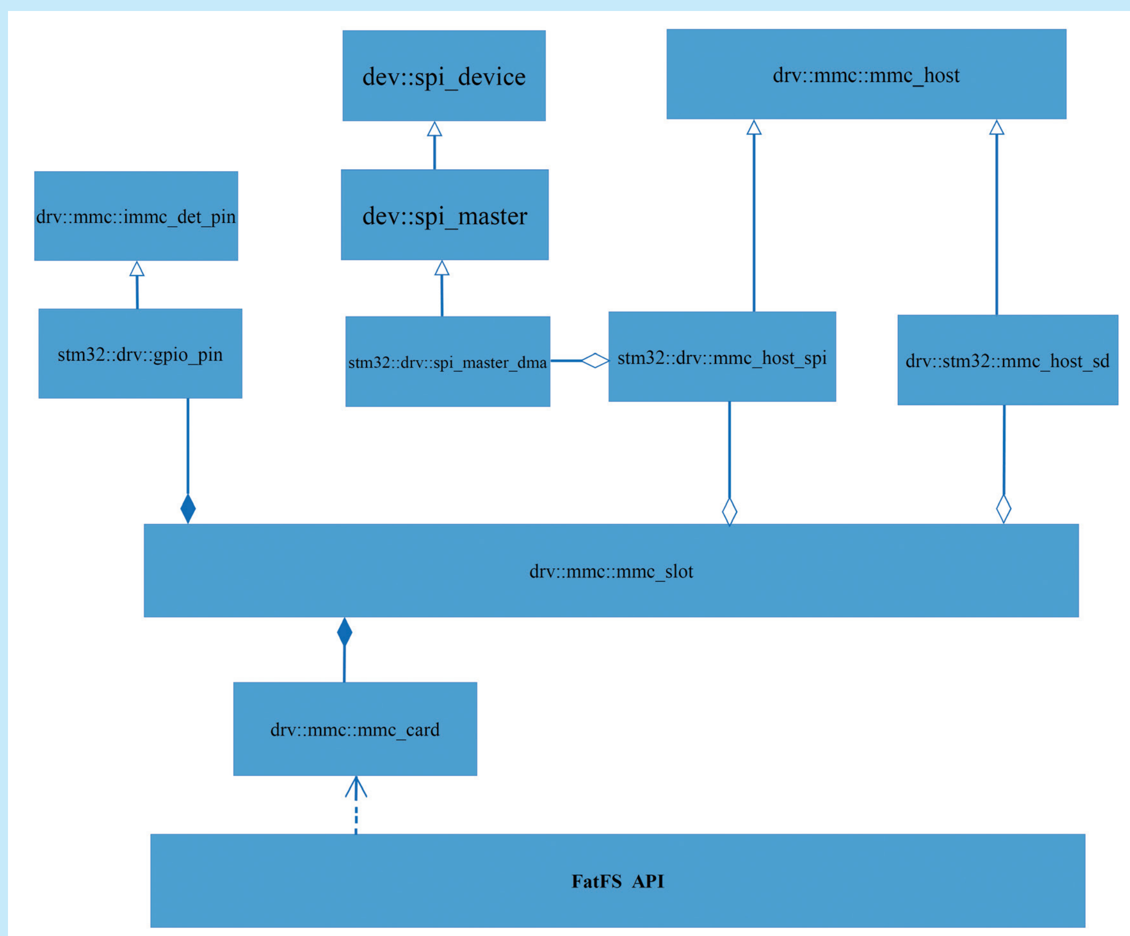
Obsługa kart SD w systemie ISIXRTOS

Niemal w każdym nowoczesnym urządzeniu elektronicznym zachodzi potrzeba przechowywania dodatkowych danych zewnętrznych. W najprostszym przypadku, jeśli są to jedynie dane konfiguracyjne, najczęściej będziemy się w stanie zadowolić niewielką pamięcią EEPROM wbudowaną w mikrokontroler lub dołączoną w postaci układu zewnętrznego. W przypadku zapisu większej ilości danych najbardziej sensownym rozwiązaniem wydaje się zastosowanie karty SD lub microSD, która w większości wypadków będzie miała najlepszy stosunek pojemności do ceny. Obecnie karty mieszczące 2 GB, możemy nabyć już w cenie kilkunastu złotych.

