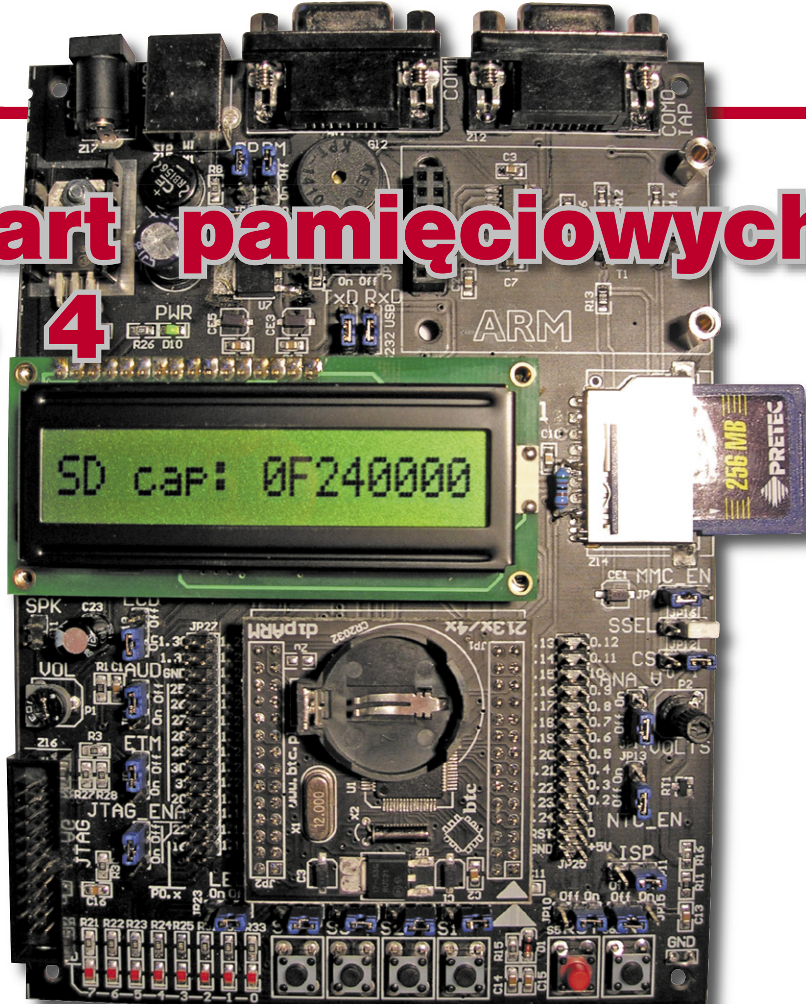
Obsługa kart pamięciowych SD, część 4



Przechodzimy do części niewątpliwie praktycznej - opisu procedur. W artykule pokazujemy jak obsłużyć karty SD za pomocą mikrokontrolera LPC2148, a dzięki napisaniu wszystkich procedur w języku C przeniesienie ich na inne mikrokontrolery nie będzie kłopotliwe.

Dołączanie kart SD do mikrokontrolerów jest stosunkowo proste. Potrzebny jest mikrokontroler wyposażony w sprzętowy interfejs SPI, ale może on być realizowany także programowo. Dodatkowo mogą być przydatne 2 linie I/O do wykrywania obecności karty w gnieździe i protekcji zapisu.

Do testów komunikacji z kartą wybrano moduł ZL9ARM z modułem dipARM wyposażonym w procesor LPC2148 (obydwa urządzenia dostarczyła firma *www.kamami.pl*). Dołączenie karty SD do modułu ZL9ARM jest łatwe ze wzgledu na to, że został on wyposażony w złącze kart SD (**rys. 15**).

