Obsługa kart pamięci Flash za pomocą mikrokontrolerów, część 3

Po miesięcznej przerwie przedstawiamy kolejną część artykułu, w którym prezentujemy sposoby obsługi kart pamięciowych Flash za pomocą mikrokontrolerów AVR. Tym razem skupiamy się na omówieniu (i pokazaniu!) procedur zapisu i odczytu kart CF.

Procedury zapisu i odczytu kart CF

W poprzednich odcinkach kursu zebraliśmy całą teoretyczną wiedzę wymaganą do osiągnięcia sukcesu - czyli do skomunikowania się z kartami CF podłączonymi do mikrokontrolera AVR. W tym miejscu dochodzimy do konkretnego sposobu implementacji komend zapisu i odczytu sektora dla kart typu *Compact Flash*, wykorzystującego procedury w języku C z ukierunkowaniem na mikrokontrolery AVR. Przykładowe procedury są przeznaczone do skompilowania za pomocą kompilatora AVR-GCC dla mikrokontrolera typu ATmega161. Zastosowanie innego kompilatora lub innego typu mikrokontrolera może wymagać dokonania niewielkich modyfikacji kodu źródłowego.

Stosując podłączenie według rys. 4 (EP2/2004), należy użyć mikrokontrolera posiadającego odpowiednią ilość wewnętrznej pamięci RAM - powyżej 600 bajtów, ponieważ potrzebujemy 512 bajtów na sam bufor sektora, no i oczywiście dodatkowo co najmniej kilkadziesiąt komórek na zmienne i stos. W obu przypadkach mikrokontroler musi posiadać interfejs dla zewnętrznej pamięci RAM - bo w ten sposób jest podłączona i obsługiwana karta. ATmega161 spokojnie spełnia te założenie - posiada 1 kB wewnętrznego RAM-u i ma interfejs do zewnętrznej pamięci. Oczywiście w przypadku procesora nieposiadającego interfejsu RAM, do komunikacji z kartą można użyć standardowych linii portów wejścia-wyjścia, a odpowiednie kombinacje sygnałów sterujących generować programowo, lecz będzie to wymagało napisania innych procedur zapisu i odczytu rejestrów karty.

Na **list. 1** umieszczono deklaracje bitów rejestru statusu oraz adresy poszczególnych rejestrów

List. 1.

```
#define NOEXTRAM        // Wstawienie komentarza w tej linii
                        // oznacza współpracę z układem według rys 5
//
// Deklaracje typów (skrótów)
//
typedef unsigned char  u08;
typedef unsigned short u16;
typedef unsigned long  u32;

//
// Bity rejestru statusu
//
#define SR_BUSY  0x80
#define SR_DRDY  0x40
#define SR_DF    0x20
#define SR_DSC   0x10
#define SR_DRQ   0x08
#define SR_CORR  0x04
#define SR_IDX   0x02
#define SR_ERR   0x01

//
// Adresy rejestrów ATA (Bloki Command i Control)
//

#ifdef NOEXTRAM

 #define ATAPI_Data     *((volatile u08*)0xF000) // Rejestr danych,   ODCZYT/ZAPIS
 #define ATAPI_ErrorReg *((volatile u08*)0xF100) // Rejestr błędów,   ODCZYT
 #define ATAPI_Features *((volatile u08*)0xF100) // Rejestr Features, ZAPIS
 #define ATAPI_SectorCount  *((volatile u08*)0xF200)     // Liczba sektorów,
                                                         // ODCZYT/ZAPIS
 #define ATAPI_Sector   *((volatile u08*)0xF300) // Numer Sektora, ODCZYT/ZAPIS
 #define ATAPI_CylLo    *((volatile u08*)0xF400) // Nr Cylindra [LSB],ODCZYT/ZAPIS
 #define ATAPI_CylHi    *((volatile u08*)0xF500) // Nr Cylindra [MSB],ODCZYT/ZAPIS
 #define ATAPI_DrvHead  *((volatile u08*)0xF600) // Numer Głowicy, ODCZYT/ZAPIS
 #define ATAPI_Status   *((volatile u08*)0xF700) // Rejestr Statusu, ODCZYT
 #define ATAPI_Cmd      *((volatile u08*)0xF700) // Rejestr Komend, ZAPIS
 #define ATAPI_AltStat  *((volatile u08*)0xFE00) // Rejestr Alt. Stat, ODCZYT
 #define ATAPI_DevCtrl  *((volatile u08*)0xFE00) // Rejestr Kontrolny, ZAPIS

#else

 #define ATAPI_Data     *((volatile u08*)0x8000) // Rejestr danych, ODCZYT/ZAPIS
 #define ATAPI_ErrorReg *((volatile u08*)0x8001) // Rejestr błędów, ODCZYT
 #define ATAPI_Features *((volatile u08*)0x8001) // Rejestr Features, ZAPIS
 #define ATAPI_SectorCount  *((volatile u08*)0x8002)     // Liczba sektorów,
                                                         // ODCZYT/ZAPIS
 #define ATAPI_Sector   *((volatile u08*)0x8003) // Numer Sektora, ODCZYT/ZAPIS
 #define ATAPI_CylLo    *((volatile u08*)0x8004) // Nr Cylindra [LSB],ODCZYT/ZAPIS
 #define ATAPI_CylHi    *((volatile u08*)0x8005) // Nr Cylindra [MSB],ODCZYT/ZAPIS
 #define ATAPI_DrvHead  *((volatile u08*)0x8006) // Numer Głowicy, ODCZYT/ZAPIS
 #define ATAPI_Status   *((volatile u08*)0x8007) // Rejestr Statusu, ODCZYT
 #define ATAPI_Cmd      *((volatile u08*)0x8007) // Rejestr Komend, ZAPIS
 #define ATAPI_AltStat  *((volatile u08*)0x800E) // Rejestr Alt. Stat, ODCZYT
 #define ATAPI_DevCtrl  *((volatile u08*)0x800E) // Rejestr Kontrolny, ZAPIS

#endif

// Prototypy funkcji
u08 cf_read_data(u32 lba, u08 *buffer);
u08 cf_write_data(u32 lba, u08 *buffer);
u08 cf_identify(u08 *buffer);
u08 cf_reset(void);

void cf_ata_command(u32 lba, u16 count, u08 cmd);
u08 cf_wait_drq(void);
```