Karty SD komunikują się z kontrolerem nadrzędnym za pomocą dedykowanej magistrali SDBus. Dodatkowo wyposażone są interfejs kompatybilny z SPI, dzięki czemu, mogą być stosunkowo łatwo obsłużone nawet przez najprostsze mikrokontrolery 8-bitowe. Największym mankamentem pracy w trybie SPI, jest mniejsza prędkość transmisji danych. Nie bez znaczenia jest również fakt, że dla kart microSD (wypierających obecnie standardowe karty SD) kompatybilność z magistralą SPI jest opcjonalna i może nie być zaimplementowana przez producenta. Nabywając kartę uSD nie mamy zatem pewności, czy będzie ona współpracowała z naszym urządzeniem korzystającym komunikację z kartą po SPI, co niejednokrotnie może być powodem frustracji. O ile magistrala SPI jest dostępna praktycznie w każdym układzie, magistrala SDBus jest domeną bardziej zaawansowanych układów. W wypadku mikrokontrolerów z rodziny STM32 większość małych układów np. STM32F100 czy nawet STM32F107 nie ma wbudowanego kontrolera. Dopiero bardziej zaawansowane układy, takie jak STM32F2/F4 mają sprzętowy układ SDBus, co zapewne jest spowodowane kwestiami licencyjnymi. Jeśli chcemy napisać uniwersalną bibliotekę zapewniającą możliwie największą wydajność oraz współpracującą z każdym mikrokon-



Rysunek 1. Diagram klas projektu

trolerem, należy zapewnić obsługę kart zarówno w trybie SDBus jak i SPI, co jest zadaniem dość skomplikowanym.

Dzięki zastosowaniu biblioteki libSD systemu ISIXRTOS obsługa i używanie kart SD niezależnie od użytego mikrokontrolera oraz trybu pracy SPI lub SDIO z poziomu API jest trywialna. Do najważniejszych cech biblioteki libSD obsługującej karty SD w systemie ISIX zaliczamy:

- Obsługa kart SD z wykorzystaniem natywnej magistrali SD w trybie 1bit oraz 4 bit.
- Obsługa kart SD w trybie kompatybilności z magistralą SPI.
- Obsługa kart SD oraz SDHC o pojemności > 2 GB.
- Łatwość rozbudowy biblioteki o obsługę dodatkowych kontrolerów, dzięki obiektowej budowie.
- Możliwość rozbudowy o inne urządzenia dołączane do magistrali SD np. moduły SDIO

Opis obsługi kart SD w systemie ISIXRTOS

Biblioteka libSD implementuje interfejs blokowy dla kart SD, czyli dostęp do przestrzeni adresowej karty na poziomie sektorów, nie ingerując w sposób organizacji danych na karcie. Za organizację danych odpowiedzialna jest warstwa systemu plików. W przypadku kart SD i SDHC standardowym systemem plików, w którym sformatowane są nowo nabyte katy jest FAT. W systemie ISIX do obsługi systemu plików postanowiono wykorzystać gotowe rozwiązanie w postaci biblioteki FATFS. Biblioteka FATFS kieruje żądania odczytu/zapisu sektorów do biblioteki libSD, które następnie przez bibliotekę przetwarzane są na odpowiednie polecenia i kierowane za pośrednictwem kontrolera do karty. Nic nie stoi na przeszkodzie aby wykorzystać inne biblioteki implementujące inne systemy plików na przykład ext2 znany z systemu linux. Biblioteka libSD została napisana w języku C++ (ISO/IEC 14882:1998). Diagram klas projektu przedstawiono na **rysunku 1**.

