Mikrokontrolery STM32

Obsługa kart SD i modułu FatFs

Ilości przechowywanych i przetwarzanych informacji nieustannie rosną. Coraz częściej systemy wbudowane są zmuszane do zarządzania dużą ilością danych. W artykule omówiono sposób wykorzystania kart pamięci SD i modułu FatFs w połączeniu z mikrokontrolerami STM32. Biorąc pod uwagę nieustanny spadek cen kart SD warto poważnie zastanowić się nad wykorzystaniem tego typu nośnika w aplikacji, która wymaga administrowania nieprzeciętną ilością danych. Przykładowe aplikacje przygotowano dla płytki ewaluacyjnej STM3210B-EVAL.

Co to jest? Dzowiesz się na końcu artykułu

Karty SD

Współcześnie najpopularniejsze i najbardziej uniwersalne są karty SD (*Secure Digital card*). Standard obejmuje karty o pojemności do 4 GB, a jego rozszerzenie, czyli SDHC (*Secure Digital High Capacity*), aż do 32 GB.

Standard kart SD został pierwotnie opracowany przez trzy firmy: Matsushita, SanDisk i Toshiba. Pierwsze nośniki danych tego typu pojawiły się pod koniec 2000 roku. Początkowo dokumentacja standardu SD była dosyć trudno dostępna, jednak sytuacja uległa zmianie wraz z nadejściem roku 2006, kiedy to stały się dostępne informacje m. in. na temat interfejsu SDIO, co w efekcie pozwoliło na implementację w mikrokontrolerach sprzętowych sterowników kart SD. Przedstawiciele najbardziej zaawansowanej grupy układów z rodziny STM32 mają wbudowany właśnie taki sterownik.

Rys. 1.

Obecnie na rynku karty SD można spotkać najczęściej w dwóch typach obudów: zwykłej SD i SDMicro. Obie przedstawiono na **rys. 1**.

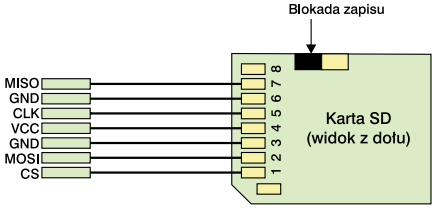
W zasadzie wszystkie karty SD dostępne na rynku obsługują dwa standardy komunikacyjne: dedykowany SDBus oraz SPI. Jak to zwykle bywa, interfejs natywny (SDBus) oferuje duże możliwości i dużą szybkość pracy, ale za cenę wzrostu stopnia skomplikowania obsługi interfejsu. Z tego powodu w kartach SD dostępna jest również komunikacja za pomocą o wiele prostszej w obsłudze magistrali SPI, z tym, że tutaj mamy nieco okrojone możliwości. Jeśli tylko aplikacja nie wymaga wyjątkowo dużej szybkości przesyłania danych, to nie ma jakiegokolwiek sensu implementacja obsługi interfejsu SDBus, wystarczy praca z magistralą SPI. Przedstawione wyżej podejście znakomicie upraszcza sprawę, ponieważ zdecydowana większość dostępnych mikrokontrolerów (w tym oczywiście **STM32F103**) jest wyposażona w sprzętowy kontroler SPI.

Sposób podłączenia karty SD przez magistralę SPI do mikrokontrolera został przedstawiony na **rys. 2**. Podobny układ został wykorzystany na płytce ewaluacyjnej STM3210B-EVAL.

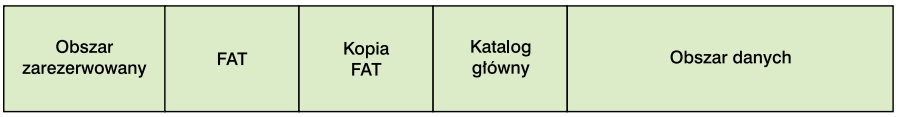
Komendy kart pamięci SD

Nie będziemy wnikać w budowę rejestrów, w jakie wyposażone są karty SD, ponieważ nie to jest celem niniejszego artykułu. Do osiągnięcia naszego celu, czyli uruchomienia obsługi systemu plików FAT w mikrokontrolerze STM32 z użyciem modułu FatFs, wystarczy znajomość przede wszystkim budowy komend oraz ich kilku typów, jakie obsługują karty SD.

Każda komenda, która ma być wysłana do karty SD, składa się z sześciu bajtów. Pierwszym bajtem jest zawsze kod komendy, kolejne cztery bajty to jej argument. Na końcu jest przesyłany bajt sumy kontrolnej CRC. O ile w trybie pracy z interfejsem SDBus CRC jest sprawdzane, to przy komunikacji za pomocą magistrali SPI, suma kontrolna jest przez kartę ignorowana. Tylko w trakcie przesyłania komendy CMD0, przełączającej tryb pracy z SDBus na SPI jest wymagany bajt CRC. Nie trzeba go w żaden sposób obliczać, ponieważ jest to stała wartość i wynosi 0x95. W **tab. 1** umieszczono kilka komend



Rys. 2.