Karta jest zasilana napięciem +3,3 V i sterowana liniami interfejsu SSP skonfigurowanego do pracy w trybie SPI Master. Wszystkie linie interfejsu są buforowane układem (U3) 74LVC541. Do konfiguracji interfejsu karty są wykorzystywane zworki JP4 (MMC_EN) i JP12 (CS). Zworka JP4 musi być tak ustawiona, żeby na wyprowadzeniach 1 i 19 układu U3 był stan niski (zwarte 1 i 2 JP4). Zworka JP12 musi łączyć linię P0.21 z wyprowadzeniem 5 U3 (zwarte 2 i 3 JP12). Zworka JP16 SSEL nie jest tu wykorzystywana, a linia P0.20 musi być ustawiona jako wejście.

Wszystkie procedury opisane w artykule zostały zweryfikowane w praktyce z wykorzystaniem kart Toshiba SD-M512 o pojemności 512 MB i Pretec 256 MB High Speed.

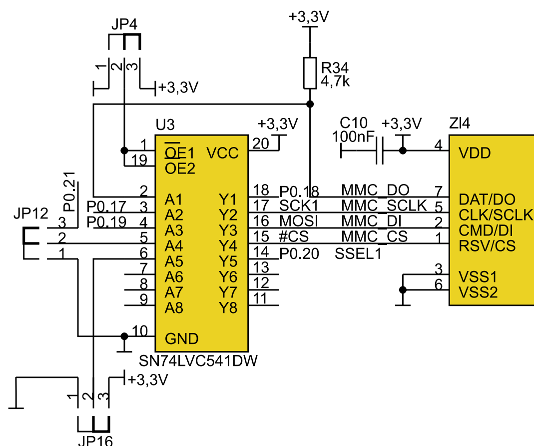
Przed wywołaniem właściwych procedur komunikacyjnych trzeba zdefiniować kody komend, odpowiadające im potwierdzenia oraz znaczniki określające, czy po otrzymaniu potwierdzenia komendy trzeba odbierać jakieś dane. Mimo iż w trybie SPI wyłączono sprawdzanie poprawności CRC, to dla każdej z komend zostały wyliczone sumy kontrolne i po włączeniu sprawdzania CRC można z nich korzystać. Wszystkie niezbędne definicje umieszczono w pliku *sd.h*, pokazano je na **list. 1**.

Procedura *SendSDCCmd* wysyła do karty SD kompletną komendę i zwraca bajt potwierdzenia R1 lub 2 bajty potwierdzenia R2. Jeżeli wykonywana komenda nie wymaga kolejnych danych (NODATA), to transakcja jest kończona wysłaniem dodatkowych 8 cykli zegarowych i przejściem sygnału MMC_CS w stan wysoki. W przypadku, kiedy są spodziewane dodatkowe dane (MOREDATA), sygnał MMC_CS pozostaje w stanie niskim.

Wysyłanie komendy rozpoczyna się wymuszeniem stanu niskiego na wejściu CS karty SD. Pierwszy argument funkcji zawiera numer komendy, który jest indeksem wiersza dwuwymiarowej tablicy *CmdTab*. Numerom wierszy odpowiadają nazwy symboliczne komend zdefiniowane na list. 1. Na przykład komendzie GO_IDLE_STATE odpowiada wiersz zerowy tablicy *CmdTab*, a symbolicznie jest to *cmdGo_IDLE_STATE* (definicja *enum*). Kolejne elementy wiersza *CmdTab* są zwracane przez funkcje pokazane na **list. 2**.

Funkcja *get_cmd(cmd)* zwraca zerowy element wiersza określonego przez *cmd*, czyli kod komendy.