Na rysunku przedstawiono kompletną hierarchię klas biblioteki, zarówno dla trybu SPI, jak i dla trybu natywnego z magistralą SD. Pomimo, że całość wydaje się stosunkowo skomplikowana, używanie biblioteki jest stosunkowo proste co pokażemy w dalszej części artykułu. Z punktu widzenia użytkownika końcowego API blokowego dostępu do karty stanowią klasy **mmc_slot** oraz klasa **mmc_card.** Klasa **mmc_slot** reprezentuje fizyczne gniazdo karty SD natomiast klasa **mmc_card** jest obiektem reprezentującym kartę pamięci SD. Obiekt klasy **mmc_card** zarządzany jest przez klasę gniazda **mmc_slot.** Zarządzanie obiektem polega na tym, że w momencie włożenia karty tworzony jest obiekt **mmc_card** odpowiedzialny za obsługę karty natomiast w momencie wyciągania karty obiekt **mmc_card** jest niszczony. W przyszłości planowane jest dodatnie innych typów karty niż karty pamięci SD np. karty SDIO, wówczas w zależności od rodzaju karty tworzony będzie odpowiedni obiekt reprezentujący odpowiedni rodzaj karty. Klasa **mmc_slot** odpowiedzialna jest również za generowanie zdarzeń w momencie wkładania lub wyciągania karty informujących program użytkownika o dostępności karty w slocie. Aby można było odwołać się do karty i zapisać lub odczytać sektory należy zaczekać na włożenie karty, a następnie pobrać od obiektu **mmc_slot** obiekt klasy **mmc_card**. Po pobraniu instancji tego obiektu możemy odwoływać się do odpowiedniego API uzyskując dostęp do zawartości pamięci karty. Wykrywanie i wkładanie karty realizowane jest poprzez klasę **mmc_slot,** która do sprawdzania stanu karty wykorzystuje klasę abstrakcyjną **immc_det_pin.** Klasa ta stanowi interfejs abstrakcji sterownika urządzeń zawierający metodę

```
virtual bool get() const = 0;
```

Zadaniem tej metody jest zwrócenie wartości **true**, w przypadku gdy karta dołączona jest do gniazda, lub **false** jeśli karty brak w slocie. Implementacja powyższego interfejsu realizowana jest w klasie **stm32:drv::gpio_pin**, która stanowi warstwę sterownika specyficzną dla danej platformy i w praktyce sprowadza się do sprawdzenia stanu odpowiedniej linii GPIO. Komunikacja z kartą realizowana jest za pomocą kontrolera SD, którego reprezentacją jest abstrakcyjna klasa bazowa **drv::mmc::mmc_host**.

Klasa slotu podczas tworzenia obiektu przyjmuje referencje do klasy kontrolera. W zależności od mikrokontrolera, którym dysponujemy (tylko SPI lub dedykowany SD) należy utworzyć klasę wywodzącą się z podanej klasy bazowej implementującej wybrany tryb pracy.

Jeśli do dyspozycji mamy układ wyposażony w natywny kontroler magistrali SD należy utworzyć instancję obiektu klasy **drv::stm32::mmc_host_sd** , a następnie przekazać referencje do niego klasie **mmc_slot**.

Klasa **mmc_host_sd** należy do warstwy sterownika urządzenia i dziedziczy z klasy bazowej **drv::mmc::mmc_host**. Jest to implementacja specyficzna dla sprzętowego kontrolera SD rodziny stm32. Tworząc inną klasę możemy stworzyć sterownik dla innego typu kontrolera w zależności od zastosowanego sprzętu. Przykładowy kod tworzący potrzebne obiekty do pracy w natywnym trybie SDIO może wyglądać następująco:

```
stm32::drv::mmc_host_sdio m_mmc_
host(config::PCLK2_HZ, 6000);
stm32_gpio m_pin;
drv::mmc::mmc_slot m_slot m_slot( m_mmc_
host, m_pin );
```

Najpierw jest tworzony obiekt klasy **mmc_host_sdio**, gdzie w konstruktorze jako argument przekazana jest częstotliwość pracy magistrali PCLK, oraz częstotliwość z jaką chcemy aby pracował interfejs SDIO wyrażoną w kHz W tym przypadku jest to 6 MHz. (Standardowe karty mogą pracować z częstotliwością do 25 MHz.) Następnie tworzony jest obiekt **stm32_gpio**, którego zadaniem jest sprawdzanie stanu dostępności karty. Po utworzeniu instancji kontrolera oraz pinu ostatnią czynnością jest utworzenie obiektu slotu.

Jeśli do dyspozycji nie mamy sprzętowego kontrolera SD a jedynie SPI, wówczas należy skorzystać z pracy karty w trybie kompatybilności. W tym celu należy stworzyć obiekt klasy **mmc_host_spi**, a następnie przekazać referencję do tego obiektu klasie **mmc_slot**. Z uwagi na to, że implementacja protokołu SD za pomocą trybu kompatybilności z SPI jest niezależna od sprzętu oraz aby uzyskać maksymalną elastyczność i możliwość ponownego użycia kodu **mmc_host_spi** zamiast bezpośredniego dostępu do kontrolera SPI, wykorzystuje dodatkową warstwę abstrakcji magistrali SPI. Klasa kontrolera hosta SPI przyjmuje referencję do abstrakcyjnej klasy bazowej **spi_device** stanowiącej generyczny interfejs obsługi magistrali SPI. Klasa pochodna implementująca ten interfejs stanowi bezpośrednio warstwę sterownika