Rys. 3.

Tab. 1.

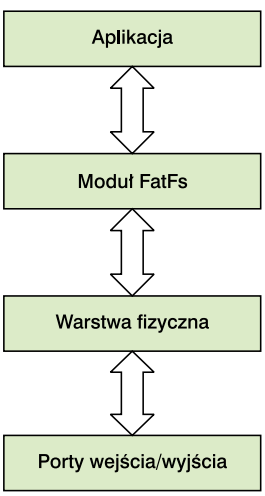
komenda	opis
CMD0	zeruje kartę, pozwala włączyć tryb pracy z magistralą SPI
CMD12 CMD16 CMD17	wymuszenie zakończenia transmisji wielu bloków danych konfiguracja długości bloku danych dla zapisu/odczytu odczyt bloku pamięci o długości określonej przez CMD16
CMD24 CMD32 CMD33	zapis bloku pamięci o długości określonej przez CMD16 w argumencie jest przesyłany adres pierwszego bloku przeznaczonego do skasowania w argumencie jest przesyłany adres ostatniego bloku przeznaczonego do skasowania
CMD38	kasuje bloki wyznaczone za pomocą CMD32 i CMD33

obsługiwanych w trybie pracy z magistralą SPI wraz z opisem argumentów. Oprócz standardowych komend CMD karty SD mogą wykorzystywać jeszcze tak zwane komendy aplikacji (ACMD). Wysłanie komendy aplikacji wymaga uprzedniego wysłania komendy CMD55, która informuje kartę SD, że następna będzie komenda ACMD.

System plików FAT

Z perspektywy systemu plików, każdy nośnik danych (dysk twardy, karta pamięci) podzielony jest na **sektory** i **klastery**. Sektorem nazywamy najmniejszą liczbę bajtów, jaką można zapisać lub odczytać. Zwykle sektor ma rozmiar 512 bajtów. Pliki zapisywane są w numerowanych klastrach. Rozmiar klastra jest zależny od systemu plików i nośnika danych. Każdy klaster jest w całości przydzielony do danego pliku, co oznacza, że nawet jeśli plik jest dużo mniejszy od rozmiarów klastra, to i tak na dysku zajmuje tyle, co pojedynczy klaster.

Kluczowym elementem systemu plików FAT (*File Allocation Table*) jest, zgodnie z jego nazwą, tablica alokacji plików. System plików FAT występuje w sumie w czterech odmianach, przy czym w systemach wbudowanych zazwyczaj wykorzystuje się dwie, w zależności od rozmiarów nośnika i wymagań aplikacji będzie to FAT16 lub FAT32.



Rys. 4.

Nośnik danych w systemie plików FAT jest podzielony na pięć części, wszystkie przedstawiono na **rys. 3**. Pierwszą logiczną częścią nośnika danych, umieszczoną w pierwszym sektorze, jest obszar zarezerwowany, który zawiera wszystkie podstawowe informacje na temat bieżącej partycji (nośnika). Do tych informacji zaliczają się m. in.: typ i rozmiar partycji, rozmiar sektora, ilość sektorów w klastrze.

Za obszarem zarezerwowanym znajdują się tablice alokacji plików, które są podstawowym źródłem informacji na temat danych zapisanych na nośniku. Zwykle, oprócz głównej tablicy alokacji, występuje również jej kopia. Czwarty obszar to katalog główny, który jest zakładany automatycznie w trakcie tworzenia systemu plików. Ostatni, piąty region, to obszar danych.

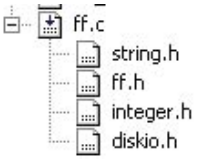
Moduł FatFs

System plików w urządzeniach wbudowanych można zaimplementować na dwa sposoby: pisząc obsługę systemu plików od podstaw, lub też wykorzystać gotowe rozwiązania. W zasadzie nie ma żadnego sensownego uzasadnienia pierwsze wymienione podejście. System plików FAT jest na tyle dobrze udokumentowany, a przy tym stosunkowo prosty, że powstało wiele darmowych narzędzi, które radzą sobie bardzo dobrze z administracją zawartości nośnika danych z systemem plików FAT. Z reguły otwarty charakter kodu pozwala na wprowadzenie koniecznych zmian i poprawek, które mogą się okazać niezbędne dla stabilności pracy urządzenia.

