



USB



dla niewtajemniczonych
w przykładach
na mikrokontrolery **STM32**



Marcin Peczarski



btc



USB

dla niewtajemniczonych
w przykładach
na mikrokontrolery STM32



Marcin Pczarski

Książka powstała z myślą o konstruktورach i programistach systemów mikroprocesorowych niewtajemniczonych w arkana USB, który jest obecnie najpopularniejszym interfejsem komunikacyjnym, zarówno w urządzeniach przenośnych jak i stacjonarnych.

Autor na praktycznych przykładach pokazał, jak zaprogramować interfejsy USB w mikrokontrolerach STM32 z rdzeniami ARM Cortex-M3 lub Cortex-M4. Przykłady przetestowano na mikrokontrolerach STM32F103, STM32F107, STM32F207, STM32F407 oraz STM32L152, co daje pełny przekrój „silników” interfejsów USB stosowanych w mikrokontrolerach STM32.

Zamiarem autora było zainteresowanie tematyką USB w systemach embedded zarówno elektroników hobbystów, studentów kierunków związanych z elektroniką lub informatyką, jak i doświadczonych konstruktörów i programistów systemów mikroprocesorowych. Żeby opanować materiał przedstawiony w książce potrzebne jest minimalne doświadczenie w programowaniu mikrokontrolerów i znajomość podstaw języka C.

Sekretarz redakcji: *mgr Katarzyna Kempista*

Redaktor merytoryczny: *mgr Anna Kubacka*

Redaktor techniczny: *mgr Delfina Korabiewska*

ISBN 978-83-60233-93-1

© Copyright by Wydawnictwo BTC
Legionowo 2013

Dodatkowe materiały i oprogramowanie wykorzystane w książce jest dostępne pod adresem
www.MIKROKONTROLER.pl/wszystko_o_USB



Wydawnictwo BTC
ul. Lwowska 5
05-120 Legionowo
fax: (22) 767-36-33
<http://www.btc.pl>
e-mail: redakcja@btc.pl

Wydanie I

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawnictwo BTC dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletnie i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawnictwo BTC nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentów niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopianie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Druk i oprawa: Drukarnia TOTEM S.C. w Inowrocławiu

Przedmowa.....	7
Ważniejsze skróty użyte w książce.....	11
1. Architektura USB.....	13
1.1. Ogólna charakterystyka interfejsu	14
1.2. Warstwa fizyczna	17
1.3. Warstwa łącza.....	24
1.4. Warstwa protokołu.....	27
1.5. Deskryptory.....	36
1.6. Warstwa aplikacji dla danych sterujących.....	45
1.7. Wiadomości uzupełniające.....	51
2. Podstawy	55
2.1. Warianty sprzętu	56
2.1.1. Mikrokontrolery STM32.....	56
2.1.2. Płytki prototypowe.....	60
2.2. Struktura archiwum z przykładami	67
2.3. Pisanie programów dla wielu wariantów sprzętu	70
2.3.1. Jeden interfejs – wiele implementacji	70
2.3.2. Daj szansę kompilatorowi.....	71
2.3.3. Kompilacja warunkowa	73
2.3.4. Pliki konfiguracyjne.....	75
2.4. Biblioteka mikrokontrolera	79
2.4.1. Odmierzanie czasu	79
2.4.2. Diody świecące	81
2.4.3. Wyświetlacz ciekłokrystaliczny	83
2.4.4. Interfejs I ² C	88
2.4.5. Interfejs I ² S	89
2.4.6. Inicjowanie programu.....	91
2.4.7. Inicjowanie sprzętu	91
2.4.8. Parametry uruchamiania aplikacji.....	93
2.4.9. Przerwania	95
2.4.10. Wsparcie dla standardowej biblioteki języka C	98
2.5. Projekt wstępny	99
2.6. Kompilowanie programów	100
2.6.1. Narzędzia	100
2.6.2. Program make	101
2.6.3. Skrypt konsolidatora.....	112

2.7.	Uruchamianie przykładowych programów	112
2.7.1.	Uwagi dla użytkowników systemu Linux	113
2.7.2.	Uwagi dla użytkowników systemu Windows	117
2.8.	Dalsza lektura	117
3.	Typowe urządzenia USB	119
3.1.	Projekt urządzenia klasy HID.....	120
3.1.1.	Deskryptory	120
3.1.2.	Żądania	122
3.1.3.	Protokół fazy rozruchu dla myszy i klawiatury.....	124
3.1.4.	Implementacja myszy	126
3.1.5.	Dżojstik	139
3.1.6.	Funkcja main.....	142
3.1.7.	Kompilowanie i testowanie	143
3.2.	Projekt wirtualnego portu szeregowego	144
3.2.1.	Deskryptory	144
3.2.2.	Żądania i powiadomienia.....	147
3.2.3.	Implementacja.....	149
3.2.4.	Dioda świecąca mocy	160
3.2.5.	Kompilowanie i testowanie	161
3.3.	Projekt odtwarzacza audio.....	164
3.3.1.	Deskryptory	164
3.3.2.	Żądania	172
3.3.3.	Synchronizacja.....	174
3.3.4.	Przetwornik cyfrowo-analogowy	175
3.3.5.	Implementacja.....	180
3.3.6.	Kompilowanie i testowanie	187
4.	Biblioteki	189
4.1.	Biblioteka urządzenia USB dla STM32	190
4.1.1.	Makra, stałe i struktury danych	191
4.1.2.	Konfigurowanie urządzenia	192
4.1.3.	Interfejs programistyczny	194
4.1.4.	Rdzeń protokołu	206
4.1.5.	Przerwania	207
4.1.6.	Abstrakcja sprzętu	208
4.1.7.	Główna funkcja programu	214
4.1.8.	Wybrane fragmenty implementacji.....	214
4.2.	Biblioteka libusb	225
4.2.1.	Inicjowanie i zwalnianie biblioteki	226
4.2.2.	Wyszukiwanie i otwieranie urządzenia	227

4.2.3.	Wybieranie konfiguracji i rezerwowanie interfejsu	230
4.2.4.	Przesyłanie blokujące	231
4.2.5.	Przesyłanie nieblokujące.....	233
4.2.6.	Pozostałe funkcje.....	237
4.3.	Projekt urządzenia własnej klasy	238
4.3.1.	Deskryptory	238
4.3.2.	Żądania	238
4.3.2.	Implementacja.....	239
4.3.3.	Kompilowanie i testowanie	249
5.	Zarządzanie zasilaniem urządzenia USB	253
5.1.	Wymagania standardu i praktyczne sposoby ich realizacji	254
5.1.1.	Komentarz do zawartości standardu	254
5.1.2.	Rozszerzenie biblioteki urządzenia.....	257
5.2.	Projekt wirtualnego portu szeregowego zasilanego z szyny	259
5.2.1.	Implementacja.....	259
5.2.2.	Zdalne budzenie.....	263
5.2.3.	Kompilowanie i testowanie	266
6.	Projekt urządzenia USB wysokiej szybkości	269
6.1.	Deskryptory i żądania.....	270
6.2.	Protokoły pamięci masowej.....	272
6.2.1.	Protokół BOT	272
6.2.2.	Protokół SCSI	274
6.3.	Implementacja	276
6.3.1.	Pamięć zewnętrzna.....	276
6.3.2.	Protokół SCSI	276
6.3.3.	Protokoły USB i BOT	286
6.4.	Kompilowanie i testowanie.....	300
7.	Projekt kontrolera HID	305
7.1.	Biblioteka kontrolera USB dla STM32	306
7.1.1.	Kody błędów	307
7.1.2.	Abstrakcja sprzętu	308
7.1.3.	Niskopoziomowe wejście-wyjście	312
7.1.4.	Rdzeń protokołu	318
7.1.5.	Funkcje pomocnicze	323
7.1.6.	Przerwania kontrolera.....	324
7.1.7.	Wybrane fragmenty implementacji.....	325

7.2.	Obsługa myszy i klawiatury.....	342
7.2.1.	Protokół fazy rozruchu	342
7.2.2.	Program demonstrujący	345
7.2.3.	Kompilowanie i testowanie	349
8.	Projekt kontrolera pamięci masowej	351
8.1.	System plików.....	352
8.1.1.	Struktura aplikacji	352
8.1.2.	Implementacja.....	356
8.2.	Obsługa pamięci USB Flash.....	368
8.2.1.	Protokół BOT.....	368
8.2.2.	Protokół SCSI	384
8.3.	Przykład użycia.....	391
8.3.1.	Program demonstrujący	391
8.3.2.	Kompilowanie i testowanie	395
Dodatek.	Instalowanie narzędzi dla ARM Cortex-M	397
Literatura		405

Przedmowa

Niniejsza książka przeznaczona jest dla niewtajemniczonych w arkana USB, a w szczególności dla tych, którzy nigdy nie pisali aplikacji wykorzystujących mikrokontroler i interfejs USB, a chcieliby zacząć takie aplikacje tworzyć. Starałem się, aby zainteresowała zarówno hobbystów, jak i studentów kierunków związanych z elektroniką lub informatyką, ale także doświadczonych konstruktorów systemów mikroprocesorowych. Od Czytelnika oczekuję tylko, choćby minimalnego, doświadczenia w programowaniu mikrokontrolerów i znajomości przynajmniej podstaw języka C.

Jeszcze całkiem niedawno układy mikrokontrolerowe z dużymi komputerami łączone głównie za pomocą interfejsu szeregowego RS-232, a czasem równoległego Centronics. Niegdyś każdy komputer osobisty (pecet) był wyposażony w te interfejsy. Obecnie zostały one całkowicie wyparte przez uniwersalny interfejs szeregowy USB (ang. *Universal Serial Bus*). Jeśli potrzebujemy podłączyć do komputera mikrokontroler, który nie obsługuje USB, zwykle stosujemy konwerter USB/RS-232. Zaletą tego rozwiązania jest wsteczna kompatybilność – można używać istniejącego i przetestowanego oprogramowania zarówno po stronie mikrokontrolera, jak i po stronie komputera. Poważną wadą jest niewykorzystywanie wszystkich możliwości, jakie oferuje USB. Można też znaleźć w Internecie całkowicie programowe implementacje USB, symulujące ten interfejs na wejściach-wyjściach ogólnego przeznaczenia, czyli GPIO (ang. *General Purpose Input-Output*). Rozwiązania te obsługują tylko bardzo ograniczony podzbiór wymagań standardu USB, więc formalnie nie są z nim zgodne. Zatem nie są godne polecenia i nie są omawiane w tej książce.

Obecnie mikrokontrolery coraz częściej wyposażane są w układ peryferyjny realizujący sprzętowo interfejs urządzenia USB. Dzięki temu można wyeliminować konwerter (zmniejszając koszt urządzenia) i wykorzystać wszystkie zalety USB. Dodatkowo, jeśli ten układ peryferyjny obsługuje protokół kontrolera (ang. *host*) USB, można do mikrokontrolera podłączać urządzenia USB: mysz, klawiaturę, pamięć Flash itp. Niestety USB jest bardzo skomplikowanym interfejsem. Podstawowy dokument, opisujący wersję 2.0 standardu, ma ponad 600 stron. Każda klasa urządzeń opisana jest w dodatkowym dokumencie, a często w kilku osobnych dokumentach. Do tego dochodzą komentarze do standardu, opisy jego rozszerzeń i erraty. Razem jest to kilka tysięcy stron. USB zaprojektowano, aby ułatwić życie użytkownikom – urządzenie podłącza się do gniazda i już wszystko działa (ang. *plug and play*). Niestety łatwość używania jest okupiona sporym utrapieniem dla programistów. Protokoły USB są bardzo rozbudowane i trzeba się mocno napracować, aby je poprawnie zaimplementować. Obsługa USB od strony mikrokontrolera też przysparza sporo trudności. Aby zaprogramować UART lub USART obsługujący RS-232, wystarczy skonfigurować kilka rejestrów. Układ peryferyjny USB ma co najmniej kilkadziesiąt rejestrów, których funkcje trzeba poznać. Wszystko to

sprawia, że przejście tej pierwszej bariery programistycznej, napisanie pierwszej aplikacji korzystającej z USB jest dla początkującego niezwykle trudne. Celem tej książki jest możliwe bezbolesne wprowadzenie Czytelnika w świat USB.

Na kilku przykładach pokazuję, jak zaprogramować USB w mikrokontrolerze STM32 z rdzeniem ARM Cortex-M3 lub Cortex-M4. Przykłady starałem się napisać możliwie ogólnie. Jeśli w przyszłości pojawią się inne modele STM32 (również z rdzeniem Cortex-M0) z interfejsem USB, to przykłady te dadzą się na nie przeneść bez większych problemów. Wyraźnie rozdzielim warstwą zależną od sprzętu od warstwy protokołu. Warstwa protokołu, z uwagi na swoją niezależność od sprzętu, daje się bardzo łatwo zaadaptować na inne mikrokontrolery. Książka ta nie aspiruje do bycia wyczerpującym kompendium wiedzy o USB. Prezentowane przykłady nie demonstrują wszystkich możliwości USB. Z uwagi na ogrom tego, co zawiera standard, jest to po prostu niemożliwe. Opisuję głównie te zagadnienia, które mogą przydać się przy programowaniu mikrokontrolerów, w nadziei że staną się one inspiracją do własnych udanych projektów. Zainteresowany Czytelnik z pewnością zechce kiedyś poznać więcej szczegółów działania USB i zajrzy do standardu, co – jak myślę – po przeczytaniu tej książki będzie łatwiejsze. W efekcie książka ta jest bardziej o USB niż o mikrokontrolerach STM32, które dały pierwotny pretekst do jej napisania.

Do omówienia przykładów wybrałem kilka modeli STM32, tak aby reprezentowały one możliwie pełne spektrum dostępnych obecnie układów z tej rodziny mikrokontrolerów. Przykłady uruchamiałem na modelach STM32F103, STM32F107, STM32F207, STM32F407 i STM32L152, stosując zestawy STM3220G-EVAL, ZL29ARM, ZL30ARM, ZL31ARM oraz moduły STM32F4-Discovery, STM32L-Discovery, MMstm32F103Vx-0-0-0, a także specjalnie zaprojektowaną na potrzeby tej książki płytę nazywaną dalej gadżetem. Dzięki elastycznemu sposobowi konfigurowania układów peryferyjnych (zarówno wewnętrznych bloków mikrokontrolera, jak i układów zewnętrznych) przykładowe programy mogą być z łatwością przeniesione na inne modele STM32 wyposażone w interfejs USB oraz inne płytki.

W rozdziale 1 opisuję architekturę i podstawowe protokoły USB. Koncentruję się głównie na wersji 2.0, która jest obsługiwana przez układy peryferyjne USB zastosowane w STM32. Jest to jedyny „teoretyczny” rozdział w książce, tzn. niezawierający żadnego przykładu. Materiał przedstawiony we wszystkie pozostałe rozdziałach ilustruję przykładami programów. Jak wspomniałem wyżej, oficjalna dokumentacja USB jest bardzo rozbudowana, a wiele jej fragmentów przeznaczonych jest głównie dla osób, które chcą projektować sprzęt realizujący interfejs USB. Programista lub konstruktor układów mikroprocesorowych, chcący tworzyć aplikacje korzystające z USB, nie potrzebuje aż tak szczegółowej dokumentacji. Przystępując do pisania tej książki, odczułem brak dobrego i zwartego kompendium na temat USB (zwłaszcza w języku polskim) przydatnego programistom i konstruktorom układów mikroprocesorowych. Celem pierwszego rozdziału jest właśnie choćby częściowe wypełnienie tej luki.

W rozdziale 2 pokazuję warianty sprzętu, którego używałem do testowania przykładowych programów. Przedstawiam stosowaną konwencję nazywaną plików

źródłowych. Opisuję, jak skompilować przykłady. Sporo miejsca poświęcam też ogólnym rozważaniom, jak poprawnie pisać programy do wielu wariantów sprzętu. Omawiam, jak dostosować projekty do dowolnej konfiguracji sprzętu. Rozdział 2 zawiera ponadto opis pomocniczych funkcji wykorzystywanych w przykładach. Rozdział ten kończy się projektem, który demonstruje konfigurowanie wariantu sprzętu i taktowania mikrokontrolera oraz obsługę błędów. Celem tego projektu jest przedstawienie procedur niezwiązanych bezpośrednio z USB, ale stosowanych we wszystkich następnych projektach. Ponadto przykład ten umożliwia szybkie przetestowanie zestawu narzędzi programistycznych (ang. *toolchain*) i sprzętu (płytki prototypowa, programator, adapter JTAG).

W rozdziale 3 przedstawiam trzy projekty prostych urządzeń USB pełnej szybkości (ang. *full speed*), demonstrujące wszystkie cztery, zdefiniowane w standardzie USB, tryby przesyłana danych (ang. *control, interrupt, bulk, isochronous*). Każde z tych urządzeń można podłączyć do komputera osobistego pracującego pod kontrolą systemu Windows lub Linux. Od strony programistycznej omawiam tylko warstwę aplikacji, odkładając dokładny opis implementacji protokołu USB do następnego rozdziału. Projekt 1 demonstruje przesyłanie pilne (ang. *interrupt transfer*). Urządzenie jest rozpoznawane jako mysz. Do zmiany położenia wskaźnika myszy i klikania jej przyciskami służą dżojstik i przyciski monostabilne zainstalowane na płytce prototypowej lub akcelerometr MEMS. Projekt 2 demonstruje przesyłanie masowe (ang. *bulk transfer*). Urządzenie jest rozpoznawane jako wirtualny port szeregowy, czyli COM w Windows, a /dev/ttyACM w Linuksie. Można się z nim skomunikować za pomocą popularnych programów: HyperTerminal (Windows), PuTTY (Windows), Minicom (Linux). Wpisując odpowiednie polecenie, można sterować diodami świecącymi podłączonymi do mikrokontrolera. Opcjonalnie, jeśli układ jest wyposażony w LCD, wyświetlane są na nim informacje diagnostyczne. Projekt 3 demonstruje przesyłanie izochroniczne (ang. *isochronous transfer*) strumienia danych audio. Urządzenie jest rozpoznawane jako głośnik, który może być używany przez dowolny program odtwarzający dźwięk. Po podłączeniu urządzenia na ekranie komputera pojawia się ikona głośnika. Kliknięcie tej ikony otwiera panel z kontrolkami regulacji głośności i wyciszania. Opcjonalnie, jeśli układ jest wyposażony w LCD, wyświetlane są na nim informacje diagnostyczne. W każdym z powyższych trzech przykładów pojawiają się też komunikaty przesyłane jako dane sterujące (ang. *control transfer*).

W rozdziale 4 przedstawiam pierwszą część, dotyczącą urządzenia USB, napisanej przeze mnie biblioteki upraszczającej korzystanie z USB na mikrokontrolerach STM32. Wszystkie prezentowane w książce przykłady bazują na tej bibliotece. Omawiam też bibliotekę libusb, która ułatwia pisane aplikacjom korzystających z urządzeń USB po stronie komputera osobistego. Rozdział kończy się projektem pokazującym, jak zaimplementować urządzenie USB własnej klasy (ang. *vendor specific*). Program demonstruje przesyłanie danych sterujących, pilnych, masowych i izochronicznych, w obu kierunkach: kontroler-urządzenie i urządzenie-kontroler.

Rozdział 5 poświęcony jest zarządzaniu zasilaniem interfejsu USB. Omawiam w nim, zamieszczone w standardzie, wymagania dotyczące poboru prądu przez urządzenie USB. Opisuję praktycznie stosowane sposoby realizacji tych wymagań.

Projekt, kończący ten rozdział, demonstruje automatyczne usypanie urządzenia przy braku aktywności na szynie (ang. *auto suspend*), budzenie urządzenia (ang. *wakeup*) i zdalne budzenie kontrolera (ang. *remote wakeup*).

W rozdziale 6 omawiam modyfikacje, jakie trzeba wprowadzić w implementacji urządzenia, aby korzystać z transmisji wysokiej szybkości (ang. *high speed*) wprowadzonej w wersji 2.0 standardu USB. Jako przykładowy projekt proponuję zaimplementowanie zewnętrznej pamięci masowej korzystającej z przesyłania masowego (ang. *bulk*). Urządzenie to jest rozpoznawane jako zewnętrzny dysk.

Dwa ostatnie rozdziały poświęcone są kontrolerom USB. W rozdziale 7 przedstawiam drugą część napisanej przeze mnie biblioteki. Opisuję moduły umożliwiające zaimplementowanie prostego kontrolera USB. Po podłączeniu do niego urządzenia USB na LCD wyświetlane są podstawowe informacje o nim: szybkość transmisji, VID, PID, klasa i podklasa oraz obsługiwany protokół, znaczniki związane z zarządzaniem energią (ang. *self powered, remote wakeup*) oraz deklarowany przez urządzenie limit poboru prądu z szyny. Jeśli są dostępne, wyświetlane są nazwa producenta i nazwa urządzenia. Ponadto kontroler ten obsługuje protokół fazy rozruchu (ang. *boot protocol*) urządzeń klasy HID (ang. *Human Interface Device*). Jeśli podłączone urządzenie jest myszą, to wyświetlany jest stan przycisków oraz współrzędne położenia jej wskaźnika. Jeśli podłączone urządzenie jest klawiaturą, to wyświetlana jest informacja o wciśniętych klawiszach. W rozdziale 8 przedstawiam projekt kontrolera obsługującego pamięć masową USB Flash. Po podłączeniu takiego urządzenia przykładowa aplikacja przeprowadza test, który polega na odczytywaniu i zapisywaniu plików.

Wszystkie pliki źródłowe prezentowanych w tej książce projektów oraz inne pliki, które są pomocne przy ich komplikowaniu i testowaniu, znajdują się w skompresowanym archiwum, które można ściągnąć ze strony Wydawnictwa BTC <http://www btc pl> lub z mojej strony domowej <http://www mimuw edu pl/~marpe/book>. Dla pełnego zrozumienia przykładów konieczne jest zjrzenie do ich tekstu źródłowego, do czego gorąco zachęcam, tym bardziej że, z uwagi na dużą objętość tych źródeł, w samej książce zamieszczam tylko kluczowe fragmenty programów, przede wszystkim te, które dotyczą bezpośrednio USB. Nie omawiam szczegółów związanych z konfigurowaniem rejestrów peryferyjnych – te można łatwo wyczytać z plików źródłowych.

W książce staram się promować polską terminologię. Nie chciałbym, aby literatura techniczna była zaśmiecona angielskimi terminami lub ich nieudolnymi tłumaczeniami. Dla wielu angielskich terminów istnieją dobre i od dawna ugruntowane, ale być może zapomniane, polskie odpowiedniki. Aby jednak ułatwić posługiwanie się dokumentacją w języku angielskim, przy pierwszym, a czasem i przy kolejnym użyciu terminu, który nie jest powszechnie znany, umieszczam w nawiasach jego angielski odpowiednik.

Marcin Peczarski, Warszawa 2013

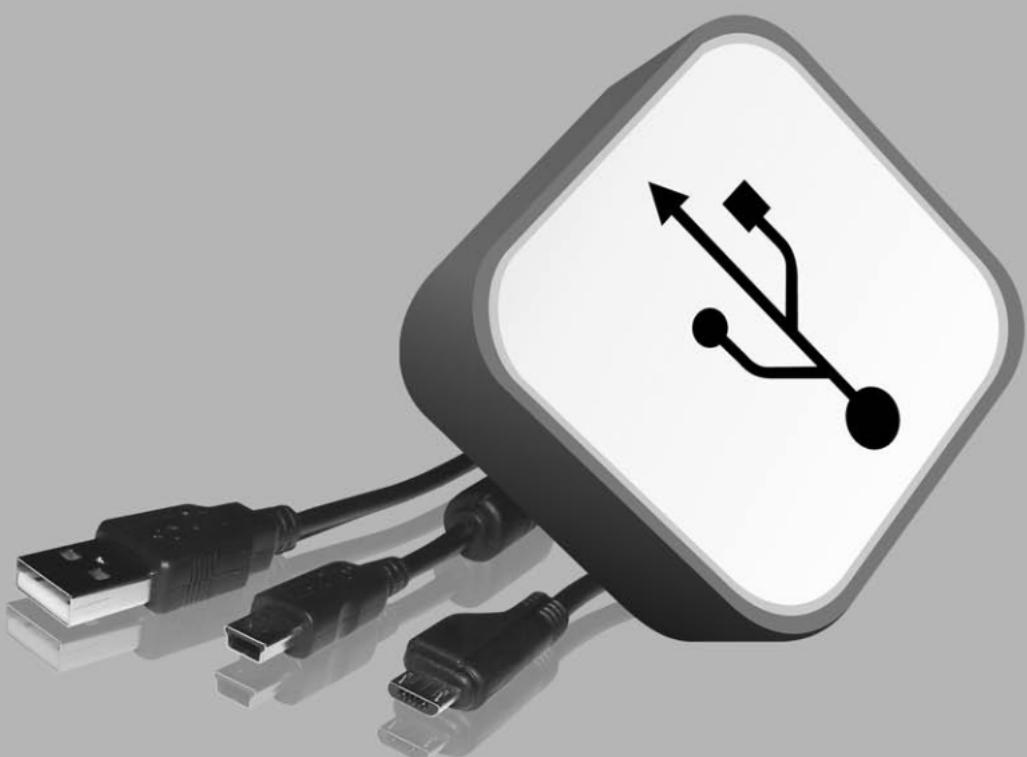
Ważniejsze skróty użyte w książce

- ADC – klasa urządzeń USB wykorzystująca transmisję izochroniczną do przesyłania strumienia audio (ang. *Audio Device Class*)
- BOT – protokół transportowy stosowany w USB i przeznaczony do pamięci masowych, multipleksujący dane i polecenia we wspólnym strumieniu danych masowych (ang. *Bulk-Only Transport*)
- CDC – klasa urządzeń komunikacyjnych i sieciowych USB (ang. *Communication Device Class*)
- DEV-FS – układ peryferyjny zastosowany w STM32 i realizujący interfejs USB urządzenia FS
- FS – pełna szybkość transmisji interfejsu USB (ang. *Full Speed*), 12 Mb/s
- HID – urządzenie USB służące do sterowania komputerem lub wprowadzania danych (ang. *Human Interface Device*), typowo mysz lub klawiatura
- HS – wysoka szybkość transmisji interfejsu USB (ang. *High Speed*), 480 Mb/s
- LS – mała szybkość transmisji interfejsu USB (ang. *Low Speed*), 1,5 Mb/s
- MSC – klasa urządzeń USB udostępniająca interfejs pamięci masowej (ang. *Mass Storage Class*)
- OTG – urządzenie USB mogące pełnić również funkcję kontrolera (ang. *host*) szyny USB (ang. *On-The-Go*)
- OTG-FS – układ peryferyjny zastosowany w STM32 i realizujący interfejs USB urządzenia FS oraz kontrolera LS/FS
- OTG-HS – układ peryferyjny zastosowany w STM32 i realizujący interfejs USB urządzenia FS/HS oraz kontrolera LS/FS/HS
- PID – identyfikator pakietu USB (ang. *Packet IDentifier*)
- PID – identyfikator produktu (ang. *Product IDentifier*)
- PID – urządzenie USB realizujące haptyczny interfejs człowiek-komputer (ang. *Physical Interface Device*), np. dżojstik z oddziaływaniem zwrotnym
- SCSI – interfejs wykorzystywany do podłączania pamięci masowych (ang. *Small Computer Systems Interface*), czyli przede wszystkim pamięci dyskowych (magnetycznych i optycznych) oraz półprzewodnikowych (*pendrive*), ale także innych rodzajów urządzeń (drukarka, nagrywarka, skaner)

- USB – uniwersalna magistrala szeregową (ang. *Universal Serial Bus*)
USB-IF – organizacja zrzeszająca firmy uczestniczące w tworzeniu standardów USB (ang. *USB Implementers Forum, Inc.*)
VID – identyfikator producenta (ang. *Vendor IDentifier*)

1

Architektura USB



Celem pierwszego rozdziału jest zapoznanie Czytelnika z architekturą i podstawami działania interfejsu USB. Pełna specyfikacja tego interfejsu wraz z dodatkami, erratami i dokumentami definiującymi poszczególne klasy urządzeń zajmuje kilka tysięcy stron (patrz <http://www.usb.org/developers/docs>). Z konieczności zatem wybrałem tylko te zagadnienia, które są niezbędne do zrozumienia materialu prezentowanego w kolejnych rozdziałach. Zaczynam od ogólnego scharakteryzowania własności interfejsu, a następnie opisuję stosowane w nim protokoły w zwyczajowym ujęciu warstwowym. W przypadku USB ten podział nie jest wyraźny. Poszczególne warstwy nie są niezależne i odseparowane wzajemnie od siebie, a oficjalna dokumentacja nie definiuje wyraźnie wszystkich warstw. Mam nadzieję, że proponowane przeze mnie podejście pomoże lepiej zrozumieć zasady działania interfejsu USB.

1.1. Ogólna charakterystyka interfejsu

USB (ang. *Universal Serial Bus*) jest uniwersalnym interfejsem szeregowym ogólnego przeznaczenia. Został on opracowany wspólnie przez czołowe firmy z branży komputerowej, aby zastąpić kilka poprzednio stosowanych interfejsów, głównie RS-232, Centronics i PS/2, wprowadzając jednocześnie nowe udogodnienia dla użytkowników. Oficjalnie umożliwiono zasilanie urządzenia bezpośrednio z interfejsu USB, choćby po to, żeby uniknąć trikowego sposobu jego zasilania, jaki na przykład stosowano w RS-232. Postarano się, aby podłączanie urządzeń wyposażonych w USB było jak najprostsze. Złącza są standaryzowane i tak skonstruowane, aby zminimalizować ryzyko błędnego podłączenia. Urządzenia mogą być podłączane przy włączonym zasilaniu (ang. *hot plug*). Po podłączeniu urządzenia USB do komputera system operacyjny automatycznie znajduje i uruchamia właściwy sterownik (ang. *device driver*) – od użytkownika nie wymaga się żadnej wiedzy na temat konfigurowania sterowników urządzeń (ang. *plug and play*). Jeden kontroler USB pozwala na podłączenie do 127 urządzeń, a podłączanie kolejnych urządzeń nie wymaga rezerwowania żadnych zasobów sprzętowych (linia przerwania IRQ, adres wejścia-wyjścia), co było poważnym problemem przed wprowadzeniem USB. Przykładowo każde nowe urządzenie używające RS-232 wymagało zainstalowania kolejnego interfejsu i zarezerwowania dla niego linii przerwania i przedziału adresów wejścia-wyjścia. Ponieważ ilość tych zasobów w komputerze jest ograniczona, to interfejsy musiały współdzielić zasoby, a przy próbie skonfigurowania kolejnego interfejsu dochodziło często do konfliktu, którego zwykły użytkownik nie potrafił samodzielnie rozwiązać. Co dla użytkownika jest udogodnieniem, staje się utratą pieniędzy dla programisty. Protokoły USB są bardzo skomplikowane. Programowanie urządzeń USB oraz pisanie dla nich sterowników należy uznać za raczej trudne. Przykładowo, uruchomienie komunikacji za pomocą RS-232 wymaga przeczytania opisu kilku rejestrów mikrokontrolera lub poznania kilku wywołań funkcji jakieś biblioteki obsługującej ten interfejs. Całość zapewne zmieści się w kilkudziesięciu liniach tekstu źródłowego. W przypadku USB, aby uruchomić jakąkolwiek komunikację, nawet gdy korzystamy z wysokopoziomowej biblioteki, trzeba napisać co najmniej kilkaset linii programu. Choć w ostatecznym rozrachunku ten dodatkowy nakład pracy się opłaca, bo otrzymujemy urządzenie, które podłącza się i obsługuje

bardzo wygodnie. Głównym konkurentem USB, o podobnych właściwościach, jest interfejs *Firewire*, znany też pod nazwą IEEE 1394.

Pierwszy praktycznie i szeroko stosowany standard USB został oznaczony numerem 1.1. Definiuje on dwie szybkości przesyłania danych:

- mała szybkość (ang. *low speed*), 1,5 Mb/s, oznaczana w dalszej części książki skrótem LS;
- pełna szybkość (ang. *full speed*), 12 Mb/s, oznaczana w dalszej części książki skrótem FS.

Standard definiuje następujące elementy składowe USB (omawiam je dokładniej w następnym podrozdziale):

- kontroler (ang. *host controller*) – zawsze jest tylko jeden;
- urządzenie (ang. *device, function*) – co najmniej jedno;
- koncentrator (ang. *hub*) – występuje, jeśli jest więcej niż jedno urządzenie;
- kable łączące wyżej wymienione elementy.

Kontroler i koncentrator muszą obsługiwać obie szybkości transmisji: LS i FS. Urządzenie może obsługiwać tylko wybraną szybkość: LS lub FS. USB działa w modelu *master-slave*. Kontroler zarządza pracą całego interfejsu, jest elementem nadrzędnym (ang. *master*), a koncentratory i urządzenia pełnią funkcję podrzędną (ang. *slave*). Wersja 2.0 standardu wraz z opublikowanymi do niej później suplementami wprowadza dodatkowo:

- wysoką szybkość przesyłania danych (ang. *high speed*), 480 Mb/s, oznaczaną w dalszej części książki skrótem HS;
- urządzenia OTG (ang. *On The Go*) mogące też pełnić funkcję kontrolera (ang. *dual role device*);
- miniaturowe złącza, które można wbudować do małych urządzeń przenośnych, np. telefonów komórkowych.

Standard USB 2.0 jest wstecznie kompatybilny ze standardem USB 1.1. Każdy produkt zgodny z wersją 1.1 powinien w założeniu poprawnie współpracować z produktem zgodnym z wersją 2.0. Każde urządzenie HS powinno móc pracować w trybie FS, jeśli kontroler lub koncentrator nie oferuje szybkości HS. Dla ściśleści należy jeszcze zaznaczyć, że interfejs USB opisany w tych standardach jest szyną o transmisji naprzemiennej (ang. *half duplex*), z czego wynika, iż podane wyżej przepływności określają maksymalną sumę przepływności w obu kierunkach dla wszystkich urządzeń i koncentratorów podłączonych do wspólnego kontrolera szyiny, z uwzględnieniem bitów synchronizacyjnych i nagłówków pakietów.

Niedawno opublikowano standard USB 3.0, który wprowadza superwysoką szybkość przesyłania danych, czyli 5 Gb/s (ang. *super speed*), oraz możliwość równoczesnej transmisji w obu kierunkach (ang. *dual simplex*). Aby uzyskać tak dużą przepływność, zastosowano nowy typ złącza oraz kodowanie 8b/10b (znane z innych interfejsów szeregowych, np. PCI Express) – każde 8 bitów danych jest kodowane jako 10 bitów liniowych, co sprawia, że rzeczywista przepustowość łącza wynosi tylko 4 Gb/s dla każdego kierunku transmisji. Ponadto istnieje też standard *Wireless USB* definiujący krótkozasięgowy, bezprzewodowy (radiowy) interfejs

o podobnych właściwościach jak przewodowa wersja USB. Jednak w dalszej części tego rozdziału omawiam wyłącznie USB 2.0, gdyż tę wersję standardu obsługują mikrokontrolery, na których można uruchomić opisywane w kolejnych rozdziałach przykłady. Poniższy opis bazuje głównie na oficjalnej dokumentacji [22] i tam należy szukać dodatkowych informacji.

Standard USB definiuje cztery rodzaje przesyłanych danych:

- sterujące (ang. *control*),
- izochroniczne (ang. *isochronous*),
- masowe (ang. *bulk*),
- pilne (ang. *interrupt*).

Dane sterujące (ang. *control transfer*) wykorzystuje się do konfigurowania urządzenia i nadzoru nad jego pracą. Każde urządzenie musi obsługiwać ten rodzaj danych. Protokół USB gwarantuje poprawność transmisji danych sterujących przez zastosowanie sumy kontrolnej, potwierdzenia dostarczenia i ewentualnej retransmisji niedostarczonych lub błędnie dostarczonych danych. Aby zagwarantować poprawność pracy całego interfejsu, kontroler USB rezerwuje pewną część pasma transmisyjnego szyny na dane sterujące.

Przesyłanie izochroniczne (ang. *isochronous transfer*) umożliwia regularne dostarczanie ustalonych porcji danych. Jego głównym zastosowaniem jest przesyłanie strumieni danych audio i wideo. Ten rodzaj transmisji gwarantuje nieprzekraczanie maksymalnego czasu pomiędzy kolejnymi porcjami danych. Nie gwarantuje natomiast poprawności dostarczonych danych, gdyż nie są one opatrywane sumą kontrolną i nie potwierdza się ich dostarczenia. Aby zagwarantować regularne dostarczanie danych, kontroler USB rezerwuje odpowiednią część pasma transmisyjnego dla każdego skonfigurowanego strumienia danych izochronicznych. Urządzenia LS nie obsługują przesyłania izochronicznego.

Przesyłanie masowe (ang. *bulk transfer*) przeznaczone jest do transmisji dużej ilości danych, gdy wymagamy poprawności ich dostarczenia, ale nie są istotne opóźnienia transmisji ani nie wymagamy regularności ich dostarczania. Poprawność zapewniana jest, podobnie jak w przypadku danych sterujących, za pomocą sumy kontrolnej, potwierdzania dostarczenia i mechanizmu retransmisji. Dane masowe są przesyłane tylko wtedy, gdy nie ma żadnych innych rodzajów danych do przesłania lub inne rodzaje przesyłań nie wykorzystują całego zarezerwowanego dla nich pasma. Przesyłanie masowe używane jest m.in. do komunikacji z pamięciami masowymi, drukarkami lub skanerami. Urządzenia LS nie obsługują przesyłania masowego.

Przesyłanie danych pilnych (ang. *interrupt transfer*) umożliwia transmisję niewielkich porcji danych w regularnych odstępach. Ten rodzaj transmisji gwarantuje nieprzekraczanie maksymalnego czasu pomiędzy kolejnymi porcjami danych i poprawność dostarczonych danych. Poprawność zapewniana jest analogicznie jak w przypadku danych sterujących i masowych. Przesyłanie pilne wykorzystywane jest głównie przez urządzenia takie jak mysz, klawiatura lub dżojstik. Należy zaznaczyć, że angielska nazwa użyta dla tego rodzaju transmisji jest nieco myląca i nie ma nic wspólnego z systemem przerwań. W USB wszystkie przesyły są inicjowane przez kontroler szyny i urządzenie nie może samo zainicjować komunikacji

z kontrolerem. W przypadku danych pilnych kontroler USB rezerwuje odpowiednią część pasma dla każdej skonfigurowanej transmisji pilnej i odpytuje (ang. *poll*) urządzenie w regularnych odstępach czasu, czy są jakieś dane do przesłania.

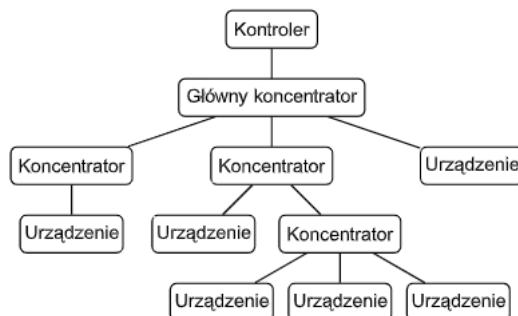
Typowo, bazując zwykle na jednym z wymienionych wyżej rodzajów danych, urządzenia realizują jakieś protokoły aplikacyjne, które zebrane są w klasy urządzeń o podobnych właściwościach. W kolejnych rozdziałach pojawią się bardziej szczegółowe opisy, poparte stosownymi przykładami implementacji, następujących protokołów:

- ADC (ang. *Audio Device Class*) – klasa urządzeń audio, np. zewnętrzny głośnik lub mikrofon, przesyłanie izochroniczne;
- CDC (ang. *Communication Device Class*) – klasa urządzeń komunikacyjnych, np. modem lub wirtualny port szeregowy, przesyłanie masowe;
- HID (ang. *Human Interface Device*) – klasa urządzeń realizujących interfejs między człowiekiem a komputerem, np. wspomniane już mysz i klawiatura, przesyłanie pilne;
- MSC (ang. *Mass Storage Class*) – klasa urządzeń pamięciowych, np. zewnętrzny dysk twardy, pamięć Flash (ang. *pendrive*), przesyłanie masowe.

Oczywiście każde urządzenie USB musi obsługiwać też dane sterujące. W terminologii USB pojawia się pojęcie funkcji, którą należy rozumieć jako urządzenie logiczne implementujące protokół określonej klasy, np. drukarka, klawiatura itd. Jeśli urządzenie USB realizuje więcej niż jedną funkcję, na przykład ADC i HID, to jest nazywane urządzeniem złożonym (ang. *compound device*).

1.2. Warstwa fizyczna

Warstwa fizyczna określa parametry mechaniczne, elektryczne, metodę ustalania szybkości przesyłania i procedurę zerowania urządzenia. Szyna USB ma fizycznie topologię drzewa. Przykładową konfigurację szyny przedstawia **rysunek 1.1**. W korzeniu drzewa znajduje się kontroler szyny oraz główny koncentrator (ang. *root hub*), który udostępnia kilka portów (zwykle od dwóch do sześciu). Do tych portów mogą być podłączone urządzenia lub kolejne koncentratory. Port koncentratora od strony kontrolera jest traktowany przez kontroler jak kolejne urządzenie

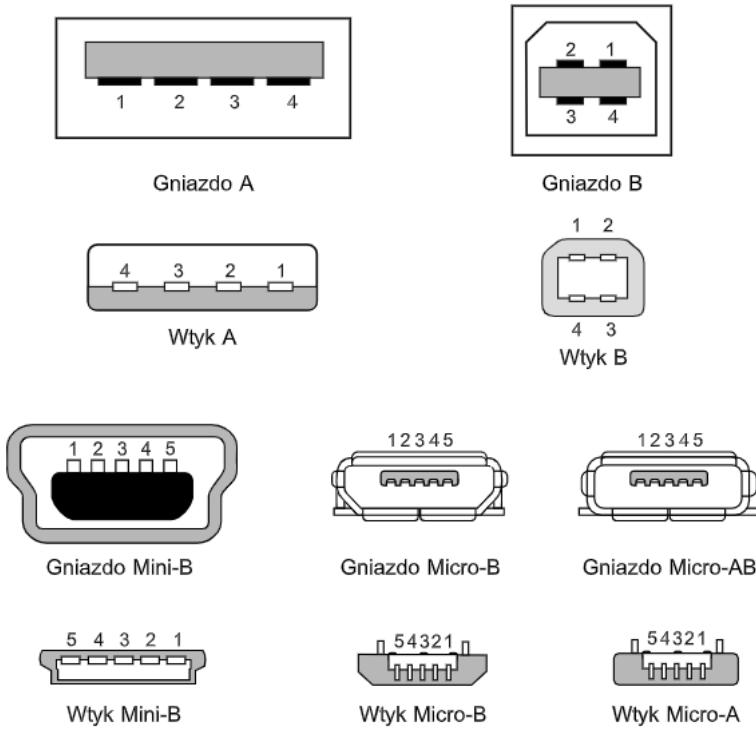


Rys. 1.1. Przykładowa konfiguracja szyny USB

i wszędzie dalej, gdzie jest mowa o urządzeniu, podane informacje dotyczą również koncentratora. Między kontrolerem a urządzeniem może pośredniczyć maksymalnie pięć koncentratorów, wliczając w to główny koncentrator. Możliwe jest też bezpośrednie połączenie kontrolera z urządzeniem, bez pośrednictwa koncentratora. Konfiguracja taka jest często spotykana, gdy łączy się ze sobą dwa urządzenia, z których przynajmniej jedno jest urządzeniem OTG. Z perspektywy oprogramowania aplikacyjnego szyna USB ma logiczną topologię gwiazdy. Oprogramowanie adresuje bezpośrednio poszczególne urządzenia i nie ma znaczenia, czy po drodze znajdują się jakieś koncentratory, które zachowują się w większości przypadków przezrocznie.

Szyna USB nie może zawierać pętli. Utrzymanie prawidłowej konfiguracji połączeń szyny ma ścisły związek ze stosowanymi złączami, które zostały zaprojektowane w taki sposób, aby uniemożliwić złe podłączenie kabli. Standard USB 1.1 przewiduje dwa typy złączy, oznaczone literami A i B. Koncentrator jest wyposażony w gniazdo typu A. Porty koncentratorów od strony kontrolera, prowadzące do góry hierarchii na rysunku 1.1, mają gniazda typu B. Porty wszystkich koncentratorów od strony urządzeń, prowadzące w dół hierarchii na rysunku 1.1, mają gniazda typu A. Urządzenie może być wyposażone w gniazdo typu B, kabel zakończony wtykiem typu A (takie jest wymaganie dla urządzeń LS) lub po prostu wtyk typu A – małe urządzenia w obudowie przypominającej pióro ze skuwką (ang. pendrive) lub patyczek (ang. stick), wkładane bezpośrednio do gniazda koncentratora. Jedyny prawidłowy kabel, w pełni zgodny ze standardem USB 1.1, ma z jednej strony wtyk typu A, a z drugiej – wtyk typu B. Możliwe, ale zdecydowanie niezalecane przez standard, jest też stosowanie przedłużacza, czyli kabla mającego z jednej strony wtyk typu A, a z drugiej – gniazdo typu A. Konstrukcja mechaniczna uniemożliwia połączenie złącza typu A ze złączem typu B, co wyklucza powstanie pętli w topologii szyny. Kontroler i główny koncentrator są często wykonane jako jeden układ elektroniczny i umieszczone we wspólnej obudowie, bez możliwości rozłączenia.

Pojawienie się małych, przenośnych urządzeń wymusiło rozszerzenie standardu o nowy typ złącza – Mini-B. Szybko okazało się, że również ono jest za duże, aby nadawało się do ciągle miniaturyzowanych telefonów komórkowych, więc opracowano złącze Micro-B. Złącza Mini-B i Micro-B pełnią taką samą funkcję jak złącze typu B, są od niego tylko istotnie mniejsze. Wraz z urządzeniami OTG wprowadzono złącze Micro-A: wtyk Micro-A i gniazdo Micro-AB. Wtyk Micro-A pełni taką samą funkcję jak wtyk typu A. Natomiast do gniazda Micro-AB można podłączyć zarówno wtyk Micro-A, jak i wtyk Micro-B. Dzięki temu urządzenie OTG może, zależnie od podłączonego kabla, być kontrolerem lub zwykłym urządzeniem. Dla porządku należy jeszcze wspomnieć, że w pewnym momencie swojego rozwoju standard USB przewidywał również możliwość stosowania złącza Mini-A (wtyk Mini-A i gniazdo Mini-AB), ale w suplementie do standardu zabroniono jego instalowania w nowych produktach. Używane obecnie wtyki i gniazda USB przedstawione są schematycznie na rysunku 1.2. Porównanie rzeczywistych wymiarów poszczególnych złączy pokazano na rysunku 1.3.



Rys. 1.2. Złącza USB

Pojawienie się dodatkowych rodzajów złączy wymusza wykorzystanie kilku rodzajów kabli. Obecnie standard przewiduje następujące warianty kabli:

- wtyk A – wtyk B,
- wtyk A – wtyk Mini-B,
- wtyk A – wtyk Micro-B,
- wtyk Micro-A – gniazdo A,
- wtyk Micro-A – wtyk Micro-B,
- kabel podłączony na stałe do urządzenia i zakończony wtykiem A,
- kabel podłączony na stałe do urządzenia i zakończony wtykiem Micro-A.



Rys. 1.3. Końcówki USB, od lewej wtyki: A, Mini-B, Micro-B

W niektórych specyficznych zastosowaniach przydatne mogą być następujące niestandardowe kable, które można też złożyć z wyżej wymienionych standardowych kabli:

- wtyk Micro-A – wtyk B,
- wtyk Micro-A – wtyk Mini-B.

Ponadto można czasem spotkać różnego rodzaju przedłużacze lub kable, które naruśają ustalenia zapisane w standardzie, np.:

- wtyk A – gniazdo A,
- wtyk A – wtyk A,
- wtyk A – wtyk Micro-A.

Oznaczenia styków w złączach USB i zalecane standardowe kolory przewodów w kablach zamieszczone są w **tabeli 1.1**. Numeracja styków jest zgodna z oznaczeniami użyтыmi na rysunku 1.2. Kabel USB ma dwie pary przewodów – jedną przesyła się dane, a druga to zasilanie i masa. Wszystkie kable, które nie są na stałe podłączone do urządzenia, powinny umożliwiać przesyłanie z szybkością HS. Jedynie do łączenia urządzeń LS można wykorzystywać tanie kable nieekranowane, zawierające pary nieskręconych przewodów, ale wtedy taki kabel musi być z jednej strony na stałe podłączony do urządzenia. We wszystkich pozostałych kablach, które są rozłączalne, para przewodów przesyłających dane musi być wykonana jako skrętka i musi być ekranowana. Różnicowa impedancja falowa tej pary powinna mieć nominalnie wartość 90Ω . Ekran oczywiście podłącza się do masy. Zależnie od jakości zastosowanych przewodów maksymalna długość kabla USB wynosi od 3 do 5 m.

Tab. 1.1. Sygnały w złączach i kablach USB

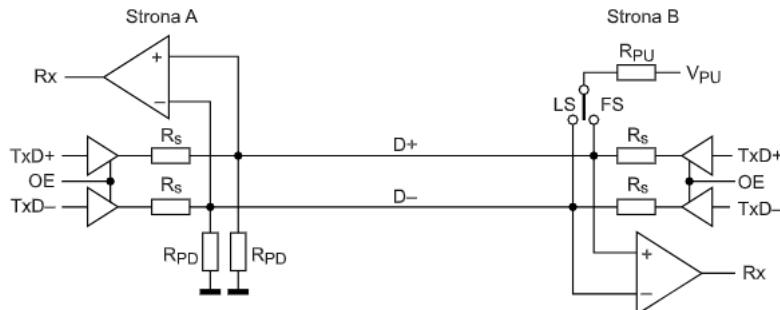
Oznaczenie	Nr styku w złączach czterostykowych	Nr styku w złączach pięciostykowych	Kolor przewodu	Opis
VBUS	1	1	Czerwony	Zasilanie, nominalnie +5 V
D-, DM	2	2	Biały	Dane
D+, DP	3	3	Zielony	Dane
ID		4	Brak przewodu	W gniazdce Micro-AB identyfikacja wtyku W gniazdach Mini-B i Micro-B nieużywany We wtyku Micro-A połączony z masą We wtykach Mini-B i Micro-B niepodłączony
GND	4	5	Czarny	Masa

Kontakty zasilania w złączach są nieco wysunięte przed kontakty danych, tak że przy wkładaniu wtyku do gniazda najpierw zostaje do urządzenia podłączone zasilanie, a dopiero potem podłączane są linie danych. Taka kolejność łączenia obwodów zapewnia poprawne zerowanie urządzeń i umożliwia łączenie elementów szyny USB bez wyłączenia zasilania. Napięcie zasilające VBUS musi mieścić się w przedziale 4,75...5,25 V. Każde urządzenie może być zasilane z szyny i pobierać do 100 mA. Urządzenie może zwiększyć pobór prądu do 500 mA, jeśli kontroler szyny na to pozwala. Kontroler i koncentratory powinny być wyposażone w układy zabezpieczające przed pobieraniem z VBUS zbyt dużego prądu. Urządzenie, które nie potrzebuje do zasilania więcej niż 100 mA, powinno poprawnie pracować przy

napięciu VBUS o wartości co najmniej 4,4 V. Urządzenie nie powinno obciążać VBUS pojemnością większą niż 10 μ F. Przy odłączaniu urządzenia mogą powstawać przepięcia, których źródłem jest indukcyjność przewodów doprowadzających zasilanie. Dlatego zaleca się, aby po stronie urządzenia blokować VBUS do masy kondensatorem o pojemności co najmniej 1 μ F. Urządzenie może stwierdzić, że zostało podłączone do szyny lub od niej odłączone, monitorując wartość napięcia na VBUS. Więcej o zasilaniu urządzeń USB i sposobach oszczędzania energii piszę w rozdziale 5, ilustrując te zagadnienia praktycznymi przykładami.

Złącza Mini i Micro mają dodatkowy piąty styk oznaczony jako ID. W kablu nie ma żadnego dodatkowego przewodu dla tego styku. Służy on do identyfikacji, jaki wtyk został podłączony do gniazda Micro-AB. We wtyku Micro-A styk ID jest połączony na stałe z masą, a we wtyku Micro-B pozostaje niepodłączony. Podobną rolę styk ID odgrywałby w gniazdach Mini-AB, gdyby były nadal używane.

Na liniach danych D+ i D- stosuje się sygnalizację różnicową, aby zminimalizować wpływ zakłóceń elektromagnetycznych. Sygnalizacja dla transmisji HS jest nieco inna niż dla transmisji LS i FS. Dlatego najpierw opiszę sygnalizację w łączu LS i FS. Uproszczony schemat obwodu sygnalizacyjnego przedstawiony jest na **rysunku 1.4**. Wyjścia buforów nadawczych są trójstanowe i w stanie aktywnym powinny mieć możliwie małą impedancję wyjściową. Do wyjść tych podłączone są rezystory szeregowe R_S o wartości nominalnej 45Ω i tolerancji 5%, w celu dopasowania impedancji wyjściowej nadajnika do różnicowej impedancji falowej linii danych. Po stronie kontrolera lub koncentratora (od strony złącza typu A) linie danych są obciążone rezystorami ściągającymi do masy R_{PD} o wartościach z przedziału $14,25 \dots 24,80 \text{ k}\Omega$. Po stronie urządzenia (po stronie złącza typu B) jedna z linii danych jest podciągana do napięcia V_{PU} z przedziału $3 \dots 3,6 \text{ V}$ rezystorem R_{PU} o wartości nominalnej $1,5 \text{ k}\Omega$ i tolerancji 5%. Rezystory podciągające służą do wyboru szybkości transmisji. Urządzenie LS podciąga do zasilania linię D-, a urządzenie FS podciąga linię D+. Jeśli żadna z linii danych nie jest podciągnięta, to urządzenie jest traktowane, jakby było odłączone. Odłączenie rezystora podciągającego ma zatem taki sam skutek jak rozłączenie kabla, umożliwia więc programową symulację odłączenia kabla. Dla porządku należy dodać, że do pierwotnej wersji standardu opublikowano suplement, który dopuszcza stosowanie przełączanej rezystancji podciągającej o wartości z przedziału $900 \dots 1575 \Omega$, gdy szyna jest w stanie spoczynku, a $1475 \dots 3090 \Omega$, gdy są przesyłane dane.



Rys. 1.4. Uproszczony schemat obwodu sygnalizacyjnego LS i FS

Tab. 1.2. Stany linii danych dla transmisji LS i FS

Stan	Poziomy sygnałów na wyjściu nadajnika	Poziomy sygnałów na wejściu odbiornika
Różnicowa jedynka K dla LS J dla FS Spoczynkowy dla FS	D+ = 2,8...3,6 V D- = 0,0...0,3 V	D+ - D- > 0,2 V D+ > 2,0 V
Różnicowe zero J dla LS K dla FS Spoczynkowy dla LS	D+ = 0,0...0,3 V D- = 2,8...3,6 V	D- - D+ > 0,2 V D- > 2,0 V
SEO Rozłączenie	D+ = 0,0...0,3 V D- = 0,0...0,3 V	D+ < 0,8 V D- < 0,8 V
SE1	D+ > 0,8 V D- > 0,8 V	D+ > 0,8 V D- > 0,8 V

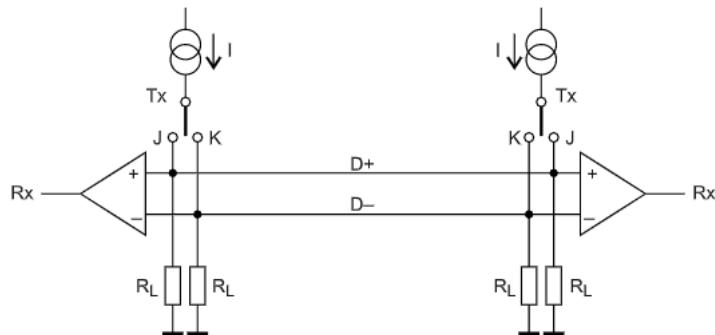
Poziomy napięć na liniach danych zebrane są w tabeli 1.2. Podczas normalnej pracy interfejsu linie danych mogą znajdować się w jednym z trzech stanów: J, K, SEO. Przynajmniej na jednej z linii danych musi być stan niski. W stanie K dla transmisji LS i w stanie J dla transmisji FS na linii D+ jest poziom wysoki, a na linii D- jest poziom niski, co w dokumentacji USB jest nazywane stanem różnicowym jedynki. W stanie J dla transmisji LS i w stanie K dla transmisji FS na linii D+ jest poziom niski, a na linii D- jest poziom wysoki, czyli stan różnicowego zera. W stanie SEO (ang. *single ended zero*) na obu liniach danych jest poziom niski. Stan SE1 (ang. *single ended one*), gdy na żadnej z linii danych nie ma poziomu niskiego, jest niepoprawnym stanem łącza USB i nie powinien nigdy wystąpić. Ponieważ jedna para przewodów służy do transmisji w obu kierunkach, w danym momencie tylko wyjścia jednego z nadajników mogą być w stanie aktywnym. Wyjścia drugiego z nich muszą być w stanie wysokiej impedancji. O kierunku transmisji decyduje protokół warstwy transportowej (który opisuję w jednym z następnych podrozdziałów). Gdy żadne dane nie są przesyłane, wyjścia obu nadajników są w stanie wysokiej impedancji. Zauważmy, że wtedy rezystory podłączone do linii danych ustalają na nich stan J, który jest też stanem spoczynkowym (ang. *idle state*) łącza.

Tab. 1.3. Stany linii danych dla transmisji HS

Stan	Poziomy sygnałów na wyjściu nadajnika	Poziomy sygnałów na wejściu odbiornika
Różnicowa jedynka J	360 mV ≤ D+ ≤ 440 mV -10 mV ≤ D- ≤ 10 mV	Dokładna definicja znajduje się w [22].
Różnicowe zero K	-10 mV ≤ D+ ≤ 10 mV 360 mV ≤ D- ≤ 440 mV	Dokładna definicja znajduje się w [22].
Spoczynkowy	Nieokreślone	-10 mV ≤ D+ ≤ 10 mV -10 mV ≤ D- ≤ 10 mV
Świergot J	700 mV ≤ D+ - D- ≤ 1100 mV	D+ - D- ≥ 300 mV
Świergot K	-900 mV ≤ D+ - D- ≤ -500 mV	D+ - D- ≤ -300 mV
Rozłączenie	Nieokreślone	D+ - D- ≥ 625 mV

Uproszczony schemat obwodu sygnalizacyjnego HS przedstawiony jest na rysunku 1.5. Obwód ten jest w pełni symetryczny i identyczny po obu stronach łącza, aby zminimalizować wpływ zakłóceń elektromagnetycznych. Z tego też powodu nie stosuje się żadnych rezystorów podciągających. Linie danych D+ i D– po obu stronach kabla są obciążone rezystorami R_L o nominalnej wartości 45Ω , w celu dopasowania do impedancji falowej kabla. Poziomy napięć na liniach danych zbrane są w tabeli 1.3. Nadajnik zrealizowany jest jako sterowane źródło prądowe I o nominalnej wartości $17,78 \text{ mA}$. Stan J uzyskuje się przez zasilanie tym prądem linii D+, a stan K – przez zasilanie linii D–. W stanie spoczynkowym żadna z linii nie jest zasilana. Ponieważ obciążenie każdej z linii danych składa się z dwóch połączonych równolegle rezystorów o wartości 45Ω , nominalna amplituda napięcia różnicowego wynosi 400 mV . Sposób rozpoznawania stanów J i K przez odbiornik jest dość skomplikowany i jego poznanie nie jest istotne do zrozumienia działania protokołów USB, więc pomijam to zagadnienie, a zainteresowanych odsyłam do przeczytania rozdziału 7.1.2.2 w specyfikacji [22]. Ponieważ nadajnik steruje linie danych ze źródła prądowego, rozłączenie obwodu (odłączenie kabla) zwiększa dwukrotnie impedancję obciążenia i skutkuje dwukrotnym wzrostem amplitudy sygnału różnicowego na wejściu odbiornika znajdującego się po tej samej stronie kabla (jest to pewne uproszczenie, w rzeczywistości należy uwzględnić efekty falowe propagacji sygnału w kablu). Efekt zwiększenia się amplitudy napięcia na liniach danych wykorzystuje się do wykrywania stanu odłączenia urządzenia w trybie HS.

Jeśli do portu kontrolera lub koncentratora nie jest podłączone urządzenie albo wykryty został stan rozłączenia (np. kabel jest nadal podłączony, ale urządzenie LS/FS odłączyło rezistor podciągający jedną z linii danych), to port ten znajduje się w trybie LS/FS, a na liniach danych jest stan SE0. Po wykryciu podłączenia nowego urządzenia do szyny, czyli po wykryciu stanu różnicowej jedynki lub różnicowego zera, należy odczekać co najmniej 100 ms . Jest to czas niezbędny do ustabilizowania się napięcia zasilającego, istotny zwłaszcza dla urządzenia zasilanego z szyny USB. Ponadto podłączane urządzenie zwykle jest sterowane mikroprocesorem i potrzebuje pewnego czasu, aby się uruchomić i być gotowe do komunikacji. Po tym czasie kontroler rozpoczyna procedurę zerowania (ang. *reset*) urządzenia. Polega ona na wymuszeniu na liniach danych stanu SE0 przez 10 do 20 ms. Jeśli zerowany jest główny koncentrator, to powinno to trwać minimum 50 ms. Urządzenie po



Rys. 1.5. Uproszczony schemat obwodu sygnalizacyjnego HS

zakończeniu procedury zerowania szyny USB powinno być gotowe do komunikacji po kolejnych 10 ms (ang. *reset recovery time*). Po zakończeniu zerowania oprogramowanie może odczytać szybkość podłączonego urządzenia, sprawdzając, która z linii danych została przez urządzenie podciągnięta do zasilania.