urządzenia odpowiedzialną za fizyczną komunikację ze sprzętem. W przypadku implementacji specyficznej dla mikrokontrolerów rodziny stm32 (tryb DMA) implementacja interfejsu bazowego realizowana poprzez klasę **stm32_spi_master_dma**. Przykładowy kod tworzący obiekty potrzebne do pracy w trybie zgodności SPI jest nieco bardziej skomplikowany:

```
stm32::drv::spi_master_dma m_spi(SPI1,CONFIG_PCLK1_HZ, CONFIG_PCLK2_HZ);
drv::mmc::mmc_host_spi m_mmc_host( m_spi,11000 );
stm32_gpio m_pin;
drv::mmc::mmc_slot m_slot( m_mmc_host, m_pin );
```

W pierwszym etapie tworzony jest obiekt klasy kontrolera SPI, który w konstruktorze przyjmuje identyfikator fizycznego kontrolera SPI, oraz częstotliwości pracy magistral PCLK1 oraz PCLK2. Następnie tworzony jest obiekt kontrolera karty SD, który w konstruktorze jako referencję przyjmuje obiekt klasy kontrolera SPI, oraz częstotliwość pracy interfejsu wyrażoną w kHz. W dalszej części inicjalizacja wygląda identycznie jak w poprzednim przypadku: tworzony jest obiekt odpowiedzialny za obsługę pinu wykrywania karty, oraz obiekt slotu, który w konstruktorze przyjmuje referencję do kontrolera hosta SD oraz utworzonej wcześniej obiektu klasy *stm32_gpio*.

Przykład praktyczny

Celem zaprezentowania możliwości biblioteki przygotowano dwa identycznie działające przykłady na dwie różne platformy: STM32Butterfly z mikrokontrolerem STM32F107 bez kontrolera SD oraz ZL41ARM_F4 wyposażonym w jednostkę STM32F417, która ma wspomniany kontroler. Przykłady działają identycznie. Jedyną różnicą (poza szybkością komunikacji) jest wykorzystanie trybu SPI w zestawie STM32Buterfly oraz natywnego interfejsu SD w zestawie ZL41ARM_F4. Ponieważ żaden z zestawów nie ma gniazda SD dodatkowo będziemy musieli użyć minimodułu KamodMMC.

Działanie obu przykładów będzie polegało na:

- Oczekiwaniu na włożenie karty SD.
- Zamontowaniu systemu plików włożonej karty SD.
- Odczytanie pliku *toread.txt* i wyświetlenie jego zawartości na konsoli szeregowej.
- Zapis do plik *towrite.txt* krótkiego tekstu.

Podłączenie karty SD do zestawu. Aby uruchomić prezentowane przykłady należy podłączyć minimoduł KamodMMC do zestawów ewaluacyjnych. Moduł karty SD zaprojektowano do współpracy z mikrokontrolerami zasilanymi napięciem 5 V i w związku z zastosowaniem dodatkowego stabilizatora 3,3 V powinien być zasilany napięciem 5 V. W **tabeli 1** przedstawiono opis wyprowadzeń minimodułu.

Minimoduł należy połączyć z zestawem za pomocą odpowiednich przewodów. Dla zestawu STM32Butterfly pracującym w trybie SPI obie płytki należy połączyć według informacji umieszczonych w **tabeli 2**.

W przypadku zestawu ZL41ARM_F4 wykorzystujemy tryb natywnej 4-bitowej magistrali SD, zapewniającej przy standardowej prędkości transmisji 25 MHz przepustowość 100 Mbps. W związku z tym minimoduł KamodMMC należy dołączyć do zestawu zgodnie z **tabelą 3**.

Tabela 1. Opis wyprowadzeń minimodułu

Nr styku	Nazwa	Opis (SD)	Opis (MMC)	Kierunek
1	+5V	Plus zasilania (zalecane +5 V)	Plus zasilania (zalecane +5 V)	Zasilania
2	DAT3 (CD/DAT3)/(DT3)	Linia danych D3	Chip Select (aktywne 0)	Wejście/wyjście
3	DAT2 (DT2)	Linia danych D2	-	Wejście/wyjście
4	CMD	Linia do wprowadzania poleceń	MOSI	Wejście/wyjście
5	DAT1 (DT1)	Linia danych D1	IRQ (opcja)	Wejście/wyjście
6	CLK	Wejście sygnału zegarowego karty	SCK	Wejście
7	CDET	Wejście sygnalizacji obecności karty w slocie (aktywne 0)	Wejście sygnalizacji obecności karty w slocie (aktywne 0)	Wyjście
8	DAT0 (DT0)	Linia danych D0	MISO	Wejście/wyjście
9	WP	Wejście sygnalizacji zabezpieczenia karty przed zapisem (aktywne 0)	Wejście sygnalizacji zabezpieczenia karty przed zapisem (aktywne 0)	Wyjście
10	GND	Masa zasilania	Masa zasilania	Zasilanie