Jednym z takich ogólnodostępnych narzędzi jest moduł FatFs, którego zadaniem jest stanowienie pomostu pomiędzy warstwą

List. 1.

```
static
void SELECT (void)        // CS w stan niski

{
        GPIO_ResetBits(GPIOC, GPIO_Pin_12);

}



static

void DESELECT (void)              // CS w stan wysoki

{
        GPIO_SetBits(GPIOC, GPIO_Pin_12);

}



static

void xmit_spi (BYTE Data)         // Wyslanie bajtu do SD

{
        // Wyslanie bajtu

        while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET);
        SPI_I2S_SendData(SPI1, Data);

}



static

BYTE rcvr_spi (void)              // Odebranie bajtu z SD

{
        u8 Data = 0;

        // Wyslanie 0xFF

        while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET);
        SPI_I2S_SendData(SPI1, 0xFF);

        // Odebranie bajtu

        while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET);
        Data = SPI_I2S_ReceiveData(SPI1);

        return Data;

}
```



Rys. 5.

fizyczną (nośnikiem pamięci), a aplikacją uruchomioną na mikrokontrolerze. Szczegółowych informacji na temat FatFs należy szukać na jego stronie internetowej – *http://elm-chan.org/fsw/ff/00index_e.html*. Rola modułu FatFs w systemie wbudowanym została zilustrowana na **rys. 4**.

Sam moduł FatFs jest napisany w języku C, a zatem jest całkowicie niezależny od sprzętu. Pliki, które są niezbędne do poprawnej pracy FatFs zostały przedstawione na **rys. 5** w formie drzewa skopiowanego z projektu wykorzystującego system plików FAT.

Teoretycznie, do poprawnej pracy moduł FatFs wymaga obecności w systemie wbudowanym zegara czasu rzeczywistego (RTC). Można ten wymóg bardzo łatwo obejść wpisując stałe wartości w miejsce daty i czasu.

Implementacja modułu FatFs w mikrokontrolerach STM32 – warstwa fizyczna

Do opracowania warstwy sprzętowej został wykorzystany przykładowy projekt, zamieszczony na stronie internetowej modułu FatFs. Wszystkie funkcje, których zadaniem jest sterowanie urządzeniami peryferyjnymi mikrokontrolera, oraz zapis i odczyt danych z karty pamięci, zostały umieszczone w jednym pliku *sd_stm32.c*.

Bezpośrednio ze sprzętem komunikują się cztery funkcje. Za odbieranie danych z kontrolera magistrali SPI odpowiada funkcja *rcvr_spi()*, która została przedstawiona, wraz z pozostałymi trzema na **list. 1**. Wysyłaniem bajtów przez SPI do karty pamięci zajmuje się funkcja *xmit_spi()*. Do zadań interfejsu sprzętowego należy jeszcze sterowanie sygnałem wyboru układu CS, co należy do obowiązków funkcji *SELECT()* i *DESELECT()*.

Czasem, gdy mikrokontroler zażąda dostępu do zasobów karty pamięci, może się okazać, że ta ostatnia jest w danym momencie zajęta wykonywaniem innych operacji. Wtedy istotna jest możliwość sprawdzania zajętości karty. W tym celu została napisana funkcja *wait_ready()*, przedstawiona na **list. 2**. Jej zadaniem jest oczekiwanie przez maksymalny czas 500 ms, aż odebrany bajt będzie miał wartość 0xFF. Jeżeli w ciągu 500 ms nie zostanie odebrany bajt 0xFF, to funkcja kończy swoje działanie, zwracając ostatnią wartość odczytaną z kontrolera SPI.

Mamy już wszystkie niezbędne funkcje zajmujące się interfejsem sprzętowym, jednak, aby mogły one w ogóle pracować, to sam sprzęt musi zostać odpowiednio skonfigurowany.

Za konfigurację kontrolera SPI, portów wejścia/wyjścia i ich sygnałów zegarowych odpowiada funkcja *power_on()*, którą zamieszczono na **list. 3**. Po zdefiniowaniu zmiennych, wykorzystywanych w dalszym kodzie funkcji, następuje włączenie sygnałów zegarowych dla wyprowadzeń (porty GPIOA i GPIOC) i kontrolera SPI. Następnie konfigurowane jest wyprowadzenie PC12 do sterowania wyborem układu (CS) oraz piny PA5, PA6, PA7 jako linie magistrali SPI.

Kontroler SPI jest ustawiany jako master do pracy w trybie *full*

List. 2.
```
static

BYTE wait_ready (void)

{

        BYTE res;


        Timer2 = 50;    // Czeka przez 500ms

        rcvr_spi();

        do

                res = rcvr_spi();

        while ((res != 0xFF) && Timer2);


        return res;
}
```