Drugi argument funkcji *SendSDCCmd* zawiera 4-bajtową liczbę będącą



Rys. 15. Sposób buforowania karty SD/MMC

List. 1. Definicje z pliku nagłówkowego *sd.h*

```
#define SDC_FLOATING_BUS    0xFF
#define SDC_BAD_RESPONSE    SDC_FLOATING_BUS
#define SDC_ILLEGAL_CMD     0x04
#define SDC_GOOD_CMD        0x00

#define BLOCKLEN_64                 0x0040
#define BLOCKLEN_128                0x0080
#define BLOCKLEN_256                0x0100
#define BLOCKLEN_512                0x0200
//definicja typow potwierdzen
#define R1 0x00
#define R1b 0x01
#define R2 0x02
#define R3 0x03
#define NODATA 0x00
#define MOREDATA 0x01

//definicje znacznikow
/* Data Token */
#define DATA_START_TOKEN            0xFE
#define DATA_MULT_WRT_START_TOK     0xFC
#define DATA_MULT_WRT_STOP_TOK      0xFD
//definicje kodow potwierdzen
/* Data Response */
#define DATA_ACCEPTED              0x05 //0b00000101
#define DATA_CRC_ERR                 0x0b //0b00001011
#define DATA_WRT_ERR                0x0d //0b00001101

// definicje kodów komend: 6 młodszych bitów numer komendy, bit B6 ma wartość 1
#define     GO_IDLE_STATE        0x40      //0 mlodsze 6-bitów numer komendy
#define     SEND_OP_COND         0x41      //1
#define     SEND_CSD               0x49            //9
#define     SEND_CID               0x4a            //10
#define     STOP_TRANSMISSION    0x4c      //12
#define     SEND_STATUS            0x4d            //13
#define     SET_BLOCKLEN         0x50      //16
#define     READ_SINGLE_BLOCK    0x51   //17
#define     READ_MULTI_BLOCK     0x52   //18
#define     WRITE_SINGLE_BLOCK   0x58   //24
#define     WRITE_MULTI_BLOCK    0x59   //25
#define     TAG_SECTOR_START     0x60      //32
#define     TAG_SECTOR_END       0x61      //33
#define     UNTAG_SECTOR         0x62      //34
#define     TAG_ERASE_GRP_START  0x63   //35
#define     TAG_ERASE_GRP_END    0x64   //36
#define     UNTAG_ERASE_GRP      0x65      //37
#define     ERASE                  0x66            //38
#define     SD_APP_OP_COND       0x69      //ACMD41
#define     LOCK_UNLOCK          0x71      //49
#define     APP_CMD              0x77      //55
#define     READ_OCR             0x7a      //58
#define     CRC_ON_OFF           0x7b      //59

/*definicja symbolicznych nazw komend używanych w wywolaniu procedury wyslania komendy */
enum
{
    cmdGO_IDLE_STATE,
    cmdSEND_OP_COND,
    cmdSEND_CSD,
    cmdSEND_CID,
    cmdSTOP_TRANSMISSION,
    cmdSEND_STATUS,
    cmdSET_BLOCKLEN,
    cmdREAD_SINGLE_BLOCK,
    cmdREAD_MULTI_BLOCK,
    cmdWRITE_SINGLE_BLOCK,
    cmdWRITE_MULTI_BLOCK,
    cmdTAG_SECTOR_START,
    cmdTAG_SECTOR_END,
    cmdUNTAG_SECTOR,
    cmdTAG_ERASE_GRP_START,
    cmdTAG_ERASE_GRP_END,
    cmdUNTAG_ERASE_GRP,
    cmdERASE,
    cmdLOCK_UNLOCK,
    cmdSD_APP_OP_COND,
    cmdAPP_CMD,
    cmdREAD_OCR,
    cmdCRC_ON_OFF,
  cmdSD
};

/*-------------------------------------------------------------------------------------
 definicja tablicy zawieracej kod komendy, sume kontrolna, typ potwierdzenia
 oraz znacznik kontynuacji odbierania danych
--------------------------------------------------------------------------------------*/
const unsigned char CmdTab[23][4] =
{
    //cmd                 crc   response
    {GO_IDLE_STATE,       0x95, R1, NODATA},
    {SEND_OP_COND,        0xF9, R1, NODATA},
    {SEND_CSD,            0xAF, R1, MOREDATA},
    {SEND_CID,            0x1B, R1, MOREDATA},
    {STOP_TRANSMISSION,   0xC3, R1, NODATA},
    {SEND_STATUS,         0xAF, R2, NODATA},
    {SET_BLOCKLEN,        0xFF, R1, NODATA},
    {READ_SINGLE_BLOCK,   0xFF, R1, MOREDATA},
```