Aby zachować kompatybilność z USB 1.1, każde urządzenie HS rozpoczyna pracę jako FS i dopiero w fazie zerowania negocjuje większą szybkość transmisji. Procedura negocjacji wykorzystuje tak zwany świergot (ang. *chirp*) i jest uruchamiana w czasie trwania sygnału zerującego, jeśli podłączane urządzenie sygnalizuje szybkość FS i kontroler lub koncentrator, który wykrył jego połączenie, umożliwia komunikację HS. Procedurę negocjacji rozpoczyna urządzenie, wymuszając na liniach danych impuls świergotu K (patrz tabela 1.3). Kontroler lub koncentrator potwierdza gotowość do pracy HS, wysyłając na przemian impulsy świergotu K i J. Jeśli procedura ta zakończy się powodzeniem, następuje przełączenie obwodu sygnalizacyjnego z wersji FS na wersję HS.

Obwód sygnalizacyjny, który ma pracować z trzema prędkościami, czyli LS, FS i HS, konstruuje się przez połączenie obwodów z rysunków 1.4 i 1.5. Przy pracy LS lub FS źródła prądowe I po obu stronach pozostają wyłączone. Przy pracy HS odłączony pozostaje rezystor podciągający R_{PU} po stronie urządzenia. Rezystory R_L są po prostu rezystorami R_S , których jedno wyprowadzenie jest zwarte do masy poprzez podanie stanu niskiego na wyjście obu buforów nadawczych po obu stronach.

1.3. Warstwa łącza

Warstwa łącza określa formaty pakietów i algorytmy obliczania sum kontrolnych pakietów. USB przesyła bity w kolejności od najmniej znaczącego do najbardziej znaczącego. Pola wielobajtowe przesyła się w porządku cienkokońcowkowym (ang. *little-endian*). Na rysunkach i schematach stosuję taką samą konwencję jak w oficjalnych dokumentach USB. Bity, symbole, bajty lub pakiety wysyłane są w takiej kolejności, jak się je czyta, czyli od lewej do prawej.

Do transmisji danych w trybach LS i FS wykorzystuje się symbole J, K i SE0. W trybie HS dane transmituje się tylko za pomocą symboli J i K. Transmisja pojedynczego symbolu polega na utrzymywaniu na liniach danych odpowiednich poziomów napięcia lub prądu przez cały czas transmisji tego symbolu. W trybie LS transmitsie się 1,5 miliona symboli na sekundę, w FS – 12 milionów symboli na sekundę, a w HS – 480 milionów symboli na sekundę. Jako kodowanie liniowe stosuje się NRZI (ang. *Non Return to Zero Inverted*) z nadziewaniem bitami (ang. *bit stuffing*). Zero przesyłane jest jako symbol przeciwny do poprzednio wysłanego. Jeśli poprzednim symbolem było J, to wysyłany jest symbol K. Jeśli poprzednio wysłano symbol K, to wysyła się J. Jedynkę wysyła się jako powtórzenie poprzedniego symbolu. Aby zapewnić synchronizację odbiornika, należy zagwarantować dostatecznie częste zmiany stanu na liniach danych. Długi ciąg jedynek kodowany byłby jako ciąg jednakowych symboli, co doprowadziłby do rozsynchonizowania się odbiornika. Dlatego po każdych sześciu kolejnych jedynkach wstawia się dodatkowe zero, które służy jedynie do synchronizacji i jest usuwane z ciągu danych przez odbiornik.

W USB dane, które mają być przesłane, dzieli się na pakiety. Wysłanie pakietu w trybach LS i FS odbywa się według następującego schematu:

- przed rozpoczęciem transmisji wyjścia nadajnika są w stanie wysokiej impedancji, a na liniach danych jest stan spoczynkowy J;
- wyjścia nadajnika zostają przełączone do stanu aktywnego, wysyłana jest sekwencja synchronizacyjna SYNC rozpoczynająca się trzykrotnie powtórzoną parą symboli KJ, po których następuje znacznik początku pakietu SOP (ang. *start of packet*) składający się z dwóch symboli K;
- wysyłana jest zawartość pakietu z uwzględnieniem ostatniego symbolu sekwencji synchronizacyjnej, tzn. jeśli pierwszy wysyłany bit ma wartość 0, to jako pierwszy zostanie wysłany symbol J, a jeśli ma wartość 1, to symbol K;
- wysyłany jest znacznik końca pakietu EOP (ang. *end of packet*), składający się z dwóch symboli SE0 i symbolu J;
- znacznik EOP nie jest wysyłany dla pakietu PRE (ang. *preamble packet*), który służy do kapsulkowania pakietów urządzenia LS w ruchu FS i za którym natychmiast rozpoczyna się sekwencja SYNC właściwego pakietu;
- wyjścia nadajnika wracają do stanu wysokiej impedancji, rezystor podciągający podtrzymuje stan spoczynkowy J.

Zobaczmy przykład transmisji fikcyjnego ośmioróżkowego pakietu:

0	1	1	1	1	1	1	0
---	---	---	---	---	---	---	---

Ta sekwencja zawiera sześć kolejnych jedynek, więc po szóstej jedynce musimy wstawić dodatkowe zero. Po tej operacji ciąg do wysłania wygląda następująco (wstawione zero znajduje się w szarej kratce):

0	1	1	1	1	1	1	1	0	0
---	---	---	---	---	---	---	---	---	---

Ponieważ sekwencja synchronizacyjna kończy się symbolem K, a pierwszy bit do wysłania ma wartość 0, pierwszy będzie wysłany symbol J. Sześć kolejnych jedynek jest kodowanych jako ciąg sześciu symboli J. Dwa końcowe zera są kodowane odpowiednio jako K i J. Cała sekwencja wygląda zatem tak:

J	J	J	J	J	J	J	K	J
---	---	---	---	---	---	---	---	---

Po uwzględnieniu sekwencji SYNC i znacznika EOP (szare kratki) cały pakiet wygląda następująco:

K	J	K	J	K	J	K	J	J	J	J	J	J	K	J	SE0	SE0	J
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	-----	-----	---

Wysłanie pakietu w trybie HS różni się tylko nieznacznie:

- przed rozpoczęciem wysyłania sekwencji synchronizującej nadajnik znajduje się w stanie spoczynkowym właściwym dla trybu HS;
- sekwencja synchronizacyjna składa się z 32 symboli: 15 razy powtórzona para symboli KJ i znacznik SOP składający się z dwóch symboli K;
- ponieważ koncentrator w trybie HS może zgubić do 4 początkowych symboli, a po drodze od kontrolera do urządzenia może być maksymalnie 5 koncentrato-

rów, mamy gwarancję, że docierająca do odbiornika sekwencja synchronizacyjna składa się co najmniej z 12 symboli;

- znacznik EOP składa się z 8 jednakowych symboli o wartości przeciwej do ostatniego symbolu pakietu;
- wyjątkiem jest znacznik EOP kończący pakiet SOF (który jest wysyłany przez kontroler cyklicznie co 125 µs i pozwala m.in. na wykrycie stanu rozłączenia, patrz niżej), składający się z 40 jednakowych symboli;
- wewnątrz znacznika EOP nie stosuje się nadziewania bitami;
- po wysłaniu znacznika EOP nadajnik przechodzi do stanu spoczynkowego właściwego dla trybu HS.

Pole	SYNC	Właściwy pakiet	EOP
Liczba symboli	8	8...101	3
Pole	SYNC	Właściwy pakiet	EOP
Liczba symboli	8	8...9574	3 (0 dla pakietu PRE)
Pole	SYNC	Właściwy pakiet	EOP
Liczba symboli	12...32	8...9584	8 (40 dla pakietu SOF)

Rys. 1.6. Pakiety (od góry) LS, FS i HS

Aby usystematyzować i podsumować przedstawiony dotąd materiał, zestawiłem na **rysunku 1.6** dopuszczalną liczbę transmitowanych symboli w pakietach dla poszczególnych szybkości transmisji. Najkrótszy pakiet zawiera 8 symboli kodujących dane. Maksymalna liczba symboli wynika z rozmiaru najdłuższego pakietu dla danej szybkości transmisji i najbardziej pesymistycznego założenia co do liczby zer, którymi te dane musiałyby zostać ponadziewane. W dalszej części książki pomijam na rysunkach pola SYNC i EOP. Przedstawiam już tylko formaty właściwych pakietów przed nadziewaniem bitami i kodowaniem liniowym. Protokół USB definiuje trzy formaty pakietów, przedstawione na **rysunku 1.7**. Pole PID (ang. *packet identifier*) identyfikuje typ pakietu i ma 8 bitów. Pole CRC5 ma 5 bitów i zawiera sumę kontrolną pola ADRES, które ma 11 lub 19 bitów. Pole CRC16 ma 16 bitów i zawiera sumę kontrolną pola DANE. Liczba bitów pola DANE jest zawsze wielokrotnością 8. Pole DANE może być puste, czyli mieć długość 0 bitów.

Pole	PID	ADRES	CRC5
Liczba bitów	8	11 lub 19	5
Pole	PID	DANE	CRC16
Liczba bitów	8	0...8192	16
Pole	PID		
Liczba bitów	8		

Rys. 1.7. Formaty pakietów (od góry): żeton, dane, potwierdzenie

Wszelkie niepoprawnie odebrane pakiety są ignorowane. Pakiet jest uznawany za niepoprawny, gdy:

- wykryto błąd nadziewania, czyli odebrano więcej niż sześć kolejnych jedynek, wyjątkiem jest tu znacznik EOP w trybie HS;
- nie zgadza się suma kontrolna pakietu, czyli CRC5 lub CRC16;
- pakiet ma niepoprawny format, np. ma złą długość lub nie jest zakończony znacznikiem EOP.

1.4. Warstwa protokołu

Warstwa protokołu USB odpowiada tradycyjnym warstwom: sieciowej, transportowej i sesji. W USB są one silnie ze sobą powiązane i trudno jest je wyodrębnić. Niemniej można się pokusić o następujące przyporządkowanie poszczególnych funkcji warstwy protokołu USB do tradycyjnych warstw:

- sieciowa – definiuje sposób adresowania urządzeń, zna topografię szyny;
- transportowa – wprowadza bardziej szczegółowy schemat adresowania, umożliwiający przesyłanie wielu strumieni danych (ang. *pipe*) między kontrolerem a pojedynczym urządzeniem, określa sposób przesyłania poszczególnych rodzajów danych, sposób podziału strumienia danych na fragmenty po stronie nadawcy i metodę ich ponownego składania w strumień po stronie odbiorcy;
- sesji – wprowadza mechanizm transakcji, definiuje wysokopoziomowe transfery danych składające się z wielu transakcji.

PID ₀	PID ₁	PID ₂	PID ₃	$\overline{\text{PID}}_0$	$\overline{\text{PID}}_1$	$\overline{\text{PID}}_2$	$\overline{\text{PID}}_3$
------------------	------------------	------------------	------------------	---------------------------	---------------------------	---------------------------	---------------------------

Rys. 1.8. Format identyfikatora pakietu

Jak już zostało wspomniane, USB przesyła dane podzielone na pakiety. Każdy pakiet rozpoczyna się identyfikatorem pakietu (patrz rysunek 1.7). Format tego pola przedstawiony jest na **rysunku 1.8**. Identyfikator pakietu składa się z dwóch czterobitowych pól. Pierwsze cztery bity zawierają kod identyfikatora, a pozostałe cztery służą do kontroli poprawności transmisji i zawierają ten sam kod, ale zanegowany. Przypominam, że bity są przesyłane od najmniej znaczącego. Protokół USB definiuje 17 identyfikatorów, które są zebrane w **tabeli 1.4**. Zauważmy, że identyfikatory PRE i ERR mają ten sam kod, ale nie prowadzi to do błędnej interpretacji, ponieważ pierwszy z nich używany jest tylko w transmisjach FS, a drugi – tylko w HS. Poszczególne pakiety nazywają się tak jak ich identyfikatory. Mamy więc pakiety OUT, IN, SOF, SETUP, DATA0, DATA1, ACK, NAK, STALL, które opisuję dokładnie w dalszej części tego podrozdziału. Poznanie ich przeznaczenia uważam za istotne dla zrozumienia działania przykładów. Pozostałe rodzaje pakietów omawiam tylko побieżnie w ostatnim podrozdziale. Zainteresowanych dokładnym poznaniem wszystkich rodzajów pakietów zachęcam do przeczytania rozdziału 8 w [22].

W celu synchronizacji pracy interfejsu USB na poziomie aplikacji, co jest potrzebne na przykład w transmisjach izochronicznych, czas podzielony jest na ramki. Jedna

ramka trwa 1 ms. Początek ramki jest sygnalizowany urządzeniu LS przez wysłanie znacznika EOP, który pełni funkcję pakietu podtrzymującego (ang. *keep alive*). Do urządzenia FS na początku każdej ramki wysyłany jest pakiet SOF. Jego pole ADRES zawiera 11-bitowy numer kolejny ramki. Ramki są numerowane od 0 do 2047 modulo 2048. W przypadku transmisji HS ramka podzielona jest dodatkowo na osiem mikroramek trwających po 125 µs. Do urządzenia HS pakiet SOF jest wysyłany na początku każdej mikroramki. Wszystkie osiem mikroramek w obrębie pojedynczej ramki ma ten sam numer. Pakiety podtrzymujące i SOF, oprócz synchronizacji czasu, służą do podtrzymywania urządzenia w stanie aktywnym. Brak trzech kolejnych pakietów powoduje wstrzymanie urządzenia (ang. *suspend*). Więcej piszę o tym w rozdziale 5 przy okazji omawiania zarządzania zasilaniem w interfejsie USB. W trybie HS w czasie wysyłania znacznika EOP pakiet SOF sprawdza się, czy urządzenie jest nadal podłączone do szyny. Jak wspomniałem

Tab. 1.4. Identyfikatory pakietów

PID	Kod	Opis
Żetony		
OUT	0001	Rozpoczyna transakcję, której celem jest przesłanie danych z kontrolera do punktu końcowego. Zawiera adres urządzenia i numer punktu końcowego
IN	1001	Rozpoczyna transakcję, której celem jest przesłanie danych z punktu końcowego do kontrolera. Zawiera adres urządzenia i numer punktu końcowego
SOF	0101	Rozpoczyna nową ramkę. Zawiera numer ramki
SETUP	1101	Rozpoczyna transakcję, której celem jest zainicjowanie transmisji danych sterujących. Zawiera adres urządzenia i numer punktu końcowego
SPLIT	1000	Rozpoczyna lub kończy transakcję dzieloną HS
PING	0100	Służy do sterowania przepływem dla danych sterujących i masowych HS
PRE	1100	Preambuła rozpoczynająca pakiet kapsułkujący pakiet LS w ruchu FS
Dane		
DATA0	0011	Identyfikuje parzysty pakiet z danymi
DATA1	1011	Identyfikuje nieparzysty pakiet z danymi
DATA2	0111	Identyfikuje pakiet z danymi izochronicznymi w trybie HS
M DATA	1111	Identyfikuje pakiet z danymi izochronicznymi w trybie HS. Używany też w transakcjach dzielonych
Potwierdzenia		
ACK	0010	Odbiorca potwierdza bezbłędne zakończenie transakcji
NAK	1010	Odbiorca nie jest gotowy, aby odebrać dane, lub nadawca nie jest gotowy, aby wysłać dane. Należy po pewnym czasie powtórzyć próbę wykonania transakcji
NYET	0111	Odbiorca potwierdza bezbłędne zakończenie transakcji dla danych masowych HS, ale w buforze odbiorczym nie ma dostatecznie dużo miejsca, aby odebrać kolejny pełny pakiet. Negatywne potwierdzenie wysypane przez koncentrator HS w odpowiedzi na próbę zakończenia transakcji dzielonej przez kontroler, gdy transakcja na porcie LS/FS jeszcze się nie zakończyła
STALL	1110	Punkt końcowy jest wstrzymany lub żądanie (dotyczy tylko danych sterujących) nie jest obsługiwane. Błąd jest permanentny i nie należy powtarzać transakcji
ERR	1100	Negatywne potwierdzenie wysypane przez koncentrator HS, gdy podczas transakcji dzielonej wystąpił błąd na porcie LS/FS
Pozostałe		
	0000	Zarezerwowany i nieużywany w USB 2.0, jest wykorzystywany w USB 3.0

wyżej, odłączenie urządzenia spowoduje zwiększenie amplitudy napięcia na liniach danych po stronie nadajnika.

Kontroler szyny USB przydziela urządzeniom i koncentratorom 7-bitowe adresy. Adres o wartości 0 jest zarezerwowany jako domyślny. Jest wykorzystywany przez urządzenie po jego poprawnym wyzerowaniu tylko w procedurze konfigurowania i przydzielania docelowego unikalnego adresu (ang. *enumeration*). Podczas tej procedury kontroler nadaje urządzeniu adres z przedziału 1...127. Opisuję to szczegółowo w podrozdziale 1.6.

Aby umożliwić przesyłanie wielu strumieni danych między kontrolerem a pojedynczym urządzeniem, wprowadzono dodatkowy poziom adresowania, mianowicie urządzenie logiczne, które w dokumentacji USB nazywane jest punktem końcowym (ang. *endpoint*). W ramach jednego urządzenia FS lub HS może być skonfigurowanych do 16 punktów końcowych. Urządzenie LS może mieć ich maksymalnie 3. Punkt końcowy może być jednokierunkowy lub dwukierunkowy. Jednokierunkowy punkt końcowy może być skonfigurowany jako wysyłający dane do kontrolera (kierunek określany po angielsku jako *in*) lub odbierający dane od kontrolera (określany po angielsku jako *out*). Dwukierunkowy punkt końcowy tworzy się jako dwa jednokierunkowe punkty końcowe powiązane ze sobą logicznie w aplikacji i przesyłające ten sam rodzaj danych. Kierunki przesyłania danych po szynie USB są podawane zawsze z perspektywy kontrolera. Punkt końcowy ma przypisany jeden z czterech dostępnych w USB rodzajów przesyłanych danych: sterujące, izochroniczne, masowe, pilne. Punkty końcowe otrzymują 4-bitowe numery, które są liczbami z przedziału 0...15. Każde urządzenie musi uruchomić przynajmniej dwukierunkowy punkt końcowy o numerze 0, przeznaczony dla danych sterujących. Ten punkt końcowy służy do konfigurowania urządzenia i zarządzania nim. Inne punkty końcowe, o niezerowych numerach, są uruchamiane w zależności od pełnionej przez urządzenie funkcji. Istnieje możliwość dynamicznej zmiany konfiguracji punktów końcowych, choć typowe urządzenia USB praktycznie z tego nie korzystają. Podsumowując, jednoznaczne zaadresowanie punktu końcowego wymaga podania w żetonie 11-bitowego pola ADRES (patrz rysunek 1.7) zawierającego 7-bitowy adres urządzenia Addr i 4-bitowy numer punktu końcowego Endp. Format pola ADRES przedstawiony jest na **rysunku 1.9**.

Addr ₀	Addr ₁	Addr ₂	Addr ₃	Addr ₄	Addr ₅	Addr ₆	Endp ₀	Endp ₁	Endp ₂	Endp ₃
-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------

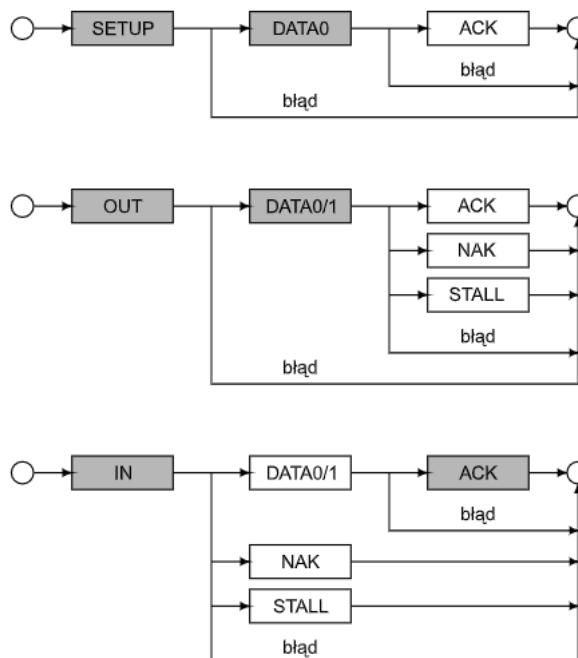
Rys. 1.9. Format pola adresowego

Pojedyncza transmisja danych odbywa się jako transakcja, w której uczestniczą dwie strony: kontroler i punkt końcowy. Transakcję zawsze rozpoczyna kontroler, wysyłając pakiet nazywany żetonem (ang. *token*), który określa typ transakcji, adres urządzenia oraz numer punktu końcowego uczestniczącego w tej transakcji. Następnie może być przesłany pakiet z danymi (ang. *data*). Transakcję może zakończyć pakiet z potwierdzeniem (ang. *handshake*). Identyfikatory żetonów, pakietów z danymi i potwierdzeń zaprezentowane są w tabeli 1.4. Schematy podstawowych rodzajów transakcji są zamieszczone na **rysunku 1.10** w postaci automatów stanowych. Pakiety wysyłane przez kontroler są oznaczane prostokątami z szarym tłem,

a pakiety wysyłane przez urządzenie – prostokątami z białym tłem. Każda transakcja rozpoczyna się i kończy w stanie spoczynkowym (ang. *idle*), który na rysunku zaznaczony jest symbolem okręgu. Nazwy transakcji pochodzą od nazw żetonów. Transakcja nazywa się tak jak żeton, który ją rozpoczyna.

Jeśli podczas transmisji żetonu (SETUP, OUT lub IN na rysunku 1.10) wystąpi błąd, np. nie zgadzają się bity kontrolne identyfikatora pakietu, to urządzenie końcowe nie odpowiada i przechodzi do stanu spoczynkowego. Podobnie, jeśli któryś kolejny odebrany pakiet jest błędny, to jest on ignorowany i kontroler lub urządzenie, które odebrało taki pakiet, powraca do stanu spoczynkowego. Pakiet jest uznawany za błędny, gdy wystąpił błąd nadziewania bitami (ang. *bit stuff error*), nie jest zakończony prawidłowym znacznikiem końca pakietu EOP, nie zgadza się wartość CRC lub jest za długi (ang. *babble error*). Transakcja nie może przekroczyć granicy ramki lub mikroramki. Zatem za błędny jest również uznawany pakiet, który przekroczył granicę ramki (ang. *frame overrun*). Jeśli kontroler lub urządzenie oczekuje na jakiś pakiet i nie pojawi się on w określonym czasie, to też uznawane jest to za błąd (ang. *timeout*) i następuje powrót do stanu spoczynkowego.

Transakcja SETUP rozpoczyna każdą transmisję danych sterujących. Kontroler wysyła żeton SETUP, a po nim pakiet z danymi DATA0. Urządzenie musi potwierdzić odebranie danych pakietem ACK. Jeśli transakcja SETUP zostanie zainicjowana do punktu końcowego, który nie jest skonfigurowany dla danych sterujących, urządzenie ignoriuje ją i powraca do stanu spoczynkowego.

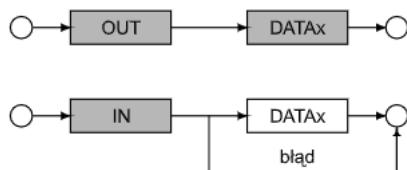


Rys. 1.10. Podstawowe transakcje

Transakcja OUT służy do przesyłania danych z kontrolera do urządzenia. Po wysłaniu żetona OUT kontroler wysyła dane w pakiecie DATA0 lub DATA1. Jeśli urządzenie odbierze poprawny pakiet z danymi, potwierdza go, wysyłając pakiet ACK. Jeśli urządzenie z jakiegoś powodu nie może zaakceptować pakietu z danymi, ale nie jest to problem permanentny, wysyła pakiet NAK, co oznacza, że kontroler powinien powtórzyć transakcję za jakiś czas. Jeśli punkt końcowy jest wstrzymany lub nieaktywny, urządzenie odsyła pakiet STALL, co oznacza permanentną niedostępność punktu końcowego. W tym przypadku kontroler nie powinien próbować powtarzać transakcji.

Transakcja IN służy do przesyłania danych z urządzenia do kontrolera. Po odebraniu żetona IN urządzenie wysyła dane w pakiecie DATA0 lub DATA1. Jeśli urządzenie nie jest gotowe i nie może wysłać danych, wysyła pakiet NAK, który oznacza, że kontroler powinien ponowić transakcję za jakiś czas. Urządzenie wysyła STALL, jeśli punkt końcowy jest wstrzymany lub nieaktywny i nie może wysłać danych, a kontroler nie powinien próbować powtarzać transakcji. Jeśli kontroler odbierze poprawny pakiet z danymi, potwierdza go, wysyłając pakiet ACK. Jeśli kontroler nie może odebrać danych, to nic nie wysyła i przechodzi do stanu spoczynkowego.

Opisana powyżej transakcja SETUP stosowana jest tylko do przesyłania danych sterujących. Natomiast transakcje OUT i IN używane są do przesyłania danych sterujących, masowych i pilnych.



Rys. 1.11. Transakcje dla danych izochronicznych

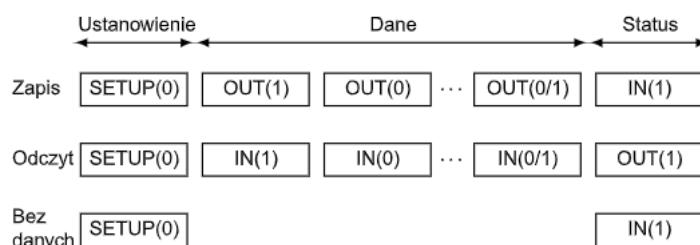
Do przesyłania danych izochronicznych, które nie wymagają potwierdzania, wykorzystuje się uproszczone transakcje przedstawione na **rysunku 1.11**. Jeśli kontroler chce wysłać dane do urządzenia, wysyła żeton OUT, a potem od razu pakiet z danymi. Jeśli kontroler chce otrzymać dane do urządzenia, wysyła żeton IN. Urządzenie, jeśli ma jakieś dane do wysłania, wysyła je, a w przeciwnym przypadku ignoruje tę transakcję. Dla transmisji FS w jednej ramce może zostać zainicjowana tylko jedna transakcja izochroniczna do konkretnego punktu końcowego i używa się tylko pakietów DATA0. Dla transmisji HS kontroler może inicjować w jednej mikroramce do trzech transakcji izochronicznych. Dla transakcji OUT identyfikator ostatniego pakietu danych wysłanego przez kontroler w danej mikroramce określa liczbę transakcji, które zostały zainicjowane, dzięki czemu urządzenie może stwierdzić, czy jakieś pakiety nie zostały utracone. W przypadku transakcji IN identyfikator pierwszego pakietu danych wysłanego przez urządzenie w danej mikroramce determinuje liczbę kolejnych transakcji, które chce zrealizować urządzenie i które kontroler powinien zainicjować. Stosowane są następujące schematy:

- jedna transakcja OUT – pakiet DATA0,
- dwie transakcje OUT – kolejno pakiety MDATA i DATA1,

- trzy transakcje OUT – kolejno pakiety MDATA, MDATA i DATA2,
- jedna transakcja IN – pakiet DATA0,
- dwie transakcje IN – kolejno pakiety DATA1 i DATA0,
- trzy transakcje IN – kolejno pakiety DATA2, DATA1 i DATA0.

Kontroler przesyła dane sterujące w postaci żądań (ang. *request*), które składają się z trzech faz. Formaty żądań przedstawione są na **rysunku 1.12**. W fazie ustanowienia (ang. *setup stage*) kontroler inicjuje transakcję SETUP(0). Numer w nawiasie nazwy transakcji oznacza rodzaj pakietu z danymi, który powinien być użyty w tej transakcji. Pakiet DATA0 tej transakcji zawiera 8-bajtową strukturę setup (opisaną w podrozdziale 1.6), której zawartość określa m.in., czy kontroler chce zapisać dane do urządzenia (ang. *write data request*), czy chce odczytać dane z urządzenia (ang. *read data request*), czy też nie będzie żadnych dodatkowych danych (ang. *no-data request*), gdyż wszystkie przesyłane dane znajdują się w strukturze setup. Faza danych (ang. *data stage*) jest opcjonalna i umożliwia przesłanie porcji danych za pomocą serii transakcji OUT(1) i OUT(0) lub IN(1) i IN(0). Maksymalna ilość danych do przesłania jest określana przez odpowiednie pole w strukturze setup. Jeśli urządzenie ma do wysłania mniej danych, niż chce odczytać kontroler, to wysyła w ostatniej transakcji IN pakiet danych zawierający mniej niż maksymalny (dla danego punktu końcowego) dopuszczalny rozmiar pola danych dla pakietów DATA0 i DATA1. Może to być zero, jeśli w poprzedniej transakcji IN pakiet z danymi był całkowicie wypełniony. Faza statusu (ang. *status stage*) służy do zakończenia realizacji żądania. W tym celu kontroler inicjuje transakcję o przeciwnym kierunku do poprzedniej, tzn. IN(1), gdy był wykonywany zapis lub nie było żadnych danych, a OUT(1), gdy był wykonywany odczyt. W fazie statusu wysyłany jest pusty (zero bajtów) pakiet DATA1. Kontroler może przerwać odczyt danych z urządzenia, zanim wyśle ono wszystkie dane, przechodząc do fazy statusu, zamiast inicjować kolejną transakcję IN.

Przesyłanie danych masowych i pilnych jest dużo prostsze. Jeśli kontroler chce wysłać (zapisać) dane do urządzenia, to inicjuje serię transakcji OUT(0) i OUT(1). Jeśli kontroler chce odebrać (odczytać) dane z urządzenia, to inicjuje serię transakcji IN(0) i IN(1). Pokazane jest to schematycznie na **rysunku 1.13**. Jeśli przesyłany ciąg danych nie mieści się w pojedynczej transakcji, to wszystkie transakcje, poza ewentualnie ostatnią, powinny przesyłać maksymalną porcję danych. Jeśli urządzenie chce zasygnalizować kontrolerowi, że nie ma już więcej danych do wysła-



Rys. 1.12. Formaty żądań dla danych sterujących

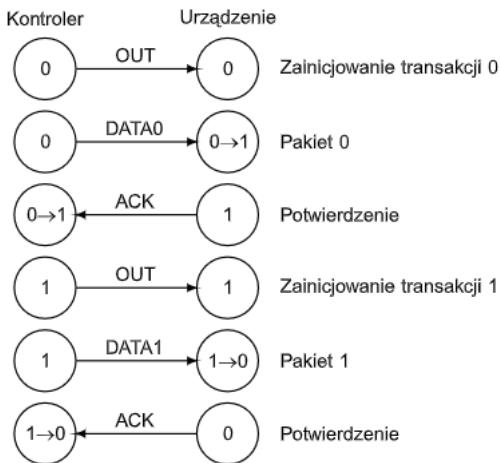


Rys. 1.13. Przesyłanie danych masowych i pilnych

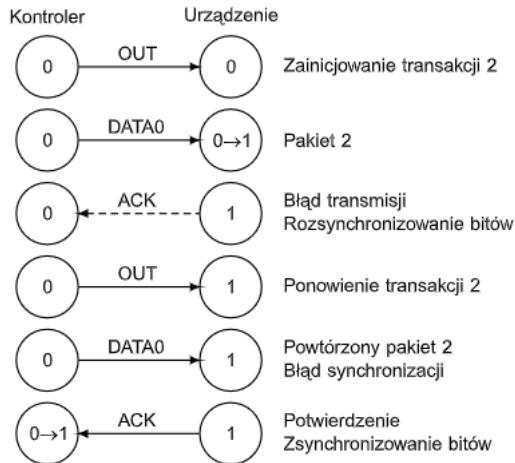
nia, to w kolejnej transakcji wysyła pakiet zawierający mniej danych, niż wynosi maksymalny rozmiar pola danych. W szczególności może wysłać pakiet pusty, bez danych. Przesyłanie danych izochronicznych jest równie proste. Kontroler w regularnych odstępach czasu inicjuje transakcje z rysunku 1.11.

Jak widać, w kolejnych transakcjach (nieizochronicznych) używa się na przemian pakietów DATA0 i DATA1. Jest to dodatkowy mechanizm synchronizacji (kontroli przepływu danych) umożliwiający poprawną interpretację potwierdzeń. Nie używa się go dla danych izochronicznych, gdyż one nie wymagają potwierdzania. Dla nieizochronicznych punktów końcowych kontroler i urządzenie utrzymują po jednym bicie synchronizacyjnym dla każdego kierunku transmisji. U nadawcy bit synchronizacyjny określa, który z pakietów DATA0 lub DATA1 ma być wysłany jako następny. U odbiorcy wartość bitu synchronizacyjnego określa, który z pakietów DATA0 czy DATA1 jest oczekiwany. Jeśli bit synchronizacyjny ma wartość 0, to nadawca wysyła pakiet DATA0. Jeśli bit synchronizacyjny ma wartość 1, to nadawca wysyła pakiet DATA1. Jeśli nadawca odbierze pakiet potwierdzający ACK, to zmienia wartość swojego bitu synchronizacyjnego na przeciwną (ang. *toggle*). Jeśli nadawca odbierze negatywne potwierdzenie NAK lub nie dostanie żadnego potwierdzenia, to nie zmienia wartości bitu synchronizacyjnego. Jeśli wartość bitu synchronizacyjnego u odbiorcy zgadza się z rodzajem odebranego pakietu DATA i odbiorca akceptuje ten pakiet, to zmienia wartość swojego bitu synchronizacyjnego na przeciwną. W przeciwnym przypadku odbiorca nie zmienia wartości swojego bitu synchronizacyjnego.

Przykłady kontroli przepływu w transakcjach OUT i IN przedstawione są na kolejnych rysunkach. Liczby w okrągach po lewej stronie oznaczają kolejne wartości bitu synchronizacyjnego kontrolera, a liczby w okrągach po prawej – urządzenie. Dwie wartości oddzielone strzałką oznaczają zmianę wartości tego bitu. Na rysunku 1.14 widzimy dwie kolejne poprawne transakcje OUT. Na rysunku 1.15 przedstawiona jest sytuacja, gdy podczas transakcji OUT zostało zgubione potwierdzenie, na przykład wskutek błędu transmisji. Brak potwierdzenia sprawia, że kontroler nie zaneguje swojego bitu i jego wartość przestaje być zsynchronizowana z bitem w urządzeniu. Urządzenie nie wykrywa jednak tej sytuacji, gdyż odebrało poprawny pakiet. Natomiast kontroler, nie doczekawszy się potwierdzenia, ponawia transakcję, w której ponownie wysyła ten sam pakiet DATA0. Jednak urządzenie oczekuje teraz pakietu DATA1, więc ignoruje odebrany pakiet i nie zmienia swojego bitu synchronizacyjnego. Jednak odsyła potwierdzenie ACK, co sprawia, że bit po stronie kontrolera zostanie zanegowany i nastąpi zsynchronizowanie wartości bitów po obu stronach. Rysunek 1.16 ilustruje przebieg komunikacji, gdy kontroler inicjuje transakcję OUT i wysyła pakiet danych, ale urządzenie odrzuca transakcję pakietem NAK, gdyż nie ma wolnej pamięci do odczytania pakietu z danymi lub nie skończyło obsługi poprzedniego pakietu. W tej sytuacji bity synchronizacyjne



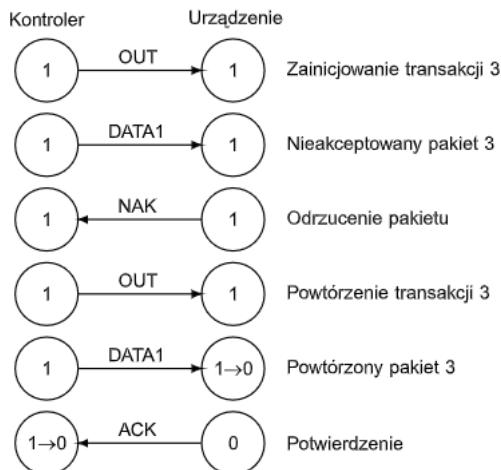
Rys. 1.14. Kontrola przepływu w transakcji OUT przy braku błędów



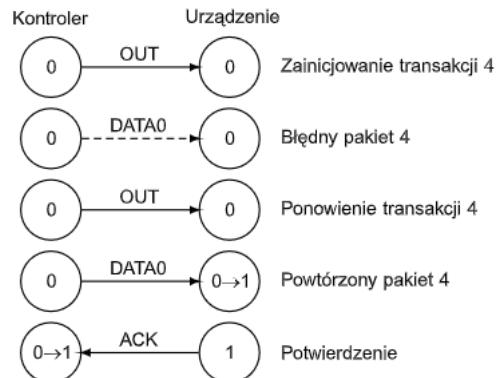
Rys. 1.15. Kontrola przepływu w transakcji OUT przy utracie potwierdzenia

nie są modyfikowane, a kontroler powtarza transakcję. Na rysunku 1.17 widzimy przebieg komunikacji, gdy urządzenie odebrało błędny pakiet danych, na przykład nie zgadza się jego suma kontrolna CRC16. W tej sytuacji bity synchronizacyjne również nie są modyfikowane i transakcja jest przerwana. Po upływie czasu przeznaczonego na oczekiwanie na potwierdzenie kontroler ponawia transakcję.

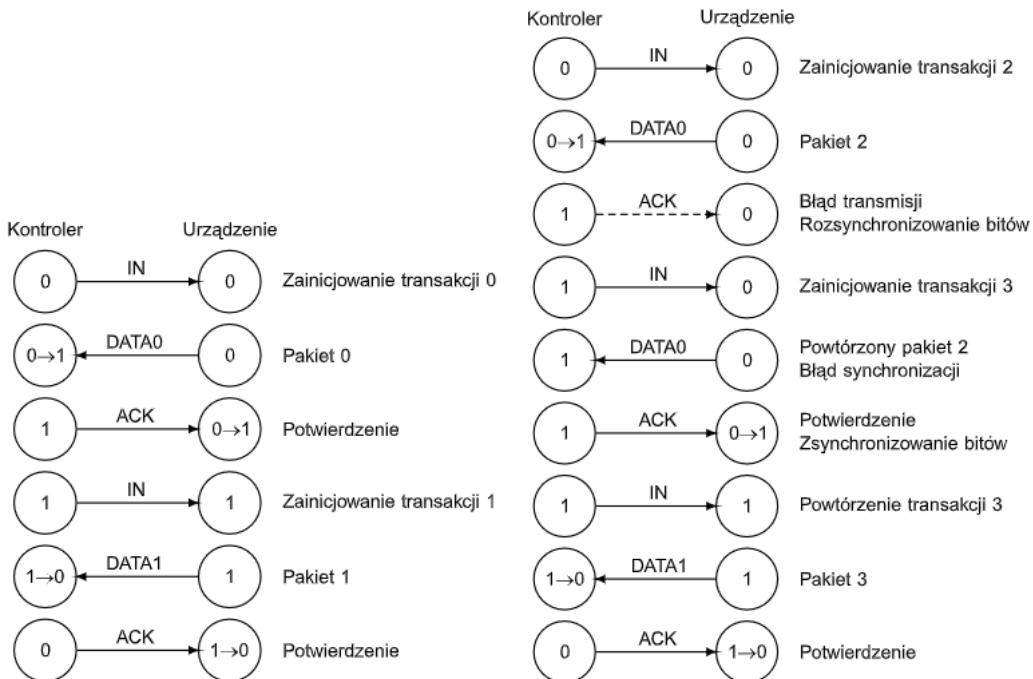
Rysunek 1.18 przedstawia dwie kolejne poprawne transakcje IN. Na rysunku 1.19 widzimy przebieg komunikacji, gdy brakuje potwierdzenia transakcji IN. Kontroler, odebrawszy pakiet DATA0, zanegował swój bit synchronizacyjny, ale urządzenie, nie otrzymawszy potwierdzenia, swojego bitu nie zanegowało, dlatego wartości bitów przestały być jednakowe. Kontroler nie wykrył jeszcze tej sytuacji, więc inicju-



Rys. 1.16. Kontrola przepływu w transakcji OUT przy odrzuceniu pakietu przez urządzenie



Rys. 1.17. Kontrola przepływu w transakcji OUT przy utracie pakietu

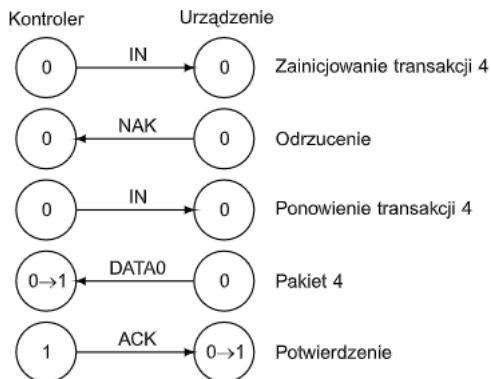


Rys. 1.19. Kontrola przepływu w transakcji IN przy utracie potwierdzenia

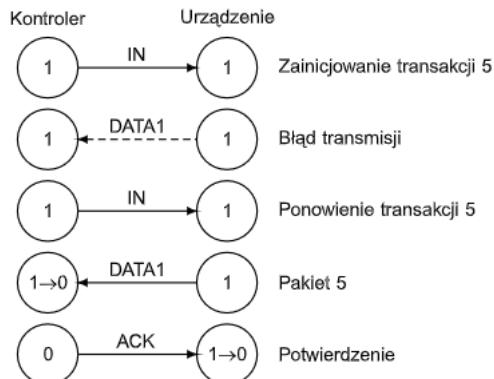
je kolejną transakcję IN, w której urządzenie powtórzy poprzedni pakiet DATA0, gdyż dla urządzenia poprzednia transakcja nie zakończyła się poprawnie. W tym momencie kontroler oczekuje pakietu DATA1, więc wykrywa błąd synchronizacji i ignoruje otrzymany pakiet. Kontroler odsyła potwierdzenie ACK, aby ponownie zsynchronizować bity. Następnie powtarza transakcję, w której poprawnie odbiera pakiet DATA1. Na rysunku 1.20 zamieszczony jest przebieg komunikacji, gdy urządzenie nie jest gotowe do zrealizowania transakcji IN, gdyż nie ma gotowego pakietu danych do wysłania. Urządzenie sygnalizuje to, odsyłając pakiet NAK. Bity synchronizacyjne nie są modyfikowane. Po pewnym czasie kontroler ponawia transakcję, w której otrzymuje pakiet DATA0. Na rysunku 1.21 widzimy komunikację, gdy kontroler odebrał błędny pakiet danych. W tej sytuacji bity synchronizacyjne również nie są modyfikowane. Kontroler ponawia transakcję i otrzymuje poprawny pakiet DATA1.

Standard USB stwierdza, że kontroler powinien podjąć maksymalnie trzy próby zrealizowania transakcji. Jeśli trzecia próba zakończy się niepowodzeniem, kontroler uzna, że punkt końcowy (urządzenie) nie działa poprawnie i wstrzymuje dalszą z nim komunikację.

Bity synchronizacyjne muszą zostać poprawnie zainicjowane. Dla punktów końcowych innych niż dla danych sterujących i izochronicznych po każdorazowym uaktywnieniu takiego punktu końcowego związane z nim bity synchronizacyjne, zarówno po stronie urządzenia, jak i po stronie kontrolera, muszą zostać wyzerowane.

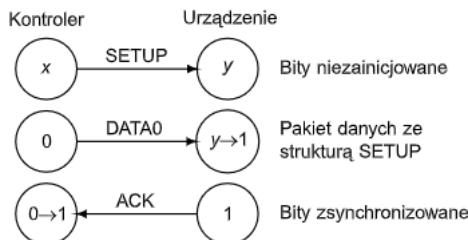


Rys. 1.20. Kontrola przepływu w transakcji IN przy jej odrzuceniu



Rys. 1.21. Kontrola przepływu w transakcji IN przy utracie pakietu

ne. W przeciwnym przypadku już pierwsza transakcja nie powiedzie się i nastąpi utrata danych. Za poprawne zainicjowanie tych bitów odpowiada aplikacja. Żeby uniknąć takiego problemu dla danych sterujących, których poprawne przesyłanie jest krytyczne dla działania całego urządzenia, wprowadzono specjalny mechanizm inicjowania bitów synchronizacyjnych dla punktu końcowego przesyłającego dane sterujące. Schemat komunikacji przedstawiony jest na rysunku 1.22. Po wysłaniu pakietu DATA0 kontroler ustawia swój bit na 0, a po odebraniu tego pakietu urządzenie ustawia swój bit na 1. Po odebraniu potwierdzenia ACK kontroler neguje swój bit. Od tego momentu bity po obu stronach są zsynchronizowane i gotowe do zrealizowania pierwszej transakcji OUT(1) lub IN(1) z pakietem DATA1.



Rys. 1.22. Bity synchronizacyjne podczas transakcji SETUP(0)

1.5. Deskryptory

Wszystkie właściwości urządzenia USB i formaty przesyłanych danych opisuje się za pomocą deskryptorów. Kontroler rozpoznaje urządzenie, odczytując z niego odpowiednie deskryptory. Dzięki temu można też automatycznie wybrać programowy sterownik urządzenia (ang. *device driver*). Zatem, być może nieco na siłę, deskryptory można uznać za element warstwy prezentacji. W tym podrozdziale opisuję tylko deskryptory wspólne dla wszystkich klas urządzeń. Deskryptory specyficzne dla konkretnych klas przedstawiam przy okazji omawiania przykładów implementujących te klasy.

Deskryptory są strukturami danych. Prezentuję je w formie tabelarycznej, podobnie jak w oficjalnej dokumentacji USB. W nazwach pól w strukturach deskryptorów zastosowano tak zwaną notację węgierską, w której prefiks nazwy zmiennej określa jej typ i przeznaczenie. Prefiks b oznacza pole jednobajtowe (ang. *byte*), prefiks w – słowo dwubajtowe (ang. *word*), prefiks i – jednobajtowy indeks będący liczbą całkowitą (ang. *integer*), prefiks bm to pole bitowe (ang. *bitmap*). Prefiks bcd oznacza dwubajtową liczbę w kodzie BCD, stosowaną do kodowania wersji standardu, protokołu, urządzenia itp. Na przykład liczba szesnastkowa 0x0110 oznacza wersję 1.1, liczba 0x0200 to wersja 2.0, wersję 4.01 należałoby zakodować jako 0x0401, a wersję 2.1.3 jako 0x0213. Prefiks id oznacza dwubajtowy identyfikator. Przypomnę, że w USB wartości wielobajtowe są zapisywane i przesyłane w porządku cienkokońcowkowym. Mimo że prefiks nazwy pola jednoznacznie określa jego rozmiar, w poniższych tabelach zamieszczam też rozmiary poszczególnych pól w bajtach, gdyż w kilku przypadkach zastosowano niestandardowe prefiksy nazw. Wszystkie deskryptory mają takie same dwa pierwsze pola. Pole **bLength** zawiera całkowity rozmiar deskryptora w bajtach. Pole **bDescriptorType** pozwala zidentyfikować deskryptor. Wartości tych dwóch pól dla poszczególnych typów deskryptorów umieszczone w nawiasach po opisie danego pola.

Każde urządzenie USB ma dokładnie jeden deskryptor urządzenia, zawierający ogólne informacje o tym urządzeniu, przedstawiony w **tabeli 1.5**. Jeśli urządzenie HS musi mieć dwa zestawy parametrów, oddzielnie dla trybu HS i FS, powinno mieć też deskryptor kwalifikujący (ang. *device qualifier descriptor*), pokazany w **tabeli 1.6**. Wersja specyfikacji USB zapisana w deskryptorach wskazuje wersję, według której zostało zaprojektowane dane urządzenie. Urządzenia HS muszą mieć wersję co najmniej 2.0. Urządzenia LS i FS powinny być zgodne z wersją 1.1 lub 2.0. Wszystkie prezentowane w tej książce przykłady są zgodne z wersją 2.0 i dlatego mają tę wersję w polu **bcdUSB** odpowiednich deskryptorów.

Tab. 1.5. Deskryptor urządzenia

Pole	Rozmiar	Opis
bLength	1	Rozmiar deskryptora (18)
bDescriptorType	1	Typ deskryptora (1)
bcdUSB	2	Wersja specyfikacji USB
bDeviceClass	1	Kod klasy urządzenia
bDeviceSubClass	1	Kod podklasy urządzenia
bDeviceProtocol	1	Kod protokołu
bMaxPacketSize0	1	Maksymalny rozmiar pola danych pakietów DATA0 i DATA1 dla punktu końcowego nr 0
idVendor	2	Identyfikator producenta
idProduct	2	Identyfikator produktu
bcdDevice	2	Wersja urządzenia
iManufacturer	1	Numer deskryptora tekstowego z nazwą producenta
iProduct	1	Numer deskryptora tekstowego z opisem produktu
iSerialNumber	1	Numer deskryptora tekstowego z numerem seryjnym urządzenia
bNumConfigurations	1	Liczba dostępnych konfiguracji

Tab. 1.6. Deskryptor kwalifikujący urządzenia

Pole	Rozmiar	Opis
bLength	1	Rozmiar deskryptora (10)
bDescriptorType	1	Typ deskryptora (6)
bcdUSB	2	Wersja specyfikacji USB
bDeviceClass	1	Kod klasy
bDeviceSubClass	1	Kod podklasy
bDeviceProtocol	1	Kod protokołu
bMaxPacketSize0	1	Maksymalny rozmiar pola danych pakietów DATA0 i DATA1 dla punktu końcowego nr 0
bNumConfigurations	1	Liczba dostępnych konfiguracji
bReserved	1	Zarezerwowane (0)

Pola bDeviceClass, bDeviceSubClass i bDeviceProtocol określają odpowiednio klasę, podklasę i protokół komunikacyjny urządzenia. Wartości 1...254 (0x01...0xFE) są przyznawane przez USB-IF (organizację standaryzującą USB). Jeśli pola te mają wartość 0, to klasa, podklasa i protokół nie są określone globalnie dla całego urządzenia (ang. *unspecified*), a są zdefiniowane na poziomie interfejsu w odpowiednim deskryptorze interfejsu (patrz niżej). Przypisanie tym polom wartości 255 (0xFF) oznacza, że urządzenie nie należy do żadnej ze standardowych klas i realizuje protokół specyficzny dla jego twórcy (ang. *vendor specific*). Umożliwia to implementowanie własnych protokołów aplikacyjnych. Inne wartości, które mogą się pojawić w polu bDeviceClass, to:

- 2 (0x02) – urządzenie CDC, np. modem lub wirtualny port szeregowy,
- 9 (0x09) – koncentrator,
- 220 (0xDC) – urządzenie diagnostyczne,
- 239 (0xEF) – rozmaite urządzenia (ang. *miscellaneous*).

Pole bMaxPacketSize0 określa maksymalny rozmiar w bajtach pola danych przy przesyłaniu danych sterujących między kontrolerem a domyślnym punktem końcowym o numerze 0. Dla urządzeń LS pole to musi mieć wartość 8. Dla urządzeń FS może przyjmować wartości 8, 16, 32 lub 64. Dla urządzeń HS jedyna dopuszczalna wartość tego pola wynosi 64.

Pole idVendor jest 16-bitowym identyfikatorem producenta VID danego urządzenia. Identyfikatory te są rozdzielane odpłatnie przez USB-IF. Pole idProduct zawiera 16-bitowy identyfikator produktu PID nadawany przez jego producenta. Producenci mikrokontrolerów wyposażonych w interfejs urządzenia USB oferują czasem możliwość używania ich VID i pewnego zakresu numerów PID za niewielką opłatą, ale tylko dla małych serii produkcyjnych. Należy wyraźnie podkreślić, że można narazić się na konsekwencje prawne, wykorzystując wykupiony przez jakąś firmę identyfikator w celach komercyjnych bez zgody tej firmy. Przez komercyjne wykorzystanie rozumiem wytwarzanie i wprowadzanie na rynek urządzeń USB. Zwykle dopuszczalne jest używanie identyfikatora bez ponoszenia opłaty w celu rozwijania lub testowania oprogramowania. Pole bcdDevice zawiera wersję urządzenia – wersja ta jest nadawana przez producenta zupełnie dowolnie i nie ma nic wspólnego z wersją USB. Pola iManufacturer, iProduct

i iSerialNumber to numery deskryptorów tekstowych (patrz niżej) zawierających czytelne dla człowieka napisy. Są to odpowiednio nazwa producenta, nazwa urządzenia i jego unikalny numer seryjny. Numer deskryptora tekstopowego jest liczbą dodatnią. Jeśli któryś z tych pól ma wartość 0, to oznacza, że odpowiedni napis nie jest dostępny.

Deskryptor kwalifikujący określa, jak zmieniają się parametry urządzenia przy zmianie szybkości. Jeśli urządzenie pracuje aktualnie w trybie FS, to deskryptor kwalifikujący zawiera informacje o parametrach w trybie HS. Odwrotnie, jeśli urządzenie pracuje w trybie HS, to zawiera on informacje o parametrach w trybie FS. Pola w deskryptorze kwalifikującym mają identyczne znaczenie jak pola o takich samych nazwach w deskryptorze urządzenia. Pole bReserved musi mieć wartość 0.

Urządzenie może mieć wiele zestawów parametrów konfiguracyjnych, nazywanych w skrócie konfiguracjami. Liczba dostępnych konfiguracji zapisana jest w deskryptorze urządzenia i deskryptorze kwalifikującym w polu bNumConfigurations. Większość typowych urządzeń nie wykorzystuje jednak tej możliwości i ma tylko jedną, a zdecydowanie rzadziej dwie konfiguracje. Dwie konfiguracje są potrzebne, gdy urządzenie może pracować z dwiema szybkościami: FS i HS. Wtedy dla każdej szybkości powinna być przewidziana osobna konfiguracja. W danym momencie aktywna może być tylko jedna z nich. Każda konfiguracja definiuje interfejsy, jakie udostępnia dane urządzenie. Zwykle konfiguracja udostępnia jeden interfejs, czasem dwa, ale może ich być więcej. Interfejs jest to zbiór punktów końcowych realizujących łącznie jakiś logiczny interfejs aplikacji. Aby zrealizować interfejs jednokierunkowy, wystarczy jeden jednokierunkowy punkt końcowy. Interfejs dwukierunkowy potrzebuje dwóch jednokierunkowych punktów końcowych, które mogą mieć różne numery lub ten sam numer. W tym drugim przypadku można mówić o jednym dwukierunkowym punkcie końcowym. Oczywiście interfejs może mieć więcej niż dwa punkty końcowe, jeśli zajdzie taka potrzeba, choć jest to dość rzadko spotykane w praktyce. Konfigurację opisuje się za pomocą standardowego deskryptora konfiguracji (ang. *standard configuration descriptor*) lub deskryptora konfiguracji dla innej szybkości (ang. *other speed configuration descriptor*), które są przedstawione w tabeli 1.7. Pierwszy z nich opisu-

Tab. 1.7. Deskryptor konfiguracji

Pole	Rozmiar	Opis
bLength	1	Rozmiar deskryptora (9)
bDescriptorType	1	Typ deskryptora (2 dla standardowego deskryptora konfiguracji, 7 dla deskryptora konfiguracji innej szybkości)
wTotalLength	2	Calkowity rozmiar tego deskryptora i wszystkich dodatkowych deskryptorów związanych z tą konfiguracją, m.in. deskryptorów interfejsów oraz punktów końcowych
bNumInterfaces	1	Liczba interfejsów opisywanych przez tę konfigurację
bConfigurationValue	1	Numer tej konfiguracji
iConfiguration	1	Numer deskryptora tekstopowego z opisem tej konfiguracji
bmAttributes	1	Atrybuty konfiguracji związane z zasilaniem
bMaxPower	1	Maksymalny prąd pobierany przez urządzenie z szyny

je bieżącą konfigurację, a drugi konfigurację dla trybu HS, gdy urządzenie pracuje aktualnie w trybie FS, lub konfigurację dla trybu FS, gdy urządzenie pracuje w trybie HS. Oba deskryptory mają identyczny format – różnią się jedynie wartością w polu `bDescriptorType`.

Pole `wTotalLength` zawiera całkowity rozmiar w bajtach danego deskryptora konfiguracji i wszystkich związanych z nim deskryptorów: standardowych i specyficznych dla danej klasy urządzenia, w tym deskryptorów interfejsów i punktów końcowych (patrz niżej). Pole `bNumInterfaces` określa liczbę interfejsów związanych z daną konfiguracją. Pole `bConfigurationValue` zawiera numer danej konfiguracji w obrębie urządzenia. Konfiguracje są numerowane od 1 do wartości z pola `bNumConfigurations` deskryptora urządzenia. Wartość 0 wykorzystuje się do oznaczenia braku aktywnej konfiguracji lub żądania dezaktywacji bieżącej konfiguracji. Pole `iConfiguration` zawiera numer deskryptora tekstowego opisującego tę konfigurację w sposób czytelny dla człowieka.

Pole `bmAttributes` zawiera atrybuty konfiguracji związane z zarządzaniem energią. Najbardziej znaczący bit tego pola – bit 7 – musi być ustawiony na 1 ze względu na kompatybilność z wersją 1.1 standardu. Jeśli bit 6 ma wartość 0, to urządzenie jest w całości zasilane z szyny VBUS. Jeśli bit ten ma wartość 1, to urządzenie ma własne zasilanie, ale może też pobierać prąd z szyny VBUS. Bit 5 ustawiony na 1 oznacza, że urządzenie może zainicjować zdalne wybudzenie (ang. *remote wakeup*) systemu komputerowego ze stanu o niskim poborze mocy. Pozostałe bity są zarezerwowane i muszą mieć wartość 0. Parametr `bMaxPower` deklaruje maksymalny prąd, jaki urządzenie chce pobierać z szyny VBUS. Wartość tego prądu podaje się w jednostkach 2 mA, czyli np. wartość 40 oznacza prąd 80 mA. Urządzenie może zażądać maksymalnie 500 mA. Urządzenie nie powinno pobierać większego prądu, niż zadeklarowało, a dodatkowo nie powinno pobierać więcej niż 100 mA, zanim kontroler nie uaktywni konfiguracji, która deklaruje pobór większego prądu. Kontroler aktywuje jedną z konfiguracji po odczytaniu wszystkich deskryptorów – patrz następny podrozdział. Więcej o zasilaniu urządzeń z szyny napisałem w rozdziale 5, gdzie znajdują się też stosowne przykłady.

Deskryptor interfejsu definiuje pojedynczy interfejs w obrębie konfiguracji. Jego format przedstawiony jest w tabeli 1.8. Pole `bInterfaceNumber` zawiera numer danego interfejsu. Interfejsy numeruje się w ramach danej konfiguracji od 0 do wartości o jeden mniejszej niż wartość pola `bNumInterfaces` w deskryptorze konfiguracji. Pole `bAlternateSetting` określa wariant ustawień danego interfejsu. Może występować wiele deskryptorów interfejsów o tym samym numerze z różnymi zestawami ustawień, które numeruje się od 0. Pole `bNumEndpoints` zawiera liczbę (jednokierunkowych) punktów końcowych przypisanych do tego interfejsu, a dokładnie liczbę deskryptorów punktów końcowych (patrz niżej) powiązanych z danym interfejsem. Nie wlicza się tu domyślnego punktu końcowego numer 0, dla którego też nie definiuje się żadnego deskryptora. Dwukierunkowe punkty końcowe liczy się podwójnie jako dwa jednokierunkowe punkty końcowe.

Pola `bInterfaceClass`, `bInterfaceSubClass` i `bInterfaceProtocol` określają odpowiednio klasę, podklasę i protokół komunikacyjny interfejsu. Używa się tu tych samych wartości, co dla odpowiednich pól w deskryptorze urządzenia. Różne war-

Tab. 1.8. Deskryptor interfejsu

Pole	Rozmiar	Opis
bLength	1	Rozmiar deskryptora (9)
bDescriptorType	1	Typ deskryptora (4)
bInterfaceNumber	1	Numer tego interfejsu
bAlternateSetting	1	Wariant ustawień
bNumEndpoints	1	Liczba (jednokierunkowych) punktów końcowych używanych przez ten interfejs, bez punktu końcowego nr 0
bInterfaceClass	1	Kod klasy
bInterfaceSubClass	1	Kod podklasy
bInterfaceProtocol	1	Kod protokołu
iInterface	1	Numer deskryptora tekstowego z opisem tego interfejsu

tości, które mogą znaleźć się w tych polach, poznamy przy omawianiu stosowanych przykładów. Wartości 1...254 (0x01...0xFE) są przyznawane przez USB-IF. Wartość 0 jest zarezerwowana i nie powinna być używana. Wartość 255 (0xFF) oznacza niestandardowy interfejs, realizujący specyficzny protokół, zdefiniowany przez wytwórcę urządzenia. (ang. *vendor specific*). Umożliwia to implementowanie własnych protokołów aplikacyjnych – stosowny przykład znajduje się w dalszej części książki. Pole **iInterface** zawiera numer deskryptora tekstowego opisującego ten interfejs w sposób czytelny dla człowieka. Wybrane wartości, które mogą się pojawić w polu **bInterfaceClass**, to:

- 1 (0x01) – interfejs ADC, np. głośnik, mikrofon, karta dźwiękowa, MIDI,
- 2 (0x02) – interfejs sterujący CDC, np. modem lub wirtualny port szeregowy,
- 3 (0x03) – interfejs HID, typowo mysz lub klawiatura,
- 5 (0x05) – interfejs PID, np. dżojstik z oddziaływaniem zwrotnym,
- 6 (0x06) – interfejs urządzenia obrazowego, np. kamery lub skanera,
- 7 (0x07) – interfejs urządzenia drukującego, wszelkiego rodzaju drukarki lub plotery,
- 8 (0x08) – interfejs MSC, pamięć masowa, np. pendrive lub zewnętrzny dysk, ale też czytnik kart pamięciowych, odtwarzacz audio lub kamera,
- 10 (0x0A) – interfejs komunikacyjny CDC, np. modem lub wirtualny port szeregowy,
- 11 (0x0B) – interfejs czytnika kart chipowych,
- 13 (0x0D) – interfejs czytnika zabezpieczeń, np. czytnika odcisków palców,
- 14 (0x0E) – interfejs wideo, np. kamery,
- 220 (0xDC) – interfejs diagnostyczny,
- 224 (0xE0) – interfejs kontrolera sieci bezprzewodowej, np. Bluetooth,
- 239 (0xEF) – rozmaite interfejsy (ang. *miscellaneous*),
- 254 (0xFE) – interfejs specyficzny dla aplikacji (ang. *application specific*).