Tabela 2. Połączenie KAMODMMC z STM32Butterfly

KAMODMMC	Złącze zestawu	PIN uC
#1	Conn6 #1	+5V
#2	Conn6 #2	PA4
#3	-	-
#4	Conn6 #3	PA7
#5	-	-
#6	Conn6 #5	PA5
#7	Conn1 #1	PD0
#8	Conn6 #4	PA6
#9	-	-
#10	Conn6 #6	GND

Tabela 3. Połączenie KAMODMMC z ZL41ARM_F4

KAMODMMC	Złącze zestawu	PIN uC
#1	Conn2 #1	+5V
#2	JP3 #7	PC11
#3	JP3 #5	PC10
#4	JP4 #5	PD2
#5	JP3 #3	PC9
#6	JP3 #4	PC12
#7	JP3 #6	PC13
#8	JP3 #1	PC8
#9	-	-
#10	Conn2 #6	GND

Poza dołączeniem modułu gniazda zewnętrznej karty SD należy również podłączyć do komputera PC konsolę szeregową, na której wypisywany będzie rezultat działania naszej aplikacji. Konsola szeregowa została skonfigurowana do pracy z prędkością 115200, 8 bitów danych, 1 bit stopu. Po stronie PC możemy wykorzystać dowolny program terminalowy na przykład **minicom**. Dla zestawu STM32Buterfly linię danych **TX** konsoli stanowi port **PD5** dostępny na pinie **#6** złącza **CONN1**. Natomiast dla zestawu ZL41ARM_F4 linię danych **TX** stanowi również port **PD5** który dostępny jest na pinie **#4** złącza **JP4**.

Oprogramowanie – dostęp do plików w systemie FAT. Wszystkie przykłady zostały skompilowane z wykorzystaniem toolchaina **GCC** w wersji **4.8.2**, który jest dostępny na mojej stronie internetowej pod adresem *http://goo.gl/CZg2fP*.

Przykłady dla obu zestawów zostało dołączone do wzorcowych projektów dla systemu ISIXRTOS dostępnych na stronie domowej projektu, oraz w dziale download na stronie EP.

Program demonstracyjny dla zestawu STM32BUTTERFLY jest umieszczony w katalogu *advanced/fatsdio*, natomiast przykład dla zestawu ZL41ARM_F4 znajduje się w katalogu *stm32f4/fatsdio*. Aby skompilować wybrany przykład, należy wejść do odpowiedniego katalogu, a następnie wydać polecenie *make*. Załadowanie programu do mikrokontrolera możliwe jest za pomocą interfejsu JTAG zgodnego z ocdlink (np. BF30) poprzez wywołanie polecenia *make program*. Możemy również posłużyć się fabrycznym bootloaderem i załadować skompilowane oprogramowanie z wykorzystaniem jednego z kanałów komunikacyjnych np. portu szeregowego. Ponieważ kod programu testowego jest podobny pomijając użycie innych klas dla kontrolera, oba przykłady zostaną omówione równocześnie.

Program rozpoczyna wykonanie of funkcji **main()**, w której tworzony jest statyczny obiekt klasy **fat_tester**, w której zaimplementowana została cała funkcjonalność. Klasą bazową dla klasy **fat_tester** stanowi klasa **isix_task_base**, implementującą zadanie systemu ISIX.