argumentem wysyłanej komendy. Następnie wysyłany jest bajt CRC. Można tu wysłać dowolna liczbę, ale funkcja *get_crc(cmd)* zwraca wcześniej wyliczoną prawidłowo wyliczoną wartość CRC. Po wysłaniu kompletnej komendy procedura oczekuje na typ potwierdzenia zapisany w *CmdTab* i zwracany funkcją *get_rsp(cmd)*. Jeżeli jest to potwierdzenie R1 lub R1b, to host próbuje odczytać 256 razy potwierdzenie z karty. Jeżeli odczytane potwierdzenie ma wartość inną niż 0xFF, to próba odczytania potwierdzenia z karty kończy się sukcesem. Jeżeli przez 256 prób nie udało się odczytać potwierdzenia, to procedura jest kończona i zwracany jest bajt 0xFF. Po odczytaniu bajtu R1 program sprawdza, czy potwierdzenie jest typu R1b i jeżeli tak, to cyklicznie sprawdza stan linii MMC_DO karty odbierając z niej dane interfejsem SPI. W trakcie odczytywania interfejsem SPI na linię zegarową karty wysyłany jest przez cały czas przebieg zegarowy. Zapobiega to permanentnemu przejściu ustawieniu danych karty w stan niski. Testowanie linii kończy się, kiedy linia przechodzi w stan wysoki, lub po upłynięciu określonego czasu (wykonaniu określonej liczby sprawdzeń). Dla potwierdzenia R2 host wysyła 8 taktów zegara, po tym odczytuje z karty 2 bajty i zapisuje w zmiennej *int res_out*.

Kiedy nie są spodziewane kolejne dane (NODATA), to procedura kończy transakcję wysyłając na linię zegarową 8 cykli zegara i wymuszając na linii CS stan wysoki. Dla komend, w których są spodziewane kolejne dane (na przykład zapisanie bloku danych) transakcja nie jest kończona – linia MMC_CS pozostaje w stanie niskim.

List. 1. c.d.

```
    {READ_MULTI_BLOCK,    0xFF, R1, MOREDATA},
    {WRITE_SINGLE_BLOCK,  0xFF, R1, MOREDATA},
    {WRITE_MULTI_BLOCK,   0xFF, R1, MOREDATA},
    {TAG_SECTOR_START,    0xFF, R1, NODATA},
    {TAG_SECTOR_END,      0xFF, R1, NODATA},
    {UNTAG_SECTOR,        0xFF, R1, NODATA},
    {TAG_ERASE_GRP_START, 0xFF, R1, NODATA},
    {TAG_ERASE_GRP_END,   0xFF, R1, NODATA},
    {UNTAG_ERASE_GRP,     0xFF, R1, NODATA},
    {ERASE,               0xDF, R1b, NODATA},
    {LOCK_UNLOCK,         0x89, R1b, NODATA},
    {SD_APP_OP_COND,      0xE5, R1, NODATA},
    {APP_CMD,             0x73, R1, NODATA},
    {READ_OCR,            0x25, R3, NODATA},
    {CRC_ON_OFF,          0x25, R1, NODATA}
};
//definicje kodow bledow
typedef enum
{
    sdcValid=0,                 // wszytko poprawnie
    sdcCardInitCommFailure,     // nie ustanowioni polaczenia z karta
    sdcCardNotInitFailure,      // karta nie przeszla fazy incjalizacji
    sdcCardInitTimeout,         // timeout inicjalizacji karty
    sdcCardTypeInvalid,         // nie mozna zdefiniowac karty
    sdcCardBadCmd,              // karta nie rozpoznala komendy
    sdcCardTimeout,             // timeout w czasie sekwencji zapisu odczyty lub kasowania
    sdcCardCRCError,            // wystapil blad CRC w czasie odczytu, dane powinny byc odrzucone
    sdcCardDataRejected,        // blad CRC danych przesylanych
    sdcEraseTimedOut            // timeout kasowania danych
}SDC_Error;
```