Deskryptor punktu końcowego opisuje rodzaj danych przesyłanych między kontrolerem a tym punktem końcowym. Pozwala też wyznaczyć potrzebne pasmo i rozmiary buforów, które kontroler musi ewentualnie zarezerwować dla tego punktu

końcowego. Domyślnego punktu końcowego numer 0 dla danych sterujących nie opisuje się za pomocą deskryptora. Wszystko, co o nim musi dowiedzieć się kontroler, to maksymalny rozmiar pola danych znajdujący się w polu `bMaxPacketSize0`, które odczytuje się z deskryptora urządzenia. Format deskryptora punktu końcowego przedstawiony jest w **tabeli 1.9**.

Tab. 1.9. Deskryptor punktu końcowego

Pole	Rozmiar	Opis
<code>bLength</code>	1	Rozmiar deskryptora (7)
<code>bDescriptorType</code>	1	Typ deskryptora (5)
<code>bEndPointAddress</code>	1	Numer i kierunek punktu końcowego
<code>bmAttributes</code>	1	Rodzaj przesyłanych danych oraz dodatkowe cechy i atrybuty punktu końcowego
<code>wMaxPacketSize</code>	2	Maksymalny rozmiar pola danych pakietów DATAx dla tego punktu końcowego
<code>bInterval</code>	1	Okres odpytywania punktu końcowego

Pole `bEndPointAddress` zawiera adres punktu końcowego, czyli numer punktu końcowego i jego kierunek. Najstarszy bit adresu punktu końcowego, czyli bit 7, oznacza kierunek przesyłania danych. Jeśli bit ten ma wartość 0, to jest to punkt końcowy wyjściowy (ang. *out*), czyli służy on do przesyłania danych z kontrolera do urządzenia. Jeśli bit kierunku ma wartość 1, to jest to punkt końcowy wejściowy (ang. *in*), czyli służy on do przesyłania danych z urządzenia do kontrolera. Cztery najmłodsze bity adresu punktu końcowego, czyli bity 3...0, zawierają numer punktu końcowego. Pozostałe bity tego pola są zarezerwowane i powinny mieć wartość 0. Dwa najmłodsze bity, czyli bity 1...0, pola `bmAttributes` definiują rodzaj przesyłanych danych według schematu: 00 – sterujące, 01 – izochroniczne, 10 – masowe, 11 – pilne. Bity 5...2 tego pola używane są tylko dla danych izochronicznych i określają sposób synchronizacji. Dla danych innych niż izochroniczne bity te powinny mieć wartości 0. Nieco o problemie synchronizacji piszę przy okazji omawiania przykładu korzystającego z danych izochronicznych. Pozostałe bity pola `bmAttributes` są zarezerwowane i powinny mieć wartości 0.

Bity 10...0 pola `wMaxPacketSize` wyznaczają maksymalny rozmiar danych przesyłanych w jednym pakiecie. Dopuszczalne wartości zależą od rodzaju danych, szybkości transmisji i wartości bitów 12...11 tego pola. Wartości, które mogą być umieszczone w tych bitach, zostały zebrane w **tabeli 1.10**. Kreska oznacza, że dana kombinacja ustawień punktu końcowego jest niedopuszczalna. Bity 12...11 mają znaczenie tylko dla trybu HS oraz danych izochronicznych i pilnych. Bity te okre-

Tab. 1.10. Dopuszczalne wartości maksymalnego rozmiaru pola danych pakietu

Szybkość	Bit 12...11	Dane sterujące	Dane izochroniczne	Dane masowe	Dane pilne
LS	00	8	–	–	1...8
FS	00	8, 16, 32, 64	1...1023	8, 16, 32, 64	1...64
HS	00	64	1...1024	512	1...1024
HS	01	–	513...1024	–	513...1024
HS	10	–	683...1024	–	683...1024

ślają maksymalną liczbę transakcji danego typu w mikroramce: 00 – jedna transakcja, 01 – dwie transakcje, 10 – trzy transakcje, 11 – wartość zarezerwowana. Dla mniejszych szybkości lub innego rodzaju danych bity 12...11 powinny mieć wartość 00. Bity 15...13 są zarezerwowane i powinny mieć wartości 0.

Ponieważ urządzenie nie może samo zainicjować transmisji danych do kontrolera, w celu zagwarantowania, że nie zostaną przekroczone maksymalne zadeklarowane opóźnienia dostarczania danych izochronicznych i pilnych, punkty końcowe skonfigurowane dla tych rodzajów danych są odpytywane okresowo przez kontroler. Okres odpytywania punktu końcowego wyraża się w ramkach (jednostka 1 ms) dla urządzeń LS i FS lub mikroramkach (jednostka 125 μ s) dla urządzeń HS. Jeśli przyjmie się, że w polu *bInterval* zapisano wartość x , sposób wyliczenia tego okresu i dopuszczalne wartości x zestawione są w **tabeli 1.11**. Kreska oznacza, że dana kombinacja parametrów jest niedopuszczalna. Dla danych sterujących i masowych oraz urządzeń LS i FS wartość umieszczona w polu *bInterval* nie ma znaczenia i jest przez kontroler ignorowana.

Tab. 1.11. Okres odpytywania punktu końcowego

Szybkość	Dane izochroniczne	Dane pilne
LS	–	x ramek, $x = 1 \dots 255$
FS	$2^x - 1$ ramek, $x = 1 \dots 16$	x ramek, $x = 1 \dots 255$
HS	$2^x - 1$ mikroramek, $x = 1 \dots 16$	$2^x - 1$ mikroramek, $x = 1 \dots 16$

Dla danych sterujących i masowych oraz urządzenia HS wartość w polu *bInterval* określa, kiedy kontroler powinien powtórzyć transakcję OUT, gdy otrzyma z danego punktu końcowego negatywne potwierdzenie NAK. Wartość 0 w tym polu oznacza, że dany punkt końcowy nigdy nie wysyła NAK w trakcie transakcji OUT. Wartość niezerowa x oznacza, że po otrzymaniu NAK kontroler powinien powtórzyć transakcję OUT po x mikroramkach, przy czym x równe 1 oznacza następną mikroramkę, a poprawne wartości x należą do przedziału 1...255. W oczywisty sposób dotyczy to tylko punktów końcowych odbierających dane od kontrolera (ang. *out*) – w przeciwnym przypadku wartość tego pola jest nieistotna. Chodzi o to, że w każdej transakcji OUT musi być wysłany pakiet DATAx i niepotrzebne powtarzanie tych transakcji oznacza marnotrawienie pasma transmisyjnego. Natomiast transakcja IN może zostać odrzucona przez urządzenie bez przesyłania danych.

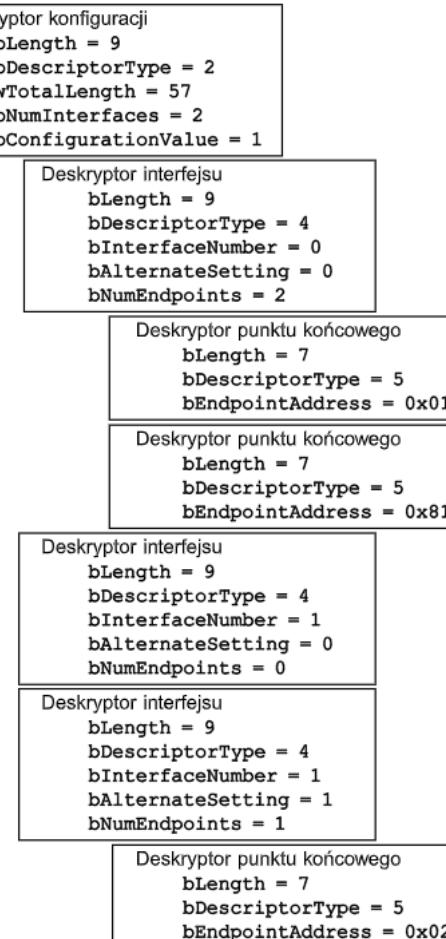
Deskryptory tekstowe umożliwiają przekazanie do kontrolera opisów urządzenia w formie czytelnej dla człowieka. Są one opcjonalne. Jeśli urządzenie nie dostarcza jakiegoś deskryptora tekstu, to w wyżej opisanych deskryptorach odpowiednie pole z numerem tego deskryptora tekstu powinno mieć wartość 0. Deskryptory tekstowe są numerowane, począwszy od 0. Deskryptor tekstu o numerze 0 zawiera listę dwubajtowych kodów języków, w których są dostępne opisy. Kody języków opisane są w dokumentacji USB. Kontroler, żądając deskryptora tekstu, podaje oprócz jego numeru również kod języka. Format deskryptora tekstu numer 0 zamieszczony jest w **tabeli 1.12**. Deskryptory o dodatnich numerach zawierają właściwe opisy. Format takiego deskryptora zamieszczony jest w **tabeli 1.13**.

Tab. 1.12. Deskryptor tekstowy nr 0

Pole	Rozmiar	Opis
bLength	1	Rozmiar deskryptora ($2N + 2$)
bDescriptorType	1	Typ deskryptora (3)
wLANGID[0]	2	Identyfikator języka 0
...
wLANGID[N-1]	2	Identyfikator języka $N - 1$

Tab. 1.13. Deskryptor tekstowy o numerze dodatnim

Pole	Rozmiar	Opis
bLength	1	Rozmiar deskryptora ($2N + 2$)
bDescriptorType	1	Typ deskryptora (3)
bString	$2N$	Opis zakodowany za pomocą Unicode

**Rys. 1.23.** Przykładowa hierarchia deskryptorów

Każdy znak tekstu zapisywany jest na dwóch bajtach w porządku cienkokońcowkowym z wykorzystaniem kodowania Unicode, co umożliwia stosowanie narodowych znaków diakrytycznych. Napisy nie są kończone terminalnym zerem. Rozmiar deskryptora musi być liczbą parzystą. Liczbę znaków wyznacza się, odejmując 2 od rozmiaru deskryptora i dzieląc tak uzyskaną wartość przez 2.

Pojedynczą konfigurację urządzenia opisuje się za pomocą hierarchii deskryptorów, zawierającej deskryptory konfiguracji, deskryptory interfejsów, deskryptory punktów końcowych oraz ewentualnie deskryptory specyficzne dla danej klasy urządzenia. Przykład konfiguracji przedstawiony jest na [rysunku 1.23](#). Pole `wTotalLength` w deskryptorze konfiguracji zawiera rozmiar w bajtach całej konfiguracji, czyli sumę wartości z pól `bLength` wszystkich deskryptorów w hierarchii. W omawianym przykładzie konfiguracja udostępnia dwa interfejsy. Interfejs numer 0 ma dwa punkty końcowe. Oba mają numer 1. Pierwszy z nich jest punktem końcowym odbierającym dane od kontrolera – ma adres 0x01. Drugi wysyła dane do kontrolera – ma adres 0x81. Interfejs numer 1 ma dwa warianty ustawień. W wariantie numer 0 interfejs ten nie ma punktów końcowych. W wariantie numer 1 ma jeden punkt końcowy o numerze 2, odbierający dane od kontrolera. Należy to interpretować tak, że wariant 0 reprezentuje interfejs w stanie wyłączonem, a wariant 1 opisuje interfejs w stanie włączonym. Wybranie odpowiedniego wariantu przez kontroler pozwala mu włączyć lub wyłączać ten interfejs. Jest dość oczywiste, że jednocześnie aktywne punkty końcowe, należące do tej samej konfiguracji, muszą mieć różne adresy.

1.6. Warstwa aplikacji dla danych sterujących

Warstwa aplikacji dla danych sterujących jest wspólna dla wszystkich zastosowań USB i dlatego opisuję ją tutaj. Dla innych rodzajów danych protokoły warstwy aplikacji są specyficzne dla konkretnych zastosowań i opisuję je przy odpowiednich przykładach.

Protokoły USB zaczynają działać po podłączeniu urządzenia do szyny. Z punktu widzenia tych protokołów stan urządzenia przed podłączeniem jest niewydefiniowany. Urządzenie może mieć wiele wewnętrznych stanów związanych z konkretną aplikacją. Jednak wyróżnia się następujące stany, które są obserwowalne z zewnątrz (ang. *visible device states*), w szczególności przez kontroler:

- szyna podłączona (ang. *attached*) – urządzenie jest podłączone do szyny, ale szyna nie jest zasilana, czyli nie ma napięcia na VBUS;
- szyna zasilana (ang. *powered*) – urządzenie jest podłączone do szyny, szyna jest zasilana (na VBUS jest prawidłowe napięcie);
- adres domyślny (ang. *default*) – urządzenie jest podłączone do szyny, szyna jest zasilana, zakończyła się procedura zerowania szyny, urządzenie ma przypisany domyślny adres o wartości 0 i może odbierać komunikaty wysypane na ten adres;
- adres przyznany (ang. *address*) – urządzenie jest podłączone do szyny, szyna jest zasilana, zakończyło się zerowanie szyny, urządzenie ma przypisany unikalny adres z przedziału 1...127, ale nie zostało jeszcze skonfigurowane;
- konfiguracja wybrana (ang. *configured*) – urządzenie jest podłączone do szyny, szyna jest zasilana, zakończyło się zerowanie szyny, urządzenie ma przypisany unikalny adres i została wybrana aktywna konfiguracja;

- urządzenie wstrzymane (ang. *suspendend*) – urządzenie jest podłączone do szyny, szyna jest zasilana, urządzenie ogranicza pobór prądu z szyny.

Wszystkie stany wraz z możliwymi przejściami między nimi przedstawione są na rysunku 1.24. Urządzenie jest podłączone do szyny, gdy podłączony jest kabel i uaktywniony jest rezystor podciągający. W naszych aplikacjach najprawdopodobniej nie będziemy rozróżniać stanów szyna podłączona i szyna zasilana, gdyż w oprogramowaniu urządzenia praktycznie nic się nie dzieje przed wykryciem zasilania na szynie. Z kolei oprogramowanie kontrolera też nie ma powodu, aby próbować wykrywać urządzenie, zanim uaktywni zasilanie szyny. Urządzenie może być zasilane z szyny (ang. *bus-powered device*) i wtedy rzeczywiście zaczyna działać, dopiero gdy pojawi się prawidłowe napięcie na VBUS. Urządzenie może mieć też własne zasilanie (ang. *self-powered device*) i wykonać pewne czynności związane z inicjowaniem interfejsu USB jeszcze przed pojawieniem się napięcia na VBUS, ale najbardziej dla nas istotne, gdyż wymagające dość rozbudowanej obsługi programowej, są stany ujęte na rysunku 1.24 w duży okrąg.



Rys. 1.24. Diagram stanowy urządzenia

W stanie *szyna zasilana* urządzenie czeka na wyzerowanie szyny przez kontroler. Jak opisałem to wcześniej, podczas zerowania szyny kontroler wykrywa szybkość, z jaką będzie pracowało urządzenie. Po zakończeniu zerowania urządzenie ma przypisany adres 0 i czeka na przyznanie przez kontroler unikalnego adresu w stanie *adres domyślny*. Po przydzieleniu adresu urządzenie przechodzi do stanu *adres przyznany*, w którym czeka na uaktywnienie konfiguracji. Po uaktywnieniu konfiguracji urządzenie przechodzi do stanu *konfiguracja wybrana* i może rozpoczęć obsługiwanie właściwego protokołu aplikacji. Bieżąca konfiguracja może być zmieniona na inną lub zostać zdezaktywowana – urządzenie wraca do stanu *adres przyznany*, w którym czeka na ponowne aktywowanie tej samej lub innej konfiguracji. Rozpoczęcie przez kontroler procedury zerowania szyny, gdy urządzenie jest w jednym ze stanów ujętych w duży okrąg, powoduje powrót do stanu *adres domyślny* – urządzenie musi dezaktywować bieżącą konfigurację i przestać używać przyznanego adresu. Przerwa w zasilaniu szyny wymusza powrót do stanu *szyna zasilana*. Jeżeli szyna przez 3 ms jest nieaktywna (urządzenie LS nie odbierze trzech kolejnych pakietów podtrzymujących, a urządzenie FS lub HS trzech kolejnych pakietów SOF rozpoczynających ramkę), to urządzenie przechodzi do stanu *wstrzymania*, w którym musi ograniczyć prąd pobierany z szyny. Powrót do poprzedniego stanu następuje, gdy urządzenie wykryje aktywność na szynie (pakietu podtrzymującego lub SOF). Więcej o zasilaniu urządzeń USB i trybach oszczędzania energii zanuduje się w rozdziale 5.

Gdy urządzenie znajdzie się w stanie *adres domyślny*, kontroler może przystąpić do jego konfigurowania.

W tym celu kontroler wysyła do urządzenia żądania przedstawione na rysunku 1.12. W fazie ustanowienia każdego żądania w pakiecie DATA0 transakcji SETUP(0) przesyłana jest 8-bajtowa struktura setup, której format przedstawiony jest w **tabell 1.14**. Pole *bmRequestType* zawiera ogólną charakterystykę żądania:

- bit 7 – bit kierunku, źródło pochodzenia danych:
 - 0 – kontroler chce wysłać dane do urządzenia lub nie ma żadnych dodatkowych danych do przesłania;
 - 1 – kontroler chce odczytać dane z urządzenia;
- bity 6...5 – typ:
 - 0 – żądanie standardowe;
 - 1 – żądanie specyficzne dla klasy urządzenia;
 - 2 – żądanie specyficzne dla dostawcy (ang. *vendor*);
 - 3 – wartość zarezerwowana;
- bity 4...0 – odbiorca:
 - 0 – urządzenie;
 - 1 – interfejs;
 - 2 – punkt końcowy;
 - 3 – inny;
 - 4...31 – wartości zarezerwowane.

Tab. 1.14. Struktura setup

Pole	Rozmiar	Opis
bmRequestType	1	Ogólna charakterystyka żądania
bRequest	1	Dokładne określenie typu żądania
wValue	2	Parametr żądania, znaczenie zależne od typu żądania
wIndex	2	Parametr żądania, znaczenie zależne od typu żądania
wLength	2	Maksymalna liczba bajtów przesyłanych w fazie danych

Jeśli żądanie kierowane jest do interfejsu, to pole wIndex zawiera numer tego interfejsu. Żądanie kierowane do punktu końcowego ma w tym polu adres punktu końcowego. Pole bRequest specyfikuje dokładnie typ żądania. Pole wLength określa ilość danych, które mają być przesłane w ramach tego żądania. Gdy realizacja żądania nie wymaga przesyłania danych, to pole wLength ma wartość 0. Kontroler powinien ustawić wtedy bit kierunku pola bmRequestType na wartość 0. Gdy urządzenie odbierze żądanie, w którym pole wLength ma wartość 0, może zignorować bit kierunku. Pozostałe pola zawierają parametry, których interpretacja zależy od typu żądania.

Standardowe typy żądań, przeznaczone dla wszystkich klas urządzeń USB, zbrane są w tabeli 1.15. Żądanie te tworzą coś w rodzaju rdzenia (ang. *core*) protokołów aplikacyjnych USB. Inne żądania, specyficzne dla konkretnych klasy urządzeń, omawiam przy okazji prezentowania przykładów, które korzystają z tych klas. Pierwsza kolumna tabeli 1.15 zawiera dopuszczalne wartości pola bmRequestType, zapisane jako liczby dwójkowe. Druga kolumna zawiera nazwy stałych używanych w oprogramowaniu dla poszczególnych typów żądania oraz odpowiadające im wartości dziesiętne. Kolejne trzy kolumny zawierają skrótowe opisy przeznaczenia parametrów żądania. Ostatnia kolumna opisuje, co jest ewentualnie przesyłane w fazie danych żądania. W dokumentacji USB bardzo szczegółowo opisano, które żądania powinny być obsługiwane przez urządzenie w poszczególnych stanach oraz jak urządzenie powinno reagować na błędne żądanie. Wiedza ta jest niezbędna, aby poprawnie zaimplementować wspomniany wyżej wspólny rdzeń sterownika urządzenia lub kontrolera. Ponieważ w dalszej części książki przedstawię gotową bibliotekę obsługującą standardowe żądania zarówno po stronie urządzenia, jak i po stronie kontrolera, tutaj poprzestanę tylko na skrótownym omówieniu przeznaczenia poszczególnych żądań.

Gdy urządzenie znajdzie się w stanie *adres domyślny*, kontroler rozpoczyna procedurę konfigurowania tego urządzenia (ang. *enumeration*). Typowa komunikacja między kontrolerem a urządzeniem rozpoczyna się od odczytania początkowych 8 bajtów deskryptora urządzenia za pomocą żądania GET_DESCRIPTOR. W ten sposób kontroler dowiaduje się, jaka jest wartość bMaxPacketSize0, czyli maksymalny rozmiar porcji danych sterujących dla danego urządzenia. Poznanie tego rozmiaru jest niezbędne do prawidłowej realizacji kolejnych żądań. Za pomocą następnego żądania GET_DESCRIPTOR kontroler odczytuje cały deskryptor urządzenia i dowiaduje się m.in., ile konfiguracji ma urządzenie. Dotyczasowa komunikacja używa domyslnego adresu urządzenia, czyli adresu 0. Kontroler inicjuje żądanie SET_ADDRESS, aby nadać urządzeniu docelowy adres, który przekazywany jest w parametrze wVa-

Tab. 1.15. Standardowe żądania

bmRequestType	bRequest	wValue	wIndex	wLength	Dane
00000000	SET_ADDRESS 5	Adres urządzenia	0	0	Brak
10000000	GET_DESCRIPTOR 6	Typ i numer deskryptora	0 Identyfikator języka	Rozmiar deskryptora	Zawartość deskryptora
00000000	SET_DESCRIPTOR 7	Typ i numer deskryptora	0 Identyfikator języka	Rozmiar deskryptora	Zawartość deskryptora
10000000	GET_CONFIGURATION 8	0	0	1	Nr aktywnej konfiguracji
00000000	SET_CONFIGURATION 9	Numer konfiguracji	0	0	Brak
10000001	GET_INTERFACE 10	0	Numer interfejsu	1	Wariant ustawień
00000001	SET_INTERFACE 11	Wariant ustawień	Numer interfejsu	0	Brak
10000000 10000001 10000010	GET_STATUS 0	0	0 Numer interfejsu Adres punktu końcowego	2	Status
00000000 00000001 00000010	CLEAR_FEATURE 1	Wyłączane ustawienie	0 Numer interfejsu Adres punktu końcowego	0	Brak
00000000 00000001 00000010	SET_FEATURE 3	Włączane ustawienie	0 Numer interfejsu Adres punktu końcowego	0	Brak
10000010	SYNCH_FRAME 12	0	Adres punktu końcowego	2	Numer ramki

lue tego żądania. Urządzenie ustawia ten adres dopiero po pomyślnym zakończeniu fazy statusu, gdyż żądanie musi być dokończone z tym samym adresem, z którym zostało ustanowione. Po pomyślnym zakończeniu żądania SET_ADDRESS urządzenie przechodzi do stanu *adres przyznany*. Wszystkie następne żądania są już wysyłane na przydzielony adres.

Gdy urządzenie znajdzie się w stanie *adres przyznany*, kontroler czyta deskryptory konfiguracji, z którego dowiaduje się m.in., ile bajtów ma cała hierarchia deskryptorów związana z tą konfiguracją. Kontroler nigdy nie czyta samych deskryptorów interfejsów i punktów końcowych. Po przeczytaniu deskryptora konfiguracji ponownie to żądanie, ale jako rozmiar wLength podaje wartość z pola wTotalLength tego deskryptora, aby przeczytać całą hierarchię deskryptorów interfejsów i punktów końcowych powiązanych z danym deskryptorem konfiguracji. Ten krok jest

powtarzany dla każdej konfiguracji. Następny krok jest opcjonalny. Kontroler czyta deskryptory tekstowe, których numery znajdują się w przeczytanych dotychczas deskryptorach. Parametr wValue żądania GET_DESCRIPTOR zawiera w starszym bajcie typ deskryptora, a w młodszym bajcie jego numer. Dla deskryptora urządzenia numer deskryptora ma zawsze wartość 0. Deskryptory konfiguracji są numerowane od 0 do wartości o jeden mniejszej niż umieszczona w polu bNumConfigurations deskryptora urządzenia. Przy żądaniu deskryptora tekstopowego pole wIndex zawiera identyfikator języka, a dla pozostałych deskryptorów ma wartość 0.

Kontroler aktywuje wybraną konfigurację za pomocą żądania SET_CONFIGURATION. W parametrze wValue przekazuje numer aktywowanej konfiguracji podany w polu bConfigurationValue deskryptora tej konfiguracji. Po aktywowaniu konfiguracji urządzenie przechodzi do stanu *konfiguracja wybrana*, co kończy procedurę *enumeration*, a urządzenie może rozpoczęć obsługę swojej właściwej funkcji. Jeśli parametr wValue żądania SET_CONFIGURATION ma wartość 0, to oznacza, że bieżąca konfiguracja powinna zostać dezaktywowana, a urządzenie powinno powrócić do stanu *adres przyznaný*.

Oprócz wyżej opisanych kontroler dysponuje jeszcze następującymi żądaniami. Za pomocą żądania SET_DESCRIPTOR można uaktualnić zawartość istniejącego deskryptora w urządzeniu. Jest to żądanie opcjonalne, nie musi być przez urządzenie obsługiwane i w praktyce nie korzysta się z niego. Za pomocą żądania GET_CONFIGURATION kontroler może odczytać wartość pola bConfigurationValue aktualnie aktywnej konfiguracji. Urządzenie zwraca wartość 0, gdy żadna konfiguracja nie jest aktywna. Żądanie GET_INTERFACE zwraca numer aktualnie wybranego wariantu ustawień interfejsu. Za pomocą żądania SET_INTERFACE kontroler wybiera wariant ustawień interfejsu, przekazując go w parametrze wValue. Parametr wIndex obu żądań zawiera numer interfejsu w ramach aktualnie wybranej konfiguracji urządzenia.

Żądanie GET_STATUS może być skierowane do urządzenia jako całości, do interfejsu lub do punktu końcowego. Jeśli jest skierowane do urządzenia, to parametr wValue zawiera 0 i zwracany jest status urządzenia. Jeśli jest skierowane do interfejsu, to parametr ten zawiera numer interfejsu i zwracany jest status tego interfejsu. Jeśli jest skierowane do punktu końcowego, to zawiera adres tego punktu końcowego i zwracany jest jego status. Status jest wartością dwubajtową, czyli 16-bitową. Status urządzenia ma ustawiony bit 0, gdy urządzenie w danym momencie nie pobiera prądu z szyny VBUS i jest zasilane autonomiczne. Gdy urządzenie jest całkowicie lub częściowo zasilane z szyny VBUS (pobiera z niej prąd), bit 0 statusu urządzenia musi być wyzerowany. Bit 1 statusu urządzenia informuje, czy aktywna jest funkcja zdalnego budzenia kontrolera (ang. *remote wakeup*). Pozostałe bity statusu urządzenia są zarezerwowane do przyszłego wykorzystania i powinny mieć wartość 0. Status interfejsu ma zawsze wszystkie bity wyzerowane. Status punktu końcowego ma w aktualnej specyfikacji zdefiniowany tylko bit 0, który informuje, czy ten punkt końcowy jest wstrzymany. Pozostałe bity statusu punktu końcowego są zarezerwowane i powinny mieć wartość 0.

Żądania CLEAR_FEATURE i SET_FEATURE służą odpowiednio do włączania i wyłączenia pewnych ustawień. Mogą być skierowane do urządzenia jako całości, do pojedynczego interfejsu lub punktu końcowego. Modyfikowane ustawienie przeka-

zywane jest w parametrze `wValue`. Opisane w standardzie ustawienia przedstawione są w **tabeli 1.16**. Jedyne zdefiniowane żądanie skierowane do punktu końcowego umożliwia jego wstrzymanie (ang. *endpoint halt*) lub ponowne aktywowanie. Jak dotąd nie przewidziano ustawień, które można byłoby zmieniać w interfejsie. Dla urządzenia jako całości można włączać lub wyłączać funkcję zdalnego budzenia kontrolera. Dla żądań skierowanych do punktu końcowego parametr `wIndex` zawiera adres tego punktu końcowego. Przy włączaniu lub wyłączaniu zdalnego budzenia kontrolera parametr `wIndex` musi mieć wartość 0. Tryb testowy (ang. *test mode*) może być tylko włączony za pomocą żądania `SET_FEATURE` i wtedy starszy bajt parametru `wIndex` określa test, który ma zostać uruchomiony, a młodszy bajt ma wartość 0.

Tab. 1.16. Konfigurowalne ustawienia

Odbiorca	wValue	Opis
Punkt końcowy	0	Wstrzymanie, aktywowanie punktu końcowego
Urządzenie	1	Włączenie, wyłączenie zdalnego budzenie kontrolera
Urządzenie	2	Włączenie trybu testowego

Żądanie `SYNCH_FRAME` może być wysłane tylko do punktu obsługującego przesyłanie izochroniczne i wykorzystującego synchronizację za pomocą wzorca. Zwracana wartość zawiera numer ramki, w której urządzenie chce rozpoczęć przesyłanie wzorca synchronizacyjnego.

1.7. Wiadomości uzupełniające

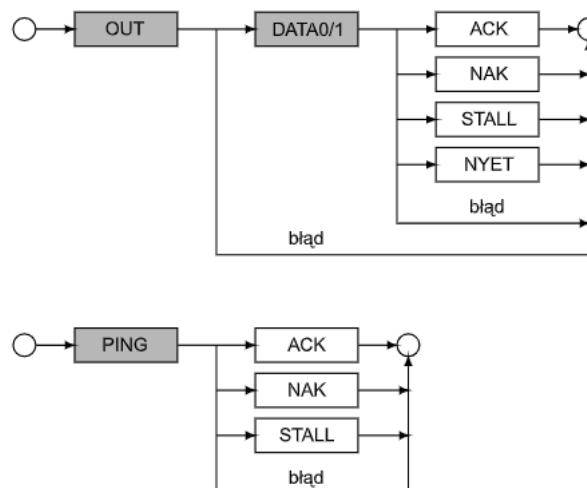
W tym podrozdziale zamieszczam skrótnie pewne dodatkowe wiadomości, które nie są niezbędne do zrozumienia przykładów zamieszczonych w dalszej części książki, ale uważam, że należy, choćby pośrodku, zapoznać się z nimi. Czytelnika pragnącego poznać więcej szczegółów odsyłam do specyfikacji USB [22]. Być może, zanim zacznie się studiować specyfikację, warto też zatrzymać się na stronie internetowej podręcznika *USB in a Nutshell, Making sense of the USB standard* [33], gdzie w bardzo przystępny sposób opisano zawartość standardu.

Jak to już zostało napisane wyżej, urządzenie OTG może być wyposażone w gniazdo Micro-AB, do którego można podłączyć wtyk Micro-A lub Micro-B. Jeśli podłączony zostanie wtyk Micro-A, to takie urządzenie jest nazywane urządzeniem typu A (ang. *A-device*) – pełni wtedy domyślnie funkcję kontrolera i odpowiada za zasilanie szyny. W przeciwnym przypadku (podłączono wtyk Micro-B lub nie podłączono żadnego kabla) jest nazywane urządzeniem typu B (ang. *B-device*) i domyślnie zachowuje się wtedy jak zwykłe urządzenie USB. W przypadku dwóch urządzeń OTG połączonych ze sobą bezpośrednio bez pośrednictwa koncentratora, jeśli aplikacja używająca urządzenia typu B wymaga przełączenia do roli kontrolera, możliwa jest zamiana ról za pomocą protokołu negocjowania kontrolera HNP (ang. *Host Negotiation Protocol*).

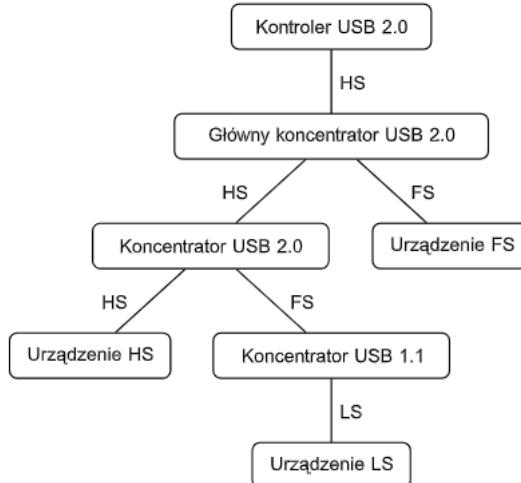
Gdy podczas transakcji OUT urządzenie FS nie może odebrać kolejnej porcji danych masowych lub sterujących, bo na przykład nie przetworzyło jeszcze poprzednio ode-

branych danych, odpowiada za pomocą pakietu NAK, ale dzieje się to dopiero po przesłaniu całego pakietu DATA0 lub DATA1. Cała transakcja musi być powtórzona, co oznacza spore marnotrawstwo przepływności łącza. Dla urządzeń HS przewidziano dodatkową możliwość kontroli przepływu danych sterujących lub masowych, przedstawioną na **rysunku 1.25**. Urządzenie może zakończyć transakcję OUT pakietem potwierdzającym NYET. Oznacza to, że transakcja zakończyła się poprawne, ale w buforze odbiorczym jest za mało miejsca, aby odebrać kolejny pakiet o maksymalnym rozmiarze. Zakończenie transakcji pakietem ACK oznacza w tym przypadku, że w buforze odbiorczym jest dostatecznie dużo miejsca, aby poprawnie odebrać dane z następnej transakcji OUT. Jeśli urządzenie odpowiedziało pakietem NYET, to kontroler może użyć dodatkowej transakcji PING, aby sprawdzić, czy pojawiło się miejsce w buforze odbiorczym. Odpowiedź ACK oznacza, że w buforze odbiorczym jest dostatecznie dużo miejsca i można zainicjować kolejną transakcję OUT. Odpowiedź NAK oznacza, że w buforze odbiorczym nadal nie ma miejsca. W każdym przypadku odpowiedź STALL oznacza, że punkt końcowy jest nieaktywny.

USB jest szyną i w danej chwili tylko jeden nadajnik może sterować liniami danych. Pozostałe nadajniki muszą być w stanie wysokiej impedancji. Kolejność włączania nadajników wynika z obserwacji przebiegu protokołu na szynie – wszystkie podłączone do szyny odbiorniki są aktywne. Taki scenariusz nie sprawia problemu, jeśli wszystkie podłączone do szyny urządzenia pracują z tą samą szybkością. Problem pojawia się, gdy chcemy podłączyć do wspólnej szyny urządzenia o różnych szybkościach. Należy się spodziewać, że urządzenie nie będzie w stanie śledzić transmisji odbywającej się z większą szybkością niż dla niego przewidziana. Aby temu zaradzić, koncentratory dzielą szynę na segmenty o różnych szybkościach. Segmentem staje się całe poddrzewo szyny lub jego fragment podłączony do pewnego portu koncentratora. Przykład podziału szyny na segmenty zamieszczono na **rysunku 1.26**.



Rys. 1.25. Dodatkowe transakcje HS dla danych sterujących i masowych



Rys. 1.26. Przykład podziału szyny USB na segmenty o różnych szybkościach

Jeśli do jednego z portów koncentratora FS przyłączone jest urządzenie LS, to nadajnik tego portu musi być blokowany podczas transakcji FS, aby zmniejszyć zakłócenia elektromagnetyczne i zapobiec błędnej interpretacji danych FS przez urządzenie LS. Jeśli kontroler chce wysłać pakiet do urządzenia LS, musi poprzedzić go żetonem PRE, po którym zamiast sekwencji EOP zaczyna się natychmiast sekwencja SYNC właściwego pakietu. Koncentrator wycina żeton PRE, włączając jednocześnie nadajnik portu. Nadajnik jest wyłączany po wysłaniu sekwencji EOP właściwego pakietu.

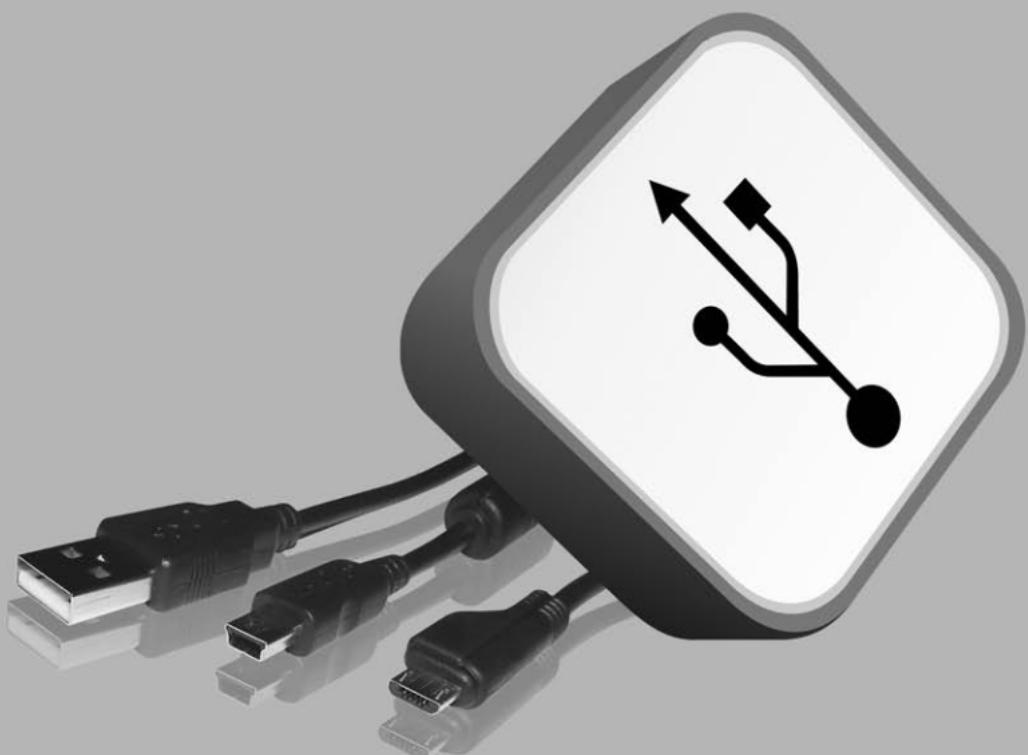
Jeśli do koncentratora HS podłączone jest urządzenie FS lub LS, to aby cała szyna nie musiała pracować w trybie FS lub LS, kontroler separuje segment szyny z takim urządzeniem. Aby transakcje FS i LS nie blokowały na zbyt długi czas pozostałej części szyny pracującej w trybie HS, kontroler HS dzieli transakcję FS lub LS w celu przesłania jej przez segment HS. Transakcja dzielona (ang. *split transaction*) składa się z dwóch części: SSPLIT – rozpoczęcie transakcji dzielonej (ang. *start split*) oraz CSPLIT – zakończenie transakcji dzielonej (ang. *complete split*). Pomiędzy nimi mogą być wykonywane inne transakcje HS. Jeśli trzeba podzielić transakcję OUT, to transakcja SSPLIT przesyła żeton OUT i pakiet DATAx, a transakcja CSPLIT powtarza ten sam żeton OUT i umożliwia odebranie pakietu potwierdzającego. Gdy dzielona jest transakcja IN, to transakcja SSPLIT przesyła żeton IN, a transakcja CSPLIT powtarza ten sam żeton IN i umożliwia odebranie pakietu DATAx, NAK lub STALL. Ewentualny pakiet ACK do urządzenia jest generowany przez koncentrator i nie jest przesyłany w segmencie HS. Pole ADRES żetona SPLIT ma 19 bitów i zawiera adres koncentratora, bit determinujący rodzaj transakcji (SSPLIT czy CSPLIT), numer portu koncentratora, pole określające szybkość (LS czy FS) i rodzaj przesyłanych danych (sterujące, izochroniczne, masowe, pilne).

Intencjonalnie nie zamieściłem w tym rozdziale szczegółowego opisu działania koncentratora. Nie jest to potrzebne do zrozumienia dalszej części książki. Programowanie urządzeń USB nie wymaga poznania protokołu stosowanego do

komunikacji z koncentratorem. Bezpośrednia interakcja zachodzi między oprogramowaniem kontrolera a oprogramowaniem urządzenia. Z perspektywy oprogramowania urządzenia koncentrator zachowuje się przezrocznie. Liczba koncentratorów pośredniczących między kontrolerem a urządzeniem nie wpływa na oprogramowanie urządzenia. Natomiast z punktu widzenia kontrolera koncentrator jest urządzeniem pośredniczącym w komunikacji z innymi urządzeniami, co wymaga wyposażenia kontrolera w odpowiednie oprogramowanie sterujące pracą koncentratora. W prezentowanych w końcowych dwóch rozdziałach książki przykładach kontrolerów USB urządzenia podłączane są bezpośrednio do kontrolera bez pośrednictwa koncentratora.

2

Podstawy



Prezentowane w tej książce przykładowe programy napisane są na mikrokontrolery STM32 firmy STMicroelectronics. Okazuje się jednak, że nawet wśród tej rodzinny układów występuje spora różnorodność peryferii. W poszczególnych modelach znajdujemy trzy warianty układów USB, dwa warianty portów wejścia-wyjścia GPIO (ang. *General Purpose Input-Output*) oraz kilka wariantów konfigurowania sygnałów zegarowych i pętli fazowych PLL (ang. *Phase Locked Loop*). Jednak same warianty peryferii to nie wszystko. Należy też wziąć pod uwagę otoczenie mikrokontrolera, czyli rozmieszczenie i przeznaczenie jego wyprowadzeń. Rozwijając oprogramowanie, korzysta się zwykle z przeróżnych modułów, płytka prototypowych i zestawów uruchomieniowych. Istotnie zwiększa to liczbę kombinacji parametrów, które trzeba uwzględnić w oprogramowaniu, aby było ono możliwie łatwo przenośne pomiędzy dostępnymi wariantami sprzętu. Ten rozdział poświęcam właśnie głównie omówieniu, jak poprawnie i skutecznie pisać oprogramowanie dostosowane do wielu konfiguracji sprzętu. Zaczynam jednak od krótkiego przedstawienia układów USB dostępnych w rodzinie STM32. Następnie opisuję strukturę archiwum z przykładami. W kolejnym podrozdziale przedstawiam zestaw pomocniczych funkcji intensywnie wykorzystywanych w prezentowanych programach. Pokazany tu materiał ilustruję przykładowym projektem, który demonstruje konfigurowanie wariantu sprzętu, inicjowanie generatora kwarcowego i układów dystrybuujących sygnały zegarowe w mikrokontrolerze oraz obsługę błędów. Ponadto przykład ten umożliwia szybkie przetestowanie zestawu narzędzi programistycznych (ang. *tool-chain*) i sprzętu (adapter JTAG, programator, płytka). Na zakończenie omawiam, jak kompilować przykłady i przedstawiam kilka programów przydatnych przy ich uruchamianiu w systemach Linux i Windows.

2.1. Warianty sprzętu

Przez wariant sprzętu rozumiem dwie rzeczy: zastosowany model mikrokontrolera oraz jego otoczenie, czyli schemat płytka. Opisuję te dwa zagadnienia kolejno, przedstawiając warianty sprzętu, na których testowałem przykładowe programy.

2.1.1. Mikrokontrolery STM32

Mikrokontrolery STM32 wyposażają się w trzy rodzaje układów peryferyjnych realizujących interfejs USB. Ich podstawowe własności są przedstawione w tabeli 2.1. Układ, który w dalszej części książki oznaczam skrótnie DEV-FS, może pracować tylko jako urządzenie FS. Układ, który oznaczam OTG-FS, może pracować jako urządzenie FS lub kontroler obsługujący urządzenia LS i FS. Układ, który oznaczam OTG-HS, może pracować jako urządzenie FS lub HS albo jako kontroler obsługujący urządzenia LS, FS i HS. Jak wskazują nazwy, układy OTG-FS i OTG-HS mogą też pracować jako urządzenie OTG. Modele mikrokontrolerów z rodzin STM32 wyposażane w interfejs USB przedstawione są w tabeli 2.2 (stan w chwili pisania tej książki).

Przykładowe programy pisałem i testowałem na układach STM32F103, STM32F107, STM32F207, STM32F407 i STM32L152. Powinny jednak dać się bardzo łatwo przenieść na pozostałe modele STM32. Programy napisane na STM32F103 daje się bardzo łatwo dostosować do układów z serii STM32F102, gdyż różnią się one od

Tab. 2.1. Stosowane w rodzinie STM32 układy peryferyjne USB

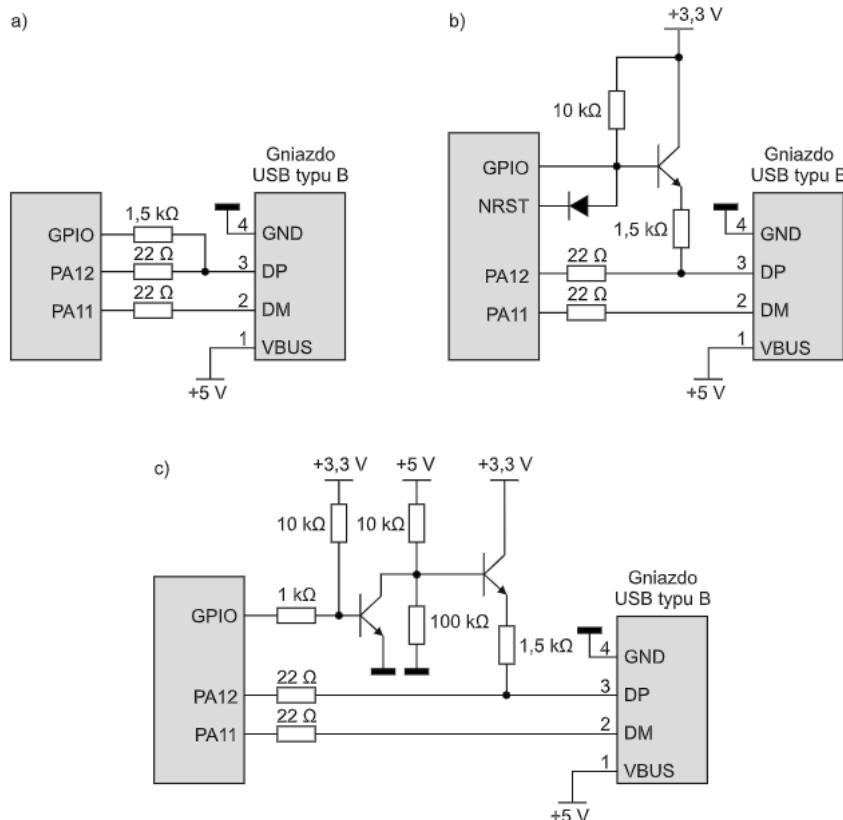
Skrótowa nazwa	Urządzenie	Kontroler	OTG
DEV-FS	FS	Brak	Nie
OTG-FS	FS	LS, FS	Tak
OTG-HS	FS, HS	LS, FS, HS	Tak

Tab. 2.2. Modele mikrokontrolerów STM32 wyposażane w interfejs USB

STM32F102 STM32F103 STM32F302 STM32F303 STM32F372 STM32F373	STM32F105 STM32F107	STM32F205 STM32F215 STM32F405 STM32F415	STM32F207 STM32F217 STM32F407 STM32F417	STM32L151 STM32L152 STM32L162
DEV-FS				DEV-FS
	OTG-FS		OTG-FS	
		OTG-HS	OTG-HS	

STM32F103 głównie mniejszą maksymalną częstotliwością taktowania. Programy napisane na STM32F107 można bez modyfikacji przenieść na STM32F105, gdyż jest to model STM32F107 pozbawiony obsługi Ethernetu. Programy napisane na STM32F207 daje się w ograniczonym zakresie przenieść na STM32F205, gdyż ten drugi ma tylko jeden interfejs USB. Układy STM32F215 i STM32F217 to odpowiednio układy STM32F205 i STM32F207, do których dodano sprzętowy akcelerator kryptograficzny, więc wszystkie przykładowe programy przenoszą się bez modyfikacji. Programy napisane na STM32F207 daje się też przenieść na układy STM32F4xx, gdyż układy te mają takie same peryferie jak odpowiednie układy STM32F2xx, ale zastąpiono w nich rdzeń Cortex-M3 rdzeniem Cortex-M4. Układy z serii STM32L151 różnią się od odpowiednich układów STM32L152 tylko brakiem obsługi LCD, a seria STM32L162 rozszerza odpowiednie układy STM32L152 o sprzętowy akcelerator kryptograficzny, więc przykładowe programy również przenoszą się zupełnie bezproblemowo.

Przykładowe programy są napisane na tyle ogólnie, że nie powinno też sprawić poważnej trudności ich przeniesienie na modele mikrokontrolerów, które pojawią się w niedalekiej przyszłości, w tym nowe modele z serii STM32F0xx z rdzeniem Cortex-M0 oraz zapowiadane w chwili pisania tej książki kolejne mikrokontrolery z rdzeniem Cortex-M4, czyli nowa linia STM32F3xx i następni przedstawiciele serii STM32F4xx. Niektóre z tych nowych układów zostaną wyposażone w interfejs USB i wszystko wskazuje na to, że będzie to jeden z trzech opisanych tu układów. Nadajnik-odbiornik (ang. *transceiver*) układu DEV-FS jest scalony w strukturze mikrokontrolera i wymaga dołączenia niewielu elementów zewnętrznych. Spedykane schematy podłączeń gniazda USB do układu DEV-FS pokazano na **rysunku 2.1**. Osobiście preferuję schemat najprostszy, czyli wariant (a). Linie danych DM i DP są podłączone odpowiednio do wyprowadzeń PA11 i PA12 mikrokontrolera. Aby zapewnić optymalne dopasowanie mocy, rezystancja wyjściowa nadajnika dla każdej z linii danych musi być równa połowie różnicowej impedancji falowej kabla, czyli powinna wynosić 45Ω . W technologii CMOS trudno wykonać rezystory,



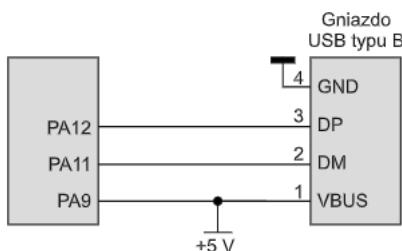
Rys. 2.1. Schematy podłączenia DEV-FS

a zwłaszcza rezystory o dokładnej wartości. Dlatego standard dopuszcza, aby w tych układach CMOS nieobsługujących trybu HS rezystancja wyjściowa nadajnika dla każdej z linii danych mieściła się w zakresie 28...44 Ω. Często jest ona mniejsza i stąd wynika, że niektóre układy wymagająłączenia w liniach danych szeregowych rezystorów o wartościach z przedziału 22...27 Ω. Można też podejrzewać, że obecność tych rezystorów pomaga zapewnić odporność wyjść nadajnika na zwarcia. Ponieważ dla trybu FS dopasowanie impedancji nie jest bardzo krytyczne, w praktyce pominięcie rezystorów szeregowych nie pogarsza jakości transmisji.

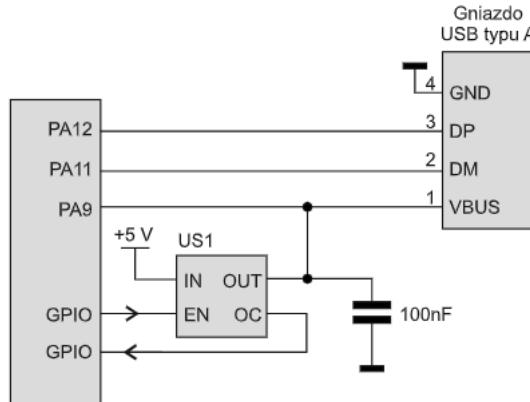
Układ DEV-FS nie zawiera rezystora podciągającego 1,5 kΩ i musi on być podłączony na zewnątrz. Trzeba na to przeznaczyć dodatkowe wyprowadzenie GPIO mikrokontrolera. W najprostszym przypadku, czyli w wariantie (a) z rysunku 2.1, rezistor podciągający podłącza się bezpośrednio między wyprowadzenie GPIO a linią danych DP. Jeśli na wyprowadzeniu GPIO jest poziom wysoki, to rezistor podciąga linię DP do napięcia zasilającego mikrokontroler. Jeśli wyprowadzenie GPIO jest w stanie wysokiej impedancji, to rezistor podciągający pozostaje odłączony. W modelach STM32L1xx chciiano wyeliminować konieczność stosowania dodatkowego zewnętrznego rezystora i scalono rezistor podciągający w strukturze mikrokontrolera, ale niestety w niektórych seriach układów ma on wartość z przedziału

810...950 Ω , co jest niezgodne ze standardem USB i jako rozwiązań obejściowe zaleca się nadal stosowanie zewnętrznego rezystora podciągającego [17]. Wariant (b) z rysunku 2.1 prezentuje inny sposób sterowania rezystorem podciągającym za pomocą dodatkowego tranzystora. Linia danych jest podciągana do zasilania, gdy na wyprowadzeniu GPIO jest poziom wysoki, a rezystor podciągający jest odłączony, gdy na tym wyprowadzeniu jest poziom niski. Dodatkowo, aby zapewnić odłączanie rezystora podciągającego w czasie zerowania mikrokontrolera, baza tranzystora jest ściągana za pomocą diody do poziomu niskiego przez wyprowadzenie NRST służące do zerowania mikrokontrolera. W STM32 wyprowadzenie zerujące jest w stanie niskim również wtedy, gdy mikrokontroler jest zerowany programowo, co pozwala na wyzerowanie układów zewnętrznych. Wariant (c) jest jeszcze bardziej skomplikowany. Zasadnicza różnica funkcjonalna w stosunku do wariantu (b) polega na zamianie poziomów sterujących podłączeniem i odłączeniem rezystora podciągającego. W wariantie (c) rezystor podciągający jest podłączony, gdy na wyprowadzeniu GPIO jest poziom niski, a odłączony, gdy na tym wyprowadzeniu jest poziom wysoki. Należy zaznaczyć, że w wariantie (c) występuje problem ze spełnieniem wymagań standardu w przypadku urządzenia zasilanego z szyny. Gdy urządzenie jest wstrzymane, rezystor podciągający musi pozostawać podłączony, a jednocześnie należy ograniczyć pobór prądu z VBUS. Wymaga to uśpienia mikrokontrolera i jednoczesnego utrzymywania na wyprowadzeniu GPIO poziomu niskiego, co powoduje przepływ z zasilania 3,3 V prądu 0,2 mA przez rezystor podciągający i ok. 0,3 mA przez łączną rezystancję 11 k Ω do wyprowadzenia GPIO oraz niewielkiego prądu ok. 40 μ A z zasilania 5 V do masy przez rezystancję 110 k Ω w obwodzie bazy tranzystora. Sumaryczny prąd pobierany z VBUS najprawdopodobniej przekroczy wtedy wartość dopuszczaną przez standard – więcej o tym w rozdziale 5.

Nadajnik-odbiornik układu OTG-FS jest scalony w strukturze mikrokontrolera i w przypadku pracy jako urządzenie nie wymaga podłączania żadnych zewnętrznych elementów. Odpowiedni schemat podłączenia gniazda typu B do układu OTG-FS znajduje się na **rysunku 2.2**. Wyprowadzenie PA9 służy do monitorowania poziomu napięcia VBUS. Jeśli układ OTG-FS ma pracować jako kontroler, to należy podłączyć do niego gniazdo typu A według schematu z **rysunku 2.3**. Zadaniem układu scalonego US1 jest zasilanie linii VBUS. Na jego wejście IN podaje się napięcie +5V. Za pomocą wejścia zezwalającego EN (ang. *enable*) steruje się włączaniem i wyłączaniem tego napięcia na wyjściu OUT. Wyjście OC (ang. *overcurrent*) służy do sygnalizowania zwarcia lub przekroczenia dopuszczalnego prądu z linii



Rys. 2.2. Schemat podłączenia OTG-FS pracującego jako urządzenie



Rys. 2.3. Schemat podłączenia OTG-FS pracującego jako kontroler

VBUS. Sterowanie napięciem na VBUS wymaga przeznaczenia do tego celu dwóch dodatkowych wyprowadzeń GPIO mikrokontrolera. Jak poprzednio wejście PA9 służy do monitorowania poziomu napięcia VBUS. Podłączenie gniazda Micro-AB do układu OTG-FS wygląda analogicznie jak na rysunku 2.3. Dodatkowo podłącza się tylko styk ID gniazda USB do wyprowadzenia PA10. Szczegółowe schematy aplikacyjne można znaleźć w dokumentacji zestawów ewaluacyjnych dostarczanych przez STMicroelectronics, np. w [19].

Układ OTG-HS może współpracować z trzema różnymi nadajnikami-odbiornikami. Jeśli nie potrzebujemy korzystać z trybu HS i zadowala nas tryb FS lub LS, możemy skorzystać z nadajnika-odbiornika scalonego w strukturze mikrokontrolera. Jest to taki sam nadajnik-odbiornik, jaki zastosowano w układzie OTG-FS, przy czym linie ID, monitorowanie VBUS, DM, DP są dostępne odpowiednio na wyprowadzeniach portu B mikrokontrolera: PB12, PB13, PB14, PB15. Drugą możliwością jest podłączenie zewnętrznego nadajnika-odbiornika FS za pomocą interfejsu I²C. Jest to zdecydowanie najmniej atrakcyjna opcja. Jeśli potrzebujemy trybu HS, musimy dołączyć zewnętrzny nadajnik-odbiornik za pomocą interfejsu ULPI. Odpowiednie schematy można znaleźć np. w [4] lub [19].

2.1.2. Płytki prototypowe

Przykłady testowałem na następującym sprzęcie:

- moduł MMstm32F103Vx-0-0-0 z procesorem STM32F103VC,
- zestaw STM3220G-EVAL z procesorem STM32F207IG,
- moduł STM32F4-Discovery z procesorem STM32F407VG,
- moduł STM32L-Discovery z procesorem STM32L152RB,
- zestaw ZL29ARM z procesorem STM32F107VC,
- zestaw ZL30ARM z procesorem STM32F103CB,
- zestaw ZL31ARM z procesorem STM32F103RB,
- gadżet z procesorem STM32F103T8.

Poniżej zebrałem uwagi dotyczące stosowania powyższych modułów i zestawów przy uruchamianiu przykładowych programów korzystających z interfejsu USB. Niestety żadna z dostępnych na rynku płyt prototypowych, które wpadły mi w ręce, nie pozwala na symulowanie urządzenia USB zasilanego z szyny w sposób w pełni zgodny ze standardem. Dlatego postanowiłem dostosować moduł STM32L-Discovery oraz zestaw ZL31ARM do wymagań standardu. Stosowne zmiany nie są trudne do wykonania dla doświadczonego elektronika, ale podejmując się ich, należy mieć świadomość, że robi się to na własną odpowiedzialność.

MMstm32F103Vx-0-0-0

Gniazdo USB podłącza się według wariantu (b) z rysunku 2.1. Tranzystor i rezystor podciągający znajdują się na module. Należy połączyć bazę tranzystora (końcówkę UDP modułu) z wybranym wyjściem, za pomocą którego chcemy sterować włączaniem i wyłączaniem rezystora podciągającego. Na potrzeby prezentowanych przykładowych programów ustalmy, że będzie to wyprowadzenie PA10. Ponadto niezbędne jest połączenie wyprowadzenia BOOT0 z masą. Końcówkę VCC modułu podłącza się do zasilacza lub do szyny VBUS. Producent podaje, że moduł można zasilać napięciem z przedziału 3,8...16 V. Należy uważać, gdyż niestety nie zawsze jest to prawda. W egzemplarzach, które widziałem, wlutowano inny niż deklarowany w dokumentacji regulator napięcia, o znacznie gorszych parametrach – większym prądzie skrośnym i dopuszczalnym napięciu wejściowym z zakresu 4,5...7,5 V. Ponadto na module przewidziano miejsce dla zewnętrznej pamięci Flash, która wykorzystuje wyprowadzenia PA4, PA5, PA6 i PA7. W wersjach modułu bez tej pamięci wlutowywany jest jednak rezystor R9 podciągający wyprowadzenie PA4 do zasilania, mimo że nie jest wtedy potrzebny, co może utrudnić użycie tego wyprowadzenia w niektórych zastosowaniach. Można jednak skorzystać z obecności tego rezystora. W przykładowych programach wejście PA4 posłuży do wyboru częstotliwości taktowania rdzenia mikrokontrolera. Podciagnięcie PA4 rezystorem do zasilania powoduje, że rdzeń będzie taktowany z częstotliwością 72 MHz. Jeśli podłączy się je do masy, to będzie taktowany zegarem o częstotliwości 48 MHz. Patrz też opis pliku *boot.h* w dalszej części tego rozdziału.