List. 3.
```
static

void power_on (void)

{
        GPIO_InitTypeDef  GPIO_InitStructure;
        SPI_InitTypeDef   SPI_InitStructure;
        u8 i, cmd_arg[6];
        u32 Count = 0xFFF;

        // Konfiguracja wyprowadzen i kontrolera SPI:

        // Wlaczenie sygnalow zegarowych dla peryferiow
        RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOC |
                                RCC_APB2Periph_SPI1 | RCC_APB2Periph_AFIO,
ENABLE);

        // PA4 jako CS
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
        GPIO_Init(GPIOC, &GPIO_InitStructure);

        //SCK, MISO and MOSI
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
        GPIO_Init(GPIOA, &GPIO_InitStructure);

        // Konfiguracja SPI
        SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
        SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
        SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
        SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;
        SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
        SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
        SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_4;
        SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
        SPI_InitStructure.SPI_CRCPolynomial = 7;
        SPI_Init(SPI1, &SPI_InitStructure);

        // Wlacz SPI
        SPI_Cmd(SPI1, ENABLE);


        // Inicjalizacja karty i przelaczenie w tryb SPI:

        DESELECT();               // CS = 1


        for (i = 0; i < 10; i++)
                xmit_spi(0xFF);   // Wyslij 0xFF 10 razy = 80 cykli zegarowych
                                  // (wymaganych co najmniej 74 cykli)
        SELECT();                 // CS = 0

        // Przygotowanie ramki inicjujacej do wyslania
        cmd_arg[0] = (CMD0 | 0x40);
        cmd_arg[1] = 0;   // Argument komendy
        cmd_arg[2] = 0;   // nawet, gdy komenda go nie ma
        cmd_arg[3] = 0;   // musi zostac wyslany w postaci zer
        cmd_arg[4] = 0;
        cmd_arg[5] = 0x95;        // CRC = 0x95

        for (i = 0; i < 6; i++)           // Wyslanie ramki
                xmit_spi(cmd_arg[i]);

        while ((rcvr_spi() != 0x01) && Count) // Czeka na 0x01
                Count--;

        DESELECT();               // CS = 1
        xmit_spi(0XFF);   // Wyslij 0xFF


        PowerFlag = 1;
}
```

List. 4.
```
void SysTick_Conf(void)
{
        // SysTick bedzie taktowany z f = 72MHz/8 = 9MHz
        SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8);

        // Przerwanie ma byc co 10ms, f = 9MHz czyli liczy od 90000
        SysTick_SetReload(90000);

        // Odblokowanie przerwania od timera SysTick
        SysTick_ITConfig(ENABLE);

        // Wlaczenie timera
        SysTick_CounterCmd(SysTick_Counter_Enable);
}
```

Tab. 2.

komenda	opis
f_mount	„montuje" (rejestruje dysk logiczny w systemie
f_open	otwieranie i/lub tworzenie pliku
f_close	zamykanie pliku
f_read	czytanie zawartości pliku
f_write	zapisywanie pliku
f_lseek	przesuwa wskaźnik zapisu/odczytu pliku
f_truncate	skraca długość pliku
f_sync	działanie podobne do f_close, z tym, że plik pozostaje otwarty, więc dalej można wykonywać na nim operacje
f_opendir	otwiera katalog
f_readdir	czyta zawartość katalogu
f_getfree	pozwala odczytać liczbę wolnych klastrów
f_stat	odczytuje informacje o pliku/katalogu
f_mkdir	tworzy katalog
f_unlink	usuwa katalog lub plik
f_chmod	zmienia atrybuty pliku lub katalogu, np. plik może być tylko do odczytu
f_utime	zmienia datę i czas dla określonego pliku lub katalogu
f_rename	zmiana nazwy lub przeniesienie pliku/katalogu
f_mkfs	tworzy system plików na nośniku
f_forward	czyta dane z nośnika i bezpośrednio przekazuje dalej

dupleks. Ramka danych będzie wynosić 8 bitów, a zatrzaskiwanie stanu linii będzie następować na zboczu narastającym sygnału zegarowego. W stanie nieaktywnym na linii SCK będzie występował stan wysoki.

Preskaler dla zegara kontrolera SPI został ustawiony na 4, co oznacza, że dane będą przesyłane z niebagatelną prędkością 18 Mbit/s. Po ustawieniu wszystkich parametrów SPI, kontroler zostaje włączony przez wywołanie funkcji *SPI_Cmd()*.

Od tego momentu mikrokontroler jest już prawidłowo skonfigurowany, natomiast karta SD domyślnie po włączeniu zasilania pracuje w trybie obsługi dedykowanego standardu SDBus. Aby komunikacja (odbieranie komend) była w ogóle możliwa, należy w pierwszej kolejności wysłać co najmniej 74 cykle zegarowe, w celu zainicjowania karty. Następnie, żeby przejść do trybu pracy z SPI, należy wysłać komendę CMD0. Jeśli inicjalizacja karty do pracy w trybie SPI zostanie przeprowadzona poprawnie, to karta zwróci bajt potwierdzenia wynoszący 0x01.