List. 2. Funkcje zwracające parametry komendy

```
char get_cmd(char sd_cmd)
{
  return(CmdTab[sd_cmd][0]);//kod komendy
}
char get_crc(char sd_cmd)
{
  return(CmdTab[sd_cmd][1]);//bajt CRC
}
char get_rsp(char sd_cmd)
{
  return(CmdTab[sd_cmd][2]);//typ potwierdzenia
}
char get_tdata(char sd_cmd)
{
  return(CmdTab[sd_cmd][3]);//kolejne dane
```

Funkcja *SendSDCCmd* (**list. 3**) zwraca kod potwierdzenia: dla potwierdzeń R1i R1b jeden bajt, a dla potwierdzenia R2 dwa bajty.

Funkcja *MediaInitialize* wykonuje inicjalizację karty. Zgodnie z wymaganiami standardu na linii MMC_CS wystawiany jest stan wysoki i wysyłanych jest 80 cykli zegarowych na linię MMC_SCLK. Po włączeniu zasilania karta komunikuje się z hostem za pomocą magistrali SDBus (z jedną linią danych D0) i żeby ją przełączyć w tryb SPI trzeba wysłać komendę CMD0 zerowania karty przy stanie niskim na wejściu CS. Ponieważ karta jest w trybie SDBus, to przy wysyłaniu CMD0 konieczne jest wysłanie prawidłowej sumy CRC, bo inaczej komenda zostanie odrzucona. Po przejściu w tryb SPI sprawdzanie CRC zostaje wyłączone.

Jeżeli potwierdzenie R1 komendy CMD0 będzie miało wartość 0xFF (same jedynki), to oznacza, że karta w ogóle nie przyjęła komendy. Karta może również być zajęta. Odpowiada wtedy potwierdzeniem z ustawionym bitem *Busy*. W obu przypadkach funkcja *MediaInitialize* kończy działanie zwracając kod błędu różny od zera (**list. 4**).

Karta odpowiada prawidłowo na komendę CMD0 potwierdzeniem R1 z wyzerowanymi wszystkimi bitami. Wtedy można wysłać komendę CMD1 SEND_OP_COND inicjalizująca kartę. Trzeba pamiętać, że część kart zamiast komendy CMD1 wymaga komendy ACMD41.

Ponieważ po zerowaniu karty może być ona zajęta przez pewien czas, to komenda CMD1 jest wysyłana wiele razy, aż do otrzymania potwierdze-