STM3220G-EVAL

Zestaw należy zasilać za pomocą dostarczonego z nim zasilacza – na złączu szpilekowym JP18 powinna być założona zwora PSU. W celu uruchamiania przykładowych programów pozostałe zwory można pozostawić w domyślnych ustawieniach fabrycznych. W szczególności, aby używać LCD i zewnętrznej pamięci SRAM podłączonych do kontrolera FSMC, nie należy zakładać zwór JP3 i JP10, a zwory JP1 i JP2 należy pozostawić w pozycji 2–3 (ustawienie bliższe środka płytki). Zestaw ten ma dwa interfejsy USB. Układ OTG-FS wykorzystuje wewnętrzny nadajnik-odbiornik podłączony do gniazda CN8 oznaczonego na płytce jako USB OTG FS, a układ OTG-HS podłączony jest za pomocą interfejsu ULPI do zewnętrznego nadajnika-odbiornika ISP1705 [4], który współpracuje z gniazdem CN9 oznaczonym na płytce jako USB OTG HS. Dokładny opis tego zestawu znajduje się w [19].

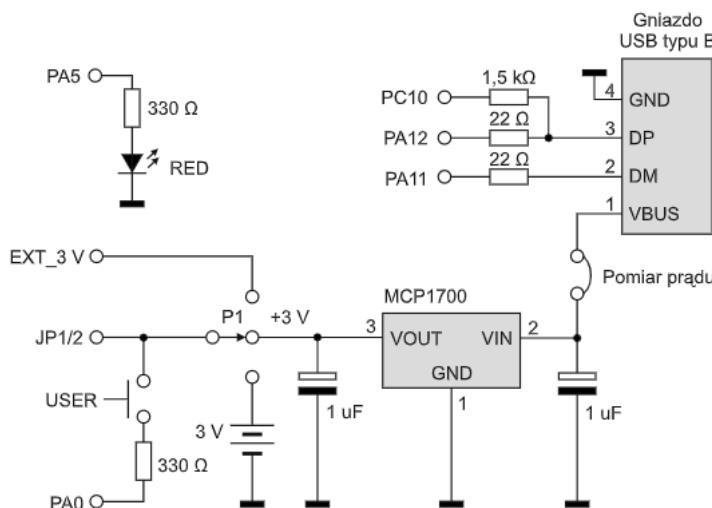
STM32F4-Discovery

Moduł można używać bez żadnych modyfikacji i dodatkowych połączeń. Jest on zasilany z gniazda USB programatora (ST-LINK/V2) i nie pozwala na testowanie zasilania urządzenia USB z szyny. Dokładny opis tego modułu znajduje się w [21].

STM32L-Discovery

Aby wygodnie korzystać z modułu, dobrze jest dobudować do niego płytę bazową z dwoma rzędami 28-końcowkowych gniazd, w które wtyka się moduł. Proponowany schemat takiej płytki zamieszczony jest na **rysunku 2.4**. Przed przystąpieniem do jej wykonania należy też dokładnie przestudiować schemat modułu [20]. Gniazdo USB podłącza się według wariantu (a) z rysunku 2.1. Aby móc symulować urządzenie zasilane z interfejsu USB, trzeba zastosować regulator LDO (ang. *low drop output*), czyli liniowy stabilizator napięcia o małym spadku napięcia między wejściem a wyjściem. Regulator ten musi ponadto mieć możliwie najmniejszy prąd skrośny (ang. *quiescent current*), czyli prąd płynący przez wyprowadzenie masy (ang. *ground current*). Dla regulatora MCP1700 prąd ten nie przekracza $4\text{ }\mu\text{A}$, gdy wyjście jest nieobciążone, a $45\text{ }\mu\text{A}$ przy pełnym obciążeniu. W prezentowanym rozwiąaniu zastosowałem regulator o napięciu wyjściowym 3 V (zamiast zwykłego stosowanego $3,3\text{ V}$), gdyż takim napięciem zasilany jest mikrokontroler w STM32L-Discovery. Dobrze jest przewidzieć możliwość rozwarcia połączenia między stykiem VBUS gniazda USB a wejściem regulatora w celu pomiaru prądu zasilania. Na płytce bazowej można też umieścić różne dodatkowe układy. W przykładowych programach przydaje się czerwona dioda świecąca (w module są dwie diody świecące: zielona i niebieska), którą podłączyłem do wyprowadzenia PA5.

W celu rozdzielenia obwodów zasilania programatora ST-LINK/V2 i właściwego mikrokontrolera należy wylutować zwoję SB21. W ten sposób programator będzie zasilany ze swojego gniazda USB, a do zasilania mikrokontrolera będzie



Rys. 2.4. Schemat płytka bazowej dla modułu STM32L-Discovery

można wybrać za pomocą przełącznika P1 jedno z trzech źródeł: programator ST-LINK/V2 (napięcie pochodzi z wyprowadzenia EXT_3V), linie VBUS lub baterię 3 V. Napięcie zasilające należy podłączyć do środkowego styku (numer 2) złącza szpilkowego JP1. Pozwala to testować prototypowany układ z odłączonym programatorem, czyli w warunkach zasilania jak w układzie docelowym. Aby wyłączonego ST-LINK/V2 nie zerował permanentnie mikrokontrolera, należy wylutować zwozę SB100, co odłącza go od wejścia NRST mikrokontrolera. W tym przypadku mikrokontroler nie zawsze będzie poprawnie zerowany przez ST-LINK/V2. Po zaprogramowaniu pamięci Flash może być potrzebne jego manualne wyzerowanie – najlepiej przez chwilowe odłączenie zasilania.

W module STM32L-Discovery przewidziano obwód mierzący prąd zasilania. Można go odłączyć, wylutowując zwory SB1, SB2 i SB14. Prąd zasilania mikrokontrolera najprościej jest zmierzyć, włączając miliamperomierz między przełącznik P1 a środkowy styk złącza JP1. Po wylutowaniu zwory SB2 uwalnia się wyprowadzenie PA4, będące wyjściem przetwornika cyfrowo-analogowego, który jest wykorzystywany w jednym z przykładowych programów. Do wyjścia PA4 można podłączyć przetwornik piezoelektryczny. W celu zlikwidowania dodatkowych pasożytniczych prąduów dobrze jest też wylutować rezystory R27 i R32. Niestety po tych zmianach przestanie działać umieszczony na płytce przycisk USER, dlatego na płycie bazowej znajduje się przycisk, który go zastępuje i jest podłączony do wyprowadzenia PA0.

Interfejs USB do poprawnej pracy wymaga taktowania sygnałem o stabilnej częstotliwości, którą można uzyskać tylko wtedy, gdy zegar mikrokontrolera jest generowany za pomocą rezonatora kwarcowego. W module można to uzyskać, taktując mikrokontroler z wyjścia MCO programatora, co wymaga zamknięcia zwory SB17. Aby móc testować układ przy wyłączonym programatorze, należy wlutować odpowiedni kwarc X3, np. 8 MHz. Można też wlutować podstawkę, w której będzie można wkładać różne rezonatory. Koniecznie trzeba też wlutować kondensatory C21 i C22 o pojemności 20 pF. Jeśli używamy podstawki, to ponieważ wprowadza ona dodatkowe pojemności montażowe, można użyć kondensatorów o mniejszych wartościach pojemności, np. 15 pF. Należy również wlutować rezistor R30 o wartości 220 Ω lub zwozę w jego miejscu. Zwory SB17, SB18 i SB20 muszą być rozłączone, gdy korzystamy z rezonatora kwarcowego.

ZL29ARM

Zestaw trzeba zasilać z zewnętrznego zasilacza – zwora JP1 PWR_SEL powinna być ustawiona w pozycji EXT. Zależnie od zastosowanego wyświetlacza ciekłokrystalicznego zwozę JP3 DISPLAY należy ustawić w pozycji CHAR (alfanumeryczny) lub GRAPH (graficzny) – patrz też dokumentacja zestawu. Przykładowe projekty były testowane z wyświetlaczem graficznym WG12864A. Aby uruchamiać programy z pamięci Flash, zwozę JP7 BOOT0 trzeba ustawić w pozycji 0. Wtedy zwora JP8 BOOT1 posłuży nam do wyboru częstotliwości taktowania rdzenia mikrokontrolera. Jeśli będzie ustalona w pozycji 0, to rdzeń będzie taktowany zegarem 48 MHz, a w pozycji 1 będzie to 72 MHz (patrz opis pliku *boot.h* w dalszej części tego rozdziału).

Zwory JP11 i JP12 przełączają nadajnik-odbiornik USB między gniazdem typu A (oznaczonym na płytce jako USB HOST i umieszczonym obok tych zwór) a gniaz-

dem typu B (umieszczonym na płytce między gniazdem zasilacza a gniazdem RS-232). W pozycji dev realizowany jest interfejs urządzenia z rysunku 2.2, a w pozycji host – interfejs kontrolera z rysunku 2.3. Przełączane są tylko linie DM (PA11) i DP (PA12). Wyprowadzenie PA9 w obu ustawieniach pozostaje podłączone do styku VBUS gniazda typu A (wyjście OUT1 układu scalonego U6). Jeśli interfejs pracuje jako urządzenie, to należałoby PA9 podłączyć do styku VBUS gniazda typu B. W większości sytuacji układ działa jednak poprawnie przy braku tego połączenia, ponieważ na niewysterowanym wejściu EN1 układu scalonego U6 jest poziom wysoki, co sprawia, że na wyjściu OUT1 tego układu, a zatem również na wejściu PA9, jest zawsze napięcie 5 V. Powoduje to jednak problem z prawidłowym rozpoznaniem zdarzenia polegającego na odłączeniu kabla USB, zwłaszcza gdy mikrokontroler powinien w takiej sytuacji obniżyć pobór prądu i przejść z zasilania z szyny USB na podtrzymujące jego pracę zasilanie autonomiczne.

ZL30ARM

Zwory i ich ustawienia są podobne jak w ZL29ARM. Zestaw należy zasilać z zewnętrznego zasilacza – zwora JP3 PWR_SEL powinna być ustawiona w pozycji EXT. Zależnie od zastosowanego wyświetlacza ciekłokrystalicznego zworę JP4 DISPLAY należy ustawić w pozycji CHAR lub GRAPH. Przykładowe projekty były testowane z wyświetlaczem alfanumerycznym o 2 wierszach po 16 znaków wyposażonym w sterownik kompatybilny z układem HD44780. Aby uruchamiać programy z pamięci Flash, zworę BOOT0 trzeba ustawić w pozycji 0. Wtedy zwora BOOT1 posłuży nam do wyboru częstotliwości taktowania rdzenia mikrokontrolera. Jeśli będzie ustawiona w pozycji 0, to rdzeń będzie taktowany zegarem 48 MHz, a w pozycji 1 będzie to 72 MHz (patrz opis pliku *boot.h* w dalszej części tego rozdziału). Zworę JP5 USB trzeba ustawić w pozycji on, aby móc programowo sterować rezystorem podciągającym do zasilania linii DP interfejsu USB. Gniazdo USB podłączone jest według wariantu (c) z rysunku 2.1.

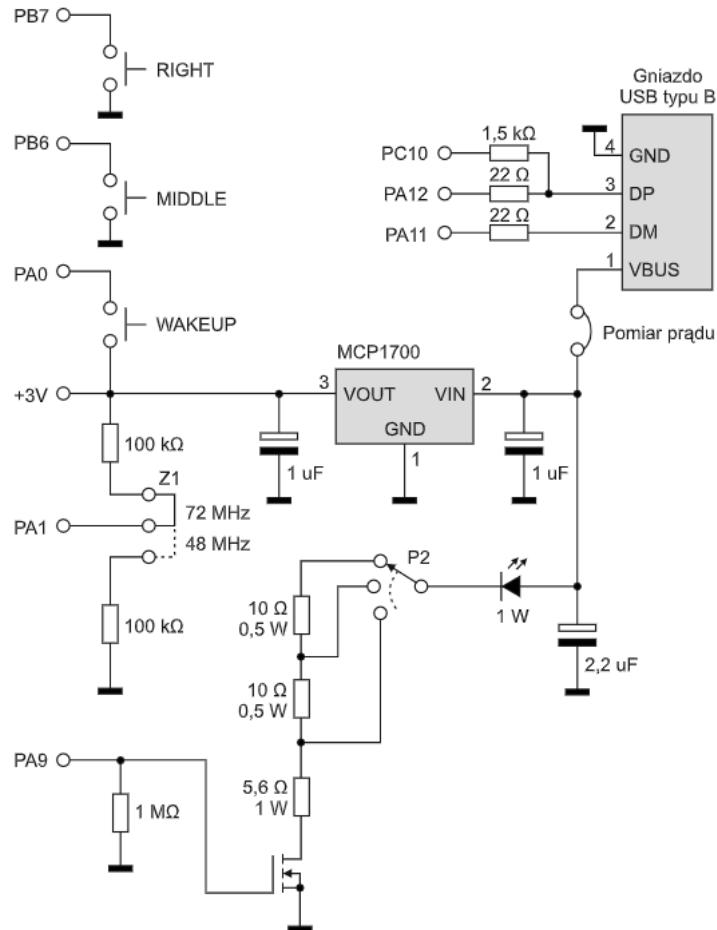
W ZL30ARM styk SRST gniazda JTAG nie jest połączony z wyprowadzeniem zeającym mikrokontrolera RST, przez co niektóre adaptery JTAG nie współpracują poprawnie z ZL30ARM albo w celu uzyskania takiej współpracy trzeba je konfigurować niestandardowo. Najprostsze rozwiążanie polega na połączeniu od spodu płytki styku SRST (numer 15) w gnieździe JTAG z przyciskiem SW4 RES.

Pewnym problemem w tym zestawie jest też podciąganie przełączników SW0 do SW3 oraz przełączników dżojstika rezystorami do napięcia +5 V. Formalnie wymaga to, aby te przełączniki były podłączane tylko do wejść mikrokontrolera tolerujących taki poziom napięcia wejściowego (ang. *five volt tolerant*). Nie wszystkie wejścia są tego typu i może ich zabraknąć, gdyż mogą być potrzebne np. do podłączenia wyświetlacza ciekłokrystalicznego. W praktyce okazuje się, że podłączenie dowolnego wejścia do napięcia +5 V przez rezystancję 10 kΩ nie uszkadza mikrokontrolera, powodując tylko przepływ pasożytniczego prądu. Dobra praktyka każe unikać takich rozwiązań.

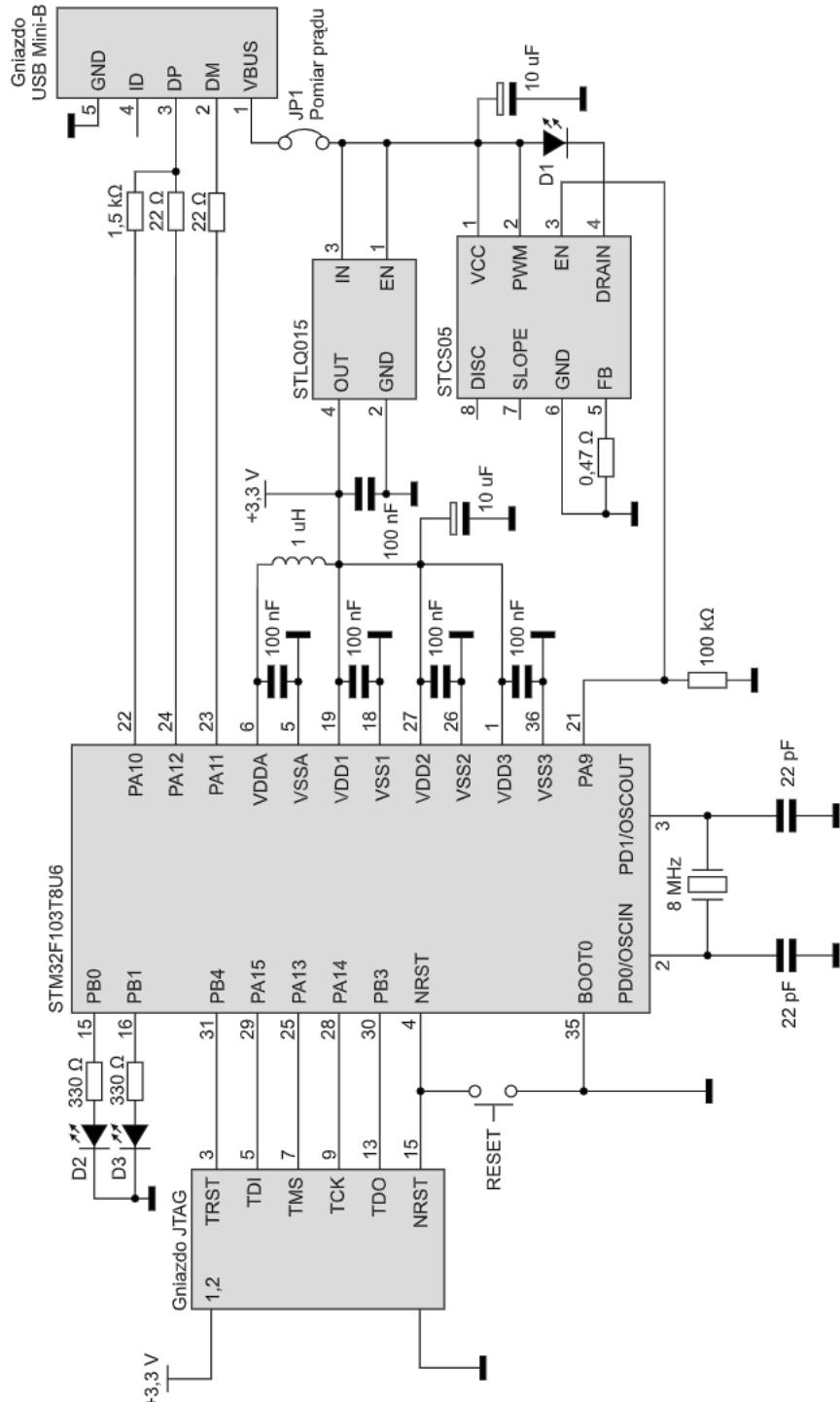
ZL31ARM

Z przedstawionych w tym podrozdziale modułów i zestawów do symulowania urządzenia zasilanego z interfejsu USB najłatwiej jest dostosować ZL31ARM. Trzeba

odciąć zasilanie od programatora. Można też odciąć cały programator i dołączyć standardowe gniazdo interfejsu JTAG. Należy też odciąć zasilanie potencjometru P1 i diody PWR, żeby nie pobierały prądu, gdy uśpiemy mikrokontroler. Schemat dołączonych do zestawu dodatkowych obwodów, z których korzystają zamieszczone w kolejnych rozdziałach przykładowe programy, przedstawia rysunek 2.5. Do zasilania mikrokontrolera trzeba zastosować regulator o małym spadku napięcia i możliwie małym prądzie płynącym przez wyprowadzenie masy. Jednowatowa dioda świecąca mocy symuluje pobór przez urządzenie dużego prądu z linii VBUS, a przy okazji może oświetlać biurko w miły sposób. Dioda musi być umieszczona na radiatorze. Za pomocą przełącznika P2 można zmieniać szeregowe rezystory ustalające prąd płynący przez tę diodę. Dla podanych na schemacie wartości rezystancji uzyskuje się prądy ok. 60, 100 i 300 mA. Do sterowania diodą świecącą można użyć dowolnego tranzystora NMOS o maksymalnym prądzie przewodzenia co najmniej 0,5 A i napięciu progowym bramki około 2 V (ang. *logic level gate drive*). Gniazdo USB podłącza się według wariantu (a) z rysunku 2.1. Warto prze-



Rys. 2.5. Dodatki do zestawu ZL31ARM



Rys. 2.6. Schemat gadżetu

widzieć zworę między stykiem VBUS tego gniazda a resztą obwodu, aby mierzyć prąd pobierany przez urządzenie z interfejsu USB. Przełączana zwora Z1 służy do wyboru częstotliwości taktowania mikrokontrolera. Między wyprowadzenie PA0 a zasilanie podłączony jest przycisk WAKEUP. Między wyprowadzenia PB6 i PB7 (dostępne są na złączu I²C) a masę podłączone są odpowiednio przyciski MIDDLE i RIGHT.

Gadżet

Bazując na przedstawionym wyżej rozwiązaniu z zestawem ZL31ARM, na potrzeby książki zaprojektowano układ spełniający wymogi standardu USB, nazywany dalej gadżetem, którego schemat jest widoczny na **rysunku 2.6**. Jest to bardzo prosty układ, ale można na nim uruchomić projekt wstępny i wszystkie projekty urządzeń FS, czyli projekty o numerach od 0 do 5. Mikrokontroler jest taktowany rezonatorem kwarcowym o częstotliwości 8 MHz. Na płytce zamontowane są przycisk RESET oraz standardowe 20-stykowe gniazdo interfejsu JTAG.

W gadżecie zastosowano stabilizator napięcia 3,3 V typu STLQ015, dla którego prąd skrośny przy braku obciążenia nie przekracza 1,7 µA, a przy maksymalnym obciążeniu, wynoszącym 150 mA, jest nie większy niż 2,4 µA. Zwora JP1 pozwala mierzyć prąd zasilania. Dioda świecąca mocy D1 symuluje pobór przez urządzenie dużego prądu z linii VBUS. Jest ona sterowana za pomocą źródła prądowego STCS05. Wartość prądu ustala się rezystorem podłączonym między wyprowadzenie FB układu STCS05 a masę. Przez rezystor ten płynie taki sam prąd jak przez diodę, a utrzymywany na nim spadek napięcia wynosi 100 mV. Diodę mocy włącza się, podając poziom wysoki na wejście EN, które jest podłączone do wyjścia PA9. Żeby dioda mocy nie świeciła, gdy wyprowadzenie PA9 nie zostało skonfigurowane lub jest w stanie wysokiej impedancji, wejście EN jest zbocznikowane do masy rezystorem.

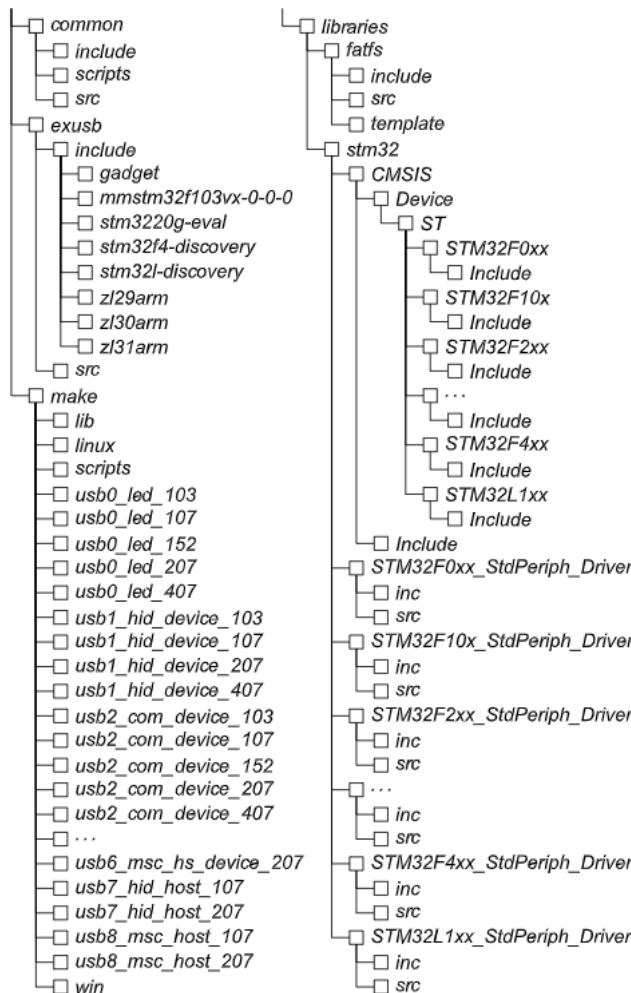
W celach diagnostycznych gadżet wyposażony jest w dwie diody świecące małej mocy: zieloną D2 i czerwoną D3. Gniazdo USB typu Mini-B jest podłączone według wariantu (a) z rysunku 2.1. Przy czym rezystor podciągający jest sterowany za pomocą wyprowadzenia PA10. Niewykorzystane wyprowadzenia mikrokontrolera, napięcie zasilania 3,3 V i masa dostępne są na złączach szpilkowych. Można je wykorzystać do podłączenia różnych elementów potrzebnych w przykładowych programach, czyli przycisków dżojstika, przetwornika piezoelektrycznego, wyświetlacza ciekłokrystalicznego itp. Wyprowadzenie PB2 (BOOT1) posłuży do wyboru częstotliwości taktowania rdzenia mikrokontrolera. Jeśli podłączy się je do masy, to będzie on taktowany zegarem o częstotliwości 48 MHz, a gdy dołączy się je do zasilania 3,3 V, to będzie taktowany z częstotliwością 72 MHz (patrz opis pliku *boot.h* w dalszej części tego rozdziału).

2.2. Struktura archiwum z przykładami

Bardzo istotnym uzupełnieniem tej książki jest archiwum z przykładowymi programami. Archiwum to będzie uaktualniane i poprawiane w miarę potrzeby. Jego najnowszą wersję w postaci skompresowanej możnaściągnąć ze strony Wydawnictwa BTC <http://www.btc.pl> lub z mojej strony domowej <http://www.mimuw.edu.pl/~>

marpe/book. Po rozpakowaniu powinniśmy zobaczyć strukturę katalogów zamieszczoną na **rysunku 2.7**. W nazwach ścieżek stosuję konwencję uniksową, w której nazwy katalogów i plików oddziela się (prawym) ukośnikiem (ang. *slash*). Jeśli podaję nazwę ścieżki względem katalogu, w którym rozpakowano archiwum, to rozpoczynam ją kropką a ukośnikiem. Dopasowanie się do konwencji stosowanych w innych systemach operacyjnych nie powinno nastręczać kłopotów.

W katalogu */common* znajdują się pliki tworzące małą bibliotekę. Są one wspólne dla wielu przykładowych programów. Napisane są w sposób ogólny, aby można je było wykorzystać w wielu projektach. Katalog */common/include* zawiera pliki nagłówkowe tej biblioteki, definiujące interfejsy poszczególnych modułów. Katalog */common/scr* zawiera pliki źródłowe z implementacją tych modułów. Katalog */common/scripts* zawiera skrypty dla konsolidatora (ang. *linker*).



Rys. 2.7. Drzewo katalogów archiwum z przykładowymi programami

Katalog *.exusb* zawiera pliki specyficzne dla konkretnych programów przykładowych. Napisane są one w sposób znacznie mniej ogólny niż pliki wyżej wspomnianej biblioteki. Niektóre z nich pełnią wręcz funkcje pomocnicze – implementują obsługę specyficznych układów zainstalowanych na płytach prototypowych. Użycie tych plików w innych projektach będzie zapewne wymagało istotnych ich modyfikacji. W katalogu *.exusb/include* znajdują się pliki nagłówkowe, które definiują interfejsy poszczególnych modułów. Ponadto w tym katalogu znajdują się podkatalogi zawierające pliki nagłówkowe z definicjami przeróżnych parametrów konfiguracyjnych dla płyt prototypowych. Nazwy tych podkatalogów jednoznacznie wskazują nazwy odpowiednich płyt. W katalogu *.exusb/src* umieszczone są pliki źródłowe z implementacją poszczególnych modułów.

W katalogu *.make/scripts* znajdują się skrypty potrzebne do skompilowania przykładowych programów i zaprogramowania pamięci Flash binarnym plikiem wynikowym. Katalog *.make/linux* zawiera programy i skrypty przydatne do testowania przykładów w systemie Linux. Katalog *.make/win* zawiera pliki przydatne do testowania przykładów w systemie Windows. Ponadto w katalogu *.make* znajdują się podkatalogi, których nazwy zaczynają się przedrostkiem *usb*. Podkatalogi te zawierają skrypty *makefile* dla przykładowych programów. Nazwy tych podkatalogów utworzone są według następującej konwencji. Po przedrostku *usb* znajduje się numer projektu, a po nim nazwa przybliżająca cel tego projektu. Nazwa kończy się przedrostkiem określającym wariant sprzętu, dla którego przeznaczony jest dany przykład. Zatem nazwa *usb0_led_103* oznacza projekt wstępny (oznaczony numerem 0), demonstrujący m.in. sterowanie diodami świecącymi i przeznaczony dla mikrokontrolera STM32F103. Pliki pośrednie tworzone podczas komplikacji za pomocą dostarczonych skryptów *makefile* przechowywane są w katalogu *.make/lib*, a dokładniej w jego podkatalogu, którego nazwa identyfikuje wariant sprzętu.

Katalog *.libraries/fatfs* zawiera bibliotekę FatFs implementującą system plików FAT, polecaną dla systemów wbudowanych. Jej autor podpisuje się ChaN. Podkatalog *inc* zawiera pliki nagłówkowe. Aplikacja korzystająca z tej biblioteki powinna włączyć znajdujący się tam plik *ff.h*, w którym zdefiniowane jest API biblioteki. W podkatalogu *src* są pliki, które zawierają właściwą implementację i które trzeba dołączyć do projektu. Podkatalog *template* zawiera dwa pliki, które trzeba zmodyfikować. W pliku *ffconf.h* definiuje się wiele parametrów, które pozwalają na elastyczne dostosowanie biblioteki do konkretnej aplikacji. Natomiast plik *diskio.c* zawiera szkielety niskopoziomowych operacji wejścia-wyjścia, które trzeba zaimplementować dla konkretnego rodzaju dysku. Więcej o tym piszę w rozdziale 8.

Katalog *.libraries/stm32* zawiera biblioteki dostarczane przez firmy ARM i STMicroelectronics. Są to CMSIS (ang. *Cortex Microcontroller Software Interface Standard*) oraz biblioteki ułatwiające obsługę peryferii mikrokontrolerów STM32 (ang. *Standard Peripherals Library Driver*). Nazwy podkatalogów dość jednoznacznie określają ich zawartość. CMSIS zawiera wyłącznie pliki nagłówkowe, umieszczone w podkatalogach *Include*. Pliki nagłówkowe dla modułów peryferyjnych znajdują się w podkatalogach *inc*, a pliki z implementacją w podkatalogach *src*. Wszystkie te biblioteki nazywam dalej skrótnie bibliotekami STM32.

W dalszej części książki opisuję poszczególne pliki znajdujące się w archiwum. Aby ułatwić orientację, nazwy plików umieszczam w ramce na marginesie w pobliżu miejsca, w którym je omawiam.

2.3. Pisanie programów dla wielu wariantów sprzętu

Pisane programów, które mają być uruchamiane na wielu wariantach sprzętu, jest sporym wyzwaniem. W tym podrozdziale przedstawiam główne zasady, którymi moim zdaniem należy się kierować już na etapie projektowania architektury oprogramowania. Dzięki temu można uniknąć wielu błędów i sprawić, że projekt będzie można łatwo dostosować do nowego wariantu sprzętu, którego nie brano pod uwagę na początku lub który wręcz nie istniał, gdy rozpoczęto projekt.

2.3.1. Jeden interfejs – wiele implementacji

Wydaje się, że najprostszym sposobem dostosowania programu do różnych wersji sprzętu jest wykorzystanie komplikacji warunkowej, czyli instrukcji warunkowych preprocesora: `#if`, `#elif`, `#else`, `#endif`. Jest to jednak złudne poczucie, gdyż zwykle prowadzi do bardzo złego rozwiązania problemu. Ponadto skłania do wprowadzania konstrukcji przyczyniających się do powstawania frustrujących błędów (trudnych do zlokalizowania i usunięcia) oraz skutkuje bardzo skomplikowaną strukturą plików źródłowych, zwłaszcza gdy instrukcje warunkowe preprocesora są zagnieżdżone lub obejmują duże i trudne do ogarnięcia wzrokiem fragmenty tekstu źródłowego. Ka da instrukcja `#if` wprowadza dodatkowy wariant programu i powoduje,  e w rzeczywistości zamiast z jednym, mamy do czynienia z dwoma lub kilkoma programami. Podczas jednego przebiegu kompilator „widzi” tylko jeden z tych wariantów, czyli jeden z tych programów. Pracuj c nad jednym wariantem, łatwo jest wprowadzi  zmiany, które spowoduj ,  e inne warianty nie b  d a dzia a c lub wr  cz nie b  d a si  kompilowa . Jednak najwa nejszy problem przy takim podejsci u spowodowany jest tym,  e narz  dzia do automatycznego wykrywania zale o ci mi dzy plikami  r  dowymi wykrywaj  ka d  zmian  i uruchamiaj  ponown  komplikacj  i konsolidacj  wszystkich plików zale onych od zmodyfikowanego pliku. Je li implementacja wszystkich wariantów sprzetu umieszczona jest we wspólnym pliku, to przestaje on by c niezale ony od sprzetu, a staje si  zale ony od wszystkich wariantów sprzetu. Dowolna zmiana lub nawet drobna poprawka w jednym wariantcie powoduje konieczno  ponownej komplikacji i konsolidacji wszystkich wariantów, nawet tych, dla których pozornie niczego nie zmieniono. Po ka dej komplikacji trzeba ponownie przetestowa  wszystkie pliki wynikowe, kt re zosta y zmodyfikowane. Podobnie jest, gdy chcemy doda  nowy wariant sprzetu. Po jego skompilowaniu musimy przetestowa  wszystkie dotychczasowe warianty. Zatem plik  r  dowy, kt re jest zale ony od wielu wariantów sprzetu, znacznie zwiększa nak ad pracy na etapie testowania oprogramowania.

Poprawne rozwi zanie problemu wielowariantowości polega na podzieleniu oprogramowania na warstwy i moduły. Warstwy tworzą niejako podzia  w pionie, a modu y – w poziomie. Cho  warstwy implementuje si  te z jako modu y. Ka dy modu  powinien mie  jasno zdefiniowany i mo liwie prosty interfejs komunikacji z nim. W j zyku C interfejs modu u to zestaw funkcji, kt re ten modu  implementuje i udo-

stępnia do wywoływania innym modułom. Podział na moduły powinien być możliwie naturalny, wynikający z zadań, które mają one pełnić. W szczególności należy unikać cyklicznych zależności wywołań funkcji między modułami. Moduły nie powinny współdzielić zmiennych globalnych, gdyż prowadzi to do zawiłych zależności między nimi. W wyjątkowych przypadkach, gdy wywoływanie funkcji wprowadza zbyt duży narzuć czasowy, możemy dopuścić, aby pewne zmienne globalne modułu były widziane na zewnątrz. Jednak konieczność stosowania wielu współdzielonych zmiennych globalnych świadczy zwykle o złym zaprojektowaniu podziału na moduły.

W języku C moduł odpowiada jednostce translacji, czyli plikowi źródłowemu z rozszerzeniem *c*. Interfejs modułu definiuje się w pliku nagłówkowym z rozszerzeniem *h*. Wszelkie szczegóły działania modułu, które nie powinny być widoczne na zewnątrz (funkcje, definicje stałych, zmienne globalne), należy ukryć wewnątrz implementacji modułu, czyli wewnątrz jednostki translacji. Osiąga się to, deklarując wszystkie zmienne globalne oraz prywatne funkcje modułu jako *static*. Zmienne globalne, które mają być widoczne na zewnątrz, deklaruje się dodatkowo (oprócz standardowej deklaracji w pliku z implementacją) w pliku nagłówkowym jako *extern*.

Jeśli moduł nie zależy od żadnych wariantów sprzętu, to zwykle składa się z jednego pliku nagłówkowego i jednego pliku z implementacją. Jeśli potrzebujemy kilku wariantów, to każdy z nich umieszczamy w osobnym pliku z implementacją. Wszystkie jednak realizują wspólny interfejs zdefiniowany w pojedynczym pliku nagłówkowym. Skompilowany i dobrze przetestowany moduł może być wielokrotnie wykorzystywany w różnych projektach jak czarna skrzynka bez konieczności ponownego testowania. Takie podejście bardzo ułatwia pielęgnację i rozwijanie oprogramowania, a dokładanie kolejnych wariantów sprzętu wymaga tylko komplikowania i testowania nowych modułów. Zastosowanie tej idei w praktyce pokazuję, prezentując w tym i dalszych rozdziałach kolejne moduły biblioteki, która znajduje się w archiwum w katalogu */common*.

2.3.2. **Daj szansę kompilatorowi**

Kolejna zasada mówi, że należy do minimum ograniczyć stosowanie preprocesora. Instrukcje preprocesora *#if*, *#elif*, *#else*, *#endif* należy starać się zastępować instrukcją *if* języka C. Jeśli warunek instrukcji *if* jest wyrażeniem stałym, kompilator potrafi zoptymalizować kod i usunąć instrukcję warunkową z kodu wynikowego. Dzięki temu, że nie używamy instrukcji preprocesora, kompilator może przeprowadzić analizę poprawności składniowej wszystkich gałęzi programu. Rozważmy następujący fragment programu, w którym konfiguracja parametrów pętli fazowej zależy od wartości stałej *HSE_VALUE*. Jeśli wartość tej stałej jest znana w czasie komplikacji, to sprawdzanie warunku i jedno z wywołań funkcji *RCC_PLLConfig* zostanie usunięte przez optymalizator, ale kompilator sprawdzi poprawność składniową obu wywołań tej funkcji.

```
if (HSE_VALUE == 8000000)
    RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_6);
else if (HSE_VALUE == 12000000)
    RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_4);
```

Natomiast wpływanie na kształt instrukcji `if` i `else` za pomocą instrukcji `#if`, `#elif`, `#else` lub mieszanie tych dwóch rodzajów instrukcji warunkowych w jednym fragmencie tekstu źródłowego jest zdecydowanie niewskazane. Taki styl programowania skutkuje nieczytelnym i trudnym do pielęgnacji kodem i wiedzie wprost ku nieuchronnej porażce.

Definiowanie stałych za pomocą instrukcji `#define` należy, jeśli to tylko możliwe, zastępować deklaracjami `static const`. Zobaczmy przykład, w którym potrzebna nam jest stała `delayTime` użyta w kilku miejscach. Zastosowane rozwiązanie sprawia, że stała ta ma określony typ, który jest sprawdzany przez kompilator, a dzięki optymalizatorowi kod wynikowy będzie równie efektywny, jak gdyby użyto definicji za pomocą instrukcji `#define`.

```
static const unsigned delayTime = 3000000U;
int i;
GreenLEDoFF();
for (i = 0; i < 5; ++i) {
    Delay(delayTime);
    GreenLEDOn();
    Delay(delayTime);
    GreenLEDoFF();
}
Delay(9U * delayTime);
```

Są jednak sytuacje, gdy język C wymaga stałych, które są literałami lub wyrażeniami stałymi składającymi się wyłącznie z literałów, jak w poniższym fragmencie tekstu źródłowego.

```
#define buffer_size 100
char buffer[buffer_size];
int i;
for (i = 0; i < buffer_size; ++i)
    buffer[i] = i;
```

Definicje makr należy zastępować funkcjami, które są przez optymalizator rozwijane w miejscu ich wywoływanego. Optymalizator podejmuje decyzję o rozwinięciu funkcji w miejscu jej wywołania na podstawie parametrów wywołania kompilatora i pewnych wewnętrznych reguł. Aby jednak ułatwić mu pracę i pozbyć się nie-rozwiniętych wersji funkcji z kodu wynikowego, funkcje, które są definiowane w module, ale nie są częścią interfejsu tego modułu, należy deklarować jako `static`, a funkcje definiowane w plikach nagłówkowych jako `static inline`. Klasycznym przykładem jest funkcja obliczająca minimum. Często spotyka się takie makro:

```
#define min(x, y) ((x) <= (y) ? (x) : (y))
```

auxiliary.h

Znacznie lepszym rozwiązaniem jest zdefiniowanie odpowiedniej funkcji. Poniższa funkcja jest zdefiniowana w pliku *auxiliary.h*.

```
static inline unsigned long min(unsigned long x, unsigned long y) {
    return x <= y ? x : y;
}
```

Przewaga funkcji nad makrem polega na tym, że argumenty i wynik funkcji mają określone typy, które są sprawdzane przez kompilator, nie mówiąc już o tym, że poniższe wywołanie w przypadku funkcji ma jasną semantykę, a w przypadku użycia makra wynik może być zaskakujący:

```
min(++x, ++y);
```

Czasem jednak stosowania makrodefinicji uniknąć się nie da. Język C nie pozwala na definiowanie stałych z użyciem wywołań funkcji. Zatem w poniższym fragmencie tekstu źródłowego makro REVERSE_UINT16 jest niezbędne.

```
#define REVERSE_UINT16(x) \
    (((x) << 8) & 0xff00U) | (((x) >> 8) & 0x00ffU)
```

```
typedef struct {
    uint16_t data_length;
    uint8_t data[12];
} data_t;
```

```
static const data_t data = {
    REVERSE_UINT16(sizeof(data_t) - 2U)
};
```

Makrodefinicja też nie zawsze da się zastąpić wywołaniem funkcji. Przykładem jest następujące makro, gdzie przyczyną tego jest użycie w nim instrukcji return.

```
#define active_check(cond, limit) { \
    int i; \
    for (i = (limit); !(cond); --i) \
        if (i <= 0) \
            return -1; \
}
```

2.3.3. Kompilacja warunkowa

stm32.h

Są jednak przypadki, w których nie da się uniknąć zastosowania komplikacji warunkowej, a czasem jest to wręcz wymagane. Musimy zapewnić ochronę przed wielokrotnym włączeniem zawartości tego samego pliku nagłówkowego w jednym pliku źródłowym (ang. *header guard*).Więcej niż jednokrotne włączenie zawartości jakiegoś pliku nagłówkowego powoduje zwykle wypisanie przez kompilator ostrzeżenia o powtórzonej deklaracji, a czasem zatrzymuje komplikację z błędem spowodowanym dwukrotną definicją. Dla przykładu rozważmy plik *stm32.h*, który włącza pliki nagłówkowe biblioteki CMSIS. Bardzo wiele plików nagłówkowych, które włączamy w naszych programach, potrzebuje włączyć tę bibliotekę. Niechybnie spowoduje to wielokrotne włączenie zawartości pliku *stm32.h*. Aby temu zapobiec,

na początku pliku nagłówkowego definiuje się stałą o unikalnej nazwie jakoś związaną z nazwą tego pliku i włącza zawartość tego pliku tylko wtedy, gdy stała ta nie była dotychczas zdefiniowana. Dla pliku *stm32.h* wygląda to tak:

```
#ifndef _STM32_H  
#define _STM32_H 1  
...  
#endif
```

Zadaniem pliku *stm32.h* jest ułatwienie włączania biblioteki CMSIS. Jak można się przekonać, oglądając implementację tej biblioteki, nazwa pliku nagłówkowego, który trzeba włączyć, zależy od modelu mikrokontrolera. Jedynym sposobem, żeby to osiągnąć i nie popaść w duże kłopoty, jest użycie instrukcji warunkowej preprocesora. Dlatego plik *stm32.h* zawiera poniższy kod, który włącza odpowiedni plik nagłówkowy zależnie od zdefiniowanej podrodziny STM32. Dodatkowo, jeśli nie zdefiniujemy żadnego ze znanych wariantów mikrokontrolera, preprocesor wypisze komunikat o błędzie.

```
#if defined STM32F0XX  
    #include <stm32f0xx.h>  
#elif defined STM32F10X_LD || defined STM32F10X_LD_VL || \  
    defined STM32F10X_MD || defined STM32F10X_MD_VL || \  
    defined STM32F10X_HD || defined STM32F10X_HD_VL || \  
    defined STM32F10X_XL || defined STM32F10X_CL  
    #include <stm32f10x.h>  
#elif defined STM32F2XX  
    #include <stm32f2xx.h>  
#elif defined STM32F30X  
    #include <stm32f30x.h>  
#elif defined STM32F37X  
    #include <stm32f37x.h>  
#elif defined STM32F4XX  
    #include <stm32f4xx.h>  
#elif defined STM32L1XX_MD || defined STM32L1XX_MDP || \  
    defined STM32L1XX_HD  
    #include <stm32l1xx.h>  
#else  
    #error STM32 device subfamily is unknown or undefined.  
#endif
```

Kod ten nie usuwa zależności komplikacji od wariantu sprzętu. Modyfikacja pliku *stm32.h* wymaga ponownego kompilowania wszystkich plików go włączających, nawet jeśli ta modyfikacja nie ma związku z tym plikiem. Takie rozwiązanie ma jednak tę zaletę, że ukrywa wszelkie zależności w jednym dobrze zdefiniowanym miejscu, gdzie łatwo jest je kontrolować. Nie trzeba używać instrukcji `#if` w każdym pliku źródłowym potrzebującym biblioteki CMSIS, co w przypadku koniecz-

ności dodania nowego wariantu sprzętu wiązałoby się z potrzebą modyfikowania wielu plików źródłowych. Wystarczy jedynie włączyć plik *stm32.h*. Wszelkie modyfikacje wykonuje się tylko w tym jednym pliku.

Instrukcje komplikacji warunkowej przydają się też przy definiowaniu domyślnych wartości stałych, jak w poniższym fragmencie programu, gdzie stała USB_B_MAX_POWER jest nadawana wartość 1, jeśli nie została ona nigdzie wcześniej zdefiniowana, a w szczególności nie została też zdefiniowana w linii poleceń kompilatora.

```
#ifndef USB_B_MAX_POWER  
    #define USB_B_MAX_POWER 1  
#endif
```

Język C nie dopuszcza zagnieźdzania komentarzy tworzonych za pomocą sekwencji znaków /* i */. Niedogodność tę można ominąć, wyłączając kompilowanie fragmentu tekstu źródłowego za pomocą instrukcji `#if`, której warunek jest zawsze fałszywy.

```
#if 0  
    ...  
#endif
```

Aby odkomentować taki fragment programu, wystarczy zamienić warunek na zawsze prawdziwy.

```
#if 1  
    ...  
#endif
```

2.3.4. Pliki konfiguracyjne

board_def.h
board_usb_def.h
usb_vid_pid.h

Oprócz konieczności pisania programów dla różnych modeli mikrokontrolerów musimy jeszcze zmierzyć się z problemem różnych schematów elektrycznych płyt, co jest związane z różnymi wariantami użycia peryferii. Dobrym sposobem poradzenia sobie z tym jest używanie w tekście źródłowym stałych, które można łatwo przedefiniować w pliku nagłówkowym w celu dostosowania programu do innego schematu płytki. Dla każdego wariantu sprzętu w katalogu *./common/include* znajduje się podkatalog, którego nazwa jednoznacznie identyfikuje ten wariant sprzętu i który zawiera pliki nagłówkowe służące do zdefiniowania jego konfiguracji. Są to pliki *board_def.h*, *board_usb_def.h* i *usb_vid_pid.h*.

Zacznijmy od końca. W pliku *usb_vid_pid.h* definiuje się następujące stałe identyfikujące urządzenie w deskryptorze konfiguracji:

- VID – identyfikator producenta, parametr `idVendor`;
- PID – identyfikator produktu, parametr `idProduct`.

Dla urządzenia w pliku *board_usb_def.h* definiuje się parametry deskryptora konfiguracji związane z zasilaniem i zarządzaniem energią:

- `USB_BM_ATTRIBUTES` – parametr `bmAttributes` określający m.in., czy urządzenie ma własne zasilanie;
- `USB_B_MAX_POWER` – maksymalny prąd pobierany z interfejsu USB (w jednostkach 2 mA), parametr `bMaxPower`.

W pliku *board_usb_def.h* dla układu periferyjnego DEV-FS trzeba też określić, jak podłączone jest gniazdo USB i jak steruje się rezystorem podciągającym. Definiują to następujące stałe:

- `USB_PULLUP_DIRECT` – sposób podłączenia rezystora podciągającego – ma wartość 1, gdy jest podłączony bezpośrednio do wyprowadzenia GPIO, a 0 w przeciwnym przypadku;
- `USB_PULLUP_ON` – poziom logiczny, który trzeba podać na wyjście, aby włączyć rezistor podciągający;
- `USB_PULLUP_GPIO_N` – literowe oznaczenie portu;
- `USB_PULLUP_PIN_N` – numer wyprowadzenia portu.

Przykładowa konfiguracja dla rezystora podciągającego podłączonego według wariantu (a) z rysunku 2.1 i sterowanego wyprowadzeniem PA10 wygląda tak:

```
#define USB_PULLUP_DIRECT    1
#define USB_PULLUP_ON         1
#define USB_PULLUP_GPIO_N     A
#define USB_PULLUP_PIN_N      10
```

Przykładowa konfiguracja dla rezystora podciągającego podłączonego według wariantu (b) z rysunku 2.1 i sterowanego wyprowadzeniem PA10 wygląda tak:

```
#define USB_PULLUP_DIRECT    0
#define USB_PULLUP_ON         1
#define USB_PULLUP_GPIO_N     A
#define USB_PULLUP_PIN_N      10
```

Przykładowa konfiguracja dla rezystora podciągającego podłączonego według wariantu (c) z rysunku 2.1 i sterowanego wyprowadzeniem PA10 wygląda tak:

```
#define USB_PULLUP_DIRECT    0
#define USB_PULLUP_ON         0
#define USB_PULLUP_GPIO_N     A
#define USB_PULLUP_PIN_N      10
```

W przypadku mikrokontrolera STM32L1xx, jeśli chcemy użyć wewnętrznego rezystora podciągającego, wystarczy następująca konfiguracja:

```
#define USB_PULLUP_DIRECT    0
```

Dla kontrolera USB trzeba w pliku *board_usb_def.h* zdefiniować następujące stałe:

- `HOST_VBUS_GPIO_N` – literowe oznaczenie portu sterującego wejściem EN (ang. *enable*) układu zasilającego linię VBUS;

- HOST_VBUS_PIN_N – numer wyprowadzenia portu sterującego wejściem EN;
- HOST_VBUS_ON – poziom logiczny, który trzeba podać na wejście EN, aby włączyć zasilanie linii VBUS;
- HOST_OVRCURR_GPIO_N – literowe oznaczenie portu, do którego podłączone jest wyjście OC (ang. *overcurrent*) sygnalizujące przekroczenie maksymalnego dopuszczalnego prądu pobieranego z VBUS;
- HOST_OVRCURR_PIN_N – numer wyprowadzenia portu, do którego podłączone jest wyjście OC;
- HOST_OVRCURR_IRQ_N – numer przerwania skojarzonego z wejściem, do którego podłączony jest sygnał OC.

Przykładowa konfiguracja, w której wejście EN podłączone jest do wyjścia PD15, a wyjście OC do wyjścia PD14, wygląda tak:

```
#define HOST_VBUS_GPIO_N      D
#define HOST_VBUS_PIN_N        15
#define HOST_VBUS_ON           1
#define HOST_OVRCURR_GPIO_N    D
#define HOST_OVRCURR_PIN_N     14
#define HOST_OVRCURR_IRQ_N     EXTI15_10
```

Ponadto w pliku *board_usb_def.h* włączane są potrzebne pliki nagłówkowe odpowiednie bibliotece STM32 oraz plik *usb_def.h* definiujący struktury danych i stałe opisane w standardzie USB.

Plik *board_def.h* pełni kilka funkcji. Po pierwsze, poprawne skompilowanie biblioteki STM32 wymaga zdefiniowania trzech stałych. Nie chcąc ingerować w pliki nagłówkowe tych bibliotek, najlepiej jest je zdefiniować w linii poleceń kompilatora (patrz niżej opis plików *makefile*). W pliku *board_def.h* umieszczone są instrukcje preprocesora sprawdzające, czy definicje tych stałych są spójne ze schematem płytki. Sprawdzamy, czy zdefiniowano stałą *USE_STDPERIPH_DRIVER*, która deklaruje chęć używania biblioteki STM32. Następnie sprawdzamy, czy zdefiniowano właściwą stałą, która określa podrodzinę zastosowanego na płytce mikrokontrolera STM32. Jest to jedna z następujących stałych: *STM32F0XX*, *STM32F10X_LD*, *STM32F10X_LD_VL*, *STM32F10X_MD*, *STM32F10X_MD_VL*, *STM32F10X_HD*, *STM32F10X_HD_VL*, *STM32F10X_XL*, *STM32F10X_CL*, *STM32F2XX*, *STM32F30X*, *STM32F37X*, *STM32F4XX*, *STM32L1XX_MD*, *STM32L1XX_MDP*, *STM32L1XX_HD*. Interfejs USB wymaga taktowania stabilnym generatorem kwarcowym. Częstotliwość tego generatora definiuje stała *HSE_VALUE*. Sprawdzamy, czy została ona zdefiniowana i czy jej wartość jest równa częstotliwości rezonatora kwarcowego podłączonego na płytce. Po drugie, w pliku *board_def.h* włączane są potrzebne pliki nagłówkowe biblioteki STM32. Po trzecie, w pliku tym definiuje się wykorzystanie peryferii (wejścia-wyjście, liczników, przerwań itp.) przez moduły biblioteki, co opisuję szczegółowo poniżej przy omawianiu poszczególnych modułów.

xcat.h

Korzystanie z układu peryferyjnego wiąże się zwykle z koniecznością użycia kilku stałych. Na przykład, jeśli zamierzamy wykorzystać licznik 3, to zapewne zechce-

my użyć stałych TIM3, TIM3_IRQn, RCC_APB1Periph_TIM3 oraz procedury obsługi przerwania TIM3_IRQHandler. Zmiana licznika wymaga przedefiniowania tych napisów w programie we wszystkich miejscach, gdzie one występują. Jest to dość żmudne i podatne na błędy. Użycie prostych instrukcji #define niewiele pomaga. Żeby modyfikacja programu była łatwa, proponuję wykorzystanie makrodefinicji xcat, xcat3, xcat4 i xcat5 znajdujących się w pliku *xcat.h*. Dzięki tym makrom wystarczy tylko w jednym miejscu zdefiniować numer licznika, na przykład tak:

```
#define MS_TIM_N 3
```

Wszystkie napisy zależne od numeru licznika definiujemy za pomocą tych makr:

```
#define MS_TIM          xcat(TIM, MS_TIM_N)
#define MS_TIM_IRQn     xcat3(TIM, MS_TIM_N, _IRQn)
#define MS_TIM_RCC       xcat(RCC_APB1Periph_TIM, MS_TIM_N)
#define MS_TIM_IRQHandler xcat3(TIM, MS_TIM_N, _IRQHandler)
```

W programie używamy napisów MS_TIM, MS_TIM_IRQn, MS_TIM_RCC, MS_TIM_IRQHandler, a powyższe definicje zachowują się tak, jak gdyby były to odpowiednio następujące definicje:

```
#define MS_TIM          TIM3
#define MS_TIM_IRQn     TIM3_IRQn
#define MS_TIM_RCC       RCC_APB1Periph_TIM3
#define MS_TIM_IRQHandler TIM3_IRQHandler
```

Różnica jest tylko taka, że teraz bardzo łatwo jest zmienić w nich numer użytego licznika, zmieniając definicję jednej stałej MS_TIM_N. Makro xcat skleja ze sobą dwa napisy, a makra xcat3, xcat4 i xcat5 sklejają odpowiednio trzy, cztery i pięć napisów. W razie potrzeby można zdefiniować analogiczne makra dla większej liczby napisów. Więcej o tym, jak działają makra xcat, można przeczytać w dodatku A12.3 do książki [1].

<i>stm32f0xx_conf.h</i>
<i>stm32f10x_conf.h</i>
<i>stm32f2xx_conf.h</i>
<i>stm32f30x_conf.h</i>
<i>stm32f37x_conf.h</i>
<i>stm32f4xx_conf.h</i>
<i>stm32l1xx_conf.h</i>

Biblioteki STM32 konfiguruje się zależnie od modelu mikrokontrolera odpowiednio za pomocą jednego z plików: *stm32f0xx_conf.h*, *stm32f10x_conf.h*, *stm32f2xx_conf.h*, *stm32f30x_conf.h*, *stm32f37x_conf.h*, *stm32f4xx_conf.h*, *stm32l1xx_conf.h* itd. Jedyną sensowną rzeczą, którą można w nich skonfigurować, jest włączenie lub wyłączenie kompilowania kodu diagnostycznego. Służy do tego makro assert_param. W przykładowych programach jest ono puste, żeby żaden kod diagnostyczny nie był generowany, co zmniejsza objętość kodu wynikowego i przyspiesza jego działanie.

2.4. Biblioteka mikrokontrolera

W tym podrozdziale opisuję pomocnicze moduły biblioteki, umieszczone w archiwum w katalogu */common*, ale niezwiązane bezpośrednio z obsługą interfejsu USB. Pisząc o module *x*, mam na myśli plik nagłówkowy *x.h* definiujący interfejs tego modułu oraz plik lub pliki źródłowe, których nazwa pasuje do wzorca *x*.c* i które zawierają implementację tego modułu. Plików nagłówkowych należy szukać w katalogu */common/include*, a pliki z rozszerzeniem *c* znajdują się w katalogu */common/src*.

2.4.1. Odmierzanie czasu

delay.h
delay.c

W wielu programach potrzebujemy odmierać czas. W tym celu moduł *delay* dostarcza bardzo prostej, ale mało dokładnej funkcji opóźniającej.

```
void Delay(volatile unsigned count);
```

Funkcja *Delay* wykonuje *count* iteracji pustej pętli *while*. Można przyjąć, że przy włączonej optymalizacji jedna iteracja trwa około 5 cykli zegara, którym taktowany jest rdzeń mikrokontrolera. Funkcja ta nie zwraca żadnej wartości. Mała dokładność odmierzania czasu wynika również z tego, że podczas wykonywania tej funkcji mogą być zgłaszane i obsługiwane przerwania i nie ma sposobu, aby oszacować zużyty na to czas procesora. Oprócz mizernej dokładności tej funkcji jej podstawową wadą jest aktywne oczekiwanie – program nie może wykonywać innych obliczeń, gdy się ona wykonuje.

timer.h
timer.c

Odmierzanie czasu z dokładnością do jednej milisekundy lub nawet jednej mikrosekundy zaimplementowane jest w module *timer*, który korzysta w tym celu z przerwań zgłaszanych przez liczniki sprzętowe. Żeby korzystać z dokładności milisekundowej, trzeba w pliku konfiguracyjnym *board_def.h* zdefiniować za pomocą stałej *MS_TIM_N* numer licznika sprzętowego, którego będziemy używać, przykładowo:

```
#define MS_TIM_N 2
```

Wtedy zostaną skompilowane następujące cztery funkcje.

```
void TimerConfigure(unsigned prio, unsigned subprio);
```

Funkcja *TimerConfigure* konfiguruje milisekundowe odliczanie czasu, a w szczególności konfiguruje wykorzystywane do tego przerwanie. Parametr *prio* określa priorytet wywieszania przerwania, a parametr *subprio* to podpriorytet, który decyduje o kolejności obsługi w obrębie przerwań o tym samym priorytecie wy-

właszczenia. Jak zwykle, im mniejsza jest wartość liczbową tych parametrów, tym większy priorytet im odpowiada. Funkcja ta nie zwraca żadnej wartości.

```
void TimerStart(int timer, void (*f)(void), unsigned time_ms);
```

Funkcja TimerStart uruchamia odliczanie czasu przez licznik i natychmiast kończy działanie. Parametr `timer` zawiera numer licznika. Po upływie czasu, zadanego za pomocą parametru `time_ms`, wywołana zostanie funkcja zwrotna, do której wskaźnik został przekazany za pomocą parametru `f`. Musi to być wskaźnik na funkcję, która nie ma żadnych parametrów i nie zwraca żadnego wyniku (typu `void`). Parametr `time_ms` określa czas w milisekundach i może mieć wartość z przedziału 1...32 767. Ponowne wywołanie tej funkcji powoduje zapamiętanie nowego wskaźnika do funkcji zwrotnej i rozpoczęcie odliczania czasu od początku. Funkcja ta nie zwraca żadnej wartości. Oprócz dokładnego odmierzania czasu zaletą tej funkcji jest brak aktywnego oczekiwania – program wyłącza licznik i może dalej wykonywać użyteczne obliczenia.

```
void TimerStop(int timer);
```

Funkcja TimerStop zatrzymuje licznik, którego numer podano jako parametr `timer`. W efekcie nie zostanie wywołana funkcja zwrotna `f` zapamiętana w wywoaniu TimerStart z tym numerem licznika. Funkcja ta nie zwraca żadnej wartości.

```
void ActiveWait(int timer, unsigned time_ms);
```

Funkcja ActiveWait zawiesza wykonanie programu na `time_ms` milisekund. Parametr ten może mieć wartość z przedziału 1...32 767. Parametr `timer` zawiera numer licznika, który ma być użyty. Funkcje TimerStart i ActiveWait nie mogą być używane jednocześnie z tym samym numerem licznika. Funkcja ta nie zwraca żadnej wartości. Funkcja ta jest użyteczna, gdy potrzebujemy odmierzyć czas dokładnie i nie przeszkadza nam aktywne oczekiwanie, gdyż nie ma żadnych użytecznych obliczeń do wykonania w tym czasie.