Moduł FatFs wymaga do pracy sygnału zegarowego, który co 10 ms będzie wywoływał funkcję *disk_timerproc()*, która jest wykorzystywana dalej do odmierzania czasu. Do cyklicznego wywoływania wymienionej wyżej funkcji został wykorzystany 24 – bitowy timer SysTick. Jego konfiguracja została przedstawiona na **list. 4**.

Domyślnie główny zegar systemowy, po powieleniu przez układ PLL, wynosi 72 MHz i z taką częstotliwością domyślnie jest taktowany SysTick. Aby uzyskać przerwanie co 10 ms zastosowano preskaler, dzielący sygnał 72 MHz przez 8, co w efekcie daje 9 MHz. Jeśli chcemy, aby funkcja obsługi przerwania od timera SysTick (*SysTickHandler()*) była wywoływana z częstotliwością 100 Hz, to należy sprawić, aby licznik liczył od 90000. Bardziej szczegółowo timer SysTick został omówiony w EP12/08, natomiast w tym przypadku funkcja obsługi jego przerwania wygląda następująco:

```
void SysTickHandler(void)
{
      disk_timerproc();
}
```

Omówione funkcje są jedynymi zależnymi od sprzętu fragmentami kodu w module FatFs, zatem teraz zajmiemy się już najwyższą jego warstwą, umożliwiającą operacje na plikach i katalogach.

Podstawowe operacje na plikach i katalogach

Gdy mikrokontroler umie się już porozumiewać z kartą SD i system plików jest należycie obsługiwany, to kolejnym krokiem są już właściwe operacje na plikach i katalogach, jakie wymaga projektowana aplikacja. Wszystkie funkcje, jakie oferuje moduł FatFs zostały omówione na jego stronie internetowej *http://elm-chan.org/fsw/ff/00index_e.html*, a ich spis umieszczono w **tab. 2**. Tutaj zajmiemy się przykładami aplikacji wykonywujących niezbędne zadania na plikach i katalogach.

Przykładowy program, którego zadaniem są podstawowe operacje na plikach i katalogach pokazano na **list. 5**. Przedstawiony kod

najpierw montuje dysk logiczny za pomocą funkcji *f_mount()*, dzięki czemu moduł FatFs będzie mógł wykonywać operacje dyskowe. W argumentach przesyłamy numer dysku (tutaj 0) oraz, przez referencję, główną zmienną systemu plików typu *FATFS*.

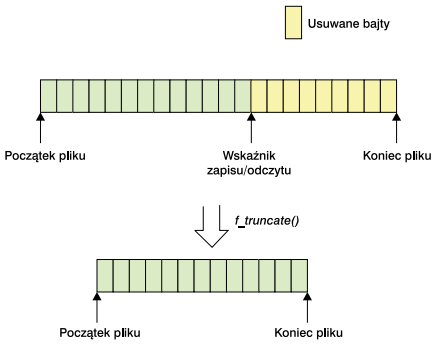
Po zamontowaniu dysku można już dowolnie zarządzać zawartością karty pamięci. W omawianym programie w pierwszej kolejności następuje utworzenie nowego pliku tekstowego w katalogu głównym karty. Wykorzystano do tego celu funkcję *f_open()*, która z pozoru (nazwy) nie ma z czynnościami tworzenia plików wiele wspólnego. Mimo tego, że nazwa funkcji na to nie wskazuje, to służy ona również do tworzenia nowych plików.

Jako pierwszy argument przekazujemy (prze referencję) adres do uchwytu pliku. Warto na to zwrócić uwagę, ponieważ, jak się przy okazji omawiania pozostałych funkcji okaże, jeśli operacje są wykonywane na pliku, to zawsze przekazujemy jego uchwyt w ten sposób. Drugim argumentem jest łańcuch znaków, który będzie stanowił nazwę pliku, w przedstawionym przykładzie będzie to plik tekstowy o *plik.txt*. Jako trzeci argument przesyłamy żądanie akcji, jaka ma być wykonana. Dla funkcji *f_open()* wszystkie możliwe wartości ostatniego argumentu pokazano w **tab. 3**. Na omawianym list. 5 widać, że program tworzy nowy plik, niezależnie, czy wcześniej istniał na karcie pamięci plik o nazwie *plik.txt*, czy nie.

Każda funkcja z modułu FatFs zwraca wartość typu FRESULT. Generalnie, jeśli zwracana wartość wynosi 0, to wszystko przebiegło prawidłowo, w przeciwnym wypadku wystąpiły błędy.

Oprócz tworzenia plików warto również dysponować mechanizmem pozwalającym na tworzenie katalogów. W module FatFs takiego mechanizmu dostarcza funkcja *f_mkdir()*. W argumencie do funkcji przekazujemy nazwę katalogu. Jeśli katalog nie ma być utworzony w jakimś innym katalogu to wystarczy jego nazwa, a jeśli chcemy utworzyć folder jako podkatalog, to należy podać całą jego ścieżkę. Ostatni przypadek przedstawia poniższa linia kodu:

```
fresult = f_mkdir(„katalog1/
katalog2”);
```



Rys. 6.