W przypadku wykorzystania kontrolera SD klasa **fat_tester** zawiera następujące obiekty prywatne:

```
stm32::drv::mmc_host_sdio m_mmc_host;
stm32_gpio m_pin;
drv::mmc::mmc_slot m_slot;
```

Ponieważ powyższe klasy nie posiadają konstruktorów domyślnych, ich konstruktory wywoływane są w konstruktorze klasy zawierającej wspomniane obiekty:

```
fat_test()
  : task_base(STACK_SIZE,TASK_PRIO),
  m_mmc_host(config::PCLK2_HZ, 6000), m_
slot( m_mmc_host, m_pin )
  {
  }
```

Pierwszy wywoływany jest konstruktor klasy bazowej, **isix_task_base**, który jako parametr przyjmuje rozmiar stosu dla zadania **ISIX**. Kolejno wywoływany jest konstruktor obiektu hosta magistrali SD **m_mmc_host**, który jako argumenty przyjmuje prędkość magistrali **PCLK2** oraz częstotliwość pracy magistrali SD. Następnie wywoływany jest konstruktor klasy slotu, który jako argumenty przyjmuje referencję do obiektu kontrolera SD oraz klasy **stm32_gpio** odpowiedzialnej za detekcję obecności karty.

Dla przykładu dla z zestawem STM32Butterfly demonstrującym tryb zgodności SPI klasa **fat_test** zawiera następujące obiekty prywatne:

```
stm32::drv::spi_master_dma m_spi;
drv::mmc::mmc_host_spi m_mmc_host;
stm32_gpio m_pin;
drv::mmc::mmc_slot m_slot;
```

Jedyną różnicą pomiędzy przykładem przeznaczonym dla mikrokontrolera STM32F1 a wersją dla STM32F4 jest to, że zamiast klasy **mmc_host_sdio** tworzony jest kontroler **mmc_host_spi** pracujący w trybie SPI. Dodatkowo tworzony jest obiekt **spi_master_dma**, który stanowi sterownik dla SPI. Inicjalizacja klas zawartych w klasie **fat_tester** realizowana jest podobnie jak poprzednio:

```
//Constructor
fat_tester()
  : task_base(STACK_SIZE,TASK_PRIO),
    m_spi(SPI1,CONFIG_PCLK1_HZ, CONFIG_
PCLK2_HZ),
    m_mmc_host( m_spi,11000 ), m_slot(
m_mmc_host, m_pin )
  {
  }
```

Najpierw inicjalizowany jest obiekt odpowiedzialny za obsługę SPI który jako parametry przyjmuje numer kontrolera sprzętowego oraz prędkości z jaką pracuje magistrala APB1 oraz APB2. Konstruktor obiektu klasy hosta SPI **m_mmc_host** przyjmuje referencję do klasy **m_spi**, oraz prędkość z jaką ma pracować magistrala komunikacyjna. Podobnie jak poprzednio ostatnią czynnością jest przekazanie referencji do obiektu **m_pin**, oraz **m_mmc_host** do obiektu **m_slot** klasy **mmc_slot.** Po zakończeniu działania konstruktorów oraz po zakończeniu inicjalizacji systemu scheduler rozpocznie wykonywanie metody *main()* klasy **fat_tester**, w której zawarty jest kod demonstracyjny. Kod jest identyczny dla obu platform. Zamieszczono go na **listingu 1**.

Metoda rozpoczyna działanie od wywołania funkcji **add_disc** biblioteki *fatFS* odpowiedzialnej za obsługę systemu plików FAT. Jako argumenty przekazywane są kolejno numer logiczny dysku, oraz wskaźnik do obiektu klasy slotu. Po dodaniu dysku do biblioteki program wywołuje metodę **check()** klasy mmc_slot, której deklaracja wygląda następująco:

```
//Wait for change status
int check( int timeout = isix::ISIX_TIME_
INFINITE );
```

Zadaniem tej metody jest oczekiwanie na zmiany statusu dostępności karty metoda ta zwraca wartość **card_inserted**, lub **card_removed** w zależności od zmiany statusu karty, lub wartość **isix::ETIMEOUT** w przypadku gdy w podanym czasie nie wystąpiła zmiana statusu karty. W **[1]** sprawdzamy czy karta jest dostępna, i jeśli metoda **check()** zwróci wartość **card_inserted**, wówczas wywoływana jest funkcja **f_mount()** biblioteki FATFS. Zadaniem tej funkcji jest zamontowanie oraz inicjalizacja systemu plików. Jeśli proces montowania przebiegł poprawnie, wówczas w **[3]** otwierany jest plik *toread.txt*. Jeśli również ta

m.ElektronikaB2B.pl
teraz zawsze pod ręką w Twoim smartfonie

Wejdź
Bądź dobrze poinformowany

operacja przebiegła bezbłędnie w **[4]**, odczytywana jest do bufora zawartość fragmentu pliku, która następnie za pomocą funkcji **tiny_printf**() wyświetlana jest na konsoli szeregowej. Po odczytaniu całości pliku wywoływana jest funkcja **f_close()** , oraz wyświetlany jest status zamknięcia pliku. W **[5]** otwierany jest nowy plik *write.txt* w trybie do zapisu, a następnie wyświetlany jest status otwarcia pliku. Jeżeli funkcja otwarcia się powiedzie, wówczas za pomocą **f_write()** do pliku wpisywana jest zawartość bufora **wstr**. Po zakończeniu zapisu plik jest zamykany (funkcja **f_close**), po czym cały cykl rozpoczyna się od początku.