W pierwszej części artykułu w tab. 2 opisującej funkcje sygnałów występujących na złączu karty CF wystąpił błąd w opisie sygnału -CSEL. Podłączenie tej linii do masy konfiguruje kartę jako MASTER, a pozostawienie jej niepodłączonej - jako SLAVE, czyli jest dokładnie na

karty CF w przestrzeni adresowej mikrokontrolera. Dyrektywa kompilacji warunkowej wraz z deklaracją *#define NOEXTRAM* umożliwia dostosowanie programu do podłączenia karty według schematu z rys. 4 lub rys. 5 (EP2/2004). Deklaracje typów *u08*, *u16* i *u32* mają na celu ułatwienie pisania programu, bo w końcu łatwiej i szybciej jest napisać *u16* niż *unsigned short*, a na dodatek jasno informują, że dany typ jest *unsigned* o długości 16 bitów.

Na **list. 2** zawarto wszystkie niezbędne procedury umożliwiające komunikację z kartą CF. Najważniejsza z nich to *void cf_ata_command(u32 lba, u16 count, u08 cmd)*. Właśnie ona powoduje zapisanie rejestrów karty odpowiednimi danymi, a następnie wywołanie odpowiedniej komendy ATA. Na początku tej funkcji mamy przeliczenie adresu interesującego nas sektora wyrażonego wartością LBA na dane wpisywane do rejestrów numerów sektora, cylindra - 2 bajty, oraz głowicy. Następnie do rejestru numeru głowicy wpisujemy obliczoną wartość na pozycje HS0...HS3 (patrz opis rejestrów z poprzedniej części artykułu) oraz ustawiamy bity D5...D7, co oznacza, że używać będziemy trybu LBA w odwołaniu do urządzenia *Master*.

Samo wpisanie danych do rejestru polega po prostu na przepisaniu wymaganej wartości pod zadeklarowany wcześniej adres rejestru, który zachowuje się w tym momencie jak komórka zewnętrznej pamięci RAM podłączonej do mikrokontrolera. Odpowiednie kombinacje na liniach adresowych, danych oraz sterujących pamięcią generowane są automatycznie przez wbudowany w mikrokontroler interfejs do zewnętrznej pamięci RAM.

W tym momencie karta rozpoznaje (po bicie D4), że wysyłane dane są przeznaczone dla niej. Następnie sprawdzamy w pętli stan flagi BUSY i DRY z rejestru statusu. Wyzerowanie BUSY i ustawienie DRY oznacza gotowość na przyjęcie reszty danych. Następnie wpisujemy pozostałą część adresu LBA do rejestrów numeru cylindra i sektora oraz żądaną liczbę sektorów do rejestru ilości sektorów. Rzutowanie 16-bitowej zmiennej *count* na 8-bitowy rejestr liczby sektorów ma na celu umożliwienie żądania 256 sektorów zarówno za pomocą wartości 0, jak i 256. No i na sam koniec wpisujemy interesującą nas komendę do rejestru komend, po czym odczekujemy około 5 µs, aby karta zdążyła wystawić flagę BUSY i rozpoczęła interpretację komendy. Czas ten nie jest krytyczny, lecz jeśli będzie zbyt krótki, to możemy się spodziewać kłopotów ze starszymi typami kart.