Żeby uzyskać jeszcze dokładniejsze odmierzanie czasu, a mianowicie z dokładnością do jednej mikrosekundy, trzeba w pliku konfiguracyjnym `board_def.h` zdefiniować numer użytego w tym celu licznika sprzętowego za pomocą stałej `US_TIM_N`, przykładowo:

```
#define US_TIM_N 3
```

Wtedy zostaną skompilowane następujące trzy funkcje.

```
void FineTimerConfigure(unsigned prio, unsigned subprio);
```

Funkcja FineTimerConfigure konfiguruje mikrosekundowe odliczanie czasu, a w szczególności konfiguruje wykorzystywane do tego przerwanie. Parametr `prio` określa priorytet wywołania przerwania, a parametr `subprio` to podpriorytet, który decyduje o kolejności obsługi w obrębie przerwań o tym samym priorytecie.

wywłaszczenia. Im mniejsza jest wartość liczbową tych parametrów, tym większy priorytet im odpowiada. Funkcja ta nie zwraca żadnej wartości.

```
void FineTimerStart(int timer, void (*f)(void), unsigned time_us);
```

Funkcja `FineTimerStart` umożliwia zaimplementowanie małych opóźnień bez aktywnego oczekiwania. Uruchamia ona odliczanie czasu przez licznik i natychmiast kończy działanie. Parametr `timer` zawiera numer licznika. Po upływie czasu, zadaneego za pomocą parametru `time_us`, wywołana zostanie funkcja zwrotna, do której wskaźnik został przekazany za pomocą parametru `f`. Podobnie jak poprzednio musi to być wskaźnik na bezparametrową funkcję typu `void`. Parametr `time_us` określa czas w mikrosekundach i może mieć wartość z przedziału 2...65 535. Ponowne wywołanie tej funkcji powoduje zapamiętanie nowego wskaźnika do funkcji zwrotnej i rozpoczęcie odliczania czasu od początku. Funkcja ta nie zwraca żadnej wartości.

```
void FineTimerStop(int timer);
```

Funkcja `FineTimerStop` zatrzymuje licznik, którego numer podano jako parametr `timer`. Zatem funkcja zwrotna `f` zapamiętana w wywołaniu `FineTimerStart` z tym numerem licznika nie zostanie wywołana. Funkcja ta nie zwraca żadnej wartości.

Można jednocześnie korzystać z liczników milisekundowych i mikrosekundowych, definiując różne numery stałych `MS_TIM_N` i `US_TIM_N`. Należy przy tym pamiętać, żeby użyć liczników, które są dostępne w danym modelu STM32. Aktualna implementacja została przetestowana na licznikach ogólnego przeznaczenia (ang. *general purpose timers*) o numerach od 2 do 5 (TIM2 do TIM5). Dla STM32 numer licznika podawany jako parametr `timer` jest w rzeczywistości numerem kanału użytego licznika sprzętowego. Dla wspomnianych liczników TIM2...TIM5 numer kanału przyjmuje wartość od 1 do 4.

2.4.2. Diody świecące

```
led.h  
led.c
```

Do sygnalizowania różnych zdarzeń i testowania aplikacji przydają się diody świecące. Przeróżne diody świecące w różnych kolorach są montowane na płytach prototypowych. Założymy, że mamy do dyspozycji dwie diody: zieloną i czerwoną. Moduł `led` implementuje proste funkcje do ich obsługi.

```
void LEDconfigure(void);
```

Funkcja `LEDconfigure` konfiguruje wyprowadzenia, do których podłączone są diody świecące. Funkcja ta nie zwraca żadnej wartości.

```
void GreenLEDon(void);  
void RedLEDon(void);
```

Funkcje GreenLEDOn i RedLEDOn włączają odpowiednio zieloną lub czerwoną diodę świeczącą. Funkcje te nie zwracają żadnej wartości.

```
void GreenLEDOff(void);  
void RedLEDOff(void);
```

Funkcje GreenLEDOff i RedLEDOff wyłączały odpowiednio zieloną lub czerwoną diodę świeczącą. Funkcje te nie zwracają żadnej wartości.

```
int GreenLEDstate(void);  
int RedLEDstate(void);
```

Funkcje GreenLEDstate i RedLEDstate pozwalają sprawdzić stan zielonej i czerwonej diody świecącej. Funkcje te zwracają 1, gdy odpowiednia dioda świeci, a 0 w przeciwnym przypadku.

```
void OK(int n);
```

Funkcja OK miga zieloną diodą n razy, po czym oczekuje czas odpowiadający około 4,5 okresom migania. Funkcja ta nie zwraca żadnej wartości.

```
void Error(int n);
```

Funkcja Error miga czerwoną diodą n razy, po czym oczekuje czas odpowiadający około 4,5 okresom migania. Funkcja ta nie zwraca żadnej wartości.

Moduł *led* wymaga zdefiniowania w pliku *board_def.h* następujących stałych:

- GREEN_LED_GPIO_N – literowe oznaczenie portu, do którego podłączona jest zielona dioda świecąca;
- GREEN_LED_PIN_N – numer wyprowadzenia portu, do którego podłączona jest zielona dioda świecąca;
- RED_LED_GPIO_N – literowe oznaczenie portu, do którego podłączona jest czerwona dioda świecąca;
- RED_LED_PIN_N – numer wyprowadzenia portu, do którego podłączona jest czerwona dioda świecąca;
- LED_ON – sposób podłączenia diod, czyli wartość logiczna, którą trzeba wpisać do portu wyjściowego, aby zaświecić diodę.

Przykładowo, jeśli dioda zielona podłączona jest do wyprowadzenia PE14, a czerwona do PE15 oraz jeśli ich katody podłączone są do tych wyprowadzeń, a anody do napięcia zasilania (oczywiście przez rezystory szeregowe), czyli zaświecenie diody wymaga podania na wyjście wartości logicznej 0, to stałe konfigurujące należy zdefiniować następująco:

```
#define GREEN_LED_GPIO_N E  
#define GREEN_LED_PIN_N 14  
#define RED_LED_GPIO_N E
```

```
#define RED_LED_PIN_N      15
#define LED_ON              0
```

error.h

W pliku nagłówkowym *error.h* zdefiniowane są dwie funkcje służące do sygnalizowania błędów, korzystające ze zdefiniowanej w module *led* funkcji *Error*.

```
static inline void ErrorPermanent(int expr, int err);
```

Funkcja *ErrorPermanent* przeznaczona jest do obsługi błędów, których przyczyna wydaje się nieusuwalna. Jeśli wartość parametru *expr* jest mniejsza od zera, wykonanie programu zostaje zawieszone (funkcja ta nigdy nie wraca), a czerwona dioda migra w rytmie: *err* mignięć, przerwa odpowiadająca około 5 okresom migania. Innymi słowy, w nieskończonej pętli wywoływana jest funkcja *Error(err)*. Jeśli wartość parametru *expr* jest nieujemna, to funkcja ta nic nie robi i nie zwraca żadnej wartości.

```
static inline void ErrorResetable(int expr, int err);
```

Funkcja *ErrorResetable* przeznaczona jest do obsługi błędów, których przyczyna może ustąpić po oczekaniu pewnego czasu i wyzerowaniu układu. Jeśli wartość parametru *expr* jest mniejsza od zera, to trzykrotnie wywoływana jest funkcja *Error(err)*, czyli czerwona dioda wykonuje trzy serie mignień, każda po *err* razy, po czym wywoywane jest programowe zerowanie mikrokontrolera. W tym przypadku funkcja ta nigdy nie wraca. Podczas zerowania wprowadzenie RST mikrokontrolera jest utrzymywane w stanie niskim, dzięki czemu również pozostałe układy na płytce mają szansę zostać wyzerowane. Jeśli wartość parametru *expr* jest nieujemna, to funkcja ta nic nie robi i nie zwraca żadnej wartości.

2.4.3. Wyświetlacz ciekłokryształiczny

lcd.h
lcd_disco_152.c
lcd_dummy.c
lcd_hd44780_10x.c
lcd_ks0108_10x.c
lcd_mb785_2xx.c

Jednym z bardziej przydatnych urządzeń wyjściowych w układzie mikroprocesorowym jest wyświetlacz ciekłokryształiczny. Moduł *lcd* dostarcza elementarnych funkcji, które pozwalają wypisywać tekst i które można zaimplementować dla większości popularnych wyświetlaczy. Plik *lcd.h* zawiera deklaracje tych funkcji, czyli definiuje wspólny interfejs modułu. Plik *lcd_hd44780_10x.c* implementuje ten interfejs dla mikrokontrolerów z podrodziny STM32F10x i wyświetlacza alfanumerycznego, który ma sterownik kompatybilny z układem HD44780 pozwalający wyświetlać 2 wiersze po 16 znaków. Zestawy uruchomieniowe ZL29ARM

i ZL30ARM mają gniazdo przystosowane do takiego typu wyświetlacza. Plik *lcd_ks0108_10x.c* implementuje interfejs modułu dla mikrokontrolerów z tej samej podrodziny STM32F10x, ale dla wyświetlacza graficznego WG12864A, który zawiera dwa sterowniki kompatybilne z układem KS0108 i w trybie tekstowym pozwala wyświetlać 8 wierszy po 21 znaków. Zestawy uruchomieniowe ZL29ARM i ZL30ARM mają również gniazdo przystosowane do tego typu wyświetlacza. Plik *lcd_mb785_2xx.c* implementuje funkcje dla modułu wyświetlacza graficznego MB785 zamontowanego w zestawie STM3220G-EVAL. Plik *lcd_disco_152.c* implementuje funkcje dla wyświetlacza ciekłokrystalicznego zainstalowanego na płytce STM32L-Discovery. Plik *lcd_dummy.c* zawiera tylko zaślepki dla funkcji modułu *lcd* i należy go dołączyć do projektu, jeśli przykładowy program, który korzysta z wyświetlacza ciekłokrystalicznego, ma być skompilowany na sprzęt, który takiego wyświetlacza nie ma. Jak wspomniałem, moduł *lcd* dostarcza tylko pięciu bardzo elementarnych funkcji.

```
int LCDconfigure(void);
```

Bezparametrowa funkcja *LCDconfigure* inicjuje porty wejścia-wyjścia, do których podłączony jest wyświetlacz, a następnie inicjuje sam wyświetlacz. Zwraca zero, jeśli wszystko poszło dobrze, a wartość ujemną, gdy wystąpił jakiś błąd.

```
void LCDclear(void);
```

Bezparametrowa funkcja *LCDclear* czyści zawartość ekranu wyświetlacza. Funkcja ta nie zwraca żadnej wartości.

```
void LCDgoto(int textLine, int charPos);
```

Funkcja *LCDgoto* ustawia bieżącą pozycję wyświetlania znaku w wierszu *textLine* i kolumnie *charPos*. Wiersze numerowane są od góry do dołu, począwszy od zera, a kolumny od lewej do prawej, też począwszy od zera. Funkcja ta nie zwraca żadnej wartości.

```
void LCDputchar(char c);
```

Funkcja *LCDputchar* wyświetla znak o kodzie ASCII *c* w bieżącej pozycji i przesuwa bieżącą pozycję o jeden znak w prawo. Funkcja ta traktuje wiersze, jakby były nieskończenie długie. Jeśli bieżąca pozycja znajduje się poza ekranem, żaden znak nie jest wyświetlany. Funkcja ta nie zwraca żadnej wartości.

```
void LCDputcharWrap(char c);
```

Funkcja *LCDputcharWrap* działa podobnie jak funkcja *LCDputchar*, z tą różnicą, że zawija wiersze. Po wyświetleniu znaku w ostatniej widocznej na ekranie pozycji przesuwa bieżącą pozycję na początek następnego wiersza. Funkcja ta również nie zwraca żadnej wartości.

Pliki *lcd_dummy.c*, *lcd_mb785_2xx.c*, i *lcd_disco_152.c* nie wymagają definiowania żadnych stałych w pliku *board_def.h*. Kompilując plik *lcd_hd44780_10x.c*, trzeba w pliku *board_def.h* zdefiniować następujące stałe:

- LCD_DATA_GPIO_N – oznaczenie literowe portu, do którego podłączone są linie danych wyświetlacza;
- LCD_D0_PIN_N – numer wyprowadzenia, do którego podłączona jest linia danych D0, pozostałe linie danych muszą być podłączone do wyprowadzeń o kolejnych numerach;
- LCD_CTRL_GPIO_N – oznaczenie literowe portu, do którego podłączone są linie sterujące wyświetlacza;
- LCD_RS_PIN_N – numer wyprowadzenia, do którego podłączona jest linia sterująca RS wyświetlacza;
- LCD_E_PIN_N – numer wyprowadzenia, do którego podłączona jest linia wyboru E (ang. *enable*) wyświetlacza;
- LCD_RW_PIN_N – numer wyprowadzenia, do którego podłączona jest linia wyboru kierunku transmisji R/W (ang. *read/write*) wyświetlacza.

Przykładowo, jeśli linie danych podłączone są do wyprowadzeń PB8...PB15, a linie RS, E i R/W odpowiednio do wyprowadzeń PB5, PB6 i PB7, to stałe konfigurujące należy zdefiniować następująco:

```
#define LCD_DATA_GPIO_N      B
#define LCD_D0_PIN_N          8
#define LCD_CTRL_GPIO_N       B
#define LCD_RS_PIN_N          5
#define LCD_E_PIN_N            6
#define LCD_RW_PIN_N           7
```

Kompilując plik *lcd_ks0108_10x.c*, należy w pliku *board_def.h*, oprócz wymienionych powyżej, zdefiniować jeszcze dwie stałe:

- LCD_CS1_PIN_N – numer wyprowadzenia, do którego podłączona jest linia wyboru pierwszego sterownika CS1 (ang. *chip select*);
- LCD_CS2_PIN_N – numer wyprowadzenia, do którego podłączona jest linia wyboru drugiego sterownika CS2.

Przykładowo, jeśli linie danych podłączone są do wyprowadzeń PE0...PE7, a linie RS, E i R/W, CS1 i CS2 odpowiednio do wyprowadzeń PE8, PE9, PE10, PE11 i PE12, to stałe konfigurujące należy zdefiniować następująco:

```
#define LCD_DATA_GPIO_N      E
#define LCD_D0_PIN_N           0
#define LCD_CTRL_GPIO_N        E
#define LCD_RS_PIN_N            8
#define LCD_E_PIN_N              9
#define LCD_RW_PIN_N             10
#define LCD_CS1_PIN_N            11
#define LCD_CS2_PIN_N             12
```

```
lcd_util.h  
lcd_util.c
```

Żeby móc korzystać z wyświetlacza ciekłokrystalicznego nieco bardziej wygodnie, moduł *lcd_util* dostarcza kilku wysokopoziomowych i niezależnych od sprzętu funkcji. Implementacja tych funkcji korzysta z funkcji dostarczanych przez moduł *lcd*, zatem używanie modułu *lcd_util* wymaga dołączenia do projektu pliku źródłowego modułu *lcd*, właściwego dla zastosowanego modelu wyświetlacza ciekłokrytalicznego. Dla wygody plik *lcd_util.h* włącza plik *lcd.h*, więc nie trzeba w plikach źródłowych włączać obu plików nagłówkowych.

```
void LCDwrite(const char *s);
```

Funkcja *LCDwrite* wyświetla napis *s*, począwszy od bieżącej pozycji. Napis musi być zakończony terminalnym zerem. Funkcja ta nie zwraca żadnej wartości.

```
void LCDwriteWrap(const char *s);
```

Funkcja *LCDwriteWrap* działa podobnie jak funkcja *LCDwrite*, z tą tylko różnicą, że zawija wyświetlany tekst, czyli przechodzi do następnego wiersza, jeśli tekst nie mieści się w bieżącym wierszu. Funkcja ta również nie zwraca żadnej wartości.

```
void LCDwriteLen(const char *s, unsigned n);
```

Funkcja *LCDwriteLen* wyświetla *n* znaków napisu wskazywanego przez *s*, począwszy od bieżącej pozycji. Napis nie jest zakończony terminalnym zerem. Funkcja ta nie zwraca żadnej wartości.

```
void LCDwriteLenWrap(const char *s, unsigned n);
```

Funkcja *LCDwriteLenWrap* działa podobnie jak funkcja *LCDwriteLen*, ale zawija wyświetlany tekst. Funkcja ta również nie zwraca żadnej wartości.

```
void LCDwriteHex(unsigned digits, unsigned hex);
```

Funkcja *LCDwriteHex* wyświetla *digits* mniejszych znaczących cyfr szesnastkowych liczby *hex*, począwszy od bieżącej pozycji. Funkcja ta nie zwraca żadnej wartości. Jest ona pomocna przy diagnozowaniu programów.

```
void LCDwriteHexWrap(unsigned digits, unsigned hex);
```

Funkcja *LCDwriteHexWrap* działa podobnie jak funkcja *LCDwriteHex*, ale zawija wyświetlany tekst. Funkcja ta również nie zwraca żadnej wartości.

```
void LCDsetRefresh(void (*cb)(void));
```

Obsługa wyświetlacza ciekłokrystalicznego jest na tyle czasochłonna, że nie powinna się odbywać wewnątrz procedury obsługującej przerwanie. Funkcja `LCDsetRefresh` rejestruje funkcję zwrotną (ang. *callback*), której zadaniem jest odświeżanie ekranu wyświetlacza ciekłokrystalicznego i która będzie wywoływana cyklicznie, gdy nie jest obsługiwane żadne przerwanie. Parametr `cb` jest wskaźnikiem na funkcję zwrotną, która musi być bezparametrowa i nie może zwracać żadnego wyniku. Funkcja `LCDsetRefresh` nie zwraca żadnej wartości.

```
void LCDrunRefresh(void);
```

Bezparametrowa funkcja `LCDrunRefresh` wywołuje zarejestrowaną funkcję zwrotną. Nic nie robi, jeśli żadna funkcja zwrotna nie została zarejestrowana. Wywołanie funkcji `LCDrunRefresh` należy umieścić w głównej pętli programu, tak aby była wywoływana, gdy tylko żadne przerwanie nie czeka na obsłużenie. Funkcja ta nie zwraca żadnej wartości.

```
font5x8.h  
font5x8.c  
fonts.h  
fonts.c
```

Wyświetlacz graficzne, których obsługa została zaimplementowana w module `lcd`, nie mają własnych generatorów znaków. Trzeba im dostarczyć rastrowe wzory fontów. Niestety zestawy znaków dla zastosowanych wyświetlaczów nie są wzajemnie kompatybilne. Dla WG12864A obrazy znaków zapisane są kolumnami bez uwzględnienia odstępu między znakami. Dla MB785 zapisane są wierszami łącznie z odstępem, który należy pozostawić między znakami. Pliki z fontami zostały wydzielone, aby łatwo można było je podmienić na inne, a także po to, aby ewentualnie można ich było użyć z innymi typami wyświetlaczów. Wszystkie fonty zawierają znaki o kodach ASCII z przedziału 32...126.

Jeśli do projektu został dołączony plik `lcd_ks0108_10x.c`, trzeba też dołączyć plik `font5x8.c` z definicją fontu dla wyświetlacza WG12864A. Ekran tego wyświetlacza ma wysokość 64 piksele i szerokość 128 pikseli, a pojedynczy znak ma wysokość 8 pikseli i szerokość 5 pikseli. Między znakami wyświetlana jest przerwa o szerokości 1 piksela. Podsumowując, umożliwia to wyświetlenie 8 wierszy po 21 znaków.

Jeśli natomiast do projektu został dołączony plik `lcd_mb785_2xx.c`, to należy też dołączyć plik `fonts.c` z definicjami fontów dla modułu wyświetlacza MB785. Ekran wyświetlacza MB785 ma wysokość 240 pikseli i szerokość 320 pikseli. W pliku `fonts.c` zdefiniowano fonty o czterech rozmiarach, których parametry zebrane są w tabeli 2.3. Bazując one na kroju pisma Terminus Font, którego autorem jest Dimitar Toshkov Zhekov. Rozmiary znaków podane są w pikselach. Domyslnie używany rozmiar wybiera się za pomocą stałej `LCD_DEFAULT_FONT` zdefiniowanej w pliku `fonts.h`, przykładowo:

```
#define LCD_DEFAULT_FONT font14x28
```

Tab. 2.3. Parametry fontów dla wyświetlacza MB785

Nazwa fontu	Rozmiar znaku	Liczba wierszy	Liczba kolumn
font8x16	8 × 16	15	40
font10x18	10 × 18	13	32
font12x24	12 × 24	10	26
font14x28	14 × 28	8	22

2.4.4. Interfejs I²C

i2c.h
i2c.c

Wiele układów peryferyjnych, np. ekspandery wejść-wyjścia czy różnego rodzaju czujniki, podłącza się za pomocą interfejsu I²C. Moduł *i2c* udostępnia trzy bardzo proste funkcje do obsługi tego interfejsu.

```
void I2Cconfigure(void);
```

Bezparametrowa funkcja *I2Cconfigure* konfiguruje interfejs jako urządzenie nadrzędne (ang. *master*). Nie zwraca żadnej wartości.żeby funkcja ta skompilowała się i działała poprawnie, w pliku *board_def.h* trzeba zdefiniować cztery stałe:

- *I2C_N* – numer wykorzystywanej interfejsu;
- *I2C_GPIO_N* – oznaczenie literowe portu, do którego podłączone są linie SCL i SDA;
- *I2C_SCL_PIN_N* – numer wyprowadzenia, do którego podłączona jest linia SCL;
- *I2C_SDA_PIN_N* – numer wyprowadzenia, do którego podłączona jest linia SDA.

Przykładowo, jeśli używamy interfejsu I²C1, którego linie SCL i SDA są wyprowadzone odpowiednio na porty PB6 i PB9, to definicje w pliku *board_def.h* powinny wyglądać następująco:

```
#define I2C_N          1
#define I2C_GPIO_N      B
#define I2C_SCL_PIN_N    6
#define I2C_SDA_PIN_N    9
```

```
int I2CwriteDeviceRegister(uint8_t addr, uint8_t reg, uint8_t value);
```

Funkcja *I2CwriteDeviceRegister* zapisuje jeden bajt o wartości *value* do rejestru o numerze *reg* w urządzeniu I²C o adresie *addr*. Jeśli transmisja zakończy się powodzeniem, wynikiem funkcji jest wartość zero. Błąd podczas transmisji sygnalizowany jest zwróceniem wartości ujemnej.

```
int I2CreadDeviceRegister(uint8_t addr, uint8_t reg, uint8_t *value);
```

Funkcja *I2CreadDeviceRegister* czyta jeden bajt z rejestru o numerze *reg* urządzenia I²C o adresie *addr*. Jeśli transmisja zakończy się poprawnie, to odczytana

wartość zapisywana jest pod adresem wskazywanym za pomocą parametru `value`, a wynikiem funkcji jest wartość zero. Błąd transmisji sygnalizowany jest zwróceniem wartości ujemnej.

2.4.5. Interfejs I²S

`i2s.h`
`i2s.c`

Interfejs I²S wykorzystywany jest do przesyłania strumienia danych audio. Można go użyć na przykład do podłączenia zewnętrznego przetwornika cyfrowo-analogowego. Do obsługi tego interfejsu przeznaczony jest moduł `i2s`. Aby transmisja nie obciążała zbytnio procesora, przesłania realizowane są za pomocą DMA.

```
void I2Sconfigure(uint32_t frequency);
```

Interfejs I²S obsługuje kilka trybów transmisji i standardów kodowania. Może pracować jako podrzędny (ang. *slave*) lub nadrzędny (ang. *master*), zgodnie ze standardem Philipsa lub PCM, z wyrównywaniem do najbardziej lub najmniej znaczącego bitu. Rozdzielcość próbkowania może być 16-, 24- lub 32-bitowa. Funkcja `I2Sconfigure` konfiguruje interfejs w trybie nadziednym w standardzie Philipsa. Nie zwraca żadnej wartości. Parametr `frequency` określa częstotliwość próbkowania w hercach. Przesyłany jest sygnał stereofoniczny – na przemian wysyłane są próbki kanału lewego i prawego. Używane jest kwantowanie 16-bitowe – pojedyncza próbka jest liczbą typu `int16_t`. Sygnał zegarowy niezbędny do taktowania interfejsu I²S jest konfigurowany w module `board_init`. Wyprowadzenia interfejsu konfiguruje się w pliku `board_def.h` za pomocą następujących stałych:

- `I2S_N` – numer wykorzystywanego interfejsu;
- `I2S_MCK_GPIO_N` – oznaczenie literowe portu, do którego podłączona jest linia MCK;
- `I2S_MCK_PIN_N` – numer wyprowadzenia, do którego podłączona jest linia MCK;
- `I2S_SCK_GPIO_N` – oznaczenie literowe portu, do którego podłączona jest linia SCK;
- `I2S_SCK_PIN_N` – numer wyprowadzenia, do którego podłączona jest linia SCK;
- `I2S_SD_GPIO_N` – oznaczenie literowe portu, do którego podłączona jest linia SD;
- `I2S_SD_PIN_N` – numer wyprowadzenia, do którego podłączona jest linia SD;
- `I2S_WS_GPIO_N` – oznaczenie literowe portu, do którego podłączona jest linia WS;
- `I2S_WS_PIN_N` – numer wyprowadzenia, do którego podłączona jest linia WS.

Interfejs I²S w mikrokontrolerach STM32 jest realizowany za pomocą odpowiednio skonfigurowanego interfejsu SPI. Przykładowo, jeśli korzystamy z interfejsu SPI3, a linie MCK, SCK, SD i WS są wyprowadzone odpowiednio na porty PC7,

PC10, PC12 i PA4, to definicje w pliku *board_def.h* powinny wyglądać następująco:

```
#define I2S_N          3
#define I2S_MCK_GPIO_N  C
#define I2S_MCK_PIN_N   7
#define I2S_SCK_GPIO_N  C
#define I2S_SCK_PIN_N   10
#define I2S_SD_GPIO_N   C
#define I2S_SD_PIN_N    12
#define I2S_WS_GPIO_N   A
#define I2S_WS_PIN_N    4
```

Ponadto, ponieważ korzystamy z DMA, w pliku *board_def.h* należy też zdefiniować numer DMA oraz numery wykorzystywanego kanału i strumienia DMA. Służą do tego stałe:

- I2S_DMA_N – numer DMA;
- I2S_DMA_CHANNEL_N – numer kanału DMA;
- I2S_DMA_STREAM_N – numer strumienia DMA.

Dla poszczególnych interfejsów dopuszczalne są tylko niektóre kombinacje tych parametrów. Informacji należy szukać w [9], [10] lub [13]. Dla wyżej wspomnianego interfejsu SPI3 można użyć następującego zestawu parametrów:

```
#define I2S_DMA_N          1
#define I2S_DMA_CHANNEL_N   0
#define I2S_DMA_STREAM_N    7
```

```
int I2Splay(int16_t const *sampleAddress, uint32_t sampleCount,
            i2s_play_t play, i2s_irq_source_t irq_source,
            unsigned prio, unsigned subprio,
            void (* callback)(int16_t **samples, uint32_t *count));
```

Funkcja I2Splay rozpoczyna wysyłanie strumienia danych audio za pomocą skonfigurowanego interfejsu I²S. Parametr sampleAddress zawiera adres początku bufora z próbkami do odtworzenia. Parametr sampleCount zawiera liczbę próbek do odtworzenia. Parametr play przyjmuje dwie wartości określające sposób odtwarzania sekwencji próbek:

- I2S_PLAY_ONCE – odtworzenie jednokrotne;
- I2S_PLAY_FOREVER – odtwarzanie w nieskończonej pętli.

Parametr irq_source decyduje, czy i kiedy ma być zgłaszane przerwanie:

- I2S_NO_IRQ – nie będzie zgłaszane;
- I2S_IRQ_TRANSFER_COMPLETE – zgłaszanego po każdym zakończeniu transmisji.

Parametr prio przekazuje priorytet wywłaszczenia, a parametr subprio podpriorytet przerwania. Parametr callback zawiera adres funkcji zwrotnej, która ma obsłu-

giwać zgłoszone przerwanie. Funkcja ta zwraca adres początku bufora z kolejnymi próbками do odtworzenia za pomocą parametru `samples`. Natomiast za pomocą parametru `count` zwraca liczbę nowych próbek do odtworzenia. Funkcja `I2Splay` zwraca zero, gdy zakończyła się sukcesem, a wartość ujemną, gdy wystąpił błąd.

```
void I2Spause(void);
```

Bezparametrowa funkcja `I2Spause` wstrzymuje transmisję strumienia danych w skonfigurowanym interfejsie I²S. Funkcja ta nie zwraca żadnej wartości.

```
void I2Sresume(void);
```

Bezparametrowa funkcja `I2Sresume` wznowia transmisję strumienia danych w skonfigurowanym interfejsie I²S. Funkcja ta nie zwraca żadnej wartości.

2.4.6. Inicjowanie programu

```
startup_stm32.c
```

W pliku `startup_stm32.c` znajdują się domyślna procedura obsługi przerwań oraz procedura startowa, uruchamiana jako pierwsza po wyzerowaniu mikrokontrolera. Procedura startowa inicjuje wszystkie zmienne globalne i statyczne programu (wszystkie zmienne niezainicjowane jawnie są zerowane), a następnie wywołuje funkcję `main`. Więcej o tym można przeczytać w mojej poprzedniej książce [2], w rozdziałach 1.8.1 i 1.8.2. Ponadto, zależnie od modelu mikrokontrolera, plik ten włącza właściwy plik z definicją tablicy adresów procedur obsługi przerwań. Poszczególne linie mikrokontrolerów STM32 obsługują różne zestawy przerwań i mają różne tablice – jak dotąd można się doliczyć co najmniej 16 wariantów tablicy przerwań. Zróżnicowanie zostało osiągnięte za pomocą instrukcji warunkowych preprocesora, co – jak napisałem powyżej – nie jest zbyt dobrą praktyką. Jednak w tym przypadku zdecydowała chęć umieszczenia wszystkiego w jednym pliku, który w dodatku praktycznie nigdy się nie zmienia i ma dość prostą strukturę, więc wprowadzone w ten sposób zależności komplikacji nie są dotkliwe.

2.4.7. Inicjowanie sprzętu

```
board_init.h  
board_init_051.c  
board_init_103.c  
board_init_107.c  
board_init_152.c  
board_init_207.c
```

Zadaniem modułu `board_init` jest zainicjowanie podstawowych układów mikrokontrolera, głównie interfejsu do pamięci Flash, generatora kwarcowego, pętli fazowych oraz dzielników wstępnych (ang. *prescaler*) dostarczających sygnałów zegarowych do układów peryferyjnych. Plik `board_init.h` deklaruje interfejs tego modułu. Pliki `board_init_051.c`, `board_init_103.c`, `board_init_107.c`, `board_init_152.c`

i *board_init_207.c* implementują ten interfejs dla poszczególnych modeli mikrokontrolerów.

```
void AllPinsDisable(void);
```

Bezparametrowa funkcja *AllPinsDisable* dezaktywuje wszystkie porty wejścia-wyjścia mikrokontrolera. Polega to na zablokowaniu wejściowego przerzutnika Schmitta w taki sposób, aby szum z wejścia nie przełączał go chaotycznie. Redukuje to prąd pobierany przez te przerzutniki i bufory wejściowe, co ma istotne znaczenie, gdy chcemy uśpić mikrokontroler w stanie niskiego poboru energii. Dodatkowo zablokowanie wejść ogranicza poziom zakłóceń elektromagnetycznych emitowanych przez układ. Funkcja ta powinna być wywołana przed wszystkimi innymi funkcjami konfigurującymi porty wejścia-wyjścia. Wtedy nieużywane wyprowadzenia pozostaną zablokowane. Funkcja ta nie zwraca żadnej wartości.

```
int ClockConfigure(int sysclk_MHz);
```

Funkcja *ClockConfigure* inicjuje wszystkie układy związane z taktowaniem mikrokontrolera oraz parametry czasowe interfejsu pamięci Flash. Jej jedynym parametrem *sysclk_MHz* jest częstotliwość taktowania rdzenia w MHz. Jak to zostało już wspomniane, dla poprawnego działania tej funkcji w linii poleceń kompilatora musi być zdefiniowana stała *HSE_VALUE*, która oznacza częstotliwość w hercach podłączonego rezonatora kwarcowego. Wartość 0 oznacza, że rezonator kwarcowy nie jest podłączony, a mikrokontroler jest taktowany wewnętrznym generatorem RC HSI (ang. *High Speed Internal*). Jednak interfejs USB do poprawnej pracy wymaga, aby częstotliwość taktowania była stabilna, a HSI nie zapewnia dostatecznej stabilności. Alternatywnie, zamiast podłączać rezonator kwarcowy, można do wyprowadzenia *OSC_IN* podłączyć zewnętrzny generator kwarcowy. Należy wtedy w pliku *board_def.h* zdefiniować stałą *HSE_BYPASS* o wartości różnej od zera. W tym przypadku w funkcji *ClockConfigure* wywołanie

```
RCC_HSEConfig(RCC_HSE_ON);
zostanie zastąpione wywołaniem
RCC_HSEConfig(RCC_HSE_Bypass);
```

Opcja ta okazuje się przydatna na przykład wtedy, gdy na płytce STM32L-Discovery chcemy taktować mikrokontroler z programatora ST-LINK/V2 (musi być wtedy zamknięta zwora SB17).

Zaimplementowane w poszczególnych plikach dopuszczalne wartości parametrów *HSE_VALUE* i *sysclk_MHz* są zebrane w **tabeli 2.4**. W drugiej kolumnie tej tabeli wymienione są typy mikrokontrolerów, dla których mogą być zastosowane poszczególne pliki. Dla STM32F2xx i STM32F4xx parametr *HSE_VALUE* może przyjmować wartość wyrażającą całkowitą liczbę MHz z przedziału od 4 do 26. Dla danego pliku dla każdej podanej w przedostatniej kolumnie wartości *HSE_VALUE* można uzyskać dowolną wartość *sysclk_MHz* podaną w ostatniej kolumnie. Przy czym uwzględnione zostały rezonatory kwarcowe najczęściej spotykane w układach z mikro-

kontrolerami STM32. Implementację można też dość łatwo dostosować do innych wartości częstotliwości, ale z pewnymi zastrzeżeniami. Trzeba uwzględnić, że niektóre modele mikrokontrolerów STM32 mają mniejsze maksymalne częstotliwości taktowania rdzenia niż podane w ostatniej kolumnie tej tabeli. Ponadto układ peryferyjny USB wymaga taktowania zegarem o częstotliwości 48 MHz, co z uwagi na dopuszczalne wartości współczynników pętli fazowych i dzielników częstotliwości ogranicza w niektórych przypadkach możliwe do zastosowania rezonatory kwarcowe i możliwe do uzyskania częstotliwości taktowania rdzenia. Jeśli chcemy korzystać równocześnie z innych interfejsów, które wymagają ustalonych częstotliwości taktowania, na przykład Ethernetu lub I²S, to pole manewru zawęża się jeszcze bardziej. Należy to wszystko sprawdzić w dokumentacji. Funkcja *ClockConfigure* zwraca zero, gdy konfigurowanie generatora kwarcowego i pętli fazowych mikrokontrolera zakończyło się sukcesem, a wartość ujemną, gdy wystąpił błąd.

Tab. 2.4. Dopuszczalne wartości parametrów funkcji *ClockConfigure*

Nazwa pliku	Model mikrokontrolera	HSE_VALUE	sysclk_MHz
<i>board_init_051.c</i>	STM32F0xx	0	24
		8000000	32
		12000000	48
		16000000	
<i>board_init_103.c</i>	STM32F102 STM32F103	8000000	48
		12000000	72
		16000000	
<i>board_init_107.c</i>	STM32F105 STM32F107	5000000	
		8000000	
		10000000	
		12000000	48
		15000000	72
		16000000	
		20000000	
		25000000	
<i>board_init_152.c</i>	STM32L1xx	8000000	24
		12000000	32
		16000000	
<i>board_init_207.c</i>	STM32F2xx STM32F4xx	4000000	48
		5000000	72
		...	120
		25000000	168
		26000000	

2.4.8. Parametry uruchamiania aplikacji

```
boot.h
boot_10x.c
boot_15x.c
boot_207.c
boot_72_fs_0.c
```

Checąc pisać programy uruchamiane na różnych platformach sprzętowych, dobrze jest zapewnić sobie możliwość łatwego modyfikowania niektórych parametrów aplikacji. Moduł *boot* umożliwia elastyczne konfigurowanie parametrów związanych z interfejsem USB. Udostępnia on tylko jedną funkcję.

```
void GetBootParams(int *sysclk, usb_speed_t *speed, usb_phy_t *phy);
```

Funkcja `GetBootParams` przyjmuje jako parametry trzy wskaźniki, za pomocą których zwraca wartości konfigurowanych parametrów. Jeśli któraś z wartości nie jest potrzebna, to jako odpowiedni parametr należy przekazać wskaźnik zerowy (`NUL`). Za pomocą parametru `sysclk` zwracana jest częstotliwość w MHz, z jaką ma być taktowany rdzeń mikrokontrolera. Za pomocą parametru `speed` przekazuje się szybkość, z jaką ma pracować interfejs USB. Odpowiednie stałe zdefiniowane są w pliku `usb_def.h`:

- `HIGH_SPEED` – wysoka szybkość,
- `FULL_SPEED` – pełna szybkość,
- `LOW_SPEED` – mała szybkość.

Za pomocą parametru `phy` wskazuje się nadajnik-odbiornik, który ma być użyty. W tym celu w pliku `usb_def.h` zdefiniowano odpowiednie stałe:

- `USB_PHY_A` – wbudowany nadajnik-odbiornik DEV-FS dostępny na wyprowadzeniach PA11 i PA12 lub wbudowany nadajnik-odbiornik OTG-FS dostępny na wyprowadzeniach PA9, PA10, PA11 i PA12;
- `USB_PHY_B` – wbudowany nadajnik-odbiornik OTG-HS, mogący pracować tylko w trybie FS, dostępny na wyprowadzeniach PB12, PB13, PB14 i PB15;
- `USB_PHY_ULPI` – zewnętrzny nadajnik-odbiornik OTG-HS, podłączany za pomocą interfejsu ULPI;
- `USB_PHY_I2C` – zewnętrzny nadajnik-odbiornik OTG-HS, mogący pracować tylko w trybie FS, podłączany za pomocą interfejsu I²C.

W katalogu `/common/src` znajdują się cztery przykładowe implementacje modułu `boot`. Plik `boot_10x.c` powstał z myślą o zestawach ZL29ARM, ZL30ARM, które mają zwoję BOOT1, ale oczywiście można go zastosować dla dowolnego modelu mikrokontrolera z podrodziny STM32F10x. Jeśli nie ma zwory BOOT1, można w tym celu wykorzystać dowolne inne wejście mikrokontrolera. Za pomocą zwory BOOT1 wybiera się częstotliwość taktowania rdzenia. Pozycja zwory jest sprawdzana podczas wywoływanego funkcji `GetBootParams`. Funkcja ta zwraca za pomocą parametru `sysclk` wartość 48 lub 72 MHz, jeśli zwora BOOT1 jest ustaliona odpowiednio w pozycji 0 lub 1. Pozostałe dwa parametry ustawiane są na jedyne możliwe dla STM32F10x wartości, czyli odpowiednio `FULL_SPEED` i `USB_PHY_A`. W pliku `board_def.h` trzeba zdefiniować położenie zwory BOOT1 lub innego wejścia, za pomocą którego chcemy konfigurować częstotliwość taktowania. Stosujemy tu tę samą konwencję, co przy konfigurowaniu innych wejść i wyjść mikrokontrolera, na przykład dla diod świecących. Standardowo zwora BOOT1 jest podłączona do wyprowadzenia PB2, czyli odpowiednie definicje wyglądają tak:

```
#define BOOT1_GPIO_N    B
#define BOOT1_PIN_N      2
```

Plik `boot_15x.c` zawiera implementację funkcji `GetBootParams` dla mikrokontrolerów z podrodziny STM32L1xx. Nie wymaga żadnych definicji w pliku `board_def.h` i zwraca ustalone wartości: 32 MHz, `FULL_SPEED`, `USB_PHY_A`.

Z myślą o zestawie STM3220G-EVAL powstał plik *boot_207.c*. Parametry aplikacji ustalane są za pomocą dwóch zwór BOOT1 i BOOT2. BOOT1 to standardowa zwora podłączona do wyprowadzenia PB2. Rolę zwory BOOT2 odgrywa przycisk USER, który podłączony jest do wyprowadzenia PA0. Przycisk USER musi być wcisnięty podczas rozpoczynania aplikacji, jeśli chcemy, aby funkcja *GetBootParams* odczytała, że zwora BOOT2 ustawiona jest w pozycji 1, a niewciśnięty, aby odczytała, że zwora BOOT2 jest w pozycji 0. Kombinacje uzyskiwanych parametrów zestawione są w **tablicy 2.5**. Wyprowadzenia, do których podłączone są zwory, należy jak poprzednio zdefiniować w pliku *board_def.h*. Dla STM3220G-EVAL stosowne definicje wyglądają następująco:

```
#define BOOT1_GPIO_N      B
#define BOOT1_PIN_N        2
#define BOOT2_GPIO_N      A
#define BOOT2_PIN_N        0
```

Tab. 2.5. Opcje konfiguracji zestawu STM3220G-EVAL

BOOT2 (PA0)	BOOT1 (PB2)	sysclk	speed	phy
0	0	48 MHz	FULL_SPEED	USB_PHY_A
0	1	120 MHz	FULL_SPEED	USB_PHY_A
1	0	120 MHz	FULL_SPEED	USB_PHY_ULPI
1	1	120 MHz	HIGH_SPEED	USB_PHY_ULPI

W pewnych przypadkach, gdy na płytce nie ma zwory BOOT1 lub żaden z wyżej wymienionych plików nie jest odpowiedni do zastosowanego typu mikrokontrolera i chcemy wymusić ustalone parametry uruchamiania aplikacji, to do projektu dodajemy plik *boot_72_fs_0.c*, który zawiera implementację funkcji *GetBootParams* niewymagającą żadnych definicji w pliku *board_def.h* i zwracającą ustalone wartości: 72 MHz, FULL_SPEED, USB_PHY_A.

2.4.9. Przerwania

System przerwań w mikrokontrolerach z rdzeniami Cortex-M3 i Cortex-M4 umożliwia bardziej elastyczne konfigurowanie priorytetów większości przerwań. Im mniejsza jest wartość liczbową priorytetu, tym wyższy jest priorytet. Priorytet pewnych krytycznych przerwań nie może być modyfikowany i jest wtedy liczbą ujemną. Są to zerowanie (ang. *reset*) o priorytecie -3, przerwanie niemaskowalne (ang. *non-maskable interrupt*) o priorytecie -2 oraz ciężkie niepowodzenie (ang. *hard fault*) o priorytecie -1. Dla pozostałych przerwań priorytet jest konfigurowalny i jest wtedy czterobitową liczbą nieujemną. Te cztery bity można podzielić na dwa pola: priorytet wywłaszczenia (ang. *preemption priority*) i podpriorytet (ang. *subpriority*). Priorytet wywłaszczenia decyduje o zagnieżdżaniu wywołań procedur obsługujących przerwania. Obsługa przerwania może zostać przerwana (wywłaszczona) tylko przez przerwanie o wyższym priorytecie wywłaszczenia, a zostanie dokonana po obsłudze wszystkich takich przerwań. Podpriorytet rozstrzyga o kolejności wywoływanego procedur obsługi przerwań o jednakowym priorytecie wywłaszczenia. System przerwań pozwala na programowe blokowanie

przerwań o zadany lub niższym priorytecie wywłaszczenia. Aktualny poziom blokowanych przerwań zapisany jest w rejestrze BASEPRI. Wszystkie przerwania o nieujemnym priorytecie wywłaszczenia blokuje się globalnie za pomocą rejestru PRIMASK.

irq.h

W pliku *irq.h* zdefiniowano, że priorytet wywłaszczenia zajmuje dwa bity, a dwa pozostałe bity przeznaczone są na podpriorytet. Umożliwia to używanie maksymalnie czterech priorytetów wywłaszczenia i maksymalnie czterech podpriorytetów. W przykładowych programach wystarczą trzy priorytety wywłaszczenia, których wartości dla czytelności tekstu źródłowego zostały zdefiniowane jako stałe. Im mniejsza wartość liczbową priorytetu, tym wyższy priorytet. Odpowiednie definicje wyglądają tak:

```
#define HIGH_IRQ_PRIO      1U
#define MIDDLE_IRQ_PRIO    2U
#define LOW_IRQ_PRIO       3U
```

W pliku *irq.h* zdefiniowane są też następujące funkcje rozwijane w miejscu wywoływania. Ich implementacja wymaga użycia instrukcji asemblera ARM. Biblioteka CMSIS od wersji 2.00 dostarcza w tym celu wygodnych funkcji rozwijanych w miejscu ich wywoływania. Nie ma więc dłużej potrzeby pisania własnych makr asemblerowych.

```
static inline void IRQprotectionConfigure(void) {
    NVIC_SetPriorityGrouping(7U - PREEMPTION_PRIORITY_BITS);
}
```

Bezparametrowa funkcja `IRQprotectionConfigure` uaktywnia możliwość blokowania przerwań zależnie od ich priorytetu. Funkcję tę należy wywołać, zanim uaktywnimy system przerwań. Nie zwraca ona żadnej wartości. Stała `PREEMPTION_PRIORITY_BITS` ma, zgodnie z tym, co napisano powyżej, wartość 2.

```
static inline irq_level_t IRQprotect(uint32_t priority) {
    irq_level_t level;
    level = __get_BASEPRI();
    __set_BASEPRI(priority << (8U - PREEMPTION_PRIORITY_BITS));
    return level;
}
```

Funkcja `IRQprotect` blokuje przerwania o priorytecie wywłaszczenia niższym lub równym `priority`. Zwraca poprzedni poziom blokowania przerwań. Funkcja `__get_BASEPRI` odczytuje zawartość rejestru `BASEPRI`, funkcja `__set_BASEPRI` zapisuje nową wartość do tego rejestru.

```
static inline void IRQunprotect(irq_level_t level) {
    __set_BASEPRI(level);
}
```

Funkcja `IRQunprotect` przywraca stan blokowania przerwań podany za pomocą argumentu `level`, którego wartość została zwrócona przez funkcję `IRQprotect`. Nie zwraca żadnej wartości.

Zobaczmy przykład użycia tych funkcji. Funkcja `DACputSample` zapisuje próbkę sygnału do cyklicznego bufora, który jest odczytywany w procedurze obsługi przerwania przetwornika cyfrowo-analogowego. Przerwanie to ma nadany wysoki priorytet wywłaszczenia `HIGH_IRQ_PRIO`. Żeby zawartość bufora była zawsze spójna, zapis do bufora musi odbywać się przy zablokowanym przerwaniem przetwornika.

```
static unsigned buffered, writePtr;
static int16_t buffer[DAC_BUFF_SIZE];

void DACputSample(int16_t sample) {
    irq_level_t level;
    level = IRQprotect(HIGH_IRQ_PRIO);
    if (buffered < DAC_BUFF_SIZE) {
        buffer[writePtr] = sample;
        ++buffered;
        if (writePtr < DAC_BUFF_SIZE - 1)
            ++writePtr;
        else
            writePtr = 0;
    }
    IRQunprotect(level);
}
```

System przerwań w Cortex-M0 został nieco uproszczony w porównaniu do Cortex-M3 i Cortex-M4. Nie ma rejestru `BASEPRI` i nie można blokować przerwań zależnie od priorytetu wywłaszczenia. Priorytet wywłaszczenia zajmuje na stałe dwa bity i nie ma podpriorytetów. Z tego powodu, aby zapewnić przenośność programów, dla Cortex-M0 funkcja `IRQprotectionConfigure` jest pusta, a pozostałe dwie funkcje mają nieco inną definicję.

```
static inline irq_level_t IRQprotect(uint32_t priority) {
    irq_level_t level;
    level = __get_PRIMASK();
    __disable_irq();
    return level;
}
```

W funkcji `IRQprotect` wartość parametru `priority` jest ignorowana. Blokowane są wszystkie przerwania o nieujemnym (konfigurowalnym) priorytecie. Aktualny

stan blokowania tych przerwań znajduje się w rejestrze PRIMASK. Funkcja `_get_PRIMASK` odczytuje jego zawartość. Funkcja `_disable_irq` blokuje wszystkie przerwania o konfigurowalnym priorytecie.

```
static inline void IRQunprotect(irq_level_t level) {
    _set_PRIMASK(level);
}
```

Funkcja `_set_PRIMASK` zapisuje do rejestru PRIMASK wartość, która została zwrocona przez funkcję `IRQprotect` i tym samym przywraca poprzedni stan blokowania przerwań.

2.4.10. Wsparcie dla standardowej biblioteki języka C

Standardowa biblioteka języka C dostarcza wiele niezwykle użytecznych funkcji, np. `atoi`, `snprintf`, `strncpy` i wielu, wielu innych. W aplikacjach mikrokontrolerowych stosuje się zwykle w tym celu bibliotekę *Newlib*, która jednak wymaga zaimplementowania specyficznych dla konkretnego sprzętu niskopoziomowych funkcji zarządzających pamięcią i plikami. Funkcje te mogą być częścią systemu operacyjnego. Jeśli nie używamy żadnego systemu operacyjnego, jak w opisywanych w tej książce programach, to musimy zaimplementować potrzebne funkcje sami.

syscalls_dummy.c

Plik `syscalls_dummy.c` zawiera minimum tego, co jest potrzebne, czyli implementację funkcji `_sbrk`, która zarządza stertą, czyli obszarem pamięci, z którego korzystają funkcje `malloc`, `calloc`, `free`. Biblioteka *Newlib* wywołuje funkcję `_sbrk`, gdy brakuje miejsca na stercie, podając jako argument `incr` liczbę bajtów, które chce alokować. Funkcja `_sbrk` sprawdza, czy jest dostatecznie dużo miejsca między pierwszym adresem nad stertą (zmienna `heap_end`) i wierzchołkiem stosu (zmienna `stack_ptr`, zadeklarowana jako synonim rejestru `SP`). Standardowo sterta znajduje się w dolnej części pamięci i rośnie w kierunku większych adresów, a stos umieszczony jest w górnej części pamięci i rośnie w dół. Jeśli jest miejsce na rozszerzenie sterty, to funkcja `_sbrk` ją rozszerza, a w przeciwnym przypadku sygnalizuje błąd braku pamięci (`ENOMEM`). Sterta jest tylko rozszerzana, nigdy zmniejszana. Zmienna `end` wskazuje koniec danych globalnych i statycznych, a tym samym początek sterty. Zmienna `end` jest zdefiniowana w skrypcie konsolidatora. W niektórych implementacjach poprzedza się jej nazwę podkreśleniem, czyli nazywa się ona `_end`. Konsolidator umieszcza zmienną `end` w pamięci tak, aby jej adres był pierwszym wolnym adresem, który może być zajęty przez stertę. Zaraz po uruchomieniu programu wskaźnik `heap_end` ma wartość zero (`NULL`). Przy pierwszym wywołaniu funkcji `_sbrk` wskaźnik `heap_end` jest inicjowany adresem zmiennej `end`, czyli na początku sterty ma zerowy rozmiar.

```
caddr_t _sbrk(int incr) {
    register char *stack_ptr asm(",sp");
    extern char end;
    static char *heap_end;
```

```

char *prev_heap_end;
if (heap_end == NULL)
    heap_end = &end;
if (heap_end + incr > stack_ptr) {
    errno = ENOMEM;
    return (caddr_t)-1;
}
prev_heap_end = heap_end;
heap_end += incr;
return (caddr_t)prev_heap_end;
}

```

syscalls_fs.c

Plik *syscalls_fs.c*, oprócz funkcji *_sbrk*, zawiera implementację funkcji obsługujących system plików: *_open*, *_close*, *_read*, *_write*, *_lseek*, *_fstat* i *_isatty*. Dzięki temu możliwe jest na przykład pisanie na wyświetlaczu ciekłokrystalicznym za pomocą funkcji *printf* i używanie standardowych funkcji *fopen*, *fclose*, *fread*, *fwrite*, *fscanf*, *fprintf* do obsługi plików w pamięci Flash podłączonej do interfejsu USB. Szczegółowy opis pliku *syscalls_fs.c* znajduje się w ostatnim rozdziale, gdyż jest wykorzystywany w omawianym tam projekcie.

2.5. Projekt wstępny

Projekt wstępny, oznaczony numerem 0, demonstruje konfigurowanie częstotliwości taktowania mikrokontrolera oraz obsługę błędów. Korzysta też z diod świecących, a ich miganie jest najbardziej widocznym efektem działania programu. Celem tego projektu jest przedstawienie procedur niezwiązanych bezpośrednio z USB, ale wykorzystywanych we wszystkich następnych projektach. Ponadto umożliwia on szybkie przetestowanie zestawu narzędzi programistycznych (ang. *toolchain*) i sprzętu, czyli adaptera JTAG lub programatora oraz płytka.

ex_led.c

Funkcja main projektu 0 znajduje się w pliku *ex_led.c*. Jej treść przedstawia poniższy wydruk.

```

int main(void) {
    static const unsigned delay_time = 2000000;
    int sysclk;
    GetBootParams(&sysclk, 0, 0);
    AllPinsDisable();
    LEDconfigure();
    ErrorResetable(ClockConfigure(sysclk), 2);
    for (;;) {

```

```
    RedLEDOn();
    Delay(delay_time);
    GreenLEDOn();
    Delay(delay_time);
    RedLEDOff();
    Delay(delay_time);
    GreenLEDOff();
    Delay(delay_time);
}
}
```

Funkcja `GetBootParams` odczytuje, z jaką częstotliwością ma być taktowany rdzeń. Funkcja `AllPinsDisable` ustawia wszystkie porty wejścia-wyjścia w taki sposób, aby pobierały najmniej prądu. Funkcja `LEDconfigure` konfiguruje wyprodadzenia, do których podłączone są diody świecące: zielona i czerwona. Funkcja `ClockConfigure` konfiguruje wszystkie niezbędne podukłady mikrokontrolera związane z dystrybucją sygnałów zegarowych i ustawia żądaną częstotliwość taktowania rdzenia. Funkcja `ErrorResetable` sprawdza, czy wywołanie funkcji `ClockConfigure` zakończyło się powodzeniem. Jeśli wystąpi błąd, czerwona dioda świecąca wykona trzy serie po dwa mignięcia, po czym mikrokontroler zostanie wyzerowany i próba uruchomienia układu zostanie ponowiona. Jeśli konfigurowanie sygnałów zegarowych zakończy się sukcesem, to zielona i czerwona dioda zaczyną migać w nieskończonej pętli. Częstotliwość ich migania zależy od skonfigurowanej częstotliwości taktowania rdzenia.

2.6. Kompilowanie programów

Przykładowe programy można w zasadzie skompilować za pomocą dowolnego narzędzia zawierającego kompilator języka C dla procesorów ARM. Zakładam, że Czytelnik programował już jakieś mikrokontrolery, może nawet ARM-y, więc zapewne ma swoje ulubione środowisko programistyczne i potrafi w nim skonfigurować projekt. Mimo wszystko w tym podrozdziale przedstawiam moją propozycję, jak kompilować przykładowe projekty, korzystając z bezpłatnych narzędzi opartych na kompilatorze z pakietu GCC. Poniższe wskazówki będą też przydatne dla użytkowników innych środowisk programistycznych. Czytelnikowie się z nich, gdzie szukać listy nazw plików potrzebnych do zbudowania poszczególnych programów oraz jak ustawić w swoim środowisku ścieżki poszukiwań plików i parametry kompilacji.

2.6.1. Narzędzia

Przykładowe programy znajdujące się w archiwum kompilowalem za pomocą narzędzi GNU, czyli kompilatora C z pakietu GCC (ang. *GNU Compiler Collection*) oraz pakietu Binutils (ang. *GNU Binary Utilities*), uzupełnionych biblioteką *Newlib*. W dodatku zamieściłem instrukcję, jak pod systemem Linux samodzielnie skonfigurować i zainstalować te pakiety, tak aby możliwie najlepiej dostroić parametry

kompilacji i biblioteki do docelowej architektury, czyli w naszym przypadku do architektury procesorów z rdzeniami ARM Cortex-M. Jeśli ktoś nie lubi instalowania pakietów ze źródeł lub chce kompilować przykładowe programy pod systemem Windows, może skorzystać z jednego z wielu gotowych zestawów narzędzi (ang. *toolchain*) bazujących na GCC i przeznaczonych dla procesorów ARM. W pliku *readme.txt*, który znajduje się w archiwum w katalogu *./make/win*, zamieściłem wskazówki, jak skompilować przykładowe programy pod systemem Windows.

2.6.2. Program make

Archiwum zawiera ponad 24 000 linii tekstu źródłowego, bez bibliotek STM32 i FatFs. Całość podzielona jest na 9 projektów. Każdy z projektów może być skompilowany na różne warianty sprzętu, co daje łącznie kilkadziesiąt przykładowych programów. Żeby je zbudować, trzeba skompilować łącznie około 180 plików źródłowych i nagłówkowych oraz kilka wariantów biblioteki STM32, a także bibliotekę FatFs. Większość plików jest wykorzystywana w więcej niż jednym projekcie. Trudno byłoby to wszystko ogarnąć bez odpowiedniego narzędzia. Najlepiej nadaje się do tego program *make*.

Przedstawione w tym podręczniku skrypty dla programu *make* umożliwiają automatyczne uruchomienie kompilacji wszystkich przykładowych programów oraz komplikację przyrostową po zmodyfikowaniu części plików źródłowych. Bazując na nich, można też stworzyć stosowne skrypty dla własnych projektów. Oprócz tego jest to krótkie, choć z pewnością niewyczerpujące, przedstawienie możliwości programu *make*. Dalszych informacji należy szukać w podręczniku do tego programu.

makefile
makefile.in

Wszystkie instrukcje, które ma wykonać program *make*, opisuje się w plikach *makefile*. Aby uruchomić instrukcje opisane za pomocą pliku *makefile*, należy w linii poleceń, będąc w katalogu, w którym znajduje się ten plik, wpisać polecenie *make*. Dla każdego projektu w archiwum znajduje się co najmniej jeden plik *makefile*. Dla większości projektów takich plików jest kilka, każdy dla innego wariantu sprzętu. Są one umieszczone w katalogach *./make/usb**, gdzie po przedrostku *usb* następuje numer projektu, jego krótka nazwa i oznaczenie wariantu mikrokontrolera, dla którego jest on przeznaczony. Dla przykładu w katalogu *./make/usb0_led_103* znajduje się przedstawiony poniżej plik *makefile* projektu wstępnego numer 0 dla mikrokontrolerów STM32F103.

```
SOURCES = board_init_103.c boot_10x.c delay.c ex_led.c led.c  
TARGETS = led_103.elf led_103.bin  
ifndef HARDWARE  
    HARDWARE = gadget  
endif  
include ../../make/scripts/makefile.in
```

Z pomocą zmiennej `SOURCES` definiuje się listę nazw plików źródłowych, które są niezbędne do zbudowania programu. Na liście tej można umieścić pliki źródłowe w języku C (z rozszerzeniem *c*), pliki źródłowe w asemblerze (z rozszerzeniem *s*), pliki pośrednie (z rozszerzeniem *o*) oraz pliki biblioteczne (z rozszerzeniem *a*).

Zmienna `TARGETS` określa, jakie pliki wynikowe mają być zbudowane i jak mają się nazywać. Może ona zawierać od jednego do trzech plików o tej samej nazwie, ale różnych rozszerzeniach. Jeśli wpiszymy plik z rozszerzeniem *elf*, to zostanie wygenerowany plik w formacie ELF (domyślny format plików binarnych dla narzędzi GNU), zawierający m.in. informacje diagnostyczne. Plik taki jest potrzebny, jeśli chcemy debugować program na przykład za pomocą GDB. Jeśli potrzebujemy pliku zawierającego binarny obraz pamięci, to wpisujemy plik z rozszerzeniem *bin* – wiele narzędzi do programowania pamięci Flash wymaga pliku w takim formacie. Jeśli chcemy, aby został wygenerowany plik w formacie Intel HEX (chyba najstarszy format używany do programowania pamięci nieulotnych), to wpisujemy jego nazwę z rozszerzeniem *hex*.

W celu rozróżnienia plików wynikowych skompilowanych dla różnych wariantów sprzętu nazwy plików wynikowych powstają przez doklejenie przedrostka zdefiniowanego za pomocą zmiennej `HARDWARE` i znaku podkreślenia do nazw zdefiniowanych w zmiennej `TARGETS`. Zmienna `HARDWARE` określa typ sprzętu, na który ma być skompilowany program. Dzięki temu, że jej definicja poprzedzona jest sprawdzeniem, czy nie została już wcześniej zdefiniowana, możemy nadpisać jej wartość w wierszu poleceń, na przykład tak:

```
make HARDWARE=z131arm
```

Wtedy zamiast plików *gadget_led_103.bin* i *gadget_led_103.elf* zostaną utworzone pliki *z131arm_led_103.bin* i *z131arm_led_103.elf*.