Pomimo stosunkowo skomplikowania implementacji biblioteki użytkowanie z punktu widzenia użytkownika końcowego jest trywialne i sprowadza się do utworzenia kilku obiektów, a następnie skorzystaniu z funkcji dostępu do plików biblioteki **fatFS**. Poza momentem tworzenia obiektów kontrolera obsługa kart SD za pomocą opisanej biblioteki jest identyczna bez względu czy karta pracuje w trybie SD czy w trybie kompatybilności SPI.

Oprogramowanie – dostęp do karty na poziomie sektorów

Dla dużej części użytkowników, użytkowanie karty SD na poziomie plików systemu FAT będzie wystarczające. Jednak w niektórych przypadkach, przy implementacji własnego systemu plików, lub podczas zapisu danych na kartę bezpośrednio w trybie surowym RAW, przydatny może być interfejs na poziomie dostępu do sektorów. Aby uzyskać fizyczny dostęp do karty należy najpierw sprawdzić (lub poczekać na dostępność karty SD co realizujemy za pomocą wywołania omówionej wcześniej metody **check()** klasy **mmc_slot**. Jeśli metoda zwróci wartość **card_inserted** należy pobrać obiekt klasy **mmc_card** na którym będziemy wykonywać operację odczytu i zapisu, co zapewnia metoda:

```
//Get card status
int get_card( mmc_card* &card );
```

Przyjmuje ona referencję do wskaźnika na obiekt klasy **mmc_card** i w przypadku powodzenia wskaźnikowi temu przypisuje adres obiektu karty SD. Operując bezpośrednio na obiekcie tej klasy za pomocą odpowiednich metod możemy uzyskać dostęp do informacji o karcie czy odczytywać i zapisywać poszczególne sektory. Do najważniejszych metod zaliczamy:

```
/* Get card capacity */
uint32_t get_sectors_count() const
```

Metoda zwraca ilość sektorów dostępnych na karcie

```
/* Get sector size */
size_t get_sector_size() const
```

Metoda zwraca wielkość pojedynczego sektora.

```
/* Get card CID */
int get_cid( cid &c ) const;
```

Metoda zwraca strukturę cid zawierającą dane producenta karty , w przypadku powodzenia zwraca wartość success (0) lub inny kod błędu w przypadku niepowodzenia.

```
/** Write the block */
int write( const void* buf, unsigned long sector,  std::size_t count );
```

Metoda *write* zapisuje dane zawarte w buforze wskazywanym przez **buf** pod początkowy sektor

o adresie zawartym w parametrze **sector**, oraz ilości sektorów zawartych w parametrze **count**. W przypadku powodzenia zwraca wartość **success** *(0)*, lub inny kod błędy w przypadku niepowodzenia.

```
/** Read the block */
int read( void* buf, unsigned long sector, std::size_t count );
```

Metoda **read** pozwala odczytać zawartość sektorów karty i przesłać je do bufora. Parametr **buf** zawiera wskaźnik na bufor, do którego będą przekazane dane, parametr **sector** zawiera adres pierwszego sektora, natomiast parametr **count** zawiera liczbę sektorów do odczytania. W przypadku powodzenia zwraca wartość **sucess** *(0)*, lub kod błędy w przypadku niepowodzenia.