Funkcja *u08 cf_wait_drq(void)* ma za zadanie sprawdzenie gotowości karty do transferu danych po wykonaniu komendy wymagającej takowego. Jeśli zwróci ona wartość równą zero, oznacza to, że mamy problem, bo komenda nie została prawidłowo wykonana. O przyczynie kłopotów możemy się dowiedzieć czytając rejestr statusu oraz rejestr błędów.

Pozostałe funkcje nie interpretują błędów, lecz przekazują informacje o jego wystąpieniu. Jest to swego rodzaju uproszczenie, ale w większości przypadków wystarcza. Najprostszym wyjściem z takiej sytuacji jest programowe wyzerowanie karty i powtórzenie żądanej komendy, bo jeśli błąd powstał z przyczyn zewnętrznych, a nie z powodu błędnych danych wejściowych, to powinniśmy w ten sposób odzyskać kontrolę nad kartą.

No i na koniec docieramy do właściwych funkcji odczytu, zapi-

List. 2.

```
#include <avr/io.h>
#include „cf.h”
// ********************************************************************************
// Opóźnienie okolo 5us (dla kwarcu 8MHz)
// ********************************************************************************

void delay5us(void)
{
   u08 i = 8;
   while(--i)
       asm volatile(„nop”);
}

// ********************************************************************************
// Zapis rejestrów karty CF i wysłanie żądanej komendy
// ********************************************************************************
void cf_ata_command(u32 lba, u16 count, u08 cmd)
{
   u16 cyl, head, sector;

   // przeliczenie adresu LBA na dane dla rejestrow cylindra, głowicy i sektora
   sector  = (u16) ( lba & 0x000000ffL );
   lba = lba >> 8;
   cyl = (u16) ( lba & 0x0000ffffL );
   lba = lba >> 16;
   head   = (u16) ( lba & 0x0fL );

   ATAPI_DrvHead = 0xE0 | head;       // rejestr głowicy, Tryb LBA / Drive0 / head

   while ((ATAPI_Status & (SR_BUSY|SR_DRDY))==SR_BUSY); // Czekaj na !BUSY i DRY

   // Ustawienie pozostałych rejestrów karty
   ATAPI_CylHi = cyl>>8;           // Cylinder MSB
   ATAPI_CylLo = cyl;              // Cylinder LSB
   ATAPI_SectorCount = (u08)count; // liczba sektorów
   ATAPI_Sector = sector;          // numer sektora

   // I na koniec wywołanie żądanej komendy
   ATAPI_Cmd = cmd;                // komenda ATA
   delay5us();                     // odczekanie 5us
}

// ********************************************************************************
// Odczekanie na gotowość do transmisji danych
// Zwraca kod błędu 0->Nastąpił Błąd, 1->OK
// ********************************************************************************
u08 cf_wait_drq(void)
{
   u08 stat;

   // odczytuj w pętli rejestr ALT STATUS dopóki BUSY = 0
   while ( (ATAPI_AltStat & SR_BUSY) == SR_BUSY );

   // odczytaj rejestr STATUS i zresetuj żądanie przerwania
   stat = ATAPI_Status;

   if ( stat & SR_ERR )            // błąd jeśli w rejestrze statusu
       return 0;                   // jest ustawiony bit błędu

   if ( ! (stat & SR_DRQ) )        // błąd jeśli nie wystawiony sygnał DRQ
       return 0;

   return 1;                       // wszystko OK
}
```