Właściwa komplikacja uruchamiana jest za pomocą skryptu *makefile.in*, który jest włączany na końcu pliku *makefile* i który znajduje się w katalogu */make/scripts*. Poniżej opisuję szczegółowo instrukcje i reguły zapisane w tym skrypcie. Na początku definiujemy nazwy potrzebnych programów narzędziowych: preprocesor, kompilator i konsolidator – zmienna `CC`, asembler – zmienna `AS`, program archiwizujący do tworzenia bibliotek – zmienna `AR`, program do konwersji formatu plików wynikowych – zmienna `OBJCOPY`. Preprocesor i konsolidator są wywoływanie niewidzialnie przez wywołanie kompilatora z odpowiednimi parametrami. Nazwy programów narzędziowych dla procesorów ARM zaczynają się zwykle od przedrostka *arm-none-eabi* lub *arm-elf*. Zależy to od używanego zestawu narzędzi. Właściwy przedrostek (z łącznikiem oddzielającym go od właściwej nazwy programu) definiujemy za pomocą zmiennej `TOOLCHAIN`. Żeby skorzystać z wartości zmiennej, jej nazwę należy ująć w nawiasy i poprzedzić znakiem dolara.

```
TOOLCHAIN = arm-elf-
# TOOLCHAIN = arm-none-eabi-
CC      = $(TOOLCHAIN) gcc
AS      = $(TOOLCHAIN) as
AR      = $(TOOLCHAIN) ar
OBJCOPY = $(TOOLCHAIN) objcopy
```

Następnie definiujemy zmienne przechowujące nazwy katalogów, które będą używane w dalszych poleceniach skryptu. Dzięki temu, że nazwy te są zgromadzone w jednym miejscu, łatwo można zmienić lokalizację plików, bez pracowitego i po-datnego na błędy edytowania całego skryptu. Ścieżki podaje się względem katalogu, z którego uruchamiany jest program `make`, czyli względem katalogów `./make/usb*`, w których znajdują się skrypty `makefile`. Zmienna `PRJDIR` wskazuje na katalog z plikami specyficznymi dla pewnej grupy projektów. Jest ona definiowana warunkowo, aby można ją było łatwo przedefiniować w pliku `makefile` dla danego projektu. Tu znajduje się jej wartość domyślna dla projektów prezentowanych w tej książce, wskazująca na katalog `./exusb`. Zmienna `COMMDIR` wskazuje, gdzie znajdują się pliki źródłowe, tworzące swego rodzaju bibliotekę i wspólne dla wszystkich projektów lub wykorzystywane w większości projektów. Zmienna `STLIBDIR` zawiera nazwę katalogu, gdzie należy szukać bibliotek STM32. Zmienna `FFLIBDIR` zawiera nazwę katalogu, gdzie należy szukać biblioteki FatFs. Zmienna `LIBDIR` zawiera nazwę katalogu roboczego, gdzie będą umieszczone pliki pośrednie (z rozszerzeniem *o*) i biblioteczne (z rozszerzeniem *a*). Nazwa tego katalogu jest inna dla każdego projektu i wariantu sprzętu w obrębie tego projektu, co umożliwia przechowywanie wielu skompilowanych z różnymi parametrami wersji tego samego pliku źródłowego. Funkcja `notdir` wycina z nazwy ścieżki nazwę katalogu, pozostawiając tylko jej fragment po ostatnim ukośniku.

```
ifndef PRJDIR
    PRJDIR = ../../exusb
endif
COMMDIR = ../../common
STLIBDIR = ../../libraries/stm32
FFLIBDIR = ../../libraries/fatfs
LIBDIR = ../../make/lib/$(notdir $(PRJDIR))/$(HARDWARE)
```

Następna instrukcja wywołuje powłokę systemu operacyjnego (ang. *shell*) w celu utworzenia katalogu `LIBDIR` za pomocą polecenia `mkdir`. Instrukcja ta działa tylko w systemach uniksowych. W systemach Windows zapewne spowoduje wypisanie komunikatu „składnia polecenia jest niepoprawna”, który należy zignorować, a odpowiedni katalog utworzyć, wywołując polecenie `make` z parametrem `mkdir` (patrz niżej).

```
$(shell mkdir -p $(LIBDIR))
```

Niektóre dalsze instrukcje definiują, z jakimi parametrami ma być wywołany pre-procesor, kompilator i konsolidator. Znaczenie wybranych parametrów jest wyjaśnione wspólnie w tabeli 2.6.

Kolejna grupa instrukcji, zależnie od wariantu sprzętu, definiuje w zmiennej `STFLAGS` stałe preprocesora oraz nazwę mikrokontrolera w zmiennej `DEVICE`:

- stała `USE_STDPERIPH_DRIVER` deklaruje, że do obsługi peryferii będziemy używać biblioteki STM32;

- stała DSTM32F10X_HD, DSTM32F2XX lub tym podobna określa, do której linii mikrokontrolerów należy zastosowany układ – jest niezbędna do prawidłowego skompilowania biblioteki STM32 (zależy od niej m.in. tablica przerwań);
- stała HSE_VALUE określa częstotliwość zastosowanego rezonatora kwarcowego w hercach lub częstotliwość zewnętrznego generatora kwarcowego podłączonego do wyprowadzenia OSC_IN (zero, gdy mikrokontroler ma być taktowany z zewnętrznego generatora RC HSI).

```

ifeq ($(HARDWARE),gadget)
    STFLAGS = -DUSE_STDEPERIPH_DRIVER -DSTM32F10X_MD -DHSE_VALUE=8000000
    DEVICE  = stm32f103t8
else ifeq ($(HARDWARE),mmstm32f103vx-0-0-0)
    STFLAGS = -DUSE_STDEPERIPH_DRIVER -DSTM32F10X_HD -DHSE_VALUE=8000000
    DEVICE  = stm32f103vc
else ifeq ($(HARDWARE),stm3220g-eval)
    STFLAGS = -DUSE_STDEPERIPH_DRIVER -DSTM32F2XX -DHSE_VALUE=25000000
    DEVICE  = stm32f207ig
else ifeq ($(HARDWARE),stm32f0-discovery)
    STFLAGS = -DUSE_STDEPERIPH_DRIVER -DSTM32F0XX -DHSE_VALUE=0
    DEVICE  = stm32f051r8
else ifeq ($(HARDWARE),stm32f4-discovery)
    STFLAGS = -DUSE_STDEPERIPH_DRIVER -DSTM32F4XX -DHSE_VALUE=8000000
    DEVICE  = stm32f407vg
else ifeq ($(HARDWARE),stm32l-discovery)
    STFLAGS = -DUSE_STDEPERIPH_DRIVER -DSTM32L1XX_MD -DHSE_VALUE=8000000
    DEVICE  = stm32l152rb
else ifeq ($(HARDWARE),z129arm)
    STFLAGS = -DUSE_STDEPERIPH_DRIVER -DSTM32F10X_CL -DHSE_VALUE=10000000
    DEVICE  = stm32f107vc
else ifeq ($(HARDWARE),z130arm)
    STFLAGS = -DUSE_STDEPERIPH_DRIVER -DSTM32F10X_MD -DHSE_VALUE=8000000
    DEVICE  = stm32f103cb
else ifeq ($(HARDWARE),z131arm)
    STFLAGS = -DUSE_STDEPERIPH_DRIVER -DSTM32F10X_MD -DHSE_VALUE=8000000
    DEVICE  = stm32f103rb
endif

```

Następna grupa instrukcji na podstawie nazwy mikrokontrolera wyznacza, do której podrodziny (zmienna STFAMILY) on należy i jaka jest wersja architektury rdzenia, co przekłada się na definicję parametrów komplikacji (zmienna FLAGS). Dla wszystkich podrodzin ustawiamy parametr -mthumb, gdyż rdzenie Cortex-M wykonują tylko zestaw instrukcji Thumb-2. Dla rdzeni Cortex-M0 ustawiamy parametr -mcpu=cortex-m0. Dla rdzeni Cortex-M3 ustawiamy parametr -mcpu=cortex-m3. Natomiast

dla rdzeni Cortex-M4, oprócz parametru `-mcpu=cortex-m4`, ustawiamy jeszcze sposób generowania kodu zmiennoprzecinkowego (parametr `-mfloating-point-abi=softfp`) i typ jednostki zmiennoprzecinkowej zastosowanej w tym rdzeniu (parametr `-mfpu=fpv4-sp-d16`).

Tab. 2.6. Wybrane parametry komplikacji

Parametr	Opis
<code>-D id</code>	Definiuje identyfikator <code>id</code> , tak jakby na początku wszystkich plików źródłowych występowała instrukcja <code>#define id</code>
<code>-D id=napis</code>	Definiuje identyfikator <code>id</code> , tak jakby na początku wszystkich plików źródłowych występowała instrukcja <code>#define id napis</code>
<code>-I path</code>	Doląca ścieżkę <code>path</code> do listy miejsc, gdzie poszukiwane są pliki nagłówkowe, włączane za pomocą instrukcji <code>#include</code>
<code>-O0</code>	Włącza wszelkie optymalizacje
<code>-Os</code>	Włącza optymalizacje zmniejszające rozmiar kodu wynikowego
<code>-O1</code>	Włącza optymalizacje zwiększące szybkość wykonywania kodu. Włącza również te optymalizacje zmniejszające rozmiar kodu wynikowego, które nie spowalniają jego wykonania. Większe wartości oznaczają bardziej agresywną optymalizację
<code>-O2</code>	
<code>-O3</code>	
<code>-ffunction-sections</code>	Umieszcza każdą funkcję w osobnej sekcji, dzięki czemu konsolidator może usunąć z kodu wynikowego funkcje, które nigdy nie są wywoływane. Wpływa na zmniejszenie rozmiaru kodu wynikowego
<code>-fdata-sections</code>	Umieszcza każdą zmienną globalną i statyczną w osobnej sekcji, dzięki czemu konsolidator może usunąć z kodu wykonywalnego zmienne, do których nie ma odwołań. Wpływa na zmniejszenie rozmiaru kodu wynikowego
<code>-Wall</code>	Włącza wypisywanie ostrzeżeń (GCC)
<code>-warn</code>	Włącza wypisywanie ostrzeżeń (GAS – GNU Assembler)
<code>-g</code>	Powoduje dołaczanie do pliku wynikowego informacji diagnostycznych, pomocnych przy poszukiwaniu błędów. Informacje diagnostyczne są wykorzystywane przez debugger, np. GDB, ale nie są zapisywane do pamięci Flash, więc nie zwiększa jej zajętości
<code>-c</code>	Nakazuje przeprowadzenie tylko komplikacji do pliku pośredniego (z rozszerzeniem <code>.o</code>)
<code>-o file</code>	Specyfikuje nazwę pliku wynikowego
<code>-mthumb</code>	Generuje kod maszynowy z użyciem zestawu instrukcji Thumb lub Thumb-2, zależnie od innych opcji specyfikujących architekturę procesora
<code>-mcpu=cortex-m3</code>	Generuje kod maszynowy z użyciem instrukcji dostępnych w architekturze Cortex-M3
<code>-mcpu=cortex-m4</code>	Generuje kod maszynowy z użyciem instrukcji dostępnych w architekturze Cortex-M4
<code>-mfloating-point-abi=softfp</code>	Deklaruje sposób przekazywania argumentów zmiennoprzecinkowych. Sprawia też, że operacje zmiennoprzecinkowe będą realizowane programowo lub sprzętowo, zależnie od dostępności i możliwości jednostki zmiennoprzecinkowej
<code>-mfpu=fpv4-sp-d16</code>	Specyfikuje jednostkę zmiennoprzecinkową, dla której mają być generowane instrukcje. Wartość <code>fpv4-sp-d16</code> oznacza jednostkę zmiennoprzecinkową pojedynczej precyzji dostępna w rdzeniach Cortex-M4 (oznaczanych też Cortex-M4F)
<code>-nostartfiles</code>	Włącza konsolidowanie domyślnego pliku z procedurą startową. Należy dostarczyć własny plik startowy. W przykładowych programach jest to plik <code>startup_stm32.c</code>
<code>-L path</code>	Doląca ścieżkę <code>path</code> do listy miejsc, gdzie poszukiwane są biblioteki
<code>-l abc</code>	Nakazuje konsolidatorowi dołączyć bibliotekę, która jest w pliku <code>/libabc.a</code>
<code>-T script</code>	Specyfikuje nazwę skryptu sterującego procesem konsolidacji
<code>-Wl,--gc-sections</code>	Uaktyniwia odśmiecanie na etapie konsolidacji, czyli usuwanie zbędnych sekcji z kodu wynikowego. Aby odśmiecanie było skuteczne, podczas komplikowania musi być ustawniona przynajmniej jedna z opcji <code>-ffunction-sections</code> lub <code>-fdata-sections</code>

```

ifneq (,$(findstring stm32f0, $(DEVICE)))
    STFAMILY = STM32F0xx
    FLAGS    = -mthumb -mcpu=cortex-m0
else ifneq (,$(findstring stm32f10, $(DEVICE)))
    STFAMILY = STM32F10x
    FLAGS    = -mthumb -mcpu=cortex-m3
else ifneq (,$(findstring stm32f2, $(DEVICE)))
    STFAMILY = STM32F2xx
    FLAGS    = -mthumb -mcpu=cortex-m3
else ifneq (,$(findstring stm32f30, $(DEVICE)))
    STFAMILY = STM32F30x
    FLAGS    = -mthumb -mcpu=cortex-m4 -mfloating-point=softfp
                -mfpu=fpv4-sp-d16
else ifneq (,$(findstring stm32f31, $(DEVICE)))
    STFAMILY = STM32F30x
    FLAGS    = -mthumb -mcpu=cortex-m4 -mfloating-point=softfp
                -mfpu=fpv4-sp-d16
else ifneq (,$(findstring stm32f37, $(DEVICE)))
    STFAMILY = STM32F37x
    FLAGS    = -mthumb -mcpu=cortex-m4 -mfloating-point=softfp
                -mfpu=fpv4-sp-d16
else ifneq (,$(findstring stm32f38, $(DEVICE)))
    STFAMILY = STM32F37x
    FLAGS    = -mthumb -mcpu=cortex-m4 -mfloating-point=softfp
                -mfpu=fpv4-sp-d16
else ifneq (,$(findstring stm32f4, $(DEVICE)))
    STFAMILY = STM32F4xx
    FLAGS    = -mthumb -mcpu=cortex-m4 -mfloating-point=softfp
                -mfpu=fpv4-sp-d16
else ifneq (,$(findstring stm32l1, $(DEVICE)))
    STFAMILY = STM32L1xx
    FLAGS    = -mthumb -mcpu=cortex-m3
endif

```

Poniższa grupa instrukcji, na podstawie dotychczas zdefiniowanych zmiennych, definiuje parametry preprocesora (zmienna `CPPFLAGS`), kompilatora (zmienna `CFLAGS`), asemblera (zmienna `ASFLAGS`) i konsolidatora (zmienna `LDFLAGS`). Należy zwrócić uwagę na uzależnienie wyboru katalogów z plikami nagłówkowymi i skryptu konsolidatora od wariantu sprzętu.

```

CPPFLAGS += $(STFLAGS)
CFLAGS    += $(FLAGS) -O2 -c -Wall -g \
                -ffunction-sections -fdata-sections \

```

```

-I$(COMMDIR)/include \
-I$(COMMDIR)/src \
-I$(PRJDIR)/include \
-I$(PRJDIR)/include/$(HARDWARE) \
-I$(STLIBDIR)/$(STFAMILY)_StdPeriph_Driver/inc \
-I$(STLIBDIR)/CMSIS/Include \
-I$(STLIBDIR)/CMSIS/Device/ST/$(STFAMILY)/Include \
-I$(FFLIBDIR)/include

ASFLAGS += $(FLAGS) -g -warn
LDFLAGS += $(FLAGS) -nostartfiles -L$(LIBDIR) -L$(COMMDIR)/scripts \
-T$(DEVICE).lds -Wl,--gc-sections

```

Zadaniem kolejnej grupy instrukcji jest zidentyfikowanie wszystkich typów plików źródłowych. Do każdego projektu trzeba dołączyć procedurę startową oraz bibliotekę STM32. Dlatego najpierw do zmiennej `SOURCES` dodajemy nazwę pliku `startup_stm32.c` oraz nazwę właściwej wersji biblioteki STM32. Następnie filtryujemy nazwy plików. Po tych operacjach zmienna `CSRCS` zawiera nazwy plików źródłowych w języku C, czyli nazwy z rozszerzeniem *c*. Zmienna `OBJECTS` zawiera nazwy plików pośrednich (z rozszerzeniem *o*), które mają powstać z plików źródłowych w języku C i w asemblerze oraz tych, które mają zostać dołączone bezpośrednio. Zmienna `LIBRARIES` zawiera nazwy bibliotek, które trzeba dołączyć, czyli nazwy plików z rozszerzeniem *a*. Zmienna `STLIBSRCS` zawiera nazwy plików źródłowych w języku C potrzebnych do zbudowania biblioteki STM32. Zmienna `STLIBOBJJS` zawiera odpowiadające im nazwy plików pośrednich. Zmienna `FFLIBSRCS` zawiera nazwy plików źródłowych w języku C potrzebnych do zbudowania biblioteki FatFs. Zmienna `FFLIBOBJJS` zawiera odpowiadające im nazwy plików pośrednich. Jako wartość zmiennej `STDCLIBS` wpisujemy listę nazw plików z bibliotekami, które ewentualnie chcemy dołączać do komplikowanych programów, ale których nie musimy budować, gdyż są częścią standardowej biblioteki języka C dostarczanej zwykle wraz ze środowiskiem programistycznym. Wymieniony tu plik `libm.a` zawiera bibliotekę funkcji matematycznych.

```

SOURCES += startup_stm32.c lib$(STFAMILY).a
CSRCS = $(filter %.c, $(SOURCES))
OBJECTS = $(patsubst %.s, %.o, $(filter %.s, $(SOURCES))) \
$(patsubst %.c, %.o, $(filter %.c, $(SOURCES))) \
$(filter %.o, $(SOURCES))
LIBRARIES = $(filter %.a, $(SOURCES))
STLIBSRCS = $(wildcard $(STLIBDIR)/$(STFAMILY)_StdPeriph_Driver/
src/*.c)
STLIBOBJJS = $(patsubst %.c, %.o, $(notdir $(STLIBSRCS)))
FFLIBSRCS = $(wildcard $(FFLIBDIR)/src/*.c)
FFLIBOBJJS = $(patsubst %.c, %.o, $(notdir $(FFLIBSRCS)))
STDCLIBS = libm.a

```

Poniższa grupa poleceń instruuje program `make`, gdzie ma szukać poszczególnych typów plików: bibliotecznych (rozszerzenie *a*), źródłowych w języku C (rozszerzenie *c*), pośrednich (rozszerzenie *o*) i źródłowych w asemblerze (rozszerzenie *s*).

```
vpath %.a $(LIBDIR)
vpath %.c $(COMMDIR)/src \
    $(PRJDIR)/src \
    $(STLIBDIR)/$(STFAMILY)_StdPeriph_Driver/src \
    $(FFLIBDIR)/src
vpath %.o $(LIBDIR)
vpath %.s $(COMMDIR)/src \
    $(PRJDIR)/src
```

Pierwsza z poniższych instrukcji mówi, że napisy `all`, `clean` i `mkdir` są tylko nazwami argumentów polecenia `make` i nie powinny być traktowane jak nazwy plików, które trzeba wygenerować. Druga z poniższych instrukcji określa, że pliki pośrednie, z których buduje się biblioteki STM32 i FatFs, są tymczasowe i mają zostać skasowane po ich utworzeniu. Biblioteki te zmieniają się bardzo rzadko, więc w praktyce wystarczy je zbudować tylko raz dla każdego wariantu sprzętu i nie trzeba przechowywać plików pośrednich.

```
.PHONY : all clean mkdir
.INTERMEDIATE : $(STLIBOBJS) $(FFLIBOBJS)
```

Wiersz zaczynający się od napisu `all` określa, co ma się stać po wywołaniu polecenia `make` bez argumentów (lub ewentualnie z argumentem `all`). W tym przypadku oczekujemy, że zostaną utworzone wszystkie pliki, których nazwy znajdują się po prawej stronie dwukropka. Lista tych nazw powstaje przez doklejenie z przodu do wszystkich nazw z listy znajdującej się w zmiennej `TARGETS` napisu znajdującego się w zmiennej `HARDWARE` i znaku podkreślenia.

```
all : $(addprefix $(HARDWARE)_, $(TARGETS))
```

Jeśli polecenie `make` nie zostało wywołane z argumentem `clean` lub `mkdir` (patrz niżej), to poniższy fragment skryptu włącza trzy pliki (z rozszerzeniem *d*) opisujące zależności (ang. *dependencies*) wynikające z włączania plików za pomocą instrukcji `#include`. Pierwszy uwzględnia zależności dla biblioteki STM32, drugi dla biblioteki FatFs (ale tylko wtedy, gdy jest ona na liście plików potrzebnych do zbudowania pliku wynikowego), a trzeci dla pozostałych plików źródłowych. Dzięki temu można stosować komplikację przyrostową. Modyfikacja któregoś z plików nagłówkowych wymusza ponowne komplikowanie tylko tych plików, które bezpośrednio lub pośrednio włączają ten plik nagłówkowy.

```
ifneq ($(MAKECMDGOALS),clean)
ifneq ($(MAKECMDGOALS),mkdir)
    include $(LIBDIR)/$(STFAMILY).d
    ifneq (,$(findstring libff.a, $(LIBRARIES)))
        include $(LIBDIR)/ff.d
```

```

endif
include $(HARDWARE).d
endif
endif

```

Szereg następnych linii skryptu opisuje reguły, jak wygenerować poszczególne pliki. Każda z tych reguł składa się z dwóch części: listy zależności w pierwszym wierszu oraz polecenia w drugim i ewentualnie kolejnych wierszach. Lista zależności zawiera oddzielone dwukropkami listy nazw plików lub wzorce nazw plików. Przed pierwszym dwukropkiem specyfikuje się nazwę pliku wynikowego (lub wzorzec nazwy pliku), który ma zostać wygenerowany. Po dwukropku wymienia się nazwy plików źródłowych, które są potrzebne do utworzenia pliku wynikowego. Program make porównuje czasy ostatniej modyfikacji plików źródłowych z czasem modyfikacji pliku wynikowego. Jeśli plik wynikowy nie istnieje lub jest starszy, uruchamiane jest polecenie. Wiersze z poleceniem muszą rozpoczynać się tabulatorem. Jeśli polecenie jest kontynuowane w kolejnym wierszu, to poprzedni wiersz musi być zakończony lewym ukośnikiem (ang. *backslash*).

Poniższy fragment skryptu opisuje reguły generowania plików z zależnościami. Przed dwukropkiem jest nazwa pliku, który ma zostać utworzony. W drugim wierszu znajduje się odpowiednie polecenie preprocesora. Zmienna \$@ jest synonimem nazwy generowanego pliku wynikowego. Dwa pierwsze pliki z zależnościami, czyli pliki dla bibliotek STM32 i FatFs, zostaną umieszczone w tym samym katalogu, w którym ma też być umieszczona skompilowana biblioteka. Plik dla biblioteki FatFs zostanie utworzony tylko wtedy, gdy znajduje się ona na liście plików źródłowych. Trzeci plik z zależnościami dla pozostałych plików źródłowych znajdzie się w bieżącym katalogu, w którym wywołano polecenie make.

```

$(LIBDIR) / $(STFAMILY).d :
    $(CC) -MM $(CPPFLAGS) $(CFLAGS) $(STLIBSRCS) > $@
ifneq (,$(findstring libff.a, $(LIBRARIES)))
$(LIBDIR)/ff.d :
    $(CC) -MM $(CPPFLAGS) $(CFLAGS) $(FFLIBSRCS) > $@
endif

$(HARDWARE).d :
    $(CC) -MM $(CPPFLAGS) $(CFLAGS) \
        $(wildcard $(CSRCS) $(addprefix $(COMMDIR)/src/, $(CSRCS)) \
        $(addprefix $(PRJDIR)/src/, $(CSRCS))) > $@

```

Kolejne dwie linie zawierają regułę, jak zbudować bibliotekę STM32. Napis po lewej stronie dwukropka w pierwszej linii to nazwa pliku wynikowego, w którym ma zostać zapisana tworzona biblioteka. Napis po prawej stronie dwukropka w pierwszej linii to lista plików pośrednich, które są niezbędne do jej utworzenia i które ewentualnie też trzeba wcześniej utworzyć, korzystając z kolejnych reguł. W drugiej linii znajduje się polecenie (nazwa programu wraz z parametrami), które trzeba wywołać, aby połączyć pliki pośrednie w jeden plik wynikowy. Zmienna \$@

jest synonimem nazwy pliku wynikowego. Zmienna $\$^$ jest synonimem listy nazw plików wymienionych w pierwszej linii po dwukropku.

```
lib$(STFAMILY).a : $(STLIBOJJS)
$(AR) -rcs $(LIBDIR)/$@ $^
```

Następne cztery linie zawierają analogiczną regułę budowania biblioteki FatFs, ale tylko wtedy, gdy jest ona potrzebna.

```
ifneq (,$(findstring libff.a, $(LIBRARIES)))
libff.a : $(FFLIBOJJS)
$(AR) -rcs $(LIBDIR)/$@ $^
endif
```

Poniższe dwie linie zawierają regułę, jak tworzyć pliki pośrednie (z rozszerzeniem o) biblioteki STM32. Napis w pierwszej linii przed pierwszym dwukropkiem jest listą nazw plików, które trzeba wygenerować. Dalej reguła mówi, że plik z rozszerzeniem o generuje się na podstawie pliku, który ma tę samą część nazwy przed kropką, ale rozszerzenie c . W drugiej linii znajduje się odpowiednie wywołanie kompilatora. Zmienna $\$<$ jest synonimem nazwy pliku źródłowego. Zmienna $\$@$ jest synonimem nazwy pliku wynikowego, który w tym przypadku jest plikiem pośrednim.

```
$(STLIBOJJS) : %.o : %.c
$(CC) $(CPPFLAGS) $(CFLAGS) $< -o $@
```

Następne cztery linie zawierają analogiczną regułę tworzenia plików pośrednich biblioteki FatFs, ale znów tylko wtedy, gdy są one potrzebne.

```
ifneq (,$(findstring libff.a, $(LIBRARIES)))
$(FFLIBOJJS) : %.o : %.c
$(CC) $(CPPFLAGS) $(CFLAGS) $< -o $@
endif
```

Kolejne dwie linie zawierają ogólną regułę generowania pliku pośredniego (z rozszerzeniem o) na podstawie pliku źródłowego w języku C (z rozszerzeniem c). Reguła ta zostanie zastosowana do wszystkich plików pośrednich, które nie zostały wygenerowane za pomocą poprzedniej reguły. Zmienna $\$<$ jest synonimem nazwy pliku źródłowego. Zmienna $\$@$ jest tu synonimem nazwy pliku pośredniego.

```
%.o : %.c
$(CC) $(CPPFLAGS) $(CFLAGS) $< -o $(LIBDIR)/$@
```

Następne dwie linie zawierają ogólną regułę generowania pliku pośredniego (z rozszerzeniem o) na podstawie pliku źródłowego napisanego w asemblerze (z rozszerzeniem s). Archiwum z przykładami nie zawiera plików w asemblerze. Reguła ta została napisana dla pokazania takiej możliwości.

```
%.o : %.s
$(AS) $(ASFLAGS) $< -o $(LIBDIR)/$@
```

Poniższe trzy linie to instrukcja, jak skonsolidować pliki pośrednie i biblioteczne do pliku wynikowego w formacie ELF. Napis po prawej stronie dwukropka w pierwszej linii jest listą nazw plików, które są potrzebne do zbudowania pliku wynikowego. Nie umieszczamy na tej liście nazw plików standardowej biblioteki języka C. Nazwy te usuwa z listy funkcja `filter-out`. Należy zwrócić uwagę, że w celu dołączenia biblioteki trzeba, podając jej nazwę w parametrze `-l`, pominąć przedrostek `lib` i rozszerzenie `a` wraz z kropką. Przykładowo, aby dołączyć bibliotekę `libabc.a`, należy podać parametr `-labc`. Część nazwy przed kropką wyłuskuje się za pomocą funkcji `basename`, zamianę przedrostka na nazwę parametru wykonuje funkcja `subst`.

```
%.elf : $(OBJECTS) $(filter-out $(STDCLIBS), $(LIBRARIES))
        $(CC) $(LDFLAGS) $(addprefix $(LIBDIR)/, $(OBJECTS)) \
        $(subst lib, -l, $(basename $(LIBRARIES))) -o $@
```

Poniższe linie instruują, jak wygenerować plik w formacie Intel HEX z pliku w formacie ELF.

```
% .hex : %.elf
        $(OBJCOPY) $< $@ -O ihex
```

Poniższe linie instruują, jak wygenerować binarny obraz pamięci na podstawie pliku w formacie ELF.

```
% .bin : %.elf
        $(OBJCOPY) $< $@ -O binary
```

Wiersz zaczynający się od `clean` określa, co ma się stać po wywołaniu polecenia `make` z argumentem `clean`. W tym przypadku usuwane są z bieżącego katalogu wszystkie pliki wynikowe, z zależnościami komplikacji, pośrednie i zapasowe kopie utworzone przez edytor tekstu.

```
clean :
        rm -f *.bin *.elf *.hex *.d *.o *.bak *~
```

Wiersz zaczynający się od `mkdir` zawiera regułę pozwalającą utworzyć katalog roboczy w systemach Windows. Aby uzyskać poprawną nazwę ścieżki w tych systemach, w parametrze przekazywanym poleceniu `mkdir` prawe ukośniki są zamieniane na lewe. Jest to ostatnia instrukcja w pliku `makefile.in`.

```
mkdir :
        mkdir $(subst /,\,$(LIBDIR))
```

Przedstawiony tu skrypt nie jest idealny. Dodanie, usunięcie lub zmiana nazwy pliku źródłowego na liście `SOURCES` wymaga wykonania najpierw polecenia `make clean`, a dopiero potem `make`. Podobnie należy postąpić, jeśli w którymś pliku źródłowym dodamy, usuniemy lub zmienimy instrukcję `#include`. Mimo tej drobnej niedoskonałości i tak ponownie skompilowane zostaną tylko absolutnie niezbędne pliki. W systemach Windows przed wywołaniem `make` trzeba wywołać `make` z parametrem `mkdir`. Ponadto w systemach Windows, zależnie od używanego zestawu narzędzi, polecenie `make` może być dostępne po inną nazwą – należy zatrzymać się do

dokumentacji posiadanej zestawu narzędzi. Patrz też plik `./make/win/readme.txt` znajdujący się w archiwum z przykładami.

Oprócz plików `makefile` w katalogach `./make/usb*`, w katalogu `./make/scripts` znajduje się globalny plik `makefile` dla systemów uniksowych. Wykonanie w tym katalogu polecenia `make` powoduje uruchomienie poleceń `make` we wszystkich katalogach `./make/usb*`, czyli zbudowanie wszystkich programów przykładowych. Wykonanie w tym katalogu polecenia `make clean` wywołuje to polecenie we wszystkich katalogach `./make/usb*`. Polecenie `make clean-all` wywołane w katalogu `./make/scripts` wykonuje to samo, co polecenie `make clean`, a dodatkowo usuwa katalog `/lib` wraz z całą jego zawartością.

2.6.3. Skrypt konsolidatora

```
cortex-m3.lds  
stm32*.lds
```

Dla każdego modelu mikrokontrolera w katalogu `./common/scripts` znajduje się odpowiedni plik dla konsolidatora. Nazwa tego pliku składa się z pełnego oznaczenia modelu mikrokontrolera, jednoznacznie identyfikującego jego wyposażenie (pamięci, peryferie), uzupełnionego rozszerzeniem `lds`: `stm32f103cb.lds`, `stm32f107vc.lds`, `stm32l152rb.lds` itp. Dla przykładu zawartość pliku `stm32f103cb.lds` wygląda następująco:

```
MEMORY
{
    FLASH (rx) : ORIGIN = 0x8000000, LENGTH = 128K
    RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 20K
}
_minimum_stack_and_heap_size = 2560;
INCLUDE cortex-m3.lds
```

Bardzo łatwo jest przygotować taki skrypt dla dowolnego modelu mikrokontrolera. Na początku definiuje się adresy początkowe i rozmiary pamięci Flash i RAM. Odpowiednich danych należy poszukać w dokumentacji. Następnie definiuje się stałą określającą minimalny rozmiar pamięci, która musi pozostać na stos i stertę. W powyższym przykładzie jest to 1/8 rozmiaru RAM. Konsolidator zgłosi błąd, gdy w RAM zostanie za mało miejsca. Zasadniczym procesem konsolidacji steruje plik `cortex-m3.lds` włączany na końcu. Wbrew nazwie jest to skrypt przeznaczony nie tylko dla rdzeni Cortex-M3, ale też Cortex-M0 i Cortex-M4. Szczegółowy opis tego pliku znajduje się w punkcie 1.8.3 innej mojej książki [2]. Dlatego też postanowiłem nie zmieniać jego nazwy.

2.7. Uruchamianie przykładowych programów

W tym podrozdziale umieściłem garść wspólnych wskazówek dotyczących testowania przykładowych programów i programowania pamięci Flash, ale z rozbiciem na dwa najpopularniejsze systemy operacyjne. Inne bardziej specyficzne uwagi zamieszczam przy omawianiu poszczególnych projektów.

2.7.1. Uwagi dla użytkowników systemu Linux

W systemie operacyjnym Linux dostęp do wszelkich urządzeń (w tym także USB) jest domyślnie zastrzeżony dla uprzywilejowanego użytkownika z uprawnieniami administratora (ang. *root*), który jest właścicielem pseudoplików reprezentujących urządzenie. Aby zezwolić innym użytkownikom na swobodne korzystanie z urządzeń USB i ich testowanie, należy skorzystać z programu udev. Jak to zwykle bywa w Linuksie, wszystko konfiguruje się za pomocą odpowiedniego skryptu. W archiwum w katalogu */make/linux* znajdują się trzy przykładowe skrypty:

- *10-openocd.rules* – reguły dla adapterów JTAG-USB i programatorów (np. ST-LINK) obsługiwanych przez program OpenOCD,
- *11-stm32.rules* – ogólne reguły dla przykładowych urządzeń opisanych w tej książce,
- *12-cdc-acm.rules* – szczególna reguła dla przykładowego urządzenia implementującego wirtualny port szeregowy.

Skrypty te można dostosować do własnych potrzeb. Najważniejsze miejsca, gdzie trzeba zmodyfikować wartości ujęte w cudzysłów, wraz z przykładowymi wartościami, są następujące:

- ATTRS{idVendor}=="0483" – identyfikator producenta VID,
- ATTRS{idProduct}=="5750" – identyfikator produktu PID,
- MODE="664" – uniksowa maska uprawnień,
- GROUP="users" – nazwa grupy użytkowników, którzy mają być uprawnieni do posługiwania się urządzeniem.

Należy zwrócić uwagę, że w dwóch pierwszych podpunktach występuje podwójny znak równości oznaczający porównanie wartości parametru, a w dwóch kolejnych tylko pojedynczy oznaczający przypisanie wartości parametrowi. Wybrane skrypty z regułami dla programu udev należy umieścić w katalogu */etc/udev/rules.d* (potrzebne są do tego uprawnienia administratora). Nowe reguły uaktywnia się za pomocą polecenia (trzeba podać hasło administratora)

```
sudo /etc/init.d/udev restart
```

lub innego podobnego, na przykład

```
sudo /etc/init.d/boot.udev restart
```

Jeśli powyższe operacje wykona się przy podłączonym urządzeniu USB, to nowe uprawnienia będą aktywne powyjęciu i ponownym włożeniu wtyczki.

W Linuksie mamy do dyspozycji kilka prostych programów diagnostycznych. Polecenie *dmesg* wypisuje informacje o zdarzeniach zarejestrowanych przez jądro systemu operacyjnego, m.in. o ostatnio podłączonych i odłączonych urządzeniach. Po podłączeniu urządzenia USB i wydaniu tego polecenia powinniśmy zobaczyć podobny do poniższego wydruk.

```
[ 134.310319] usb 2-1.3: new full speed USB device using ehci_hcd  
and address 4
```

```
[ 134.397373] usb 2-1.3: New USB device found, idVendor=0483, idProduct=5752
[ 134.397377] usb 2-1.3: New USB device strings: Mfr=1, Product=2,
SerialNumber=3
[ 134.397381] usb 2-1.3: Product: USB virtual serial port example
[ 134.397383] usb 2-1.3: Manufacturer: Marcin Peczarski
[ 134.397386] usb 2-1.3: SerialNumber: 0000000001
[ 134.429286] cdc_acm 2-1.3:1.0: ttyACM0: USB ACM device
[ 134.429715] usbcore: registered new interface driver cdc_acm
[ 134.429717] cdc_acm: v0.26:USB Abstract Control Model driver for
USB modems and ISDN adapters
```

Widzimy na nim, że zostało podłączone urządzenie FS i został mu przydzielony adres 4. Numery 2-1.3 oznaczają po kolej numer szyny USB i numery portów kolejnych koncentratorów, począwszy od głównego koncentratora. Dalej następują informacje odczytane z deskryptora urządzenia, m.in. VID i PID. Potem widzimy nazwę urządzenia w systemie, czyli w tym przypadku `ttyACM0`, a na koniec informacje o załadowanym sterowniku. Po odłączeniu tego urządzenia zobaczymy następujący wydruk:

```
[ 1346.061454] usb 2-1.3: USB disconnect, address 4
```

Polecenie `dmesg` umożliwia diagnozowanie pewnych problemów, aczkolwiek wy pisywane numery błędów nie są zbyt użyteczne. Na poniższym wydruku widzimy komunikaty, gdy nie udało się odczytać deskryptora urządzenia.

```
[ 2894.027290] usb 2-1.3: new full speed USB device using ehci_hcd
and address 9
[ 2894.101318] usb 2-1.3: device descriptor read/64, error -32
[ 2894.275337] usb 2-1.3: device descriptor read/64, error -32
```

Inne informacje możemy uzyskać za pomocą polecenia `lsusb`. Wydając go bez argumentów, zobaczymy listę wszystkich podłączonych urządzeń USB, jak na poniższym wydruku. Nas najbardziej interesuje urządzenie rozpoznane jako SGS Thomson Microelectronics (poprzednia nazwa firmy STMicroelectronics).

```
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 002: ID 8087:0020
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 002: ID 8087:0020
Bus 002 Device 003: ID 15d9:0a4c Unknown
Bus 002 Device 004: ID 0483:5752 SGS Thomson Microelectronics
```

Widzimy tu m.in., że są dwie szyny USB. Do pierwszej podłączone są dwa urządzenia, a do drugiej – cztery.

Fizyczną topologię szyn zobaczymy, wydając polecenie `lsusb -vt`.

```
/: Bus 01.Port 1: Dev 1, Class=root_hub, Driver=ehci_hcd/2p, 480M
|__ Port 1: Dev 2, If 0, Class=hub, Driver=hub/6p, 480M
```

```
/: Bus 02.Port 1: Dev 1, Class=root_hub, Driver=ehci_hcd/2p, 480M
|__ Port 1: Dev 2, If 0, Class=hub, Driver=hub/8p, 480M
    |__ Port 2: Dev 3, If 0, Class=HID, Driver=usbhid, 1.5M
    |__ Port 3: Dev 4, If 0, Class=comm., Driver=cdc_acm, 12M
    |__ Port 3: Dev 4, If 1, Class=data, Driver=cdc_acm, 12M
```

Widzimy tu schematycznie narysowane dwa drzewa, odpowiadające dwóm szynom USB. Do pierwszej podłączony jest główny koncentrator HS (oznaczenie 480M), a do niego inny koncentrator HS. Podobną konfigurację widzimy dla drugiej szyyny, ale do koncentratora podłączone są dwa urządzenia. Urządzenie numer 3 ma tylko jeden interfejs numer 0 i jest to urządzenie LS (oznaczenie 1.5M) klasy HID (w tym przypadku jest to mysz). Urządzenie numer 4 ma dwa interfejsy oznaczone numerami 0 i 1. Jest to urządzenie FS (oznaczenie 12M), wirtualny port szeregowy, emulujący RS-232.

Szczegółowy wydruk deskryptorów dostaniemy, wydając polecenie `lsusb -v -d 0483:5752`, gdzie podane liczby to odpowiednio VID i PID urządzenia. Polecenie to wypisuje zwykle dość dużo linii tekstu. Na poniższym wydruku widzimy typowy początkowy fragment takiego wydruku (tylko deskryptor urządzenia).

```
Bus 002 Device 004: ID 0483:5752 SGS Thomson Microelectronics
Device Descriptor:
```

bLength	18
bDescriptorType	1
bcdUSB	2.00
bDeviceClass	2 Communications
bDeviceSubClass	0
bDeviceProtocol	0
bMaxPacketSize0	64
idVendor	0x0483 SGS Thomson Microelectronics
idProduct	0x5752
bcdDevice	1.00
iManufacturer	1 Marcin Peczarski
iProduct	2 USB virtual serial port example
iSerial	3 0000000001
bNumConfigurations	1

Podczas uruchamiania urządzenia USB bardzo przydatna okazuje się możliwość monitorowania komunikacji USB za pomocą modułu `usbmon`. Najpierw trzeba zamontować pseudokatalog poleceniem `mount` (to i niektóre kolejne polecenia wymagają uprawnień administratora).

```
sudo mount -t debugfs none_debugfs /sys/kernel/debug
```

Moduł `usbmon` może być wkompilowany na stałe w jądro systemu lub skompilowany jako moduł ładowalny. W tym drugim przypadku trzeba go załadować polecienniem `modprobe`.

```
sudo /sbin/modprobe usbmon
```

Żeby sprawdzić, czy moduł `usbmon` jest aktywny, sprawdzamy, czy istnieje katalog `/sys/kernel/debug/usb/usbmon` i wyświetlamy jego zawartość.

```
ls /sys/kernel/debug/usb/usbmon
```

Powinniśmy zobaczyć podobną do poniższej listę pseudoplików, w których będą się pojawiały komunikaty przesypane za pomocą USB. Liczba rozpoczynająca nazwę pliku to numer szyny USB. Zero oznacza wszystkie szyny.

```
0s 0u 1s 1t 1u 2s 2t 2u
```

Szynę, do której podłączone jest interesujące nas urządzenie, możemy zidentyfikować omówionym wyżej poleceniem `lsusb` lub zaglądając do pliku `/sys/kernel/debug/usb/devices`.

```
cat /sys/kernel/debug/usb/devices
```

Załóżmy, że nasze urządzenie podłączone jest do szyny numer 2. Rozpoczynamy monitorowanie komunikacji po tej szynie, wydając polecenie `cat` kopiące zawartość odpowiedniego pseudopliku do wybranego pliku, np. `usb2.log`.

```
sudo cat /sys/kernel/debug/usb/usbmon/2u > usb2.log
```

Monitorowanie kończymy, przerwując polecenie `cat` za pomocą kombinacji klawiszy *Ctrl-C*. Więcej szczegółów na temat działania modułu `usbmon` i dokładny opis formatu wypisywanych przez niego danych można znaleźć w dokumentacji jądra systemu Linux pod adresem <http://www.mjmwired.net/kernel/Documentation/usb> w dokumencie `usbmon.txt`.

Urządzenia emulujące port szeregowy są tematem projektu 2. Do ich testowania potrzebny będzie program obsługujący komunikację szeregową, na przykład Minicom lub PuTTY.

Do programowania pamięci Flash można użyć programu OpenOCD i dowolnego obsługiwanej przez niego adaptera JTAG. Wydaje się, że OpenOCD po okresie pewnych zawirowań stał się w końcu dojrzałym i stabilnym narzędziem. Opis, jak go skompilować i zainstalować, zamieściłem w dodatku. W katalogu `./make/scripts` znajdują się przykładowe skrypty do szybkiego programowania pamięci Flash za pomocą tego programu:

- `qfl` – dla płyt z układami STM32F10x i złączem JTAG,
- `qf2` – dla płyt z układami STM32F2xx i złączem JTAG,
- `qf4` – dla modułu STM32F4-Discovery wyposażonego w ST-LINK/V2,
- `qfl` – dla modułu STM32L-Discovery wyposażonego w ST-LINK/V2.

Skrypty te należy wywoływać z katalogu, w którym znajduje się plik `makefile` projektu.

```
./scripts/qfl
```

Jako argument można podać nazwę pliku z binarnym obrazem pamięci Flash. Jeśli nie podamy żadnego argumentu, to pamięć jest programowana zawartością pliku z rozszerzeniem *bin*, który zostanie znaleziony w bieżącym katalogu – musi być tylko jeden taki plik. Skrypty te korzystają ze skryptów z rozszerzeniem *cfg* dostarczanych wraz z programem OpenOCD. Wykorzystywane skrypty *cfg* są dla wygody umieszczone również w katalogu */make/scripts*. Skrypty *qfl* i *qf2* należy dostosować do posiadanej adaptera JTAG. Trzeba w nich zmienić nazwę skryptu konfiguruującego adapter. Podobnie skrypty *qf4* i *qfl* można dostosować do programowania za pomocą ST-LINK/V1. Potrzebnych skryptów należy szukać w katalogu *tcl/interface* (lub jego podkatalogu) wersji źródłowej pakietu OpenOCD. Skrypty *qfl*, *qf2*, *qf4* i *qfl* można też łatwo dostosować do innego modelu mikrokontrolera, zmieniając nazwę skryptu konfiguruującego programowany mikrokontroler. Skrypty opisujące inne mikrokontrolery znajdują się w katalogu *tcl/target* wersji źródłowej pakietu OpenOCD.

2.7.2. Uwagi dla użytkowników systemu Windows

Jeżeli w systemie Windows podłączymy nowe urządzenie identyfikowane parą VID i PID, która była już wcześniej wykorzystywana przez urządzenie o innej funkcji, to nowe urządzenie może nie działać. W takim przypadku należy odnaleźć to urządzenie w panelu sterowania i przeinstalować jego sterownik, wybierając opcję rozwiązywania problemów z menu rozwijanego prawym klawiszem myszy.

Do testowania urządzeń USB emulujących port szeregowy (COM), będących tematem projektu 2, potrzebny będzie program obsługujący komunikację szeregową. Ze starszymi wersjami systemu Windows był standardowo dostarczany program HyperTerminal. Niestety w najnowszych już go nie ma. Można próbować go skopiować z którejś ze starszych instalacji systemu. Potrzebne są dwa pliki: *hypertrm.dll* i *hypertrm.exe*, które można po prostu umieścić na pulpicie. Innym, jeszcze prostszym rozwiązaniem, jest użycie programu PuTTY. Potrzebny jest tylko jeden plik wykonywalny *putty.exe*, który można скачать ze strony <http://www.putty.org>. Żeby zainstalować sterownik wirtualnego portu szeregowego, potrzebny jest plik *stmcdc.inf*, który znajduje się w archiwum z przykładami w katalogu */make/win*.

Do programowania pamięci Flash można oczywiście używać windowsowej wersji programu OpenOCD. Można też korzystać z programu STM32 ST-LINK Utility, który obsługuje programatory wbudowane w płytach STM3220G-EVAL, STM32L-Discovery, STM32F4-Discovery i innych oferowanych przez STMicroelectronics oraz programatory ZL30PRG i ZL30PRGv2. Program ten można скачать ze strony <http://www.st.com>.

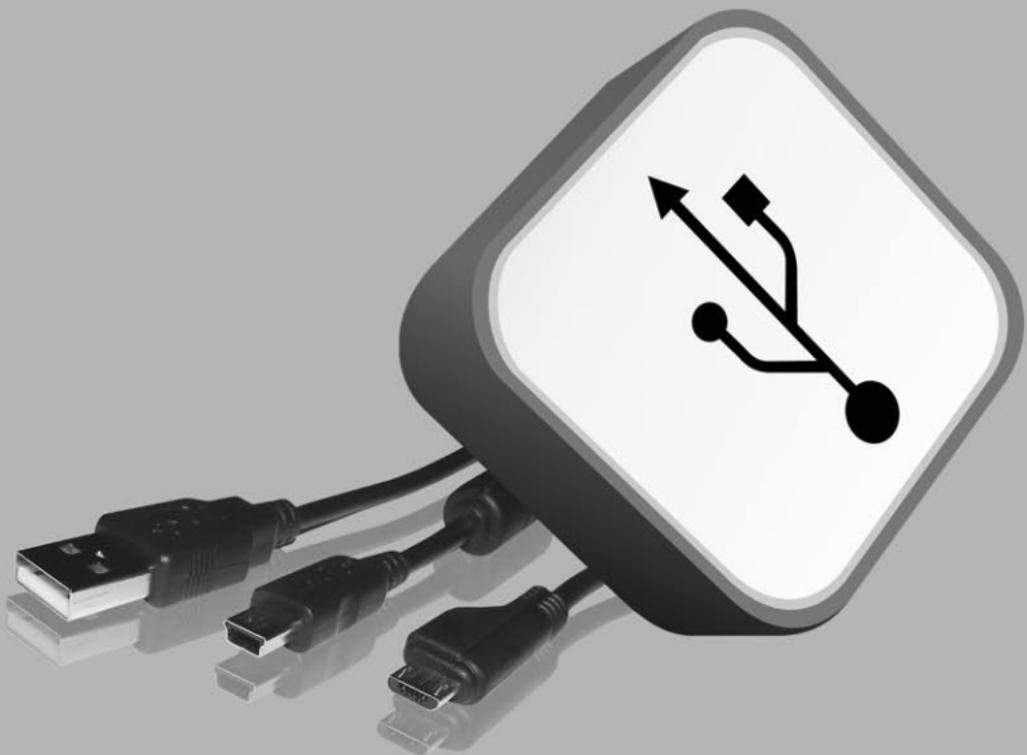
2.8. Dalsza lektura

Dodatkowych wiadomości na temat programowania mikrokontrolerów z rdzeniem Cortex-M3 należy szukać w [6], a mikrokontrolerów z rdzeniem Cortex-M4 w [7].

Rdzeń Cortex-M0 opisano w [8]. Szczegółowy opis układów peryferyjnych mikrokontrolerów STM32F10x, STM32F2xx, STM32L1xx, STM32F4xx, STM32F05x znajduje się odpowiednio w [9], [10], [11], [13] i [14]. Wyjątkiem są układy peryferyjne podrodziny STM32F100 opisane w [12]. Układy peryferyjne mikrokontrolerów STM32F3xx są opisane w [15] i [16].

3

Typowe urządzenia USB



W tym rozdziale przedstawiam trzy projekty typowych urządzeń USB pełnej szybkości (ang. *full speed*), demonstrujące wszystkie cztery, zdefiniowane w standardzie USB, sposoby przesyłania danych (ang. *control, isochronous, bulk, interrupt*). Każde z tych urządzeń współpracuje bez problemu z komputerem osobistym pracującym pod kontrolą systemu Linux lub Windows. Omawiam tutaj tylko implementację warstwy aplikacji tych urządzeń, odkładając dokładny opis implementacji protokołu USB do następnego rozdziału.

3.1. Projekt urządzenia klasy HID

Pierwszy projekt demonstruje przykład urządzenia z interfejsem klasy HID (ang. *Human Interface Device*) wykorzystującego dane pilne (ang. *interrupt*). Klasa HID obejmuje urządzenia używane przez człowieka do sterowania komputerem lub wprowadzania danych. Mogą to być:

- różnorakie klawiatury;
- urządzenia wskaźnikowe jak mysz, manipulator kulowy (ang. *trackball*) lub dżojstik;
- różnego rodzaju panele z przyciskami mechanicznymi, dotykowymi, suwakami, pokrętlami;
- moduły zdalnego sterowania;
- manipulatory wykorzystywane w grach, np. kierownica, pedały;
- czytniki kodów kreskowych;
- czujniki, np. temperatury.

Przykładowe urządzenie nie jest jakieś wyjątkowe, jest rozpoznawane przez komputer osobisty jako zwykła mysz. Wystarczy to jednak do pokazania najważniejszych cech klasy HID. Dalszych informacji należy szukać w [23].

3.1.1. Deskryptory

Urządzenie udostępniające interfejs klasy HID, jak każde urządzenie USB, musi mieć deskryptor urządzenia, deskryptor konfiguracji oraz deskryptor interfejsu. Obowiązkowo musi mieć też deskryptor wejściowego punktu końcowego dla danych pilnych. Ten punkt końcowy służy do przesyłania z urządzenia do kontrolera danych związanych z interfejsem obsługiwany przez to urządzenie. Opcjonalnie urządzenie może też mieć deskryptor wyjściowego punktu końcowego, który służy do przesyłania z kontrolera do urządzenia danych zwrotnych wymagających małego opóźnienia transmisji (ang. *latency*). W deskryptorze urządzenia z interfejsem klasy HID pola określające klasę, podklasę i protokół, czyli pola *bDeviceClass*, *bDeviceSubClass* i *bDeviceProtocol*, wypełnia się zerami. Właściwe wartości ustawia się w deskryptorze interfejsu, zgodnie z tabelą 3.1. Pozostałe pola powyższych deskryptorów wypełnia się zgodnie z opisem zamieszczonym w rozdziale 1. W przypadku klasy HID protokół definiuje format pakietów danych przesyłanych między urządzeniem a kontrolerem. Pakiety te w nomenklaturze USB nazywa się raportami.

Tab. 3.1. Parametry deskryptora interfejsu

Pole	Rozmiar	Wartość	Opis
bInterfaceClass	1	3	Wartość oznaczająca klasę HID
		0	Podklaśa nieokreślona
		1	Podklaśa obsługująca uproszczony protokół stosowany w fazie rozruchu komputera (ang. <i>boot interface subclass</i>)
bInterfaceProtocol	1	0	Protokół nieokreślony
		1	Klawiatura
		2	Mysz

Pewne urządzenia (klawiatura i mysz) muszą być uruchomione już w fazie rozruchu (ang. *boot*) komputera. Nie ma wtedy jeszcze załadowanych żadnych sterowników, a obsługą wszelkich urządzeń zajmuje się bezpośrednio BIOS. Nie ma tam miejsca na skomplikowane protokoły i analizowanie zawartości wielu deskryptorów. Dlatego urządzenia, które powinny działać w fazie rozruchu, muszą obsługiwać domyślny format raportu, co zaznacza się, umieszczając w polu bInterfaceSubClass wartość 1. Wtedy w polu bInterfaceProtocol musi być wpisana wartość 1 lub 2, oznaczająca odpowiednio klawiaturę lub mysz. Jeśli urządzenie nie obsługuje protokołu fazy rozruchu, to w obu tych polach wpisuje się zero.

Tab. 3.2. Główny deskryptor HID

Pole	Rozmiar	Opis
bLength	1	Rozmiar tego deskryptora ($3N + 6$)
bDescriptorType	1	Typ tego deskryptora (33)
bcdHID	2	Wersja specyfikacji klasy HID
bCountryCode	1	Kod kraju
bNumDescriptors	1	Liczba N dodatkowych deskryptorów, co najmniej 1
bDescriptorType[0]	1	Typ dodatkowego deskryptora 0
bDescriptorLength[0]	2	Rozmiar dodatkowego deskryptora 0
bDescriptorType[1]	1	Typ dodatkowego deskryptora 1
bDescriptorLength[1]	2	Rozmiar dodatkowego deskryptora 1
...
bDescriptorType[N-1]	1	Typ dodatkowego deskryptora $N - 1$
bDescriptorLength[N-1]	2	Rozmiar dodatkowego deskryptora $N - 1$

Wszystkie właściwości interfejsu klasy HID przedstawia się za pomocą deskryptorów specyficznych dla tej klasy. W hierarchii deskryptorów opisujących konfigurację (patrz rysunek 1.23) pomiędzy deskryptorem interfejsu a pierwszym (zwykle jedynym) deskryptorem punktu końcowego wstawia się główny deskryptor HID, którego format przedstawiony jest w tabeli 3.2. Jego zasadniczym celem jest wskazanie kolejnych deskryptorów zawierających bardziej szczegółowe informacje o urządzeniu. Pole bLength zawiera rozmiar w bajtach głównego deskryptora. Rozmiar ten zależy od liczby N dalszych deskryptorów i wlicza się go do wartości wpisanej w polu bTotalLength deskryptora konfiguracji. Pole bDescriptorType identyfikuje główny deskryptor HID i zawsze jest w nim wpisana wartość 33. Pole bcdHID

określa wersję specyfikacji klasy HID [23]. Jest to wersja, z którą w zamierzeniu ma być zgodne urządzenie przedstawiające się tym deskryptorem. Aktualna wersja specyfikacji HID ma numer 1.11, zatem należy w tym polu wpisać wartość 0x0111. Pole `bCountryCode` zawiera kod kraju, dla którego przeznaczone jest dane urządzenie. W zamierzeniu kod ten ma określać pewne specyficzne własności urządzenia zależne na przykład od języka. Dla Polski przewidziano wartość 21. W praktyce jednak nie korzysta się z tej możliwości, umieszczając w tym polu wartość zero, co oznacza, że urządzenie nie wspiera żadnego standardu krajowego. Taką wartość w tym polu ma na przykład typowa komputerowa klawiatura USB, która zwraca kod wciśniętego klawisza, niezależny od umieszczonego na tym klawiszku znaku, a wsparcie dla różnych narodowych układów klawiatury, czyli przekodowanie na odpowiedni znak, realizowane jest przez system operacyjny.

Pole `bNumDescriptors` zawiera liczbę N dodatkowych deskryptorów opisujących kolejne własności urządzenia. Po nim występuje N par pól. Pole `bDescriptorType` w każdej parze określa typ deskryptora. Natomiast pole `bDescriptorLength` określa jego rozmiar w bajtach. Musi istnieć przynajmniej jeden dodatkowy deskryptor, czyli N musi mieć wartość co najmniej jeden. Dodatkowe deskryptory numeruje się od 0 do $N - 1$. Standard definiuje dwa ich typy: deskryptor raportu (ang. *report descriptor*) i deskryptor fizyczny (ang. *physical descriptor*). Deskryptor raportu jest obowiązkowy. Opisuje format danych przesyłanych między urządzeniem klasy HID a kontrolerem po uruchomieniu systemu operacyjnego, czyli po zakończeniu fazy rozruchu. Innymi słowy, opisuje pełny protokół urządzenia. Format raportu obejmuje wielkości poszczególnych pól, ich przeznaczenie, zakresy dopuszczalnych wartości, stosowane jednostki miary itp. Deskryptor fizyczny jest opcjonalny i opisuje przyporządkowanie części ludzkiego ciała do elementów sterujących. Może na przykład określać, że prawym kciukiem obsługuje się lewy przycisk manipulatora. Budowa tych deskryptorów jest odmienna niż pozostałych deskryptorów USB. Po szczegóły odsyłam do [23]. W dalszym ciągu będziemy korzystać z domyślnego protokołu fazy rozruchu, niepotrzebującego deskryptora fizycznego, a deskryptor raportu ustalimy tak, aby definiował protokół identyczny z tym protokołem.

3.1.2. Żądania

Urządzenie udostępniające interfejs klasy HID musi obsługiwać standardowe żądania opisane w rozdziale 1, w tym żądanie `GET_DESCRIPTOR` dla standardowych deskryptorów urządzenia, konfiguracji i tekstowych, a ponadto dla deskryptorów specyficznych dla klasy HID. Główny deskryptor HID, deskryptor raportu i deskryptor fizyczny identyfikuje się, podając odpowiednio wartość 33, 34 lub 35 w starszym bajcie parametru `wValue`. W przypadku deskryptora głównego i raportu w młodszym bajcie podaje się zero. Dla deskryptora fizycznego w młodszym bajcie parametru `wValue` podaje się numer żądanego deskryptora, przy czym deskryptor numer 0 określa liczbę i rozmiar dostępnych deskryptorów. Oprócz żądań standardowych urządzenie klasy HID musi obsługiwać żądania specyficzne dla tej klasy, zebrane w tabeli 3.3. Wszystkie specyficzne żądania kierowane są do interfejsu klasy HID, czyli do interfejsu, który ma w polu `bInterfaceClass` wartość 3. W parametrze `wIndex` żądania umieszcza się numer tego interfejsu.

Tab. 3.3. Żądania specyficzne dla klasy HID

bmRequestType	bRequest	wValue	wIndex	wLength	Dane
10100001	GET_REPORT 1	Typ i numer raportu	Numer interfejsu	Rozmiar raportu	Raport
00100001	SET_REPORT 9	Typ i numer raportu	Numer interfejsu	Rozmiar raportu	Raport
10100001	GET_IDLE 2	Numer raportu	Numer interfejsu	1	Wartość
00100001	SET_IDLE 10	Wartość i numer raportu	Numer interfejsu	0	Brak
10100001	GET_PROTOCOL 3	0	Numer interfejsu	1	Protokół
00100001	SET_PROTOCOL 11	Protokół	Numer interfejsu	0	Brak

Zasadniczo raporty przesyła się za pomocą punktu końcowego dla danych pilnych. Kontroler może też odbierać i wysyłać raporty za pomocą domyślnego punktu końcowego numer 0 dla danych sterujących. Służą do tego odpowiednio żądania GET_REPORT i SET_REPORT. Stosuje się je przede wszystkim w fazie inicjowania urządzenia, ale także na przykład do sterowania diodami świecącymi klawiatury. W starszym bajcie parametru wValue przesyła się typ raportu, odpowiednio 1 dla raportu wejściowego (ang. *input*), czyli przesyłanego z urządzenia do kontrolera, 2 dla raportu wyjściowego (ang. *output*), czyli przesyłanego z kontrolera do urządzenia, a 3 dla raportu z własnościami (ang. *feature*). W młodszym bajcie umieszcza się identyfikator raportu. Urządzenie może ignorować żądania z bezsensowną dla niego kombinacją typu żądania i typu raportu. Żądanie GET_REPORT służy do przesyłania raportu wejściowego i raportu z własnościami. Musi być ono obsługiwane przez każde urządzenie mające interfejs klasy HID. Żądanie SET_REPORT służy do przesyłania raportu wyjściowego i jest obowiązkowe tylko dla urządzeń deklarujących w deskryptorze raportu obsługę tego typu raportu. Pole wLength zawiera rozmiar raportu.