List. 5.

```
int main(void)
{
        FRESULT fresult;
        FIL plik;
        WORD zapisanych_bajtow;

        RCC_Conf();
        GPIO_Conf();
        SysTick_Conf();

        fresult = f_mount(0, &g_sFatFs);

        // Tworzenie pliku
        fresult = f_open (&plik,”plik.txt”, FA_CREATE_ALWAYS);
        fresult = f_close (&plik);

        // Tworzenie katalogu
        fresult = f_mkdir(„katalog1”);

        // Zapis pliku
        fresult = f_open (&plik,”plik.txt”, FA_WRITE);
        fresult = f_write(&plik, „zawartosc pliku”, 15, &zapisanych_bajtow);
        fresult = f_close (&plik);

        // Usuniecie pliku
        fresult = f_unlink(„plik.txt”);

        while(1);
}
```

Tab. 3.

sposób otwarcia	opis
FA_READ	plik otwierany do odczytu
FA_WRITE	plik otwierany do zapisu
FA_OPEN_EXISTING	otwarcie pliku, jeśli plik nie istnieje to zostanie zgłoszony błąd
FA_OPEN_ALWAYS	otwarcie pliku, jeśli nie istnieje to zostanie stworzony nowy plik
FA_CREATE_NEW	utworzenie nowego pliku, jeżeli plik istnieje to zostanie zgłoszony błąd
FA_CREATE_ALWAYS	utworzenie nowego pliku, jeżeli już istnieje to nadpisanie na stary plik

Jeśli można tworzyć pliki i katalogi, to wypadałoby również móc je usuwać. Zapewnia to wywołanie funkcji *f_unlink()*, w argumencie należy podajać nazwę pliku lub katalogu przeznaczonego do usunięcia. Zapis do pliku, po jego wcześniejszym otwarciu do zapisu, wykonuje się przez wywołanie funkcji *f_write()*. Oprócz uchwytu do pliku przekazujemy tablice z bajtami przeznaczonymi do zapisu oraz liczbę zapisywanych bajtów. Ostatnim argumentem jest przekazanie przez referencję zmiennej typu DWORD, do której zostanie wpisana faktyczna ilość zapisanych do pliku bajtów.

Administrowanie zawartością karty pamięci może niekiedy wymagać skracania długości pliku, czyli innymi słowy zmniejszenia jego rozmiarów. Można tego dokonać dzięki funkcji *f_truncate()* (*truncate* – skracać), której wywołanie może wyglądać podobnie poniżej zamieszczonej linii kodu:

```
fresult = f_truncate(&plik);
```

Jak widać, jedynym wymaganym do przekazania argumentem jest adres do uchwytu pliku, czyli zmiennej (struktury) zawierającej wszystkie informacje na temat pliku, niezbędne do jego poprawnego egzystowania w systemie plików. Funkcja *f_truncate()* do skracania długości pliku wykorzystuje aktualny wskaźnik zapisu/odczytu. Mechanizm zmniejszania rozmiarów pliku przedstawio-

List. 6.

```
/* File status structure */
typedef struct _FILINFO {
        DWORD fsize;            /* Size */
        WORD fdate;             /* Date */
        WORD ftime;             /* Time */
        char fname[13];         /* Name (8.3 format) */
} FILINFO;
```

List. 7.

```
fresult = f_opendir(&Dir, „katalog1”);

if(fresult != FR_OK)
        return(fresult);

for(;;)
{
        fresult = f_readdir(&Dir, &plikInfo);

        if(fresult != FR_OK)
                return(fresult);

        if(!plikInfo.fname[0])
                break;

        // p jest wkaznikiem na tablice elemetnow typu FILINFO
        *p++ = plikInfo;
}
```

