_rx(0xff);           // wyślij pusty bajt
   spi_tx_rx(0xFE);           // wyślij „data token”
   for(i=0 ; i<512 ; i++)     // wyślij 512 bajtów danych
      spi_tx_rx(mmc_sbuf[i]);

   spi_tx_rx(0);       // wyślij 1 bajt CRC (ignorowany przez kartę)
   spi_tx_rx(0);       // wyślij 2 bajt CRC

   while((spi_tx_rx(0xff) & 0x1F) != 0x05); // odbierz potwierdzenie
                                            // (data response)
   while(spi_tx_rx(0xff) == 0xff);  // czekaj na koniec sygnału busy
   mmc_finish();                    // koniec transakcji
   return 0;
}

// ***********************************************************
//  Odczyt rejestru CID
// ***********************************************************

u08 mmc_get_cid(void)
{
   u08 i;
   if(mmc_cmd(MMC_SEND_CID, 0, Leave))
      return 1;

   if(read_tag())                   // czekaj na „data token”
      return(1);

   for(i=0 ; i<16 ; i++)            // odbierz 16 bajtów rejestru CID
      mmc_sbuf[i] = spi_tx_rx(0xff);

   flush_mmc(2);                    // odrzuć CRC
   mmc_finish();                    // koniec transakcji
   return 0;
}

// ***********************************************************
//  Odczyt i obliczenie pojemności karty w sektorach
// ***********************************************************

u32 mmc_capacity(void)
{
   u16 size;
   u08 mult;
   if(mmc_cmd(MMC_SEND_CSD, 0, Leave))
      return 0;

   if(read_tag())                   // czekaj na „data token”
      return 0;
   flush_mmc(6);                // odrzuć pierwsze 6 bajtów rejestru CSD
   size = (spi_tx_rx(0xff) & 3) << 10;  // najstarsze 2 bity C_SIZE
   size |= spi_tx_rx(0xff) << 2;        // kolejne 8 bitów C_SIZE
   size |= (spi_tx_rx(0xff) >> 6) & 3;  // najmłodsze 2 bity C_SIZE
   mult = (spi_tx_rx(0xff) & 3) << 1;   // starsze 2 bity C_SIZE_MULT
   mult |= (spi_tx_rx(0xff) >> 7) & 1;  // najmłodszy bit C_SIZE_MULT
   flush_mmc(7);            // odrzuć resztę rejestru CSD i bajty CRC
   mmc_finish();            // koniec transakcji MMC
   return (u32)(size+1)<<(mult+2);   // oblicz i zwróć pojemność karty
}
```

List. 3. Przykład wykorzystania omówionych procedur

```
#include <avr/io.h>
#include „mmc.h”

extern u08 mmc_sbuf[];                      // Bufor danych w pamięci RAM mikrokontrolera

void send_buf(u16 count)
{
   u16 i;
   for(i=0; i<count; i++)
   {
      while( !(UCSR0A & (1<<UDRE)) );       // Czekaj na gotowość nadajnika
      UDR0 = mmc_sbuf[i];                   // Wyślij bajt z bufora
   }
}

void printu32(u32 u_val)                    // wyślij wartość liczby u32 przez uart
{
   u08 scratch[16];
   u08 *ptr;

   ptr = scratch + 16;
   *--ptr = 0;
   do
   {
      *--ptr = u_val % 10 + ‚0’;
      u_val /= 10;
   }while (u_val);

   while (*ptr)
   {
      while( !(UCSR0A & (1<<UDRE)) );
      UDR0 = *ptr++;
   }
}

void eol(void)
{
   while( !(UCSR0A & (1<<UDRE)) );          // Czekaj na gotowość nadajnika
   UDR0 = 13;                               // Wyślij CR
   while( !(UCSR0A & (1<<UDRE)) );          // Czekaj na gotowość nadajnika
   UDR0 = 10;                               // Wyślij LF
}

int main(void)
{
   u32 poj;
   UCSR0B = (1<<TXEN);                      // Inicjalizacja nadajnika RS232
   UBRRH  = 0;
   UBRR0  = 25;                             // Ustawienie 19200 bodów przy kwarcu 8MHz

   u08 i;
   mmc_reset();                             // Reset karty

   mmc_get_cid();                           // Identyfikacja karty
   send_buf(16);                            // Wyślij CID przez uart
   eol();

   poj = mmc_capacity();                    // pobierz ilość sektorów
   printu32(poj);                           // wyślij przez uart
   eol();

   for(i=0; i<10; i++)
   {
      mmc_read_sector(i);                   // Odczytaj sektor o adresie w zmiennej i
      send_buf(512);                        // Wyślij dane przez UART
      eol();
   }
   while(1);                                // Koniec pracy
}
```

poprzez wywołanie funkcji *mmc_finish*. Zapisując dane do karty, musimy jeszcze odebrać z karty potwierdzenie *data response*, a następnie poczekać na zakończenie wewnętrznych procedur zapisu do pamięci Flash karty, czyli poczekać na koniec sygnału BUSY.

Funkcja *mmc_get_cid* pozwala na odczytanie zawartości rejestru CID karty, czyli danych identyfikacyjnych. Wygląda ona praktycznie tak samo jak funkcja odbioru zwykłych danych z karty, lecz wysyła inną komendę oraz odbiera tylko 16 bajtów danych.

Ostatnia z funkcji *mmc_capacity* umożliwia odczyt pojemności karty wyrażonej w blokach. Do tego celu wykorzystywany jest rejestr CSD, a właściwie jego fragment, w którym zakodowano wartości *C_SIZE* i *C_SIZE_MULT*. Po ich odczytaniu i uporządkowaniu dokonuje ona obliczeń zgodnych z wzorem na pojemność karty, który podawałem przy okazji opisu zawartości rejestru CSD, a następnie zwraca obliczoną pojemność jako 32-bitową wartość funkcji. W odróżnieniu od poprzednich funkcji, w przypadku wystąpienia błędu, zwraca ona wartość 0, czyli określa pojemność jako 0. Pozostałe funkcje zwracają zero jako wyznacznik prawidłowego ich wykonania.

Procedury napisane w języku C, choć są dość dobrze optymalizowane przez kompilator, zajmują jednak sporo miejsca w pamięci Flash mikrokontrolera. Z tego też względu na CD-EP9/2004B zamieszczamy takie same procedury jak pokazano na list. 2, lecz napisane w asemblerze procesora AVR i zoptymalizowane pod kątem długości generowanego kodu (plik *mmc.asm*). Funkcjonalnie odpowiadają one w 100% procedurom zamieszczonym na list. 2 i oprócz tego, że zajmują mniej miejsca, są jeszcze nieco szybsze od swoich odpowiedników napisanych w języku C.

Na koniec, **na list. 3** przedstawiam malutki przykładzik wykorzystania omówionych procedur w postaci krótkiego programu wysyłającego poprzez szeregowy interfejs RS232 dane identyfikacyjne karty, jej wielkość wyrażoną w sektorach, a następnie zawartość pierwszych 10 sektorów karty.

Romuald Biały