Żądania GET_IDLE i SET_IDLE służą odpowiednio do odczytania lub zapisania parametru IDLE ograniczającego częstotliwość wysyłania raportu przez wejściowy punkt końcowy dla danych pilnych. Jest to maksymalny czas, przez który urządzenie nie generuje nowego raportu (odpowiada NAK na żądania przesłania raportu), jeśli stan elementów sterujących (np. klawiszy) się nie zmienił. Parametr IDLE przyjmuje wartości od 0 do 255, przy czym 0 oznacza czas nieskończony, a pozostałe wartości mnożą się przez 4 ms. Można zatem ustawić czas od 4 do 1020 ms z rozdzielcością 4 ms. Młodszy bajt parametru wValue zawiera identyfikator raportu, którego dotyczy żądanie. W przypadku żądania SET_IDLE nową wartość przesyła się w starszym bajcie tego parametru. Dla żądania GET_IDLE starszy bajt ma wartość zero, a odpowiednią wartość przesyła się w fazie danych obsługi żądania. Standard wymaga, aby klawiatura obsługiwała żądania GET_IDLE i SET_IDLE. Dla innych urządzeń żądania te są opcjonalne. Rekomendowana wartość IDLE dla klawiatury wynosi 125, czyli 500 ms, a dla myszy i dżojstika 0, czyli nieskończoność.

Żądania GET_PROTOCOL i SET_PROTOCOL służą odpowiednio do sprawdzenia lub ustawienia aktywnego protokołu. Zdefiniowane są dwie wartości: zero oznacza protokół fazy rozruchu (ang. *boot protocol*), a jeden oznacza protokół zdefini-

wany za pomocą deskryptora raportu (ang. *report protocol*). W przypadku żądania SET_PROTOCOL nową wartość przesyła się w parametrze `wValue`. Dla żądania GET_PROTOCOL parametr ten ma wartość zero, a odpowiednią wartość przesyła się w fazie danych obsługi żądania. Żądania GET_PROTOCOL i SET_PROTOCOL są obowiązkowe dla urządzeń deklarujących obsługę protokołu fazy rozruchu (wartość 1 w polu `bInterfaceSubClass` deskryptora interfejsu). Dla pozostałych urządzeń żądania te są opcjonalne.

3.1.3. Protokół fazy rozruchu dla myszy i klawiatury

Format raportu myszy dla protokołu fazy rozruchu przedstawiony jest w tabeli 3.4. Składa się on z trzech bajtów. Trzy najmłodsze bity w zerowym bajcie zawierają stan przycisków myszy. Jedynka oznacza przycisk wcisnięty. Pozostałe bity są zarezerwowane i powinno się je zerować. Dwa kolejne bajty zawierają względną zmianę położenia myszy w stosunku do poprzedniego raportu. X i Y zapisuje się jako liczby całkowite ze znakiem w kodowaniu uzupełnieniowym do dwójki. Patrząc na mysz z góry, dodatnia wartość X oznacza przesunięcie w prawo, a ujemna – w lewo. Dodatnia wartość Y oznacza przybliżenie myszy do siebie, a ujemna – odsunięcie jej od siebie. Wartości w polach X i Y powinny być z przedziału od -127 do 127. Dla zachowania symetrii nie używa się wartości -128, która też może być zapisana na 8 bitach. Każde przyciśnięcie lub puszczenie przycisku albo poruszenie myszą powoduje wygenerowanie nowego raportu. Długi ruch może spowodować wygenerowanie kilku kolejnych raportów.

Tab. 3.4. Format raportu myszy

Bajt	Bit	Opis
0	0	Przycisk 1, lewy
	1	Przycisk 2, prawy
	2	Przycisk 3, środkowy
	3...7	Zarezerwowane
1	0...7	Przesunięcie w poziomie X
2	0...7	Przesunięcie w pionie Y

Format raportu zawierającego stan klawiatury dla protokołu fazy rozruchu przedstawiony jest w tabeli 3.5. Składa się on z ośmiu bajtów. Bajt zerowy zawiera w kolejnych bitach stan klawiszy modyfikujących. Są to kolejno cztery klawisze umieszczone po lewej stronie klawiatury: *Ctrl*, *Shift*, *Alt* i *GUI* (klawisz z logo systemu Windows), a następnie odpowiednie cztery klawisze umieszczone po prawej stronie klawiatury. W konkretnym wykonaniu klawiatury nie wszystkie one muszą istnieć. Gdy klawisz jest wcisnięty, odpowiedni bit jest ustawiony. Bajt numer 1 jest zarezerwowany i powinien mieć wartość zero. Kolejne sześć bajtów zawiera kody wcisniętych klawiszy. Przy czym wartość zero nie reprezentuje żadnego klawisza i oznacza, że dany bajt należy pominąć. Wartości 1, 2 i 3 oznaczają błąd. Zwykle należy wtedy zignorować cały raport. Pozostałe wartości należy interpretować jako kody wcisniętych klawiszy. Typowa klawiatura komputerowa USB zwalca kody o wartościach od 4 do 101. Nie są to jednak kody znaków umieszczone na klawiszach, a umownie ustalone wartości związane z fizycznymi klawiszami.

Przykładowo na typowej klawiaturze QWERTY kody od 4 do 29 oznaczają klawisze opisane literami od A do Z, kody 30 do 39 to kody klawiszy oznaczonych odpowiednio cyframi 0 do 9, kod 40 przydzielono klawiszowi *Enter*, 58 do 69 to kody klawiszy F1 do F12, a 101 to kod klawisza *Menu*. Pełną tablicę kodów można znaleźć w [5]. Pojawi się ona też w przykładzie kontrolera obsługującego urządzenia z interfejsem klasy HID w rozdziale 7. Każde przyciśnięcie lub puszczenie klawisza powoduje wygenerowanie nowego raportu. Dodatkowo, jeśli nawet stan klawiatury nie uległ zmianie od wysłania poprzedniego raportu, kolejny raport generowany jest po upływie czasu określonego parametrem **IDLE**. Zauważmy, że wśród klawiszy modyfikujących nie ma *Caps Lock* ani *Num Lock*. Klawiatura traktuje je jak zwykłe klawisze (o kodach odpowiednio 57 i 83). Ich specjalne funkcje muszą być obsługiwane programowo. Zauważmy też, że teoretycznie protokół umożliwia, aby jednocześnie było wciśniętych aż 14 klawiszy (8 modyfikujących i 6 zwykłych). W praktyce jednak typowa klawiatura komputerowa nie potrafi obsługiwać jednoczesnego wciśnięcia większej liczby klawiszy, niż mamy razem palców u obu rąk, a najczęściej potrafi znacznie mniej.

Tab. 3.5. Format raportu wejściowego klawiatury

Bajt	Bit	Opis
0	0	Lewy <i>Ctrl</i>
	1	Lewy <i>Shift</i>
	2	Lewy <i>Alt</i>
	3	Lewy <i>GUI</i>
	4	Prawy <i>Ctrl</i>
	5	Prawy <i>Shift</i>
	6	Prawy <i>Alt</i>
	7	Prawy <i>GUI</i>
1	0...7	Zarezerwowane
2...7	0...7	Kody wciśniętych klawiszy

Oprócz klawiszy nawet najzwyklejsza klawiatura komputerowa ma co najmniej dwie diody świecące przy klawiszach *Caps Lock* i *Num Lock*, często też trzecią związaną z klawiszem *Scroll Lock*, a czasem jeszcze czwartą przy klawiszu *Compose* (jeśli ma taki klawisz). Japońskie klawiatury mają diodę świecącą *Kana*. Sterowanie tymi diodami w protokole fazy rozruchu polega na wysłaniu do urządzenia raportu przedstawionego w **tabeli 3.6**. Jest to tylko jeden bajt, którego kolejne bity informują, które diody mają być zaświecone (bit ustawiony) lub zgaszone (bit wyzerowany). Nieużywane bity powinny być wyzerowane.

Tab. 3.6. Format raportu wyjściowego klawiatury

Bajt	Bit	Opis
0	0	<i>Num Lock</i>
	1	<i>Caps Lock</i>
	2	<i>Scroll Lock</i>
	3	<i>Compose</i>
	4	<i>Kana</i>
	5...7	Zarezerwowane

Ponieważ w BIOS-ie nie ma miejsca na pełną implementację kontrolera USB, a w szczególności na analizę zawartości wszystkich deskryptorów, w fazie rozruchu stosuje się bardzo uproszczony protokół komunikacji. Kontroler, wykrywający urządzenie USB, nie odczytuje deskryptorów urządzenia i konfiguracji, skąd mógłby się dowiedzieć, z jakiej klasy urządzeniem ma do czynienia. Musi jednak jakoś stwierdzić, czy podłączone urządzenie jest klawiaturą lub myszą. W tym celu kontroler wysyła do urządzenia żądanie `GET_DESCRIPTOR`, prosząc o przesyłanie głównego deskryptora HID. Nie musi nawet analizować jego zawartości, gdyż tylko urządzenie z interfejsem klasy HID odpowie poprawnie na to żądanie. Żeby stwierdzić, czy urządzenie obsługuje protokół fazy rozruchu, kontroler wysyła najpierw żądanie `GET_PROTOCOL`, a następnie `SET_PROTOCOL`, próbując ustawić ten protokół. Jeśli urządzenie poprawnie zrealizuje oba żądania, to można przyjąć, że mamy do czynienia z klasą HID i obsługiwany jest protokół fazy rozruchu. Analizując przysłany raport, łatwo odróżnić klawiaturę od myszy, a właściwie to wystarczy tylko sprawdzić jego długość. W fazie rozruchu kontroler korzysta jeszcze z żądania `SET_IDLE`.

3.1.4. Implementacja myszy

`ex_hid_dev.c`

Implementacja warstwy aplikacji dla urządzenia z interfejsem klasy HID znajduje się w pliku `ex_hid_dev.c`. Poniżej omawiam najważniejsze fragmenty tego pliku. Na początek musimy zdefiniować wszystkie deskryptory. Potrzebne do tego struktury i stałe są zdefiniowane w pliku `usb_def.h`.

```
static usb_device_descriptor_t const device_descriptor = {
    sizeof(usb_device_descriptor_t), /* bLength */
    DEVICE_DESCRIPTOR,             /* bDescriptorType */
    HTOUSBS(0x0200),              /* bcdUSB */
    0,                            /* bDeviceClass */
    0,                            /* bDeviceSubClass */
    0,                            /* bDeviceProtocol */
    8,                            /* bMaxPacketSize0 */
    HTOUSBS(VID),                /* idVendor */
    HTOUSBS(PID + 1),              /* idProduct */
    HTOUSBS(0x0100),              /* bcdDevice */
    1,                            /* iManufacturer */
    2,                            /* iProduct */
    3,                            /* iSerialNumber */
    1,                            /* bNumConfigurations */
};
```

Jak wskazuje nazwa, struktura `device_descriptor` zawiera deskryptor urządzenia. Poszczególne jej pola inicjujemy zgodnie z opisem przedstawionym w rozdziale 1.

Urządzenie ma w założeniu być zgodne z wersją 2.0 standardu USB, dlatego w polu `bcdUSB` wpisujemy wartość `0x0200`. Makro `HTOUSBS` jest zdefiniowane w pliku `usb_endianness.h` i zamienia porządek bajtów lokalnej maszyny na porządek obowiązujący w USB dla liczby 16-bitowej (ang. *Host TO USB Short*). Przypominam, że w architekturze ARM Cortex-M i w USB obowiązuje ten sam cienkokońcowkowy (ang. *little-endian*) porządek bajtów, więc w tym przypadku makro to jest puste. Zostało jednak użyte dla zachowania przenośności tekstu źródłowego. Jeśli będziemy stosować to makro konsekwentnie, to chcąc przenieść implementację na maszynę z porządkiem grubokońcowkowym (ang. *big-endian*), nie trzeba będzie modyfikować wielu plików źródłowych, wystarczy tylko zmienić plik `usb_endianness.h`. Zgodnie z opisem zamieszczonym w rozdziale 3.1.1 pola `bDeviceClass`, `bDeviceSubClass` i `bDeviceProtocol` wypełniamy zerami. W celu zachowania kompatybilności z typowymi implementacjami myszy pole `bMaxPacketSize0` zawiera najmniejszą dopuszczalną dla niego wartość (patrz też tabela 1.10). Pola `idVendor` i `idProduct` inicjujemy odpowiednio stałymi VID i PID zdefiniowanymi w pliku `usb_vid_pid.h`. Przyjęłem zasadę, że identyfikator produktu dla poszczególnych urządzeń to wartość stałej PID zwiększoną o numer projektu. W ten sposób urządzenia w poszczególnych projektach otrzymują różne identyfikatory. Pole `bcdDevice` według intencji opisanej w standardzie powinno zawierać numer wersji urządzenia. Zaczynamy, jak to często bywa, od wersji 1.0. W polach `iManufacturer`, `iProduct` i `iSerialNumber` wpisujemy numery deskryptorów tekstowych z czytelnymi dla człowieka nazwami producenta i produktu oraz numerem seryjnym. W polu `bNumConfigurations` wpisujemy liczbę konfiguracji naszego urządzenia. Przykładowe urządzenie ma tylko jedną konfigurację.

Tablica `usb_hid_report_descriptor` zawiera deskryptor raportu, który opisuje identyczny format jak ten zamieszczony w tabeli 3.4. Poniższy wydruk przedstawia tylko początkowy fragment tego deskryptora.

```
static uint8_t usb_hid_report_descriptor[] = {
    0x05, 0x01, /* Usage Page (Generic Desktop) */
    0x09, 0x02, /* Usage (Mouse) */
    ...
};
```

Żądanie odczytu deskryptora konfiguracji powinno umożliwiać odczytanie całej hierarchii deskryptorów związanych z konfiguracją opisywaną tym deskryptorem. Dlatego definiujemy strukturę danych `usb_hid_configuration_t`, która zawiera kolejno deskryptor konfiguracji, deskryptor interfejsu, główny deskryptor HID i deskryptor punktu końcowego. Atrybut `_packed` nakazuje kompilatorowi ciasno układać w pamięci poszczególne pola struktury. Zatem położenie pól nie jest wyrównywane do adresów, które zapewniają optymalizację operacji odczytu i zapisu. Wyrównywania nie można zastosować, gdyż prowadzi ono czasem do pozostawiania odstępów między polami. Ponieważ standard języka C nie przewiduje atrybutu sterującego rozmieszczeniem pól struktury i w poszczególnych kompilatorach używa się ten efekt w różny sposób, atrybut ten jest zdefiniowany dla popularnych kompilatorów na początku pliku `usb_def.h`.

```
typedef struct {
    usb_configuration_descriptor_t cnf_descr;
    usb_interface_descriptor_t      if_descr;
    usb_hid_main_descriptor_t      hid_descr;
    usb_endpoint_descriptor_t       ep_descr;
} packed usb hid configuration t;
```

Mając już zdefiniowaną strukturę dla hierarchii deskryptorów konfiguracji, trzeba ją pracowicie wypełnić, co przedstawia poniższy wydruk. Stałe `USB_BM_ATTRIBUTES` i `USB_B_MAX_POWER` są zdefiniowane w pliku `board_usb_def.h`. Pierwsza z tych stałych określa m.in. sposób zasilania urządzenia. Jeśli ma własne zasilanie, to stała ta ma wartość (`SELF_POWERED | D7_RESERVED`). Jeśli urządzenie jest zasilane tylko z szyny USB, to powinna mieć wartość `D7_RESERVED`. Druga stała określa, ile prądu (w jednostkach 2 mA) urządzenie ma zamiar pobierać z interfejsu USB.

```

static usb_hid_configuration_t const hid_configuration = {
{
    sizeof(usb_configuration_descriptor_t), /* bLength */
    CONFIGURATION_DESCRIPTOR, /* bDescriptorType */
    HTOUSBS(sizeof(usb_hid_configuration_t)), /* wTotalLength */
    1, /* bNumInterfaces */
    1, /* bConfigurationValue */
    0, /* iConfiguration */
    USB_BM_ATTRIBUTES, /* bmAttributes */
    USB_B_MAX_POWER /* bMaxPower */
},
{
    sizeof(usb_interface_descriptor_t), /* bLength */
    INTERFACE_DESCRIPTOR, /* bDescriptorType */
    0, /* bInterfaceNumber */
    0, /* bAlternateSetting */
    1, /* bNumEndpoints */
    HUMAN_INTERFACE_DEVICE_CLASS, /* bInterfaceClass */
    BOOT_INTERFACE_SUBCLASS, /* bInterfaceSubClass */
    MOUSE_PROTOCOL, /* bInterfaceProtocol */
    0 /* iInterface */
},
{
    sizeof(usb_hid_main_descriptor_t), /* bLength */
    HID_MAIN_DESCRIPTOR, /* bDescriptorType */
    HTOUSBS(0x0111), /* bcdHID */
    0, /* bCountryCode */
    1, /* bNumDescriptors */
}
}
```

```

    HID_REPORT_DESCRIPTOR,           /* bDescriptorType */
    HTOUSBS(sizeof(usb_hid_report_descriptor))/* wDescriptorLength */
},
{
    sizeof(usb_endpoint_descriptor_t),      /* bLength */
    ENDPOINT_DESCRIPTOR,                 /* bDescriptorType */
    ENDP1 | ENDP_IN,                   /* bEndpointAddress */
    INTERRUPT_TRANSFER,               /* bmAttributes */
    HTOUSBS(sizeof(hid_mouse_boot_report_t)), /* wMaxPacketSize */
    10                                /* bInterval */
}
};

```

Ponieważ w deskryptorze urządzenia wpisaliśmy niezerowe numery deskryptorów tekstowych, musimy odpowiednie deskryptory zdefiniować. Pomocne będzie makro `usb_string_descriptor_t` zdefiniowane w pliku `usb_def.h`. Definiuje ono strukturę deskryptora tekstuowego zależnie od wartości argumentu, który jest liczbą dwubajtową znaków zapisanych w tym deskryptorze. Jeśli definiujemy jakieś deskryptory tekstowe, to musimy zdefiniować specjalny deskryptor tekstowy o numerze zero deklarujący, w jakich językach będą dostępne właściwe deskryptory tekstowe. Ten specjalny deskryptor nazwiemy `string_lang`. Nie będziemy się silić i wszystkie napisy będą w amerykańskiej wersji języka angielskiego – stała `LANG_US_ENGLISH`. Dla języka polskiego należy użyć stałej `LANG_POLISH`.

```

static usb_string_descriptor_t(1) const string_lang = {
    sizeof(usb_string_descriptor_t(1)),
    STRING_DESCRIPTOR,
    {HTOUSBS(LANG_US_ENGLISH)}
};

```

Zgodnie z tym, co zostało napisane wyżej, potrzebujemy trzech deskryptorów tekstowych, zawierających nazwę producenta, nazwę produktu oraz numer seryjny. Nazwiemy je odpowiednio `string_manufacturer`, `string_product` i `string_serial`. Ponieważ ich definicje wyglądają bardzo podobnie, poniżej zamieszczam tylko jedną z nich. Używamy makra `HTOUSBS`, gdyż znaki są wartościami 16-bitowymi w kodowaniu UTF-16. Jako ćwiczenie pozostawiam Czytelnikowi zaimplementowanie mechanizmu, który pozwoliłby łatwo nadawać indywidualny numer seryjny każdemu egzemplarzowi urządzenia.

```

static usb_string_descriptor_t(10) const string_serial = {
    sizeof(usb_string_descriptor_t(10)),
    STRING_DESCRIPTOR,
    {
        HTOUSBS('0'), HTOUSBS('0'), HTOUSBS('0'), HTOUSBS('0'),
        HTOUSBS('0'), HTOUSBS('0'), HTOUSBS('0'), HTOUSBS('0'),
        HTOUSBS('0'), HTOUSBS('1')
    }
};

```

Ponieważ żądania GET_DESCRIPTOR operują numerami deskryptorów, wygodnie jest stworzyć pomocniczą tablicę w taki sposób, aby jej element o indeksie n wskazywał deskryptor o numerze n . Elementy tablicy są typu string_table_t. Każdy element zawiera wskaźnik do deskryptora oraz jego rozmiar. Wszystkie deskryptory tekstowe są zebrane w tablicy strings. Stała stringCount określa liczbę tych deskryptorów albo inaczej maksymalny numer deskryptora tekstowego.

```
typedef struct {
    uint8_t const *data;
    uint16_t length;
} string_table_t;

static string_table_t const strings[] = {
    {(uint8_t const*)&string_lang,           sizeof string_lang},
    {(uint8_t const*)&string_manufacturer,   sizeof string_manufacturer},
    {(uint8_t const*)&string_product,         sizeof string_product},
    {(uint8_t const*)&string_serial,          sizeof string_serial}
};

static uint32_t const stringCount = sizeof(strings)/sizeof(strings[0]);
```

Jeśli oprogramowanie protokołu odbierze żądanie, które wymaga jakieś akcji ze strony aplikacji, to wywołuje odpowiednią funkcję zwrotną (ang. *callback*). Implementacja warstwy aplikacji dla przykładowych urządzeń USB prezentowanych w tej książce wymaga zdefiniowania wymaganych deskryptorów i zaimplementowania potrzebnych funkcji zwrotnych. Obsługę żądania GET_DESCRIPTOR realizuje funkcja zwrotna GetDescriptor. Pierwsze dwa parametry wValue i wIndex to wartości ze struktury setup (typu usb_setup_packet_t) przesłanej w fazie ustanowienia żądania (patrz rozdział 1). Identyfikują one deskryptor, którego żąda kontroler. Za pomocą parametru data zwraca się wskaźnik na deskryptor do przesłania, a za pomocą parametru length przekazuje się jego rozmiar. Należy zwrócić uwagę, że przekazany wskaźnik musi być ważny również po zakończeniu funkcji (co wydaje się oczywiste) aż do zakończenia fazy transmisji danych, gdyż oprogramowanie USB nie ma obowiązku kopiowania całe zawartości deskryptora do jakiegoś pośredniego bufora. Z perspektywy języka C oznacza to, że przekazany wskaźnik musi wskazywać na strukturę, która jest zadeklarowana jako statyczna lub globalna, aby nie zniknęła po zakończeniu wywołania funkcji. Funkcje zwrotne zwykle zwracają wartość typu usb_result_t. Może to być REQUEST_SUCCESS, gdy żądanie jest poprawne i funkcja go realizuje, a REQUEST_ERROR, gdy wystąpił błąd lub funkcja nie obsługuje danego żądania. Poniższy wydruk przedstawia kompletną implementację funkcji GetDescriptor dla omawianego urządzenia.

```
usb_result_t GetDescriptor(uint16_t wValue, uint16_t wIndex,
                           uint8_t const **data, uint16_t *length) {
    uint32_t index = wValue & 0xff;
    switch (wValue >> 8) {
        case DEVICE_DESCRIPTOR:
```

```
if (index == 0 && wIndex == 0) {
    *data = (uint8_t const *)&device_descriptor;
    *length = sizeof(device_descriptor);
    return REQUEST_SUCCESS;
}
return REQUEST_ERROR;
case CONFIGURATION_DESCRIPTOR:
if (index == 0 && wIndex == 0) {
    *data = (uint8_t const *)&hid_configuration;
    *length = sizeof(hid_configuration);
    return REQUEST_SUCCESS;
}
return REQUEST_ERROR;
case STRING_DESCRIPTOR:
if (index < stringCount) {
    *data = strings[index].data;
    *length = strings[index].length;
    return REQUEST_SUCCESS;
}
return REQUEST_ERROR;
case HID_MAIN_DESCRIPTOR: /* Requested when boot up. */
if (index == 0 && wIndex == 0) {
    *data = (uint8_t const *)&hid_configuration.hid_descr;
    *length = sizeof(hid_configuration.hid_descr);
    return REQUEST_SUCCESS;
}
return REQUEST_ERROR;
case HID_REPORT_DESCRIPTOR:
if (index == 0 && wIndex == 0) {
    *data = usb_hid_report_descriptor;
    *length = sizeof(usb_hid_report_descriptor);
    return REQUEST_SUCCESS;
}
return REQUEST_ERROR;
default:
return REQUEST_ERROR;
}
}
```

Stan aplikacji przechowujemy w zmiennych globalnych. Typ hid_mouse_boot_report_t definiuje format raportu z tabeli 3.4. Zmienna report przechowuje aktualny raport, czekający na wysłanie. Zmienna configuration zawiera numer wybranej

konfiguracji. W tym przypadku może to być 0, gdy kontroler nie wybrał jeszcze żadnej konfiguracji, albo 1, gdy kontroler wybrał jedyną dostępną konfigurację za pomocą żądania SET_CONFIGURATION. Zmienna protocol zawiera aktualnie realizowany protokół, wybrany przez kontroler za pomocą żądania SET_PROTOCOL. Zmienna busy przyjmuje wartość 1, gdy raport został wysłany, ale jego transmisja jeszcze się nie zakończyła, więc trzeba poczekać z wysyaniem kolejnego raportu. Zmienna ta jest ustawiana na 0 po zakończeniu transmisji raportu. Zmienne button1, button2 i button3 pomagają likwidować drgania styków przycisków myszy (ang. *debouncing*). Zmienne x, y, xrepeat i yrepeat służą do uzyskania efektu akceleracji ruchu wskaźnika myszy w poziomie i pionie. Zmienna idle_value przechowuje maksymalny czas (w ms) między wysłaniem kolejnych raportów, ustawiany przez kontroler za pomocą żądania SET_IDLE. Jeśli czas ten ma być nieskończony, trzeba ustawić wartość minus jeden. Zmienna idle_timer odlicza czas (w ms), jaki pozostał do wysłania kolejnego raportu.

```
static hid_mouse_boot_report_t report;
static uint8_t configuration, protocol, busy;
static int button1, button2, button3, x, y, xrepeat, yrepeat;
static int idle_value, idle_timer;
```

Pomocnicza funkcja ResetState inicjuje wartości wszystkich zmiennych globalnych. Początkowa wartość zmiennej protocol wskazuje, że chcemy używać protokołu opisanego deskryptorem raportu. Stała MAX_REPEAT ma wartość 180. Później pojawi się jeszcze stała MIN_REPEAT o wartości 3. Bez wchodzenia w szczególty oznacza to, że szybkość ruchu wskaźnika myszy będzie w przybliżeniu mieścić się w przedziale od 1 piksela na 180 ms do 1 piksela na 3 ms. Początkowa wartość zmiennych idle_value i idle_timer oznacza czas nieskończony.

```
static void ResetState(void) {
    report.buttons = 0;
    report.x = report.y = 0;
    configuration = 0;
    protocol = 1;
    busy = 0;
    button1 = button2 = button3 = x = y = 0;
    xrepeat = yrepeat = MAX_REPEAT;
    idle_value = idle_timer = -1;
}
```

Funkcja zwrotna Configure jest wywoływana tylko raz podczas uruchamiania mikrokontrolera. Jej zadaniem jest uruchomienie sprzętu specyficznego dla aplikacji. Funkcja ta jest bezparametrowa i powinna zwrócić zero, gdy zakończy się powodzeniem, a wartość ujemną, gdy wystąpi jakiś błąd. Jak widać na poniższym wydruku, najpierw ustawiamy wartości początkowe wszystkich zmiennych globalnych. Działanie myszy będziemy symulować za pomocą różnego rodzaju dżojstików. W podstawowej wersji będzie to po prostu siedem przycisków monostabilnych: trzy emulujące odpowiednio lewy, prawy i środkowy przycisk myszy, a pozostałe czte-

ry dla uzyskania ruchu wskaźnika myszy odpowiednio w górę, dół, prawo i lewo. Dżojstik konfigurujemy za pomocą funkcji JoystickConfigure. Szczegółowo opisuję ją w następnym podrozdziale.

```
int Configure() {
    ResetState();
    if (JoystickConfigure() < 0)
        return -1;
    return 0;
}
```

Funkcja zwrotna Reset jest wywoływana za każdym razem, gdy kontroler wykona procedurę zerowania szyny USB. Jej argumentem jest szybkość, z jaką ma pracować urządzenie. Z uwagi na ograniczenia sprzętowe nie możemy zrealizować urządzenia LS, a mysz HS byłaby sporą przesadą. Dopuszczamy zatem jedynie tryb FS. Następnie przywracamy wartości początkowe wszystkich zmiennych globalnych i zerujemy stan dżojstika za pomocą funkcji JoystickReset. Za pomocą funkcji USBDendPointConfigure, której szczegółowy opis odkładam do następnego rozdziału, konfigurujemy punkt końcowy zero (parametr ENDPO), który jest domyślnym punktem końcowym dla danych sterujących (parametr CONTROL_TRANSFER). Ten punkt końcowy musi zostać uruchomiony po wyzerowaniu szyny USB, aby umożliwić komunikację z urządzeniem. Jeśli coś poszło źle, to wywołujemy funkcję ErrorResetable, która zasygnałizuje problem, odczeka pewien czas, a następnie wyzeruje mikrokontroler, podejmując w ten sposób ponowną próbę uruchomienia urządzenia. Funkcja Reset zwraca wartość z pola bMaxPacketSize0 deskryptora urządzenia, czyli maksymalny rozmiar pola danych pakietów sterujących.

```
uint8_t Reset(usb_speed_t speed) {
    ErrorResetable(speed == FULL_SPEED ? 0 : -1, 6);
    ResetState();
    ErrorResetable(JoystickReset(), 9);
    if (USBDendPointConfigure(ENDPO, CONTROL_TRANSFER,
                               device_descriptor.bMaxPacketSize0,
                               device_descriptor.bMaxPacketSize0) != REQUEST_SUCCESS)
        ErrorResetable(-1, 7);
    return device_descriptor.bMaxPacketSize0;
}
```

Żądanie GET_CONFIGURATION jest realizowane za pomocą bezparametrowej funkcji zwrotnej GetConfiguration, która po prostu zwraca numer aktualnie wybranej konfiguracji (zero, gdy żadna konfiguracja nie została wybrana).

```
uint8_t GetConfiguration() {
    return configuration;
}
```

Żądanie SET_CONFIGURATION jest realizowane za pomocą funkcji zwrotnej SetConfiguration. Jej parametrem jest numer konfiguracji, która ma być ustaliona. Najpierw sprawdzamy, czy parametr ten ma poprawną wartość. Jeśli kontroler żąda ustawienia konfiguracji, której nie obsługujemy, odrzucamy żądanie, sygnalizując to przez zwrócenie wartości REQUEST_ERROR. Następnie zapisujemy nowy numer konfiguracji w zmiennej configuration. Po czym dezaktywujemy za pomocą funkcji USBDdisableAllNonControlEndPoints wszystkie punkty końcowe z wyjątkiem domyślnego punktu końcowego dla danych sterujących – ustawienie nowej konfiguracji wymaga dezaktywowania punktów końcowych związanych z poprzednią konfiguracją i zainicjowania punktów końcowych związanych z nową konfiguracją. Jeśli kontroler żąda ustawienia konfiguracji opisanej w którymś z przekazanych mu deskryptorów konfiguracji, aktywujemy związane z tą konfiguracją punkty końcowe za pomocą funkcji USBDendPointConfigure. W tym konkretnym przypadku jest tylko jedna konfiguracja i opisuje tylko jeden punkt końcowy numer 1 (stała ENDP1) dla danych pilnych (stała INTERRUPT_TRANSFER). Poprawne zakończenie żądania sygnalizujemy, zwracając wartość REQUEST_SUCCESS. Zwróćmy uwagę, że tę wartość zwracamy również wtedy, gdy kontroler zażądał dezaktywacji konfiguracji, tzn. gdy parametr confValue ma wartość zero.

```
usb_result_t SetConfiguration(uint16_t confValue) {
    if (confValue > device_descriptor.bNumConfigurations)
        return REQUEST_ERROR;
    configuration = confValue;
    USBDdisableAllNonControlEndPoints();
    if (confValue == hid_configuration.cnf_descr.bConfigurationValue)
        return USBDendPointConfigure(ENDP1, INTERRUPT_TRANSFER, 0,
                                      sizeof(hid_mouse_boot_report_t));
    return REQUEST_SUCCESS;
}
```

Funkcja zwrotna GetStatus obsługuje żądanie GET_STATUS i zwraca informację o aktualnym źródle zasilania urządzenia zgodnie z ustawieniem parametru bmAttributes deskryptora konfiguracji. Urządzenie, które zadeklarowało jednocześnie zasilanie własne (ustawiony bit SELF_POWERED w polu bmAttributes) i z szyny USB (dodatnia wartość pola bMaxPower), może dynamicznie zmieniać źródło zasilania. Wtedy funkcja GetStatus powinna zwrócić stan aktualny.

```
uint16_t GetStatus() {
    if (hid_configuration.cnf_descr.bmAttributes & SELF_POWERED)
        return STATUS_SELF_POWERED;
    else
        return 0;
}
```

Dotychczas przedstawione funkcje zwrotne obsługują standardowe żądania. Żądania specyficzne dla konkretnej klasy i niewymagające przesłania danych obsługują funkcja zwrotna ClassNoDataSetup. Jej argumentem jest wskaźnik do struktury setup przesłanej w fazie ustanowienia żądania w transakcji SETUP. Na podstawie

zawartości tej struktury dekoduje się żądanie i jego parametry. Funkcja zwraca wartość REQUEST_SUCCESS, jeśli zakończyła się sukcesem, a wartość REQUEST_ERROR, jeśli nie obsługuje danego żądania. Poniższy wydruk przedstawia obsługę żądań SET_IDLE i SET_PROTOCOL.

```
usb_result_t ClassNoDataSetup(usb_setup_packet_t const *setup) {
    if (setup->bmRequestType == (HOST_TO_DEVICE |
                                    CLASS_REQUEST |
                                    INTERFACE_RECIPIENT)) {
        if (setup->bRequest == SET_IDLE &&
            setup->wIndex == 0 &&
            setup->wLength == 0 &&
            (setup->wValue & 0xff) == 0 /* report ID == 0 */) {
            if (setup->wValue & 0xff00)
                idle_value = idle_timer = (setup->wValue & 0xff00) >> 6;
            else
                idle_value = idle_timer = -1; /* infinity */
            return REQUEST_SUCCESS;
        }
        else if (setup->bRequest == SET_PROTOCOL &&
                  setup->wIndex == 0 &&
                  setup->wLength == 0) {
            protocol = setup->wValue;
            return REQUEST_SUCCESS;
        }
    }
    return REQUEST_ERROR;
}
```

Żądania specyficzne dla konkretnej klasy, ale wymagające przesłania danych z urządzenia do kontrolera, obsługuje funkcja zwrotna ClassInDataSetup. Jej pierwszym argumentem jest wskaźnik do struktury setup przesłanej w fazie ustanowienia żądania w transakcji SETUP. Na podstawie zawartości tej struktury dekoduje się żądanie i jego parametry. Za pomocą parametru data zwraca się wskaźnik na dane do przesłania, a za pomocą parametru length przekazuje się ich rozmiar. Przekazany wskaźnik musi być ważny również po zakończeniu funkcji aż do zakończenia fazy transmisji danych. Zatem przekazany wskaźnik musi wskazywać na dane statyczne lub globalne, które nie znikną po zakończeniu wywołania funkcji. Funkcja zwraca wartość REQUEST_SUCCESS, jeśli zakończyła się sukcesem, a wartość REQUEST_ERROR, jeśli nie obsługuje danego żądania. Poniższy wydruk przedstawia obsługę żądań GET_REPORT i GET_PROTOCOL.

```
usb_result_t ClassInDataSetup(usb_setup_packet_t const *setup,
                             uint8_t const **data,
                             uint16_t *length) {
```

```

if (setup->bmRequestType == (DEVICE_TO_HOST |
                                CLASS_REQUEST |
                                INTERFACE_RECIPIENT)) {
    if (setup->bRequest == GET_REPORT &&
        setup->wIndex == 0 &&
        setup->wValue == 0x0100 /* input report, report ID == 0 */) {
        *data = (uint8_t const *)&report;
        *length = sizeof(report);
        return REQUEST_SUCCESS;
    }
    else if (setup->bRequest == GET_PROTOCOL &&
              setup->wValue == 0 &&
              setup->wIndex == 0 &&
              setup->wLength == 1) {
        *data = &protocol;
        *length = 1;
        return REQUEST_SUCCESS;
    }
}
return REQUEST_ERROR;
}

```

Na początku każdej ramki lub mikroramki jest wywoływana funkcja zwrotna SoF (ang. *start of frame*). Zatem w przypadku naszego urządzenia FS jest ona wywoływana co 1 ms. Wykorzystamy tę okoliczność do zaimplementowania zasadniczej części algorytmu działania myszy. Argumentem funkcji SoF jest numer kolejnej ramki, jednak nie będziemy z niego korzystać. Funkcja ta nie zwraca żadnej wartości. W naszym przypadku funkcja SoF rozpoczyna się odczytaniem stanu dżojstika za pomocą funkcji JoystickGetState, którą opisuję szczegółowo w następnym podrozdziale. Dla ustalenia uwagi przyjmijmy, że jest to stan, wspomnianych wyżej, siedmiu przycisków monostabilnych. Dla każdego z trzech przycisków symulujących przyciski myszy zaimplementowany jest ten sam algorytm, którego zadaniem jest likwidacja efektu drgania styków. Inny algorytm, którego zadaniem jest uzyskanie efektu akceleracji ruchu, zastosowany jest do obsługi przesuwania wskaźnika myszy w poziomie i pionie. Na poniższym wydruku uwidoczniony jest tylko tekst źródłowy dla lewego przycisku myszy i przesuwania w poziomie. Dla pozostałych dwóch przycisków i przesuwania w pionie jest on analogiczny.

```

void SoF(uint16_t frameNumber) {
    unsigned state;
    state = JoystickGetState();
    if (!(state & JOYSTICK_BUTTON_1) !=
        !(report.buttons & MOUSE_LEFT_BUTTON)) {
        ++button1;
        if (button1 >= DEBOUNCE) {

```

```
        button1 = 0;
        report.buttons ^= MOUSE_LEFT_BUTTON;
        idle_timer = 0;
    }
}
else {
    button1 = 0;
}
...
if (state & JOYSTICK_RIGHT) {
    ++x;
    if (x >= xrepeat && report.x < 127) {
        x = 0;
        xrepeat = (xrepeat + MIN_REPEAT) >> 1;
        ++report.x;
        idle_timer = 0;
    }
}
else if (state & JOYSTICK_LEFT) {
    --x;
    if (x <= -xrepeat && report.x > -127) {
        x = 0;
        xrepeat = (xrepeat + MIN_REPEAT) >> 1;
        --report.x;
        idle_timer = 0;
    }
}
else {
    xrepeat = MAX_REPEAT;
}
...
if (idle_timer > 0)
    --idle_timer;
if (configuration > 0 && idle_timer == 0 && busy == 0) {
    USBWrite(ENDP1, (uint8_t const *)report, sizeof(report));
    report.x = report.y = 0;
    busy = 1;
    idle_timer = idle_value;
}
}
```

Bieżący stan przycisków myszy przechowujemy w polu buttons struktury report. Odczytany stan dżojstika zapisany jest w zmiennej state. Porównujemy stan pierwszego przycisku dżojstika (stała JOYSTICK_BUTTON_1) ze stanem lewego przy-

cisku myszy (stała MOUSE_LEFT_BUTTON). Zmienna button1 zlicza, przez ile kolejnych milisekund te stany się różnią. Jeśli zmieniony stan jest stabilny przez co najmniej DEBOUNCE milisekund, to znaczy, że drgania styków już nie występują i nowy stan jest zapisywany w polu buttons. Zerowana jest wtedy zmienna idle_timer, aby zmieniony raport został wysłany przy najbliższej nadarzającej się okazji. Stała DEBOUNCE ma wartość 5. Można oczywiście poeksperymentować z innymi jej wartościami.

Pole x struktury report przechowuje liczbę pikseli, o które ma być przesunięty kursor myszy w poziomie. Zmienna x przechowuje skumulowaną liczbę milisekund, przez które dżojstik był wychylony. Wartość tej zmiennej jest dodatnia dla wychylenia w prawo, a ujemna dla wychylenia w lewo. Jeśli osiągnie wartość xrepeat lub -xrepeat oraz pole x struktury report nie zawiera wartości ekstremalnej, to kursor myszy będzie przesunięty odpowiednio o jeden piksel w prawo lub lewo. Przy długim przechyleniu dżojstika w jedną stronę wartość zmiennej xrepeat jest stopniowo zmniejszana do wartości MIN_REPEAT dla uzyskania efektu coraz szybszego ruchu wskaźnika myszy. Jeśli przyciski JOYSTICK_RIGHT i JOYSTICK_LEFT powróćą do położenia neutralnego, to zmienna ta jest ponownie inicjowana wartością MAX_REPEAT. Po każdej zmianie zawartości raportu zerowana jest zmienna idle_timer, aby zainicjować wysłanie zmienionego raportu przy najbliższej nadarzającej się okazji. Można poeksperymentować z wartościami stałych MIN_REPEAT i MAX_REPEAT, aby uzyskać inne przyspieszenia ruchu wskaźnika myszy.

W każdym wywołaniu funkcji SoF, jeśli zmienna idle_timer ma wartość dodatnią, zmniejszamy jej wartość o jeden. Jeśli konfiguracja urządzenia jest aktywna (zmienna configuration ma dodatnią wartość), nadszedł czas wysłania kolejnego raportu (zmienna idle_timer ma wartość zero) oraz poprzedni raport został wysłany (zmienna busy ma wartość zero), to wysyłamy nowy raport przez skonfigurowany wcześniej punkt końcowy (stała ENDP1) za pomocą funkcji USBDwrite, którą opisuję szczegółowo w następnym rozdziale. Funkcja ta ma semantykę kopирования, to znaczy, że przekazane jej do wysłania dane są kopowane do wewnętrznego bufora lub kolejki nadawczej USB. Jest to wygodne, gdyż natychmiast po powrocie z tej funkcji możemy wyzerować pola x i y raportu report. Pola te przechowują względne przesunięcie w stosunku do poprzedniego raportu i dlatego, przygotowując strukturę report do przesłania nowego raportu, musimy zacząć od wartości zerowych. Zakończenie funkcji USBDwrite nie oznacza jednak, że dane zostały faktycznie wysłane. Dlatego wpisujemy jedynkę do zmiennej busy, oznaczając w ten sposób, że dane oczekują na wysłanie. Po każdym wstawieniu raportu do wysłania ponownie inicjujemy też zmienną idle_timer, odliczającą czas do wysłania kolejnego raportu.

```
void EP1IN() {
    busy = 0;
}
```

Jeśli dane wstawione do wysłania przez punkt końcowy numer 1 zostaną wysłane, to zostaniemy o tym poinformowani przez wywołanie funkcji zwrotnej EP1IN. Jedyną rzeczą, którą musimy w tym przypadku zrobić, jest wyzerowanie zmiennej

busy, co pozwoli wstawić do wysłania kolejną porcję danych, czyli kolejny raport. Rozwiążanie ze zmienną `busy` jest konieczne, gdyż w deskryptorze punktu końcowego wpisaliśmy w polu `bInterval` wartość 10. Zatem kontroler odpytuje (ang. *poll*) raz na 10 ramek, czy są jakieś dane do wysłania przez ten punkt końcowy, więc w pesymistycznym przypadku wstawiony do wysłania raport będzie czekał na wysłanie co najwyżej 10 ms. Trwające krócej wciśnięcia przycisków zostaną zignorowane. Natomiast przesunięcie myszy kumuluje się i zostanie wysłane w następnym raporcie. Nie sprawia to zatem żadnego problemu. Podobnie w przypadku raportu dla klawiatury, gdzie można zignorować bardzo krótkie wciśnięcia klawiszy modyfikujących, a w raporcie zebrać do sześciu wciśnień zwykłych klawiszy.

```
usbd_callback_list_t const * USBDgetApplicationCallbacks() {
    return &ApplicationCallBacks;
}
```

Żeby posługiwanie się funkcjami zwrotnymi było dostatecznie elastyczne, nie są one bezpośrednio eksportowane na zewnątrz modułu. W rzeczywistości wszystkie one są zadeklarowane jako `static` i są prywatnymi funkcjami jednostki translacji, w której zostały zadeklarowane. Adresy funkcji zwrotnych umieszcza się w specjalnej strukturze danych typu `usbd_callback_list_t`. Dzięki temu trzeba zaimplementować tylko rzeczywiście używane funkcje zwrotne i można im nadać dowolne nazwy. Nieużywane funkcje zwrotne oznacza się, wpisując jako ich adres wskaźnik zerowy. Jedyną funkcją, którą musimy wyeksportować, jest wypisana wyżej funkcja zwrotna `USBDgetApplicationCallbacks`, która zwraca wskaźnik do struktury zawierającej wskaźniki do pozostałych funkcji zwrotnych. Więcej o tym piszę w następnym rozdziale.

3.1.5. Dżojstik

<code>joystick.h</code>
<code>joystick_gpio_10x.c</code>

Do zmiany położenia wskaźnika myszy i symulowania jej przycisków posłuży nam urządzenie, które będziemy umownie nazywać dżojstikiem. Interfejs dżojstika zdefiniowany jest w pliku `joystick.h`. Podstawowa implementacja dla mikrokontrolerów STM32F10x znajduje się w pliku `joystick_gpio_10x.c`. Implementacja ta używa siedmiu przycisków monostabilnych. Trzy przyciski symulują odpowiednio lewy, prawy i środkowy przycisk myszy. Pozostałe cztery przyciski służą do przesywania wskaźnika myszy odpowiednio w górę, dół, prawo i lewo. Aktywnym stanem przycisków jest poziom niski, czyli wciśnięty przycisk podaje na wejście poziom logiczny 0. Moduł `joystick` udostępnia trzy funkcje.

<code>int JoystickConfigure(void);</code>

W podstawowej implementacji bezparametrowa funkcja `JoystickConfigure` konfiguruje porty wejściowe dżojstika. W innych implementacjach może też konfigurować inne potrzebne układy peryferyjne. Funkcja ta zwraca zero, gdy konfigurowanie zakończyło się sukcesem, a wartość ujemną, gdy podczas konfigurowania

wystąpił błąd. Podstawowa implementacja zawsze zwraca zero. Możliwość poinformowania o błędzie przydaje się w innych implementacjach dżojstika, gdzie trzeba skonfigurować układy peryferyjne inne niż porty wejścia-wyjścia, a taka operacja już nie zawsze musi się zakończyć powodzeniem.

Nie zawsze uda się znaleźć siedem wolnych wprowadzeń na jednym porcie wejścia-wyjścia, dlatego podstawowa implementacja dżojstika przewiduje możliwość użycia dwóch 16-bitowych portów, które wspólnie tworzą wirtualny port 32-bitowy. W pliku *board_def.h* trzeba zdefiniować następujące stałe:

- JOYSTICK_L_GPIO_N – literowe oznaczenie portu wejścia-wyjścia, który zostanie użyty jako 16 młodszych bitów portu wirtualnego;
- JOYSTICK_H_GPIO_N – literowe oznaczenie portu wejścia-wyjścia, który zostanie użyty jako 16 starszych bitów portu wirtualnego – definiowana tylko wtedy, gdy potrzebny jest drugi port.

Ponadto trzeba zdefiniować siedem masek bitowych. Każda z nich ma ustalony tylko jeden bit, który wskazuje pozycję odpowiedniego przycisku dżojstika w 32-bitowym porcie wirtualnym. Są to następujące stałe:

- PIN_JOYSTICK_BUTTON_1 – pierwszy przycisk dżojstika – lewy przycisk myszy;
- PIN_JOYSTICK_BUTTON_2 – drugi przycisk dżojstika – prawy przycisk myszy;
- PIN_JOYSTICK_BUTTON_3 – trzeci przycisk dżojstika – środkowy przycisk myszy;
- PIN_JOYSTICK_UP – przycisk dżojstika do góry;
- PIN_JOYSTICK_DOWN – przycisk dżojstika w dół;
- PIN_JOYSTICK_RIGHT – przycisk dżojstika w prawo;
- PIN_JOYSTICK_LEFT – przycisk dżojstika w lewo.

Przykładowo, jeśli wszystkie przyciski są podłączone do portu PC, a mianowicie: PC9 – pierwszy, PC12 – drugi, PC13 – trzeci, PC5 – do góry, PC6 – w dół, PC7 – w prawo, PC8 – w lewo, to definicje powinny wyglądać tak:

```
#define JOYSTICK_L_GPIO_N      C
#define PIN_JOYSTICK_BUTTON_1   0x00000200U
#define PIN_JOYSTICK_BUTTON_2   0x00001000U
#define PIN_JOYSTICK_BUTTON_3   0x00002000U
#define PIN_JOYSTICK_UP         0x00000020U
#define PIN_JOYSTICK_DOWN       0x00000040U
#define PIN_JOYSTICK_RIGHT      0x00000080U
#define PIN_JOYSTICK_LEFT       0x00000100U
```

Jeśli przyciski podłączone są z wykorzystaniem dwóch portów, na przykład: PA9 – pierwszy, PC13 – drugi, PC15 – trzeci, PA4 – do góry, PA5 – w dół, PA7 – w prawo, PA8 – w lewo, to definicje wyglądają następująco:

```
#define JOYSTICK_L_GPIO_N      A
#define JOYSTICK_H_GPIO_N       C
#define PIN_JOYSTICK_BUTTON_1   0x00000200U
```

```
#define PIN_JOYSTICK_BUTTON_2 0x20000000U
#define PIN_JOYSTICK_BUTTON_3 0x80000000U
#define PIN_JOYSTICK_UP        0x00000010U
#define PIN_JOYSTICK_DOWN      0x00000020U
#define PIN_JOYSTICK_RIGHT     0x00000080U
#define PIN_JOYSTICK_LEFT      0x00000100U

int JoystickReset(void);
```

Bezparametrowa funkcja `JoystickReset` przywraca początkowy stan dżojstika. Funkcja ta zwraca zero, gdy operacja zakończyła się sukcesem, a wartość ujemną, gdy wystąpił błąd. W podstawowej implementacji ta funkcja nic nie robi i zawsze zwraca zero. W zamyśle funkcja ta przewidziana jest dla innych implementacji, które mają stan wewnętrzny wymagający wyzerowania w pewnych sytuacjach, na przykład po wyzerowaniu szyny USB.

```
unsigned JoystickGetState(void);
```

Bezparametrowa funkcja `JoystickGetState` zwraca aktualny stan przycisków dżojstika. Zwracana wartość jest bitową alternatywą poniższych stałych, których znanie powinno być oczywiste.

```
#define JOYSTICK_BUTTON_1    0x0001
#define JOYSTICK_BUTTON_2    0x0002
#define JOYSTICK_BUTTON_3    0x0004
#define JOYSTICK_UP          0x0010
#define JOYSTICK_DOWN         0x0020
#define JOYSTICK_RIGHT        0x0040
#define JOYSTICK_LEFT          0x0080
```

<i>ioe_stm322xg.h</i>
<i>ioe_stm322xg.c</i>
<i>joystick_322xg.c</i>
<i>joystick_mems_4xx.c</i>
<i>lis302dl.h</i>
<i>lis302dl_disco_4xx.c</i>

Oprócz podstawowej implementacji dżojstika w archiwum z przykładami znajdują się jeszcze dwie implementacje zachowujące ten sam interfejs zdefiniowany w pliku `joystick.h`. Plik `joystick_322xg.c` zawiera implementację przeznaczoną dla zestawu ewaluacyjnego STM3220G-EVAL. W zestawie tym dżojstik podłączony jest do eksplandera wejścia-wyjścia, który z kolei podłączony jest do mikrokontrolera za pomocą szyny I²C. Użycie tej implementacji wymaga dołączenia do projektu jeszcze dwóch plików: `ioe_stm322xg.c` – obsługa eksplandera wejścia-wyjścia zainstalowanego w zestawie, `i2c.c` – obsługa interfejsu I²C (opisana w rozdziale 2). Potrzebne są też odpowiednie pliki nagłówkowe: `ioe_stm322xg.h` i `i2c.h`. W pliku `joystick_mems_4xx.c`

znajduje się implementacja przeznaczona dla modułu STM32F4-Discovery, w którym dżojstik emulowany jest za pomocą akcelerometru wykonanego w technologii MEMS. Użycie tej implementacji wymaga dołączenia do projektu pliku *lis302dl_disco_4xx.c* z implementacją obsługi akcelerometru LIS302DL zainstalowanego na tym module. W pliku nagłówkowym *lis302dl.h* zdefiniowano interfejs obsługi tego akcelerometru.

3.1.6. Funkcja main

usb_main.c

Implementacja funkcji `main` dla wszystkich przedstawionych w tej książce urządzeń USB znajduje się w pliku *usb_main.c*. Na początku odczytujemy parametry aplikacji: częstotliwość w megahercach, z jaką ma być taktowany rdzeń (zmienna `sysclk`), szybkość, z jaką ma działać interfejs USB (zmienna `speed`) oraz wariant nadajnika-odbiornika, który ma być użyty (zmienna `phy`). Następnie przystępujemy do konfigurowania poszczególnych podukładów. Przez cały czas konfigurowania świeci czerwona dioda. Konfigurowanie USB jest podzielone na dwa etapy. Pierwszy realizuje funkcja `USBConfigure`, która musi być wywołana możliwie szybko po włączeniu zasilania. Drugi, zasadniczy etap konfigurowania USB zaimplementowany jest w funkcji `USBConfigure`. Taki podział powodowany jest tym, że specyfikacja USB wymaga dość krótkiego czasu między rozpoznaniem podłączenia urządzenia do szyny a uzyskaniem przez niego gotowości do działania. Może się jednak okazać, że przed uruchomieniem interfejsu USB trzeba skonfigurować wiele innych peryferii, co zajmuje sporo czasu. Aby symulować długi czasu konfigurowania peryferii, wprowadzono wywołanie funkcji `Delay`. Głównym zadaniem funkcji `USBConfigure` jest skonfigurowanie rezystora podciągającego i pozostawienie go w stanie odłączonym, aby kontroler nie wykrył podłączenia naszego urządzenia. Rezystor podciągający jest włączany dopiero w funkcji `USBConfigure`, co rozpoczęyna właściwą procedurę konfigurowania interfejsu USB. Niektóre wersje sprzętu zapewniają, że rezistor podciągający jest domyślnie odłączony. Wtedy taki podział konfigurowania na dwa etapy jest zbędny i funkcja `USBConfigure` jest (prawie) pusta, ale pozostawiłem ją, aby zachować jednolite API dla wszystkich wariantów sprzętu.

```
int main(void) {
    int          sysclk;
    usb_speed_t speed;
    usb_phy_t   phy;

    GetBootParams(&sysclk, &speed, &phy);
    AllPinsDisable();
    LEDconfigure();
    RedLEDon();
    ErrorResetable(USBConfigure(speed, phy), 1);
    Delay(2000000);
    IRQprotectionConfigure();
```

```
ErrorResetable(ClockConfigure(sysclk), 2);  
ErrorResetable(LCDconfigure(), 3);  
ErrorResetable(PWRconfigure(HIGH_IRQ_PRIO, 0, sysclk), 4);  
ErrorResetable(USBConfigure(MIDDLE_IRQ_PRIO, 0, sysclk), 5);  
RedLEDDoff();  
for (;;)  
    LCDrunRefresh();  
}
```

Błąd podczas konfigurowania urządzenia jest sygnalizowany miganiem czerwonej diody, po czym mikrokontroler jest zerowany – zadanie to wykonuje funkcja ErrorResetable. Po poprawnym zakończeniu konfigurowania wszystkich układów i modułów czerwona dioda gaśnie, a następnie w nieskończonej pętli wywoływana jest funkcja LCDrunRefresh odświeżająca ekran wyświetlacza ciekłokrystalicznego. Funkcja ta nic nie robi, jeśli nie zainstalowano wyświetlacza ciekłokrystalicznego lub nie zarejestrowano funkcji zwrotnej obsługującej jego odświeżanie. Taki sposób obsługi wyświetlacza wynika z dwóch przesłanek. Z jednej strony odświeżanie wyświetlacza jest zwykle dość czasochłonne, więc nie powinno odbywać się wewnętrz procedury obsługującej przerwanie. Jest to bardzo istotne w przypadku interfejsu USB, który ma bardzo restrykcyjne zależności czasowe i przetrzymywanie sterowania w procedurze obsługującej jego przerwanie może doprowadzić do jego błędnego funkcjonowania. Z drugiej strony zadanie odświeżania wyświetlacza może wykonywać się z bardzo małym priorytetem, gdy nie ma żadnych przerwań do obsłużenia.

Funkcja PWRconfigure konfiguruje zarządzanie energią urządzenia USB. Dla urządzeń prezentowanych w tym i następnym rozdziale jest ona pusta. Zarządzanie energią jest tematem rozdziału 5 i dopiero w przykładzie prezentowanego tam urządzenia zostanie omówiona jej właściwa implementacja. Dokładny opis funkcji konfiguujących USB, czyli USBConfigure i USBDConfigure, znajduje się w rozdziale 4. Opisu pozostałych wywoływanych funkcji należy szukać w rozdziale 2.

3.1.7. Kompilowanie i testowanie

Archiwum z przykładami zawiera kilka wersji projektu myszy. W katalogu *./make* znajdują się podkatalogi, których nazwy rozpoczynają się przedrostkiem *usb1_hid_device*. W tych podkatalogach umieszczone są pliki *makefile* umożliwiające skompilowanie tego projektu dla przykładowych wariantów sprzętu. Można je oczywiście łatwo dostosować do innego sprzętu, posługując się wskazówkami z rozdziału 2. Pliki *makefile* zawierają listę plików źródłowych niezbędnych do skompilowania programu, są zatem przydatne również dla tych, którzy nie chcą korzystać bezpośrednio z programu *make*. Wtedy należy pamiętać, aby do projektu dołączyć również plik *startup_stm32.c* i właściwą wersję biblioteki STM32.

We wszystkich popularnych systemach operacyjnych sterownik urządzeń z interfejsem klasy HID jest jednym z najlepiej dopracowanych, dlatego zarówno w systemie Linux, jak i Windows nasze urządzenie jest automatycznie rozpoznawane jako

mysz. Bez problemu można też równocześnie używać dwóch myszy – tej zwykle podłączonej do komputera i tej zaimplementowanej w tym projekcie.

3.2. Projekt wirtualnego portu szeregowego

Drugi projekt demonstruje przykład urządzenia CDC (ang. *Communication Device Class*) wykorzystującego dane masowe (ang. *bulk*). Klasa ta obejmuje urządzenia telekomunikacyjne i sieciowe:

- modem analogowy i ADSL;
- telefon analogowy, cyfrowy, terminal ISDN;
- adapter lub koncentrator ethernetowy.

Urządzenie omówione w tym podręczniku jest rozpoznawane przez komputer osobisty jako wirtualny port szeregowy, czyli COM w Windows, a /dev/ttyACM w Linuksie. Jest to emulacja bardzo starego, ale wciąż popularnego interfejsu RS-232. Wizualna demonstracja działania tego urządzenia polega na sterowaniu diodami świecącymi zainstalowanymi na płytce prototypowej oraz wyświetlaniu na LCD stanu interfejsu RS-232.

Współcześnie sprzedawane komputery osobiste nie są już wyposażane w interfejs RS-232, za to standardowo wszystkie mają interfejs USB. Odwrotna sytuacja ma miejsce w przypadku oprogramowania. Nadal używa się wielu programów, które obsługują RS-232, ale za to nie obsługują USB. Prezentowany projekt można wykorzystać jako szkielet aplikacji, która zapewni komunikację między mikrokontrolerem a programem przystosowanym do obsługi RS-232 bez stosowania konwertera USB/RS-232 w sytuacji, gdy w komputerze dostępny jest jedynie interfejs USB, co ma jeszcze tę zaletę, że USB – w przeciwieństwie do RS-232 – pozwala bezproblemowo zasilać mikrokontroler. Dalszych informacji na temat CDC należy szukać w [24] i [25].

3.2.1. Deskryptory

Dla urządzeń komunikacyjnych CDC w polu bDeviceClass deskryptora urządzenia wpisuje się wartość 2. Pól bDeviceSubClass, bDeviceProtocol nie używa się i mają one zawsze wartość 0. Realizowaną przez urządzenie funkcję opisuje się na poziomie interfejsu. Specyfikacja CDC definiuje dwie klasy interfejsów przeznaczone odpowiednio do:

- sterowania komunikacją (ang. *communication interface class*);
- przesyłania właściwych danych (ang. *data interface class*).

Dla obu tych klas zdefiniowano wiele podklas i protokołów. Wybrane wartości, które można wpisać w polach deskryptora interfejsu określających jego klasę, podklasę i protokół, zamieszczone są w tabelach 3.7 i 3.8. Pozostałe pola deskryptora urządzenia i deskryptorów interfejsów wypełnia się zgodnie ze wskazówkami z rozdziału 1. W przypadku interfejsu sterującego komunikacją podklaśa wyznacza model, według którego odbywa się to sterowanie. Na przykład model sterowania telefonem obejmuje m.in. podniesienie słuchawki, odłożenie słuchawki, wybranie numeru czy sygnalizację połączenia przychodzącego (dzwonienie), a odpowiedni

protokół definiuje żądania realizujące te czynności. Najpopularniejszym protokołem realizującym ten model jest protokół AT, który opracowano pierwotnie do komunikacji z modemem analogowym. Jego nazwa pochodzi stąd, że wszystkie polecenia są napisami ASCII zaczynającymi się od liter AT. Rozszerzona wersja tego protokołu wykorzystywana jest do komunikacji z modułami i modemami GSM.