List. 8.
```
void LCD_BMP(u8 * nazwa_pliku)
{
        u32 i = 0, j = 0, liczba_pikseli = 0, liczba_bajtow =0;
        u16 piksel;
        u8 temp[4];
        WORD ile_bajtow;
        FRESULT fresult;
        FIL plik;

        // Otwarcie do odczytu pliku bitmapy
        fresult = f_open (&plik, (const char *) nazwa_pliku, FA_READ);

        // Opuszczenie dwoch pierwszych bajtow
        fresult = f_read (&plik, &temp[0], 2, &ile_bajtow);
        // rozmiar pliku w bajtach
        fresult = f_read (&plik, (u8*) &liczba_bajtow, 4, &ile_bajtow);

        // Opuszczenie 4 bajtow
        fresult = f_read (&plik, &temp[0], 4, &ile_bajtow);
        // Odczytanie przesuniecia (offsetu) od poczatku pliku do
        // poczatku bajtow opisujacych obraz
        fresult = f_read (&plik, (u8*) &i, 4, &ile_bajtow);
        // Opuszczenie liczby bajtow od aktualnego miejsca
        // do poczatku danych obrazu, wartosc 14, bo odczytane zostalo
        // juz z pliku 2+4+4+4=14 bajtow
        for(j = 0; j < (i - 14); j++)
                fresult = f_read (&plik, &temp[0], 1, &ile_bajtow);
        // Liczba pikseli obrazu = (rozmiar pliku - offset)/2 bajty na pisel
        liczba_pikseli = (liczba_bajtow - i)/2;

        // Ustawienie parametrow pracy LCD (m. in. format BGR 5-6-5)
        LCD_WriteReg(R3, 0x1008);
        LCD_WriteRAM_Prepare();

        // Odczyt bajtow z karty SD i wyslanie danych do LCD
        for(i = 0; i < liczba_pikseli; i++)
        {
                fresult = f_read (&plik, (u8*) &piksel, 2, &ile_bajtow);
                LCD_WriteRAM(piksel);
        }

        LCD_CtrlLinesWrite(GPIOB, CtrlPin_NCS, Bit_SET);
        // Przywrocenie ustawien LCD
        LCD_WriteReg(R3, 0x1018);

        // Zamyka plik
        fresult = f_close (&plik);
}
```

no na **rys. 6**. Mechanizm skracania długości pliku może znaleźć zastosowanie na przykład w systemach akwizycji dużej ilości danych, gdzie niekiedy zachodzi potrzeba odrzucenia części zebranych informacji.

Przeglądanie zawartości karty pamięci, informacje o plikach i katalogach

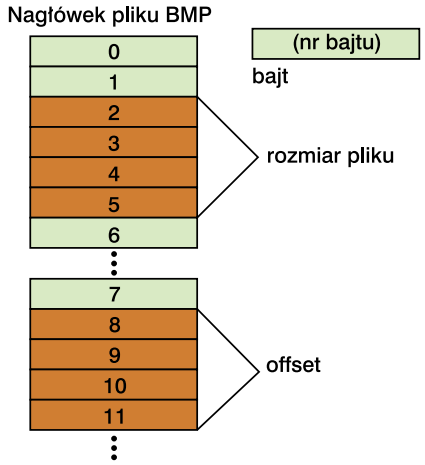
Informacje na temat plików w module FatFs można uzyskać poprzez wywołanie funkcji *f_stat()*, wystarczy w programie umieścić następującą linię kodu:

```
fresult = f_stat(„plik.txt”, &plikInfo);
```

Jako pierwszy argument należy podać łańcuch znaków zawierający nazwę pliku, ściślej wskaźnik na początek tablicy z nazwą pliku.

Informacje zostają zapisane do struktury typu *FILINFO*, której budowa pokazano na **list. 6**, a jej adres jest przekazywany jako drugi argument w wywołaniu funkcji *f_stat()*. Dane na temat pliku, jakie otrzymujemy to: rozmiar, data ostatniej modyfikacji, czas modyfikacji, atrybuty pliku, nazwa w tablicy 13 elementowej (format 8+3).

Przeglądu zawartości całego katalogu można dokonać z wykorzystaniem funkcji *f_readdir()*. Przykładowy program, który zbiera informację na temat tego, co znajduję się w folderze *katalog1* został przedstawiony na **list. 7**.



Rys. 7.

Po otwarciu katalogu za pomocą funkcji *f_opendir()* w nieskończonej pętli następuje odczytywanie informacji o kolejnych plikach i folderach przez skopiowanie struktury wyżej omówionego typu (*FILINFO*). Kolejne wywołania funkcji *f_readdir()* powodują samoczynne czytanie informacji o następujących po sobie elementach katalogu. Jeśli pole nazwy elementu będzie puste, to oznacza wtedy, że nie ma już w danym folderze elementów i wykonywanie pętli zostaje przerwane.

List. 9.
```
int main(void)
{
        FRESULT fresult;
        FATFS g_sFatFs;
        DIR Dir;
        FILINFO PlikInfo;

        RCC_Conf(); NVIC_Conf(); GPIO_Conf(); SysTick_Conf();

        // Inicjalizuj LCD
        STM3210B_LCD_Init();

        // Wyczysc LCD, tlo czarne
        LCD_Clear(Black);

        fresult = f_mount(0, &g_sFatFs);
        // „Otworzenie” katalogu glownego
        fresult = f_opendir(&Dir, „/”);

        if(fresult != FR_OK)
                return(fresult);

        while (1)
        {
                if(!GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_9))
                {
                        fresult = f_readdir(&Dir, &PlikInfo);

                        // Jesli ostatli plik to przegladanie
                        // od poczatku
                        if(!PlikInfo.fname[0])
                        {
                                fresult = f_opendir(&Dir, „/”);
                                fresult = f_readdir(&Dir, &PlikInfo);
                        }

                if(fresult != FR_OK)
                                return(fresult);

                        LCD_DrawBMP((u8*) PlikInfo.fname);
                }
        }
}
```

Przeglądarka obrazów

Płytka ewaluacyjna STM3210B-EVAL jest wyposażona w graficzny wyświetlacz LCD o rozmiarach 320 na 240 pikseli, który umożliwia wyświetlanie obrazów w 262000 kolorów. Mając do dyspozycji karty pamięci i graficzny LCD można zbudować przyjazny, graficznie rozbudowany interfejs użytkownika. Do obsługi wyświetlacza LCD zostały wykorzystane funkcje dostarczane przez firmę STMicroelectronics.