List. 3. Funkcja wysyłająca komendę do karty SD

```
unsigned int SendSDCCmd(unsigned char cmd, unsigned int
address)
{
char resp;
unsigned char res_byte_l=0,res_byte_h=0,index;
unsigned int res_out;
unsigned int timeout = 8;
    SDC_CS_0 //CS=0
    Write_SPI(get_cmd(cmd));   //wyslij kod komendy
   Write_SPI(address>>24);    //najstarszy bajt argu-
mentu
    Write_SPI(address>>16);   //kolejne bajty argumentu
   Write_SPI(address>>8);
    Write_SPI(address);       //najmlodszy bajt argu-
mentu
    Write_SPI(get_crc(cmd));  //wyslij CRC
     resp=get_rsp(cmd);       //typ odpowiedzi karty
    if(resp==R1 || resp==R1b)
    {
   while(timeout!=0)
   {res_byte_l=ReadMedia();   //czekaj na odpowiedz
rozna od 0xFF
    if(res_byte_l!=0xff)
    break;
      timeout--;}
    }
    else
  if(resp == R2)
        {ReadMedia();
        res_byte_l = ReadMedia();
        res_byte_h = ReadMedia();}
    if(resp == R1b)           //dodatkowo czekaj na
zwolnienie linii danych dla R1b
      {
  res_byte_l = 0x00;
  for(index =0; index < 0xFF && res_byte_l == 0x00; in-
dex++)
  {timeout = 0xFFFF;
          do
          {res_byte_l = ReadMedia();
            timeout--;
          }while((res_byte_l == 0x00) && (timeout !=
0));
    }
  }

    if(get_tdata(cmd)==NODATA)  //nie sa spodziewane
kolejne dane
      {SDC_CS_1                 //CS=1
   Write_SPI(0xff);}
   if(resp == R1 || resp == R1b)
    return(res_byte_l);         //potwierdzenie 1 baj-
towe
  if(resp == R2)                //poteirdzenie 2 bajto-
we, lub 3 bajtowe
  {res_out=res_byte_h;
   res_out=res_out<<8;
   res_out=res_out|res_byte_l;
    return(res_out);}
}
```

List. 4. Funkcja inicjalizacji karty do pracy z magistralą w trybie SPI

```
#define SDC_CS 1<<21
#define SPI_SEL 0x00100000
#define SDC_CS_1 IOSET0|=SDC_CS;                    //definicje makr sterowania linią CS
#define SDC_CS_0 IOCLR0|=SDC_CS;
unsigned int MediaInitialize(void)
{
unsigned short timeout=0;
unsigned int CSDstatus=0;
unsigned int resp=0;
  IO0DIR|=SDC_CS;                                   //linia CS wyjsciowa
  status=0;
  SDC_CS_1                                          // makro CS=1
  delay_ms(100);
    for(timeout=0; timeout<10; timeout++)           //karta wymaga 80 cykli zegra przy inicjalizacji
  Write_SPI(0xff);
    SDC_CS_0                                        //CS=0 SC aktywny
    delay_ms(10);
    resp = SendSDCCmd(cmdGO_IDLE_STATE,0x0);        //wyslij komende CMD0 reset karty (adres 0)
    if(resp == SDC_BAD_RESPONSE)//odpowiedź 0xFF
    {status = sdcCardInitCommFailure;               //nic nie odebralismy z karty
     goto InitError;}
    if(resp != 0x01)                                //nie ma błedu,ale karta zajęta (BUSY)
     {status = sdcCardNotInitFailure;               //nic nie odebraliśmy z karty
     goto InitError;}
     timeout = 0x1FFF;                              //reset karty poprawny - wysylamy CMD1 procedura inicjalizaci karty
(adres 0)
  do{resp = SendSDCCmd(cmdSEND_OP_COND,0x0);        //CMD1
   timeout--;
     }while(resp != 0); //&& timeout != 0);         //probujemy 4096 razy az mnie czas lub otrzymamy potwierdzenie 00
    if(timeout == 0)                                //nie otrzymalismy potwierdzenia
    {status = sdcCardInitTimeout;                   //nie odebralismy nic z karty - blad przekroczenia czasu
     goto InitError;
    }
    else
  {CSDstatus=CSDRead();                             //odczytaj rejestr CID - CMD2
      if(CSDstatus==0);
    else
      {status=sdcCardTypeInvalid;
     goto InitError;}
  }
   resp=SendSDCCmd(cmdSET_BLOCKLEN,BLOCKLEN_512); //ustawienie długości sektora do odczytu na 512 bajtow
     if(resp!=0)
  {status = sdcCardInitTimeout;                     //nie odebralismy nic z karty - blad przekroczenia czasu
      goto InitError;}
     for(timeout = 0xFF; timeout > 0; timeout--)
     {resp=SectorRead(0x0,msd_buffer);
     if(resp==0)
   break;}
    if(timeout == 0)
    {status = sdcCardNotInitFailure;
        goto InitError;}
  return(status);
InitError:
    SDC_CS_1                                        //karta nieaktywna
    return(status);
}//end MediaInitialize
```