Listing 1. Przykładowy program – dostęp do plików w systemie FAT

```
static char buf[513];
FATFS fs;
int err;
FIL f;
UINT ccnt;
disk_add( 0, &m_slot );
for(;;)
{
  const int cstat = m_slot.check();
  if( cstat == drv::mmc::mmc_slot::card_inserted ) //[1]
  {
    std::memset(buf, 0, sizeof(buf) );
    err = f_mount(0, &fs); //[2]
    dbprintf(„Fat disk mount status %i”, err );
    if( !err )
    {
      int err = f_open(&f, „toread.txt”, FA_READ );  //[3]
      dbprintf( „Open file for read status=%i”, err );
      if( !err )
      {
        for(;;) //[4]
        {
          err = f_read(&f, buf,sizeof(buf)-1,&ccnt );
          if( err || ccnt == 0 ) break;
          if( ccnt > 0 )
          {
            buf[ccnt] = ‚\0’;
            fnd::tiny_printf(„%s”, buf);
          }
        }
        fnd::tiny_printf(„\r\n”);
        dbprintf(„Read finished ret=%i count=%u”, err, ccnt );
      }
      err = f_close( &f );
      dbprintf( „Close status %i”, err );
    }
    if(!err)
    {
       int err = f_open(&f, „write.txt”, FA_WRITE | FA_CREATE_ALWAYS );//[5]
      dbprintf( „Open file for write status=%i”, err );
      if(!err )
      {
        static const char wstr[] = „Ala ma kota a kot ma ale\r\n”;
        err = f_write( &f, wstr, sizeof(wstr)-1, &ccnt );
        dbprintf(„Write finished ret=%i count=%u”, err, ccnt );
        err = f_close( &f );
        dbprintf( „Close status %i”, err );
      }
    }
  }
  isix::isix_wait_ms( 100 );
}
```

Uwagi końcowe. Kilka praktycznych uwag na temat wydajności zapisu i odczytu

Zaprezentowana biblioteka systemu ISIX pozwala prosty dostęp do obsługi kart SD z poziomu użytkownika uniezależniając aplikację od zastosowanego interfejsu komunikacyjnego. Uwalnia ona użytkownika od żmudnego oprogramowywania komunikacji z kartą i umożliwia przygotowanie aplikacji karty SD w bardzo krótkim czasie. Dzięki integracji z ISIX-em biblioteki **fatFS** dostępnej na licencji *Open Source*, dostajemy dostęp do systemu plików FAT, dzięki czemu dane zapisywane na karcie, będą dostępne dla systemu operacyjnego w komputerze PC.

Praktyczne próby wykazały, że komunikacja za pośrednictwem natywnej 4-bitowej magistrali SD zapewnia dużo większą wydajność niż w przypadku pracy w trybie kompatybilności SPI.

Dodatkowego komentarza wymaga wielkość bufora pamięci RAM wykorzystywanego do zapisu oraz odczytu sektorów. W przypadku odczytu już bufor o wielkości pojedynczego sektora zapewnia przyzwoitą wydajność. Zwiększanie rozmiaru bufora podnosi wydajność z uwagi na mniejszy narzut na przygotowanie transferu, np. przeprogramowanie kontrolera DMA. Jednak zmiany wielkości bufora nie powodują drastycznych zmian wydajności. Zupełnie inaczej wygląda sprawa zapisu danych. W tym przypadku rozmiar bufora sektorów ma drastyczny wpływ na wydajności i fizycznie związany jest z rozmiarem strony pamięci Flash karty SD. Jeśli rozmiar bufora będzie dużo mniejszy niż rozmiar strony pamięci FLASH wówczas kontroler będzie musiał dzielić każdy transfer na części wykonywać dodatkowe operacje kasowania, i przeprogramowywania sektora, co ma drastyczny wpływ na wydajność oraz trwałość karty. Im karta pamięci posiada większy rozmiar tym strona pamięci FLASH jest większa i może nawet przyjmować wielości rzędu 128 kB i więcej dla kart 16 lub 32 GB. Z uwagi na to, iż pracując jedynie na wewnętrznych zasobach mikrokontrolera pamięć RAM jest deficytowym zasobem, dużo lepsze rezultaty przyniesie przy zapisie zastosowanie kart o stosunkowo niewielkiej z dzisiejszego punktu widzenia pojemności np. 2 GB

W zaprezentowanym przykładzie istnieje możliwość praktycznych prób wydajności poszczególnych kart podczas odczytu czy zapisu, poprzez utworzenie w aplikacji głównej obiektu klasy **mmc_host_tester**. Klasa ta przeprowadza szereg odczytów i zapisów fizycznych sektorów dla różnych wielkości bufora i wyświetla na ekranie prędkość, z którą została wykonana dana operacja. UWAGA! Test ten jest destruktywny dla danych zawartych na karcie, i po jego uruchomieniu dane zawarte na karcie zostaną utracone, a karta będzie wymagała ponownego sformatowania.

Lucjan Bryndza, EP