List. 2. - cd.

```
// ********************************************************************************
// Odczyt sektora o numerze w "lba" do bufora
// Zwraca kod błędu 0->Nastąpił Błąd, 1->OK
// ********************************************************************************
u08 cf_read_data(u32 lba, u08 *buffer)
{
   u08 r;
   u16 i;

   cf_ata_command(lba, 1, 0x20);    // komenda odczytu sektora
   r = cf_wait_drq();               // odczekanie na sygnał DRQ

   for (i=0;i<512;i++)              // transfer danych
       buffer[i] = ATAPI_Data;
   return r;
}

// ********************************************************************************
// Zapis bufora do sektora o numerze w "lba"
// Zwraca kod błędu 0->Nastąpił Błąd, 1->OK
// ********************************************************************************
u08 cf_write_data(u32 lba, u08 *buffer)
{
   u08 r;
   u16 i;

   cf_ata_command(lba, 1, 0x30);    // komenda zapisu sektora
   r = cf_wait_drq();               // odczekanie na sygnał DRQ

   for (i=0;i<512;i++)              // transfer danych
        ATAPI_Data = buffer[i];
   return r;
}

// ********************************************************************************
// Identyfikacja karty CF. Zwrócone dane znajdą się w buforze „buffer"
// Zwraca kod błędu 0->Nastąpił Błąd, 1->OK
// ********************************************************************************
u08 cf_identify(u08 *buffer)
{
   u08 r;
   u16 i;

   cf_ata_command(0, 0, 0xEC);      // komenda odczytu danych identyfikacyjnych
   r = cf_wait_drq();               // odczekanie na sygnał DRQ

   for (i=0;i<512;i++)              // transfer danych
       buffer[i] = ATAPI_Data;
   return r;
}

// ********************************************************************************
// Wybranie karty CF jako Drive 0 oraz programowy reset karty CF
// Zwraca kod błędu 0->OK, 1->Błąd, 2->Błąd sygnatury resetu karty
// ********************************************************************************
u08 cf_reset(void)
{
   ATAPI_DevCtrl = 0x06;            // Ustaw bity SW RST i -IEn
   delay5us();                      // odczekanie 5us
   delay5us();                      // odczekanie 5us
   ATAPI_DevCtrl = 0x02;            // Ustaw bit -IEn i wyzeruj SW RST
   delay5us();                      // odczekanie 5us
   delay5us();                      // odczekanie 5us
   // Czekaj na gotowość karty
   while ( (ATAPI_Status & (SR_BUSY|SR_DRDY)) != SR_DRDY );

   if (ATAPI_Sector != 1 )          // Sprawdź sygnaturę resetu
       return (2);                  // Wróć z błędem sygnatury resetu
   return (ATAPI_Status & 1);       // Zwróć status błędu
```

List. 3.

```
#include <avr/io.h>
#include „cf.h"
u08 buf[512];                   // Bufor danych w pamięci RAM mikrokontrolera

void send_buf(void)
{
   u16 i;
   for(i=0; i<512; i++)
   {
       while( !(UCSR0A & (1<<UDRE)) );    // Czekaj na gotowość nadajnika
       UDR0 = buf[i];           // Wyślij bajt z bufora
   }
}

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

   u08 i;
   cf_reset();                  // Reset karty
   cf_identify(buf);            // Identyfikacja karty
   send_buf();                  // Wyślij dane z identyfikacji
   for(i=0; i<10; i++)
   {
       cf_read_data(i, buf);    // Odczytaj sektor o adresie LBA w zmiennej i
       send_buf();              // Wyślij dane przez UART
   }
   while(1);                    // Koniec pracy
}
```

su i identyfikacji karty. Wszystkie trzy są do siebie podobne. Najpierw wywołują *cf_ata_command*, przekazując do niej adres LBA, żądanie jednego sektora oraz odpowiednią komendę (0x20, 0x30 lub 0xEC). W przypadku komendy identyfikacji, adres LBA jest ustawiany na zero. Kolejno następuje odczekanie na gotowość do transferu danych i na koniec 512 odczytów lub zapisów rejestru danych karty CF. Przy odczycie dane są umieszczane w buforze **buffer*, a przy zapisie do karty - kolejno z niego pobierane.

Ach, byłbym zapomniał... Oczywiście przyda się nam jeszcze funkcja zerująca kartę CF. Działa ona według algorytmu opisanego w poprzedniej części tego kursu, więc nie będę się na jej temat zbytnio rozpisywał. Dodam jedynie, że zwraca ona status karty po zerowaniu, i jeśli wynosi on 0, to karta jest gotowa do pracy, jeśli 1, to powód błędu znajduje się w rejestrze błędów, a jeśli 2, to karta wykonała komendę, ale zerowanie nie przebiegło poprawnie i rejestr numeru sektora nie przyjął wartości 1, tak jak powinien. Jeśli z jakichś powodów procedura zerowania karty nie powiodła się, może to oznaczać, że np. przypadkowo wprowadziliśmy kartę w tryb *True IDE*, podając stan niski na styk 9 karty (sygnał -OE/-ATA_SEL) podczas załączania zasilania karty. Jest na to tylko jeden sposób: wyłączyć i ponownie załączyć zasilanie karty lub całego urządzenia. Wprawdzie według specyfikacji karta powinna na tym wejściu posiadać wewnętrzny rezystor podciągający je do VCC, ale jeśli przez przypadek wystąpią opisane wyżej problemy, można spróbować im zaradzić poprzez dodanie zewnętrznego rezystora o wartości około 10 kΩ pomiędzy styk 9 karty a VCC.

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

W następnej części kursu przedstawię drugi z omawianych typów kart pamięci, czyli karty *Multimedia Card* (MMC).

Romulad Biały