Jedną z możliwych aplikacji, która pokazuje sposób współpracy karty SD i wyświetlacza graficznego, jest przeglądarka obrazów zapisanych na karcie pamięci. Zadaniem przeglądarki obrazów będzie odczytywanie zawartości plików w formacie BMP z karty SD, a następnie pokazywanie obrazu na LCD.

Żeby dobrze zrozumieć działanie programu, najpierw należy zapoznać się z budową plików w formacie bitmapy (BMP). W pierwszym przybliżeniu każdy plik BMP składa się z nagłówka oraz właściwych danych opisujących obraz. Nagłówek zawiera informacje m. in. na temat rozmiaru pliku, rozmiarów obrazu w pikselach itd. Do zaprogramowania przeglądarki obrazów z nagłówka wystarczy wyłuskać informacje na temat liczby pikseli z jakiej składa się obraz oraz miejsce w pliku, gdzie rozpoczynają się bajty opisujące piksele.

Budowę aplikacji znacznie uproszczą wstępne wymagania co do plików graficznych. Zakładamy, że obrazy mają być dopasowane do wyświetlacza, a zatem ich wymiary muszą mieć 320×240 pikseli. Plik graficzny musi być zakodowany w formacie 16 bitowej bitmapy, ponieważ taki typ wspiera wyświetlacz graficzny dostarczany wraz z STM3210B-EVAL. Taki tryb kodowania oznacza, że na każdy piksel przypadają dwa bajty. Niestety standardowe narzędzia Windowsa nie umożliwiają zapisu pliku bitmapy w formacie 16-bitowym, jest jednak wiele programów dostępnych w Internecie, które umożliwiają konwersję bitmap. Po przygotowaniu kilku obrazów do testów i wgraniu ich na kartę pamięci do katalogu głównego, można już przystąpić do budowy i testowania aplikacji.

Cały kod, jaki jest niezbędny do poprawnego odczytania i pokazania na LCD zawartości pliku bitmapy, został zapisany w funkcji *LCD_BMP()*, którą przedstawiono na **list. 8**. Jako argument do funkcji należy przesłać łańcuch znaków reprezentujący nazwę pliku. Funkcja po zadeklarowaniu niezbędnych zmiennych otwiera, znajdujący się w katalogu głównym karty, plik o nazwie podanej przesłanej w argumencie wywołania. Pierwszą niezbędną informacją, jaką należy uzyskać jest rozmiar całego pliku w bajtach, który jest zapisany na czterech bajtach z przesunięciem 2 bajtów (**rys. 7**). Podczas odczytywania rozmiaru pliku wykorzystano ciekawy sposób. Ponieważ rozmiar jest reprezentowany przez cztery bajty, to nie ma możliwości bezpośredniego uzyskania rozmiaru w zmiennej 32 bitowej. Problem został rozwiązany przez przekazanie do funkcji czytającej bajty z pliku zmiennej 32-bitowej z wcześniejszym rzutowaniem na wskaźnik 8-bitowy. Dzięki temu funkcja *f_read()* będzie zapisywać pojedyncze komórki pamięci tak, jakby była to tablica czteroelementowa. W efekcie uzyskany zostanie efekt zapisu całej zmiennej 32-bitowej wartością określającą rozmiar pliku.

Zgodnie z przedstawionym nieco wcześniej rys. 7 teraz należy opuścić cztery bajty, by następnie odczytać przesunięcie danych obrazu w stosunku do początku pliku. Na podstawie uzyskanego offsetu zostaje wyznaczona liczba piskeli, z jakiej składa się obraz, po czym w pętli odczytywane są bajty pikseli i wysyłane do wyświetlacza LCD.

Główna funkcja programu została zamieszczona na **list. 9**. Jej zadanie polega na konfiguracji urządzeń peryferyjnych do pracy oraz na reagowaniu na naciśnięcie przycisku użytkownika na płycie ewaluacyjnej. Z każdym naciśnięciem będzie wyświetlany kolejny obraz z karty SD. Działanie aplikacji, z przykładowym obrazem na wyświetlaczu LCD, ilustruje fotografia otwierająca artykuł.

Krzysztof Paprocki
poprocki.krzysztof@gmail.com