List. 4. Funkcja odczytania pojedynczego bloku danych z karty

```
unsigned int SectorRead(int sector_addr, unsigned char *buffer)
{
int index;
unsigned short resp=0;
unsigned char data_token;
    resp = SendSDCCmd(cmdREAD_SINGLE_BLOCK,(sector_addr << 9));
    if(resp != 0x00)
    {
        status = sdcCardBadCmd;
    }
    else
    {
        index = 0x2FF;
        do
        {
            data_token = ReadMedia();
            index--;
        }while((data_token == SDC_FLOATING_BUS) && (index != 0));

        if((index == 0) || (data_token != DATA_START_TOKEN))//0xFE
            status = sdcCardTimeout;
        else
        {
            for(index = 0; index < SDC_SECTOR_SIZE; index++)
     buffer[index] = ReadMedia();                  //czytanie 512 bajtow danych

     ReadMedia();                                  //CRC
     ReadMedia();
        }
    }
  SDC_CS_1                                         //koniec transakcji
  Write_SPI(0xff);
    return(status);
}//end SectorRead
```

nia R1 z wyzerowanymi bitami. Jeżeli takiego potwierdzenia nie otrzymamy po wysłaniu 4096 razy komendy CMD1, to funkcja *MediaInitialize* jest kończona i zwraca błąd przekroczenia czasu inicjalizacji.

Prawidłowe wykonanie komendy CMD1 kończy funkcję inicjalizacji i można wysłać do karty komendy odczytania rejestru CSD i dla pewności ustawić długość bloku dla odczytu karty na 512 bajtów. Funkcją *MediaInitialize* kończy się próba prawidłowego odczytania bloku o długości 512 bajtów. Jeżeli się powiedzie, to funkcja zwraca wartość zero oznaczającą poprawne zainicjowanie karty.

Funkcja *SectorRead* (**list. 5**) odczytuje pojedynczy blok danych (sektor) o długość 512 bajtów i zapisuje go do bufora *msd_buffer* w pamięci RAM mikrokontrolera. Adres początkowy bloku musi być wielokrotnością długości bloku danych. Argument *sector_addr*

zawiera numer bloku danych, a nie fizyczny adres jego początku. Adres fizyczny jest wyliczany przez pomnożenie numeru bloku przez 512 (przesunięcie o 9 bitów w lewo).

W pierwszym kroku po wysłaniu komendy READ_SINGLE_BLOCK spodziewane jest odebranie zerowego potwierdzenia R1. Jeżeli je otrzymamy, to są odczytywane dane cyklicznie z karty, aż do odebrania bajtu startu DATA_START_TOKEN równego 0xFE. Potem można już odczytać 512 bajtów danych z karty i dodatkowo 2 bajty sumy kontrolnej CRC. Transakcja kończy się wysłaniem 8 cykli zegarowych na linię SCLK i przejściem linii CS w stan wysoki.

Funkcja zapisująca pojedynczy blok danych (**list. 6**) rozpoczyna się od wysłania komendy WRITE_SINGLE_BLOCK. Argumentem komendy jest adres początkowy bloku danych wyliczony na podstawie numeru bloku danych w taki sam sposób jak w procedurze odczytania bloku danych. Jeżeli karta odeśle potwierdzenie R1 z wyzerowanymi wszystkimi bitami, to wysyłanych jest 8 cykli zegarowych na linię SCLK i bajt startu (0xFE) DATA_START_TOKEN. Następnie host wysyła 512 bajtów bloku danych i 2 bajty CRC, a karta potwierdza ich prawidłowe przyjęcie wysłaniem bajtu *Data Response* o wartości 0x05.

Stan zajętości karty w czasie zapisywania bloku danych w pamięci Flash jest testowany przez cykliczne odczytywanie danych z karty. Trwa to tak długo, aż odczytywane dane będą różne od zera. Testowanie linii MMC_DO przez odczytywanie danych z karty (a nie testowanie stanu linii portu) powoduje, że na linię zegarową jest wysyłany niezbędny sygnał zegarowy. Jeżeli zapisywanie zostało wykonane prawidłowo, to procedura zwraca wartość zerową, w przeciwnym wypadku zwraca kod błędu różny od zera.

Funkcja *CSDRead* (**list. 7**) wysyła komendę SEND_CSD z zerowym argumentem i czeka na zerowe potwierdzenie R1. Jeżeli je otrzyma, to oczekuje na bajt startu 0xFE dokładnie tak samo, jak przy odczytywaniu bloku danych. W następnym kroku jest odczytywanych 16 bajtów rejestru CSD, 2 bajty sumy CRC i karta kończy transakcję.

Tomasz Jabłoński, EP
tomasz.jablonski@ep.com.pl

List. 6. Procedura zapisująca pojedynczy blok danych

```
unsigned int SectorWrite(short sector_addr, unsigned char *buffer)
{
    int index;
    unsigned char data_response;
  unsigned char resp;

    resp = SendSDCCmd(cmdWRITE_SINGLE_BLOCK,(sector_addr << 9));
    if(resp != 0x00)
        status = sdcCardBadCmd;
    else
    {
        Write_SPI(0xff);
        Write_SPI(DATA_START_TOKEN);              //wyślij data start token

        for(index = 0; index < 512; index++)      //wyślij 512 bajtów danych
        Write_SPI(buffer[index]);

        Write_SPI(0xff);                          //wyślij 2 bajty CRC
     Write_SPI(0xff);
        index=16;
        while(index>0)
        {data_response = ReadMedia();             //czytaj odpowiedź
      if((data_response & 0x0F) == DATA_ACCEPTED)
         break;}
        if(index==0)
        {
            status = sdcCardDataRejected;
        }
        else
        {index = 0;
            do                                    //czekaj na koniec zapisu
            {data_response = ReadMedia();
             index++;
            }while((data_response == 0x00) && (index != 0));
            if(index == 0)                        //nie wykryto końca zapisu
             status = sdcCardTimeout;
        }
    }

    SDC_CS_1 //CS=1
  Write_SPI(0xff);

     return(status);
} //end SectorWrite
```

List. 7. Procedura odczytu rejestru CSD

```
unsigned int CSDRead()
{
    unsigned short index, timeout=0x2ff;
  unsigned int resp;
    unsigned char data_token;

    SDC_CS_0
    Write_SPI(SEND_CSD);
    Write_SPI(0);
    Write_SPI(0);
    Write_SPI(0);
    Write_SPI(0);
    Write_SPI(0xaf);

    do//czekaj na odpowiedz
      {resp = ReadMedia();
       timeout--;}
  while((resp == 0xFF) && (timeout != 0));

    if(resp != 0x00)
     status = sdcCardBadCmd;
    else//komenda zostala zaakceptowana
    {
        index = 0x2FF;
        do//czekaj na data token (0xFE)
        {
            data_token = ReadMedia();
            index--;
        }while((data_token == SDC_FLOATING_BUS) && (index != 0));

       for(index = 0; index < CSD_SIZE; index++)//zerowanie bufora CSD
         csd_buffer[index] = 0;

        if((index == 0) || (data_token != DATA_START_TOKEN))//nie odebrano
data token
            status = sdcCardTimeout;
        else
        {
         for(index = 0; index < CSD_SIZE; index++)//czytaj 16 bajtow CSD
       csd_buffer[index] = ReadMedia();
        }
     ReadMedia();
     ReadMedia();
    }
    SDC_CS_1
  Write_SPI(0xff);
    return(status);
}//end CSDRead
```