Tab. 3.7. Wybrane wartości parametrów deskryptora interfejsu sterującego komunikacją

Pole	Rozmiar	Wartość	Opis
bInterfaceClass	1	2	Interfejs sterujący komunikacją (ang. <i>communication interface class</i>)
bInterfaceSubClass	1	1	Model sterowania łączem bezpośrednim (ang. <i>direct line control model</i>)
		2	Abstrakcyjny model sterowania (ang. <i>abstract control model</i>)
		3	Model sterowania telefonem (ang. <i>telephone control model</i>)
		5	Model sterowania CAPI (ang. <i>CAPI control model</i>)
		11	OBEX
		13	Model sterowania siecią (ang. <i>network control model</i>)
bInterfaceProtocol	1	0	Specyficzny protokół nie jest wymagany
		1...6	Różne warianty protokołu AT
		7	Emulacja Ethernetu

W przypadku wirtualnego portu szeregowego interfejs sterujący zawiera wejściowy punkt końcowy dla danych pilnych. Za pomocą tego punktu końcowego przesyła się do kontrolera USB powiadomienia o stanie interfejsu szeregowego – patrz następny podrozdział. Natomiast właściwe dane przesyła się za pomocą dwukierunkowego punktu końcowego dla danych masowych związanego z interfejsem dla transmisji danych.

Tab. 3.8. Wybrane wartości parametrów deskryptora interfejsu dla transmisji danych

Pole	Rozmiar	Wartość	Opis
bInterfaceClass	1	10	Interfejs dla danych (ang. <i>data interface class</i>)
bInterfaceSubClass	1	0	Nieużywane
		0	Specyficzny protokół nie jest wymagany
		48	I.430
		49	HDLC
		50	Przezroczysty (ang. <i>transparent</i>)
		80	Q.921M
		81	Q.921
		82	Q.921TM
		144	V.42bis
		145	Q.932/Euro-ISDN
		146	V.120
		147	CAPI2.0

Standard USB CDC definiuje kilkadziesiąt specyficznych deskryptorów, nazywanych deskryptorami funkcjonalnymi. Ich ogólny format przedstawiony jest w **tablicy 3.9**. Pierwsze pole bFunctionLength zwyczajowo zawiera rozmiar deskryptora

w bajtach. W polu `bDescriptorType` umieszcza się typ deskryptora, wartość 36 oznacza deskryptor funkcjonalny interfejsu, a wartość 37 deskryptor funkcjonalny punktu końcowego. Wartość umieszczona w polu `bDescriptorSubtype` rozróżnia podtypy w obrębie typu. Potem następuje zawartość deskryptora specyficzna dla danego typu i podtypu. W praktyce deskryptorów CDC używa się tylko w połączeniu z interfejsem sterującym komunikacją, umieszczając je w hierarchii deskryptorów konfiguracji bezpośrednio po standardowym deskryptorze interfejsu. Szczegółowe omawianie wszystkich podtypów deskryptorów nie wydaje się celowe. Ograniczę się zatem tylko do przedstawienia przykładu.

Tab. 3.9. Ogólny format deskryptora funkcjonalnego CDC

Pole	Rozmiar	Opis
<code>bFunctionLength</code>	1	Rozmiar deskryptora ($N + 3$)
<code>bDescriptorType</code>	1	Typ deskryptora (36 lub 37)
<code>bDescriptorSubtype</code>	1	Podtyp deskryptora
...	N	Właściwa zawartość deskryptora

W tabeli 3.10 przedstawiono hierarchię deskryptorów CDC dla wirtualnego portu szeregowego. Składa się ona z czterech deskryptorów specyficznych dla klasy. Pierwszy deskryptor to nagłówek (ang. *header functional descriptor*) deklarujący w polu `bcdCDC` wersję specyfikacji CDC, z którą w zamierzeniu ma być zgodne urządzenie przedstawiające się tym deskryptorem. W tym konkretnym przypadku jest to wersja 1.2.

Drugi deskryptor opisuje interfejs, którym przesyła się wybierany numer telefonu (ang. *call management functional descriptor*). Jest to istotne dla urządzeń korzystających z sieci telefonicznej. Szczegóły zawarte są w polu bitowym `bmCapabilities`:

- Bit 0 determinuje, czy urządzenie obsługuje wybieranie numeru. Zero oznacza, że nie obsługuje, a jedynka, że obsługuje.
- Jeśli bit 0 ma wartość jeden, to bit 1 określa interfejs, za pomocą którego wybierany jest numer, a przeciwnym przypadku wartość tego bitu jest ignorowana. Jeśli bit 1 ma wartość zero, to numer wysyłany lub odbierany jest tylko za pomocą interfejsu sterującego komunikacją. Jeśli bit 1 ma wartość jeden, to numer może być wysyłany lub odbierany interfejsem dla danych wskazanym za pomocą parametru `bDataInterface`.
- Bity 2...7 są zarezerwowane i powinny mieć wartość zero.

Trzeci deskryptor doprecyzowuje zestaw żądań obsługiwanych przez abstrakcyjny model komunikacji (ang. *abstract control model functional descriptor*). Szczegóły zawarte są w polu bitowym `bmCapabilities` tego deskryptora:

- Bit 0 jest ustawiony, jeśli urządzenie obsługuje żądania `SET_COMM_FEATURE`, `CLEAR_COMM_FEATURE` i `GET_COMM_FEATURE`.
- Bit 1 jest ustawiony, jeśli urządzenie obsługuje żądania `SET_LINE_CODING`, `GET_LINE_CODING`, `SET_CONTROL_LINE_STATE` oraz powiadamianie o stanie interfejsu szeregowego.
- Bit 2 jest ustawiony, jeśli urządzenie obsługuje żądanie `SEND_BREAK`.

- Bit 3 jest ustawiony, jeśli urządzenie obsługuje powiadamianie NETWORK_CONNECTION.
- Bity 4...7 są zarezerwowane i powinny mieć wartość zero.

Tab. 3.10. Hierarchia deskryptorów CDC dla wirtualnego portu szeregowego

Pole	Rozmiar	Wartość	Opis
bFunctionLength	1	5	Rozmiar deskryptora
bDescriptorType	1	36	Deskryptor interfejsu specyficzny dla klasy
bDescriptorSubtype	1	0	Nagłówek
bcdCDC	2	0x0120	Wersja 1.2 specyfikacji CDC
bFunctionLength	1	5	Rozmiar deskryptora
bDescriptorType	1	36	Deskryptor interfejsu specyficzny dla klasy
bDescriptorSubtype	1	1	Wybieranie numeru telefonicznego
bmCapabilities	1	0 lub 3	Pole bitowe określające interfejs używany do wybierania numeru telefonicznego
bDataInterface	1	1	Numer interfejsu dla danych, jeśli ten interfejs ma być używany do wybierania numeru telefonicznego
bFunctionLength	1	4	Rozmiar deskryptora
bDescriptorType	1	36	Deskryptor interfejsu specyficzny dla klasy
bDescriptorSubtype	1	2	Abstrakcyjny model komunikacji
bmCapabilities	1	2	Pole bitowe, żądania obsługiwane przez model komunikacji
bFunctionLength	1	5	Rozmiar deskryptora
bDescriptorType	1	36	Deskryptor interfejsu specyficzny dla klasy
bDescriptorSubtype	1	6	Grupa funkcjonalna interfejsów
bControlInterface	1	0	Numer interfejsu sterującego komunikacją
bSubordinateInterface0	1	1	Numer interfejsu podziemnego

Czwarty deskryptor opisuje zależności pomiędzy deskryptorami, które tworzą funkcjonalną grupę (ang. *union functional descriptor*). Parametr bControlInterface zawiera numer interfejsu sterującego komunikacją. Do tego interfejsu należy kierować wszystkie żądania dotyczące całej grupy. Również powiadomienia dotyczące całej grupy są wysyłane za pomocą tego interfejsu. Parametr bSubordinateInterface0 i ewentualnie kolejne zawiera numer interfejsu podziemnego (dla transmisji danych) należącego do grupy funkcjonalnej. Numery interfejsów odpowiadają wartościami z pola bInterfaceNumber odpowiednich standardowych deskryptorów interfejsów.

3.2.2. Żądania i powiadomienia

Urządzenia komunikacyjne CDC muszą obsługiwać standardowe żądania zgodnie z opisem z rozdziału 1. Nie muszą jednak obsługiwać żądania GET_DESCRIPTOR dla deskryptorów specyficznych dla tej klasy. Kontroler może zażądać jedynie deskryptora urządzenia, konfiguracji i tekstopowego. Deskryptory specyficzne wysyła się w odpowiedzi na żądanie deskryptora konfiguracji, przesyłając całą hierarchię deskryptorów związanych z daną konfiguracją. Wielość podklas interfejsów i obsługiwanych protokołów powoduje, że standard USB CDC definiuje mnóstwo żądań specyficznych dla poszczególnych podklas. Nie ma tu miejsca na omówienie

ich wszystkich, gdyż zajęłoby to kilkudziesiąt stron, a przydatność tych informacji byłaby wątpliwa. Dlatego w tabeli 3.11 zestawiam tylko żądania wykorzystywane w projekcie wirtualnego portu szeregowego. Wszystkie żądania są kierowane do interfejsu sterującego komunikacją i wymienionego w polu bControlInterface deskryptora CDC opisującego grupę interfejsów, patrz tabela 3.10. Numer tego interfejsu jest przekazywany w parametrze wIndex.

Tab. 3.11. Wybrane specyficzne żądania CDC

bmRequestType	bRequest	wValue	wIndex	wLength	Dane
00100001	SET_LINE_CODING 32	0	Numer interfejsu	7	Kodowanie liniowe
10100001	GET_LINE_CODING 33	0	Numer interfejsu	7	Kodowanie liniowe
00100001	SET_CONTROL_LINE_STATE 34	Pole bitowe	Numer interfejsu	0	Brak

Żądania SET_LINE_CODING i GET_LINE_CODING służą do przesłania siedmiobajtowej struktury opisującej kodowanie liniowe stosowane w interfejsie RS-232. Format tej struktury opisany jest w tabeli 3.12.

Tab. 3.12. Kodowanie liniowe RS-232

Pole	Rozmiar	Opis
dwDTERate	4	Szybkość transmisji w bitach na sekundę
bCharFormat	1	Liczba bitów stopu: 0 – jeden, 1 – półtora, 2 – dwa
bParityType	1	Bit parzystości: 0 – brak, 1 – nieparzysty (ang. <i>odd</i>), 2 – parzysty (ang. <i>even</i>), 3 – jedynka (ang. <i>mark</i>), 4 – zero (ang. <i>space</i>)
bDataBits	1	Liczba bitów danych: 5, 6, 7, 8 lub 16

W przypadku wirtualnego portu szeregowego, jeżeli chcemy zbudować pełny konwerter USB/RS-232, czyli jeżeli urządzenie USB ma dokładnie emulować DTE (ang. *Data Terminal Equipment*), to kontroler USB steruje liniami RTS i DTR za pomocą żądania SET_CONTROL_LINE_STATE. Nowe ustawienia przekazuje się w parametrze wValue, który jest traktowany jako pole bitowe zgodnie z tabelą 3.13. Natomiast stan interfejsu RS-232 oraz stan linii, którymi steruje DCE (ang. *Data Communication Equipment*), urządzenie USB przesyła do kontrolera USB za pomocą mechanizmu powiadamiania (ang. *notification*). Powiadomienie składa się ze struktury notification, po której występują opcjonalne dane. Struktura notification (typu `usb_notification_packet_t` zdefiniowanego w pliku `usb_def.h`) jest praktycznie identyczna ze strukturą setup używaną w żądaniach. Jedyna różnica polega na zamianie nazwy pola `bRequest` na `bNotification`. Powiadomień jednak nie przesyła się za pomocą punktu końcowego numer 0, a za pomocą specjalnie do tego celu skonfigurowanego wejściowego punktu końcowego. W rozważanym przypadku korzystamy tylko z powiadomienia typu `SERIAL_STATE`, przesyłanego za pomocą wejściowego punktu końcowego dla danych pilnych związanego z interfejsem sterującym komunikacją. Format tego powiadomienia zamieszczony jest w tabeli 3.15. Pole danych, czyli stan interfejsu, jest dwubajtowym polem bitowym opisany w tabeli 3.14. Oprócz stanu linii DCD, DSR i RI raportowane są też

błedy wykryte podczas odbioru ramki oraz wystąpienie sygnału *break*, czyli stanu logicznego zera na linii RxD trwającego dłużej niż czas trwania pełnej ramki. Błąd przepełnienia bufora dotyczy sytuacji, gdy kontroler USB nie odebrał na czas danych otrzymanych z RS-232 i pojawiły się kolejne dane, które już nie mieszczą się w buforze odbiorczym urządzenia USB. Na koniec zauważmy, że w opisany tu modelu komunikacji nigdzie nie pojawia się sygnał CTS. Jeżeli chcemy zaimplementować pełną kontrolę przypływu za pomocą sygnałów RTS i CTS, to sygnał CTS należy obsługiwać lokalnie w urządzeniu, buforując wysypane przez kontroler USB dane, które jeszcze nie zostały wysłane przez RS-232 (linia TxD) i wstrzymywać ich transmisję, jeśli sygnał CTS nie jest aktywny. Aby nie spowodowało to przepełnienia bufora nadawczego w urządzeniu USB, należy wstrzymywać wysyłanie przez kontroler USB kolejnych porcji danych przez ustawienie negatywnego potwierdzenia NAK (w wyjściowym punkcie końcowym związanym z interfejsem dla transmisji danych).

Tab. 3.13. Linie RS-232 sterowane przez DTE

Bit	Opis
0	RTS – sterowanie przepływem
1	DTR – gotowość terminalu
2...15	Zarezerwowane, zawsze zero

Tab. 3.14. Stan RS-232 oraz linie sterowane przez DCE

Bit	Opis
0	DCD – nośna wykryta
1	DSR – gotowość modemu
2	Sygnal <i>break</i>
3	RI – sygnał dzwonka
4	Błąd ramki
5	Błąd parzystości
6	Błąd przepełnienia bufora
7...15	Zarezerwowane, zawsze zero

Tab. 3.15. Powiadomienie CDC

bmRequestType	bNotification	wValue	wIndex	wLength	Dane
10100001	SERIAL_STATE 32	0	Numer interfejsu	2	Stan interfejsu

3.2.3. Implementacja

ex_com_dev.c

Implementacja warstwy aplikacji dla wirtualnego portu szeregowego znajduje się w pliku *ex_com_dev.c*. Spora część tekstu źródłowego zawartego w tym pliku jest bardzo podobna do omówionej poprzednio implementacji urządzenia klasy HID. Dlatego poniżej omawiam tylko najważniejsze różnice między tymi implementacjami. Na początek musimy zdefiniować wszystkie deskryptory. Zaczynamy od

deskryptora urządzenia. Jest on bardzo podobny do tego, który widzieliśmy przy okazji omawiania implementacji myszy. Inne wartości są tylko w polach `bDeviceClass`, `bMaxPacketSize0` i `idProduct`.

```
static usb_device_descriptor_t const device_descriptor = {
    sizeof(usb_device_descriptor_t), /* bLength */
    DEVICE_DESCRIPTOR,           /* bDescriptorType */
    HTOUSBS(0x0200),            /* bcdUSB */
    COMMUNICATION_DEVICE_CLASS, /* bDeviceClass */
    0,                          /* bDeviceSubClass */
    0,                          /* bDeviceProtocol */
    64,                         /* bMaxPacketSize0 */
    HTOUSBS(VID),               /* idVendor */
    HTOUSBS(PID + 2),           /* idProduct */
    HTOUSBS(0x0100),             /* bcdDevice */
    1,                          /* iManufacturer */
    2,                          /* iProduct */
    3,                          /* iSerialNumber */
    1                           /* bNumConfigurations */
};
```

Ostatnie pole deskryptora urządzenia informuje, że jest tylko jedna konfiguracja. Odpowiednią hierarchię deskryptorów definiujemy jako strukturę typu `usb_com_configuration_t`.

```
typedef struct {
    usb_configuration_descriptor_t      cnf_descr;
    usb_interface_descriptor_t          if0_descr;
    usb_cdc_header_descriptor_t        cdc_h_descr;
    usb_cdc_call_management_descriptor_t cdc_cm_descr;
    usb_cdc_acm_descriptor_t          cdc_acm_descr;
    usb_cdc_union_descriptor_t         cdc_u_descr;
    usb_endpoint_descriptor_t          ep2in_descr;
    usb_interface_descriptor_t          if1_descr;
    usb_endpoint_descriptor_t          eplout_descr;
    usb_endpoint_descriptor_t          eplin_descr;
} __packed usb_com_configuration_t;
```

Konfiguracja rozpoczyna się deskryptorem konfiguracji `cnf_descr`. Potem umieszczamy deskryptor interfejsu sterującego komunikacją `if0_descr`. Interfejs ten ma numer 0. Bezpośrednio za nim znajdują się deskryptory interfejsów specyficznych dla klasy CDC i opisane w tabeli 3.10. Są to kolejno: nagłówek – `cdc_h_descr`, deklaracja interfejsu wykorzystywanego do wybierania numeru telefonicznego – `cdc_cm_descr`, opis własności abstrakcyjnego modelu sterowania – `cdc_acm_descr`, definicja grupy interfejsów – `cdc_u_descr`. Po deskryptorach CDC znajdują się deskryptory interfejsów EP2IN, EP1OUT i EP1IN.

je się deskryptor wejściowego punktu końcowego, który jest wykorzystywany do przesyłania powiadomień do kontrolera. Jest to punkt końcowy dla danych pilnych. Po nim umieszczamy deskryptor interfejsu dla danych if1_descr, który ma numer 1. Na koniec mamy deskryptory eplout_descr i eplin_descr opisujące dwukierunkowy punkt końcowy, którym są przesyłane właściwe dane masowe. Fragment tekstu źródłowego inicjujący strukturę konfiguracji przedstawiony jest na poniższym wydruku. Stałe BLK_BUFF_SIZE i INT_BUFF_SIZE określają rozmiary pakietów dla danych masowych i pilnych. Mają one odpowiednio wartości 64 i 16.

```
static usb_com_configuration_t const com_configuration = {
{
    sizeof(usb_configuration_descriptor_t), /* bLength */
    CONFIGURATION_DESCRIPTOR, /* bDescriptorType */
    HTOUSBS(sizeof(usb_com_configuration_t)), /* wTotalLength */
    2, /* bNumInterfaces */
    1, /* bConfigurationValue */
    0, /* iConfiguration */
    USB_BM_ATTRIBUTES, /* bmAttributes */
    USB_B_MAX_POWER /* bMaxPower */
},
{
    sizeof(usb_interface_descriptor_t), /* bLength */
    INTERFACE_DESCRIPTOR, /* bDescriptorType */
    0, /* bInterfaceNumber */
    0, /* bAlternateSetting */
    1, /* bNumEndpoints */
    COMMUNICATION_INTERFACE_CLASS, /* bInterfaceClass */
    ABSTRACT_CONTROL_MODEL_SUBCLASS, /* bInterfaceSubClass */
    0, /* bInterfaceProtocol */
    0 /* iInterface */
},
{
    sizeof(usb_cdc_header_descriptor_t), /* bFunctionLength */
    CS_INTERFACE_DESCRIPTOR, /* bDescriptorType */
    CDC_HEADER_DESCRIPTOR, /* bDescriptorSubtype */
    HTOUSBS(0x120) /* bcdCDC */
},
{
    sizeof(usb_cdc_call_management_descriptor_t), /* bFunctionLength */
    CS_INTERFACE_DESCRIPTOR, /* bDescriptorType */
    CDC_CALL_MANAGEMENT_DESCRIPTOR, /* bDescriptorSubtype */
    3, /* bmCapabilities */
    1 /* bDataInterface */
},
```

```

{
    sizeof(usb_cdc_acm_descriptor_t),           /* bFunctionLength */
    CS_INTERFACE_DESCRIPTOR,                   /* bDescriptorType */
    CDC_ACM_DESCRIPTOR,                      /* bDescriptorSubtype */
    2                                         /* bmCapabilities */
},
{
    sizeof(usb_cdc_union_descriptor_t),          /* bFunctionLength */
    CS_INTERFACE_DESCRIPTOR,                   /* bDescriptorType */
    CDC_UNION_DESCRIPTOR,                    /* bDescriptorSubtype */
    0,                                         /* bControlInterface */
    1                                         /* bSubordinateInterface0 */
},
{
    sizeof(usb_endpoint_descriptor_t),           /* bLength */
    ENDPOINT_DESCRIPTOR,                     /* bDescriptorType */
    ENDP2 | ENDP_IN,                         /* bEndpointAddress */
    INTERRUPT_TRANSFER,                     /* bmAttributes */
    HTOUSBS(INT_BUFF_SIZE),                /* wMaxPacketSize */
    3                                         /* bInterval */
},
{
    sizeof(usb_interface_descriptor_t),          /* bLength */
    INTERFACE_DESCRIPTOR,                   /* bDescriptorType */
    1,                                         /* bInterfaceNumber */
    0,                                         /* bAlternateSetting */
    2,                                         /* bNumEndpoints */
    DATA_INTERFACE_CLASS,                  /* bInterfaceClass */
    0,                                         /* bInterfaceSubClass */
    0,                                         /* bInterfaceProtocol */
    0                                         /* iInterface */
},
{
    sizeof(usb_endpoint_descriptor_t),           /* bLength */
    ENDPOINT_DESCRIPTOR,                   /* bDescriptorType */
    ENDP1 | ENDP_OUT,                       /* bEndpointAddress */
    BULK_TRANSFER,                          /* bmAttributes */
    HTOUSBS(BLK_BUFF_SIZE),                /* wMaxPacketSize */
    0                                         /* bInterval */
},
{
    sizeof(usb_endpoint_descriptor_t),           /* bLength */
    ENDPOINT_DESCRIPTOR,                   /* bDescriptorType */
}

```

```

        ENDP1 | ENDP_IN,
        BULK_TRANSFER,
        HTOUSBS(BLK_BUFF_SIZE),
        0
    }
};


```

Implementacja deskryptorów tekstowych jest taka sama dla wszystkich urządzeń i opisałem ją przy okazji omawiania urządzenia z interfejsem klasy HID. Choć oczywiście dla każdego urządzenia nieco inna jest sama zawartość tych deskryptorów. Funkcja GetDescriptor jest nieco prostsza niż w przypadku klasy HID, gdyż nie musi obsługiwać żadnych specyficznych deskryptorów, dlatego nie zamieszczam jej implementacji.

```

static uint16_t ep2queue;
static uint8_t configuration, refresh, rs232state;
static usb_cdc_line_coding_t rs232coding;


```

Stan aplikacji przechowujemy w kilku zmiennych globalnych. Zmienna `ep2queue` zawiera liczbę powiadomień czekających na wysłanie. Zmienna `configuration` przechowuje numer aktywnej konfiguracji urządzenia USB. Zmienna `rs232coding` przechowuje parametry kodowania liniowego RS-232 (szybkość transmisji, parzystość, liczba bitów stopu, liczba bitów danych). Typ `usb_cdc_line_coding_t` definiuje strukturę przedstawioną w tabeli 3.12. Zmienna `rs232state` jest polem bitowym i zawiera stan linii interfejsu RS-232 (DTR, DCD, DSR, RTS, CTS). Znaczenie poszczególnych bitów jest zdefiniowane za pomocą szeregu instrukcji `#define` zamieszczonych poniżej.

```

#define DTR 0x01
#define DCD 0x02
#define DSR 0x04
#define RTS 0x08
#define CTS 0x10


```

Jeśli płytka prototypowa wyposażona jest w wyświetlacz ciekłokrystaliczny, to w celach diagnostycznych będziemy na nim wyświetlać wartości zmiennych `rs232coding` i `rs232state`. Zmienna `refresh` informuje, czy należy odświeżyć zawartość ekranu wyświetlacza. Wyświetlaniem zajmuje się funkcja zwrotna `LCDrefresh` wywoływana cyklicznie w funkcji `main`.

```

static void LCDrefresh(void) {
    if (refresh) {
        refresh = 0;
        ...
    }
}


```

Pomocnicza funkcja `ResetState` nadaje wartości początkowe wszystkim zmiennym globalnym.

```
static void ResetState(void) {
    ep2queue = 0;
    configuration = 0;
    rs232coding.dwDTERate = 38400;
    rs232coding.bCharFormat = ONE_STOP_BIT;
    rs232coding.bParityType = NO_PARITY;
    rs232coding.bDataBits = 8;
    rs232state = 0;
    refresh = 1;
}
```

Funkcja `Configure` jest wywoływana jednokrotnie podczas inicjowania sprzętu. Inicjuje ona zmienne globalne. Następnie rejestruje funkcję zwrotną `LCDrefresh` za pomocą funkcji `LCDsetRefresh`. Na koniec konfiguruje dodatkową diodę świecącą, o czym piszę szczegółowo w następnym podrozdziale.

```
int Configure() {
    ResetState();
    LCDsetRefresh(LCDrefresh);
    PowerLEDconfigure();
    return 0;
}
```

Funkcja zwrotna `Reset` jest wywoływana za każdym razem, gdy kontroler wykona procedurę zerowania szyny USB. Jej argumentem jest szybkość, z jaką ma pracować urządzenie. Chwilowo dopusczamy jedynie tryb FS. Realizacja urządzeń HS zostanie przedstawiona w jednym z kolejnych rozdziałów. Następnie przywracamy wartości początkowe wszystkich zmiennych globalnych. Za pomocą funkcji `USBDDepPointConfigure` konfigurujemy punkt końcowy zero (parametr `ENDP0`), będący domyślnym punktem końcowym dla danych sterujących (parametr `CONTROL_TRANSFER`). Ten punkt końcowy musi zostać uruchomiony po wyzerowaniu szyny USB, aby umożliwić komunikację z urządzeniem. Jeśli coś poszło źle, to wywołujemy funkcję `ErrorResetable`, która zasygnalizuje problem, odczeka pewien czas, a następnie wyzeruje mikrokontroler, podejmując w ten sposób ponowną próbę uruchomienia urządzenia. Funkcja `Reset` zwraca maksymalny rozmiar pola danych pakietów sterujących.

```
uint8_t Reset(usb_speed_t speed) {
    ErrorResetable(speed == FULL_SPEED ? 0 : -1, 6);
    ResetState();
    if (USBDDepPointConfigure(ENDP0, CONTROL_TRANSFER,
        device_descriptor.bMaxPacketSize0,
        device_descriptor.bMaxPacketSize0) != REQUEST_SUCCESS)
        ErrorResetable(-1, 7);
    return device_descriptor.bMaxPacketSize0;
}
```

Implementację funkcji zwrotnych GetConfiguration i SetConfiguration przedstawiłem już przy omawianiu implementacji myszy. Pierwsza z nich zwraca wartość zmiennej configuration. Druga inicjuje pozostałe potrzebne punkty końcowe. W przypadku wirtualnego interfejsu szeregowego najpierw konfigurujemy dwukierunkowy punkt końcowy dla danych masowych – pierwsze wywołanie funkcji USBxDendPointConfigure z parametrami ENDP1 i BULK_TRANSFER. Następnie konfigurujemy wejściowy punkt końcowy dla danych pilnych – drugie wywołanie funkcji USBxDendPointConfigure z parametrami ENDP2 i INTERRUPT_TRANSFER. Poprawne zakończenie żądania sygnalizujemy, zwracając wartość REQUEST_SUCCESS, a błąd sygnalizujemy za pomocą wartości REQUEST_ERROR.

```
usb_result_t SetConfiguration(uint16_t confValue) {
    if (confValue > device_descriptor.bNumConfigurations)
        return REQUEST_ERROR;
    configuration = confValue;
    USBDisableAllNonControlEndPoints();
    if (confValue == com_configuration.cnf_descr.bConfigurationValue) {
        usb_result_t r1, r2;
        r1 = USBxDendPointConfigure(ENDP1, BULK_TRANSFER,
                                     BLK_BUFF_SIZE, BLK_BUFF_SIZE);
        r2 = USBxDendPointConfigure(ENDP2, INTERRUPT_TRANSFER,
                                     0, INT_BUFF_SIZE);
        if (r1 == REQUEST_SUCCESS && r2 == REQUEST_SUCCESS)
            return REQUEST_SUCCESS;
        else
            return REQUEST_ERROR;
    }
    return REQUEST_SUCCESS; /* confValue == 0 */
}
```

Do kompletu funkcji zwrotnych obsługujących żądania standardowe potrzebujemy jeszcze funkcji GetStatus, której implementacja jest identyczna jak w przypadku myszy. Zanim przejdziemy do omawiania obsługi żądań specyficznych dla urządzeń CDC, zobaczymy definicję struktury state, która przeznaczona jest do wysyłania powiadomień o stanie interfejsu RS-232. Struktura state składa się ze struktury notification oraz dwubajtowego pola danych wData. Odpowiednie definicje są w pliku *usb_def.h*. Pola struktury notification inicjujemy zgodnie z tabelą 3.15, a pole danych zerujemy.

```
static usb_cdc_serial_state_t state = {
    {DEVICE_TO_HOST | CLASS_REQUEST | INTERFACE_RECIPIENT,
     SERIAL_STATE, 0, 0, 2}, 0
};
```

Prezentowany przykład pokazuje, jak emulować interfejs RS-232 za pomocą USB, ale nie implementuje pełnego RS-232. Dlatego upraszczamy sobie życie i w odpo-

wiedzi na ustawienie linii DTR ustawiamy linie DCD i DSR, a ustawienie linii RTS powoduje ustawienie linii CTS. Z punktu widzenia oprogramowania uruchamianego na komputerze implementowany interfejs zachowuje się w taki sposób, jakby w gnieździe urządzenia zwarte były linie DTR, DCD i DSR oraz RTS i CTS, co jest często stosowane, gdy podłączane urządzenie nie obsługuje tych linii. Korzystając z tego przykładu, można bez trudu zaimplementować inne zachowanie się tych linii.

```
usb_result_t ClassNoDataSetup(usb_setup_packet_t const *setup) {
    if (setup->bmRequestType == (HOST_TO_DEVICE |
                                  CLASS_REQUEST |
                                  INTERFACE_RECIPIENT) &&
        setup->bRequest == SET_CONTROL_LINE_STATE &&
        setup->wIndex == 0 &&
        setup->wLength == 0) {
        uint8_t new_rs232state;
        new_rs232state = rs232state;
        if (setup->wValue & 1) /* DTR set */
            new_rs232state |= (DTR | DSR | DCD);
        else
            new_rs232state &= ~(DTR | DSR | DCD);
        if (setup->wValue & 2) /* RTS set */
            new_rs232state |= RTS | CTS;
        else
            new_rs232state &= ~(RTS | CTS);
        if ((rs232state ^ new_rs232state) & (DCD | DSR)) {
            state.wData = 0;
            if (new_rs232state & DCD)
                state.wData |= 1;
            if (new_rs232state & DSR)
                state.wData |= 2;
            if (ep2queue == 0)
                USBDwrite(ENDP2, (uint8_t const *)&state, sizeof(state));
            if (ep2queue < 2)
                ++ep2queue;
        }
        if (rs232state != new_rs232state) {
            rs232state = new_rs232state;
            refresh = 1;
        }
        return REQUEST_SUCCESS;
    }
    return REQUEST_ERROR;
}
```

Funkcja zwrotna ClassNoDataSetup obsługuje żądania niewymagające przesyłania dodatkowych danych. W tym przypadku jest tylko jedno takie żądanie, mianowicie SET_CONTROL_LINE_STATE. Parametr setup jest wskaźnikiem do struktury opisującej to żądanie i przesyłanej w fazie jego ustanowienia w transakcji SETUP. Zależnie od wartości bitów w polu wValue modyfikowane są ustawienia linii interfejsu RS-232. Jeśli zmienił się stan którejkolwiek linii, to ustawiana jest zmienna refresh w celu odświeżenia zawartości ekranu wyświetlacza. Jeśli zmienił się stan linii DCD lub DSR, to generowane jest nowe powiadomienie. Jeśli żadne powiadomienie nie czeka na wysłanie, to nowe jest wstawiane natychmiast do wysłania za pomocą funkcji USBDwrite. O wysłaniu powiadomienia zostaniemy poinformowani przez wywołanie funkcji zwrotnej EP2IN. W funkcji tej zmniejszana jest liczba powiadomień czekających na wysłanie. Jeśli nadal jest ona dodatnia, to oczekujące powiadomienie jest wstawiane do wysłania. Kolejka powiadomień ma długość dwa. Pierwsze czeka wstawione do wysłania, a drugie czeka w zmiennej state. W zupełności wystarcza to do poprawnej pracy, gdyż w przypadku dwóch czekających powiadomień modyfikacja zmiennej state zmienia powiadomienie czekające na wysłanie, zamiast generować nowe. Jeśli kolejka powiadomień jest pełna, to oznacza, że kontroler nie czyta powiadomień wystarczająco często i nie ma sensu informowanie go o nieaktualnym stanie interfejsu.

```
void EP2IN() {
    if (ep2queue > 0)
        --ep2queue;
    if (ep2queue > 0)
        USBDwrite(ENDP2, (uint8_t const *)&state, sizeof(state));
    refresh = 1;
}
```

Ponieważ nie implementujemy rzeczywistego interfejsu RS-232, ignorujemy parametry kodowania liniowego – są one tylko wyświetlone na LCD. Parametry te przesyłane są za pomocą żądań GET_LINE_CODING i SET_LINE_CODING. Pierwsze z nich wymaga przesyłania dodatkowych danych z urządzenia do kontrolera i jest obsługiwane przez funkcję zwrotną ClassInDataSetup. Parametr setup jest wskaźnikiem do struktury przesłanej w fazie ustanawiania żądania w transakcji SETUP i opisującej to żądanie. Za pomocą parametru data przekazuje się wskaźnik na dane do wysłania – musi to być wskaźnik na dane statyczne, które nie znikną po zakończeniu wywołania funkcji zwrotnej. Parametr length służy do przekazania rozmiaru danych do wysłania.

```
usb_result_t ClassInDataSetup(usb_setup_packet_t const *setup,
                               uint8_t const **data,
                               uint16_t *length) {
    if (setup->bmRequestType == (DEVICE_TO_HOST |
                                  CLASS_REQUEST |
                                  INTERFACE_RECIPIENT) &&
        setup->bRequest == GET_LINE_CODING &&
```

```

        setup->wValue == 0 &&
        setup->wIndex == 0) {
    *data = (const uint8_t *)&rs232coding;
    *length = sizeof(rs232coding);
    return REQUEST_SUCCESS;
}
return REQUEST_ERROR;
}

```

Żądanie SET_LINE_CODING wymaga przesłania dodatkowych danych z kontrolera do urządzenia. Jest ono obsługiwane za pomocą funkcji zwrotnej ClassOutDataSetup. Parametr setup jest wskaźnikiem do struktury przesłanej w fazie ustanawiania żądania w transakcji SETUP i opisuje to żądanie. Parametr data służy do przekazania wskaźnika do statycznego bufora, do którego mają być skopiowane dane wysłane przez kontroler. Jednak z bufora tego będzie można czytać dopiero po zakończeniu obsługi żądania.

```

usb_result_t ClassOutDataSetup(usb_setup_packet_t const *setup,
                               uint8_t **data) {
if (setup->bmRequestType == (HOST_TO_DEVICE |
                                CLASS_REQUEST |
                                INTERFACE_RECIPIENT) &&
    setup->bRequest == SET_LINE_CODING &&
    setup->wValue == 0 &&
    setup->wIndex == 0 &&
    setup->wLength == sizeof(rs232coding)) {
    *data = (uint8_t *)&rs232coding;
    return REQUEST_SUCCESS;
}
return REQUEST_ERROR;
}

```

O zakończeniu obsługi żądania, czyli o zakończeniu jego fazy statusu, zostaniemy poinformowani przez wywołanie funkcji zwrotnej ClassStatusIn. Jej wywołanie oznacza, że dane zostały odebrane i znajdują się w buforze, do którego wskaźnik przekazaliśmy za pomocą funkcji ClassOutDataSetup.

```

void ClassStatusIn(usb_setup_packet_t const *setup) {
if (setup->bmRequestType == (HOST_TO_DEVICE |
                                CLASS_REQUEST |
                                INTERFACE_RECIPIENT) &&
    setup->bRequest == SET_LINE_CODING &&
    setup->wValue == 0 &&
    setup->wIndex == 0 &&
    setup->wLength == sizeof(rs232coding)) {
    refresh = 1;
}
}

```

W prezentowanej przykładowej aplikacji komunikacja z urządzeniem odbywa się za pomocą tekstów ASCII czytelnych dla człowieka. Po otrzymaniu przez urządzenie danych wywoływana jest funkcja zwrotna EP1OUT. Odebrane dane czytamy za pomocą funkcji USBDread. Zależnie od przysłanego znaku włączamy lub wyłączzamy odpowiednią diodę świecącą. Jeśli odebrany znak nie dotyczy żadnej diody, odsyłamy tekst pomocy znajdujący się w tablicy help. Białe znaki (spacja, koniec wiersza, tabulator) ignorujemy.

```
static uint8_t const help[] =
    "Press:\r\n"
    "  G to switch green LED on,\r\n"
    "  g to switch green LED off,\r\n"
    "  R to switch red LED on,\r\n"
    "  r to switch red LED off,\r\n"
    "  W to switch white LED on,\r\n"
    "  w to switch white LED off.\r\n";
void EP1OUT() {
    uint8_t buffer[BLK_BUFF_SIZE];
    uint16_t i, len;
    len = USBDread(ENDP1, buffer, BLK_BUFF_SIZE);
    for (i = 0; i < len; ++i) {
        switch (buffer[i]) {
            case 'G':
                GreenLEDon();
                break;
            case 'g':
                GreenLEDoff();
                break;
            case 'R':
                RedLEDon();
                break;
            case 'r':
                RedLEDoff();
                break;
            case 'W':
                PowerLEDon();
                break;
            case 'w':
                PowerLEDoff();
                break;
            case ' ':
            case '\n':
            case '\r':
            case '\t':
                break;
        }
    }
}
```

```

        default:
            USBDwriteEx(ENDP1, help, sizeof(help) - 1);
            return;
        }
    }
}

```

Jak poprzednio wskaźnik do struktury zawierającej wskaźniki do funkcji zwrotnych przekazujemy za pomocą funkcji `USBDgetApplicationCallbacks`, której implementacja jest taka sama jak w przypadku myszy, więc ją pomijam.

3.2.4. Dioda świecąca mocy

<code>pwr_periph.h</code>
<code>pwr_periph_10x.c</code>
<code>pwr_periph_15x.c</code>
<code>pwr_periph_2xx.c</code>

Prezentowane urządzenie, oprócz sterowania diodami świecącymi zieloną i czerwoną, o których zakładamy, że są standardowo zainstalowane na płytce prototypowej, steruje też białą diodą świecącą mocy, której włączenie symuluje pobór dużego prądu z szyny zasilającej interfejsu USB. Będzie to potrzebne w rozdziale 5 do zademonstrowania różnych aspektów zasilania urządzeń USB. W tym przykładzie korzystamy z niej jak ze zwykłej diody świecącej. Na płytach prototypowych, które takiej diody mocy nie mają, zastępujemy ją zwykłą diodą świecącą malej mocy. Sterowanie diodą świecącą mocy zaimplementowane jest w module `pwr_periph`. Interfejs tego modułu znajduje się w pliku `pwr_periph.h`. Implementacja dla mikrokontrolerów STM32F10x jest w pliku `pwr_periph_10x.c`, dla STM32Lxxx – w pliku `pwr_periph_15x.c`, a dla STM32F2xx i STM32F4xx – w pliku `pwr_periph_2xx.c`.

<code>void PowerLEDconfigure(void);</code>
--

Bezparametrowa funkcja `PowerLEDconfigure` konfiguruje port wejścia-wyjścia, który steruje białą diodą świecącą mocy. Funkcja ta nie zwraca żadnej wartości. W pliku `board_def.h` muszą być zdefiniowane stałe:

- `POWER_LED_GPIO_N` – literowe oznaczenie portu wejścia-wyjścia sterującego diodą;
- `POWER_LED_PIN_N` – numer wyprowadzenia portu wejścia-wyjścia sterującego diodą;
- `POWER_LED_ON` – poziom na wyjściu, który włącza diodę.

Przykładowo, jeśli diodę mocy włącza się przez podanie poziomu wysokiego na wyjście PA1, to stałe te powinny być zdefiniowane następująco:

<code>#define POWER_LED_GPIO_N A</code>
<code>#define POWER_LED_PIN_N 1</code>
<code>#define POWER_LED_ON 1</code>

```
void PowerLEDon(void);
```

Bezparametrowa funkcja `PowerLEDon` włącza białą diodę świecącą mocy i nie zwraca żadnej wartości.

```
void PowerLEDoff(void);
```

Bezparametrowa funkcja `PowerLEDoff` wyłącza białą diodę świecącą mocy i nie zwraca żadnej wartości.

```
int PowerLEDstate(void);
```

Bezparametrowa funkcja `PowerLEDstate` zwraca 1, gdy biała dioda mocy świeci, a 0 w przeciwnym przypadku, czyli gdy dioda nie świeci.

```
void WakeupButtonConfigure(void);
```

Moduł `pwr_periph` udostępnia jeszcze bezparametrową funkcję `WakeupButtonConfigure`, która konfiguruje przycisk budzący mikrokontroler ze stanu niskiego poboru energii. W tym przykładzie nie korzystamy z tej funkcji, ale będzie nam ona potrzebna w rozdziale 5. Funkcja ta nie zwraca żadnej wartości. Aby skonfigurować przycisk, trzeba w pliku `board_def.h` zdefiniować stałe:

- `PUSH_BUTTON_GPIO_N` – literowe oznaczenie portu wejścia-wyjścia, do którego podłączony jest przycisk;
- `PUSH_BUTTON_PIN_N` – numer wyprowadzenia portu wejścia-wyjścia, do którego podłączony jest przycisk;
- `PUSH_BUTTON_IRQ_N` – oznaczenie przerwania związanego z przyciskiem;
- `PUSH_BUTTON_EDGE` – zbocze, które ma wyzwalać przerwanie związane z przyciskiem.

Przykładowo, jeśli przycisk budzenia podłączony jest do wejścia PC15 i po jego wcisnięciu na tym wejściu jest stan niski, czyli przerwanie ma być wyzwalane zbozem opadającym, to stałe te należy zdefiniować następująco:

```
#define PUSH_BUTTON_GPIO_N C
#define PUSH_BUTTON_PIN_N 15
#define PUSH_BUTTON_IRQ_N EXTI15_10
#define PUSH_BUTTON_EDGE EXTI_Trigger_Falling
```

3.2.5. Kompilowanie i testowanie

Archiwum z przykładami zawiera kilka wersji projektu wirtualnego portu szeregowego. W katalogu `.make` znajdują się podkatalogi, których nazwy rozpoczynają się przedrostkiem `usb2_com_device`. W tych podkatalogach umieszczone są pliki `makefile` umożliwiające skompilowanie tego projektu dla przykładowych wariantów sprzętu. Można je oczywiście łatwo dostosować do innego sprzętu, posługując się wskazówkami z rozdziału 2. Pliki `makefile` są również przydatne dla tych, którzy

nie chcą korzystać bezpośrednio z programu `make`, gdyż zawierają listę plików źródłowych niezbędnych do skompilowania programu. Należy przy tym pamiętać, aby dołączyć plik `startup_stm32.c` i właściwą wersję biblioteki STM32.

System Linux rozpoznaje nasz wirtualny port szeregowy jako urządzenie `/dev/ttyACM0`. Oczywiście jego numer może być inny, jeśli do komputera jest podłączone więcej niż jedno urządzenie tego typu. Do komunikacji możemy użyć programu Minicom. Jego domyślna konfiguracja zapisana jest w pliku `.minirc.dfl`, który znajduje się w katalogu domowym użytkownika. Przykładowa zawartość tego pliku wygląda następująco:

```
pu port          /dev/ttyACM0
pu baudrate     115200
pu bits          8
pu parity         N
pu stopbits      1
pu minit
pu mreset
```

Nie zaleca się bezpośredniego edytowania tego pliku. Wszelkie zmiany konfiguracji należy przeprowadzać poprzez menu w Minicomie. Program Minicom działa w trybie tekstowym – uruchamia się go z konsoli systemu poleceniem `minicom`. Wpisywany tekst jest wysyłany do naszego urządzenia. Tekst odsyłany przez urządzenie jest wypisywany na ekranie. Komunikację kończy się, wciskając kombinację klawiszy `Ctrl-A`, a następnie bezpośrednio klawisz `X`. Pojawia się okienko z prośbą o potwierdzenie, czy na pewno chcemy zakończyć połączenie. Wciśnięcie klawisza `Enter` definitelynie kończy sesję komunikacyjną.

Jeśli używamy systemu Windows, to po podłączeniu naszego urządzenia do gniazda USB najprawdopodobniej zostaniemy poinformowani o niepowodzeniu instalowania sterownika. Musimy sami wskazać odpowiedni sterownik wirtualnego portu szeregowego. W tym celu należy odszukać urządzenie w panelu sterowania w menu *Sprzęt i dźwięk*, *Urządzenia i drukarki*, *Menedżer urządzeń* lub podobnym. Powinniśmy w kategorii *Inne urządzenia* zobaczyć *nieokreślone* urządzenie o nazwie *USB virtual serial port example*. Klikamy na nie dwukrotnie, aby otworzyć jego okno dialogowe. W zakładce *Ogólne* lub *Sterownik* wybieramy *Aktualizuj sterownik*. Alternatywnie można w zakładce *Sprzęt* wybrać *Właściwości*, a następnie *Zmień ustawienia i aktualizuj sterownik*. W obu przypadkach powinno pojawić się okno instalatora. Wskazujemy opcję *Przeglądaj mój komputer w poszukiwaniu oprogramowania sterownika* lub *Odszukaj i zainstaluj oprogramowanie sterownika ręcznie*. W następnym oknie wpisujemy lub wyszukujemy nazwę folderu, w którym znajduje się plik `stmcdc.inf`. Plik ten jest umieszczony w archiwum z przykładami w katalogu `./make/win`. Klikamy przycisk *Dalej*. Wtedy zapewne zostaniemy poinformowani, że „system Windows nie może zweryfikować wydawcy tego oprogramowania”. Ignorujemy to ostrzeżenie i wydajemy polecenie „zainstaluj oprogramowanie mimo to”. Po chwili powinniśmy zobaczyć komunikat, że „system Windows pomyślnie zaktualizował oprogramowanie sterownika”. Klikamy przycisk *Zamknij*, co kończy instalację. Po poprawnym zainstalowaniu sterownika powinniśmy zo-

baczyć, że system umieścił urządzenie w kategorii „porty (COM i LPT)” i zmienił mu nazwę na „STM Virtual Com Port (COM8)”. Oczywiście numer portu COM będzie zapewne inny. Nazwę urządzenia i numer przydzielonego mu portu można też zobaczyć, klikając we właściwości urządzenia i zaglądając do zakładki *Sprzęt*. Wyżej opisany proces instalowania może przebiegać nieco odmiennie, zależnie od wersji systemu Windows, w szczególności mogą pojawiać się komunikaty o nieco innej treści, a poszczególne opcje mogą być rozmieszczone w innych menu lub zakładkach.

W systemie Windows do komunikacji z naszym urządzeniem możemy użyć programu PuTTY. Po jego uruchomieniu musimy go najpierw skonfigurować. Przykładowa, skrócona konfiguracja polega na wybraniu następujących opcji:

- *Serial line: COM8,*
- *Speed: 115200,*
- *Connection type: serial.*

W szczególności bardzo ważne jest wybranie właściwego numeru portu COM. Pełna konfiguracja dostępna jest w okienku *Category* po kliknięciu kontrolki *Serial*. Możemy wtedy skonfigurować następujące parametry:

- *Serial line to connect to: COM8,*
- *Speed (baud): 115200,*
- *Data bits: 8,*
- *Stop bits: 1,*
- *Parity: none,*
- *Flow control: none.*

Po skonfigurowaniu klikamy przycisk *Open*, aby rozpocząć sesję. Jeśli nie można otworzyć połączenia, należy sprawdzić, czy w głównym oknie programu wybrano typ połączenia *serial*. Otwiera się okno, w którym możemy wpisywać wysyłany do urządzenia tekst. W oknie tym wyświetlany jest też tekst odesłany przez urządzenie. Aby zakończyć sesję, należy po prostu zamknąć to okno.

W systemach Windows, zwłaszcza tych starszych, do komunikacji z naszym urządzeniem możemy też użyć programu HyperTerminal. Po jego uruchomieniu jesteśmy najpierw proszeni o wpisanie nazwy sesji. Wpisujemy cokolwiek i klikamy *OK*. Następnie musimy wybrać port, przykładowo COM8. W kolejnym okienku wpisujemy konfigurację portu:

- *Liczba bitów na sekundę: 115200,*
- *Bity danych: 8,*
- *Parzystość: brak,*
- *Bity stopu: 1,*
- *Sterowanie przepływem: brak.*

Po zaakceptowaniu konfiguracji za pomocą przycisku *OK* otwiera się okno, w którym wpisujemy tekst do wysłania i w którym wyświetla się tekst odsyłany przez urządzenie.

Jeśli płytka prototypowa wyposażona jest w wyświetlacz ciekłokrystaliczny, to wyświetlane są na nim informacje diagnostyczne. Po otwarciu sesji komunikacyjnej z urządzeniem powinniśmy na ekranie wyświetlacza zobaczyć coś takiego:

```
115200 8N1      0  
RTS CTS DTR DSR DCD
```

W pierwszej linii wyświetlane są parametry transmisji, czyli po kolei szybkość transmisji, liczba bitów danych, parzystość i liczba bitów stopu. Ostatnia wartość po prawej stronie to liczba powiadomień wstawionych do wysłania i czekających na ich odebranie przez kontroler USB. W drugiej linii wyświetlany jest stan linii interfejsu RS-232. Nazwa danej linii jest wyświetlana tylko wtedy, gdy linia ta jest aktywna.

3.3. Projekt odtwarzacza audio

Trzeci projekt demonstruje przykład urządzenia ADC (ang. *Audio Device Class*) wykorzystującego dane izochroniczne (ang. *isochronous*). W klasie tej mieszą się następujące urządzenia:

- głośnik monofoniczny, stereofoniczny zestaw głośnikowy, zestaw głośników dla kina domowego,
- słuchawki monofoniczne lub stereofoniczne,
- mikrofon,
- zestaw słuchawkowy składający się z co najmniej jednego głośniczka i co najmniej jednego mikrofonu,
- słuchawka telefoniczna,
- rejestrator lub odtwarzacz dźwięku,
- instrument muzyczny,
- mikser.

Urządzenie zademonstrowane w tym podrozdziale będzie rozpoznawane przez komputer osobisty jako zewnętrzny głośnik z możliwością regulacji poziomu głośności dźwięku i wyciszczeniem – po podłączeniu urządzenia w aplikacji miksera pojawiają się odpowiednie kontrolki. W prezentowanym projekcie wykorzystano wersję 1.0 specyfikacji USB ADC. Istnieje też wersja 2.0 tej specyfikacji, ale nie jest ona kompatybilna z wersją 1.0 i nie wszystkie systemy operacyjne obsługują ją poprawnie. Dalszych informacji na temat innych urządzeń ADC należy szukać w [26], [27], [28], [29] i [30].

3.3.1. Deskryptory

Urządzenie audio często jest urządzeniem złożonym. Oprócz interfejsów przeznaczonych wyłącznie do transmisji dźwięku może też udostępniać interfejs klasy HID (panel sterujący) lub interfejs pamięci dyskowej (odtwarzacz płyt). Dlatego w deskryptorze urządzenia pola określające klasę, podklasę i protokół, czyli pola bDeviceClass, bDeviceSubClass, bDeviceProtocol, mają wartości zerowe. Pozostałe pola deskryptora urządzenia wypełnia się zgodnie z opisem zamieszczonym w roz-

dziale 1. Natomiast właściwą klasą i podkласę ustawia się na poziomie deskryptora interfejsu. Pojedyncze urządzenie może realizować wiele funkcji. Z każdą funkcją musi być skojarzony jeden interfejs sterujący (ang. *audio control interface*) oraz opcjonalne interfejsy do transmisji danych dźwiękowych (ang. *audio streaming interface*) lub danych w formacie MIDI (ang. *MIDI streaming interface*). Producenci mogą też rozszerzać standard i definiować własne interfejsy (ang. *vendor specific interface*). Dla klasy audio nie zdefiniowano żadnych protokołów. Wartości umieszczane w deskryptorze interfejsu zebrane są w **tabeli 3.16**. W klasie audio zdefiniowano wiele specyficznych deskryptorów, które wraz z deskryptorami interfejsów zebrane są jak zwykle w konfigurację rozpoczęającą się od deskryptora konfiguracji. Zanim jednak przejdziemy do omawiania przykładowej konfiguracji, warto w kilku zdaniach przedstawić ogólną ideę tworzenia konfiguracji urządzenia audio.

Konfiguracja odzwierciedla budowę urządzenia audio, czyli występujące w nim bloki funkcjonalne (ang. *functional entities*) oraz połączenia między nimi. Rozróżnia się dwa typy bloków: terminal i jednostkę (ang. *unit*). Zdefiniowano dwa rodzaje terminali:

- wejściowy (ang. *input terminal*),
- wyjściowy (ang. *output terminal*),

Zdefiniowano następujące rodzaje jednostek:

- mikser (ang. *mixer unit*),
- przełącznik (ang. *selector unit*),
- regulator (ang. *feature unit*),
- procesor dźwięku (ang. *processing unit*),
- rozszerzenie (ang. *extension unit*).

Tab. 3.16. Parametry deskryptora interfejsu

Pole	Rozmiar	Wartość	Opis
bInterfaceClass	1	1	Wartość oznaczająca klasę audio
bInterfaceSubClass	1	0	Interfejs o niekreślonej podklasie
		1	Interfejs sterujący
		2	Interfejs do transmisji danych
		3	Interfejs MIDI
		255	Interfejs specyficzny dla producenta
bInterfaceProtocol	1	0	Protokół niezdefiniowany

Każdy blok funkcjonalny ma pewną liczbę wejść i wyjść. Wyjście bloku musi być podłączone z wejściem innego bloku. Połączenie między blokami przesyła strumień danych dźwiękowych. W pojedynczym strumieniu można przesyłać jeden (mono) lub kilka kanałów dźwiękowych (stereo i więcej). Kanały strumienia mogą być analogowe lub cyfrowe. Dla kanałów cyfrowych musi być podany format kodowania (PCM, MPEG itp.) oraz jego parametry (częstotliwość próbkowania, liczba bitów na próbce, sposób kodowania próbki itd.). Z oczywistych powodów do wejścia danego bloku można podłączyć tylko wyjście innego bloku mające identyczną liczbę kanałów i stosujące ten sam standard kodowania.

Terminal wejściowy reprezentuje źródło dźwięku z punktu widzenia urządzenia USB. Może to być wyjściowy punkt końcowy dla danych izochronicznych, czyli punkt końcowy, przez który kontroler USB wysyła dane do urządzenia. Może to być także gniazdo wejściowe (ang. *line-in connector*), mikrofon lub tym podobne źródło. Terminal wejściowy ma tylko jedno wyjście. Integralną częścią terminalu wejściowego może być przetwornik analogowo-cyfrowy, gdy źródło dźwięku ma naturę analogową (np. mikrofon), ale jego wyjście jest cyfrowe. Terminal wyjściowy reprezentuje ujście, czyli punkt, w którym sygnał dźwiękowy opuszcza urządzenie USB. Może to być wejściowy punkt końcowy dla danych izochronicznych, czyli punkt końcowy, przez który kontroler USB odbiera dane z urządzenia. Może to być także gniazdo wyjściowe (ang. *line-out connector*), głośnik lub tym podobne. Terminal wyjściowy ma tylko jedno wejście. Integralną częścią terminalu wyjściowego może być przetwornik cyfrowo-analogowy, gdy na wejście podawany jest sygnał cyfrowy, ale sygnał wychodzący z terminalu ma charakter analogowy (np. głośnik). Niektóre typy urządzeń, na przykład zestaw słuchawkowy z mikrofonem, zawierają zarówno źródło dźwięku, jak i jego ujście. Wymagają one terminalu dwukierunkowego. Tego rodzaju terminal reprezentuje się przez skojarzenie dwóch terminali: wejściowego i wyjściowego.

Pozostałe rodzaje bloków funkcjonalnych, czyli jednostki, mają zarówno wejścia, jak i wyjścia. Jednostki mają różnego rodzaju atrybuty. Typowymi atrybutami są głośność, wzmacnienie, poziom tonów niskich lub wysokich. Z każdym atrybutem związane są cztery wartości, które można odczytać i ewentualnie zmieniać. Nie wszystkie wartości wszystkich atrybutów są modyfikowalne – niektóre są tylko do odczytu. Każdy atrybut może mieć:

- wartość aktualną (ang. *current setting*),
- wartość minimalną (ang. *minimum setting*),
- wartość maksymalną (ang. *maximum setting*),
- rozdzielcość (ang. *attribute resolution*), czyli wartość, której wielokrotnością są wartości pozostałych atrybutów.

Mikser jest typową jednostką w klasie audio. Ma on zwykle wiele wejść, ale tylko jedno wyjście. Mikser przesyła do poszczególnych kanałów na wyjściu zsumowane sygnały z kanałów wejściowych. Sumowanie odbywa się z zadanimi współczynnikami wzmacnienia (tłumienia). Współczynniki te określa się za pomocą macierzy, która ma tyle wierszy, ile jest łącznie kanałów wejściowych we wszystkich wejściach i tyle kolumn, ile jest kanałów wyjściowych. Każdy współczynnik określa wzmacnienie między odpowiednim kanałem wejściowym a wyjściowym. Inną bardzo typową jednostką jest przełącznik. Podobnie jak mikser ma on zwykle wiele wejść, ale tylko jedno wyjście. Jak sama nazwa wskazuje, jego zadaniem jest przełączanie sygnałów z wejść na wyjście bez ich modyfikacji.

Pod nazwą regulator kryją się: układ wyciszający (ang. *mute*), regulator głośności (ang. *volume*), regulator tonów niskich (ang. *bass*), średnich (ang. *mid*) lub wysokich (ang. *treble*), graficzny regulator barwy dźwięku (ang. *graphic equalizer*), układ automatycznej regulacji wzmacnienia (ang. *automatic gain*), układ opóźniający (ang. *delay*), układ podbijający basy (ang. *bass boost*), układ fizjologicznej regulacji głośności (ang. *loudness*). Regulator ma dokładnie jedno wejście i dokładnie jedno wyj-

ście. Wejście i wyjście mają jednakową liczbę kanałów i format. Procesor dźwięku też ma jedno wejście i jedno wyjście, ale wyjście może mieć inną liczbę kanałów niż wejście, może też mieć inne kodowanie. Typowymi procesorami dźwięku są różnego rodzaju wielokanałowe dekodery stosowane w kinie domowym, ekspander stereo, kompresor dynamiki, pogłos, chorus i tym podobne efekty.

Jednostka rozszerzająca służy do realizacji funkcji, która nie mieści się w żadnej z wyżej przedstawionych kategorii. Ponieważ zastosowanie takiej jednostki wymaga załadowania specyficznego sterownika urządzenia, to musi ona udostępniać atrybut umożliwiający jej ominięcie lub wyłączenie domyślnego zachowania, jeśli sterownik urządzenia nie będzie w stanie jej poprawnie rozpoznać i obsługiwać.

Standard definiuje bardzo dużo specyficznych deskryptorów dla klasy audio. W szczególności każdy blok funkcjonalny ma własny deskryptor. Przepisywanie całego standardu nie ma sensu, dlatego pominię szczegółowy opis wszystkich deskryptorów. Przedstawię tu tylko ogólną ideę opisu konfiguracji na przykładzie urządzenia przedstawionego schematycznie na **rysunku 3.1**. Rysunek ten jest zgodny z propozowaną w standardzie konwencją, według której terminale rysuje się w okręgach, a jednostki w prostokątach. Rozważane urządzenie ma terminal wejściowy, będący izochronicznym punktem końcowym, przez który kontroler USB wysyła cyfrowy sygnał dźwiękowy. Sygnał ten przesyłany jest do regulatora głośności z funkcją wyciszania. Sygnał z regulatora jest przesyłany do terminalu wyjściowego, którym formalnie jest głośnik, ale kryje się w nim przetwornik cyfrowo-analogowy i ewentualnie też wzmacniacz mocy. Każdy blok funkcjonalny musi mieć unikalny identyfikator, który jest liczbą z przedziału 1...255. Identyfikatory te są umieszczone na rysunku 3.1 pod odpowiednimi symbolami. Hierarchia deskryptorów opisujących konfigurację przykładowego urządzenia zamieszczona jest w **tabeli 3.17**. Jak widać, nawet tak proste urządzenie, składające się tylko z trzech bloków, wymaga dość długiego opisu. Hierarchia deskryptorów rozpoczyna się od deskryptora konfiguracji, który jest wypełniony standardowo, zgodnie z opisem z rozdziału 1.

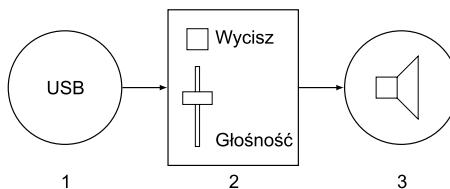
Po deskryptorze konfiguracji pojawia się deskryptor interfejsu sterującego klasy audio. Jego pola wypełnia się zgodnie z opisem z rozdziału 1 z uwzględnieniem wartości wymienionych w tabeli 3.16. Interfejs ten nie ma żadnych punktów końcowych. Komunikacja z nim odbywa się przez domyślny dla całego urządzenia punkt końcowy dla danych sterujących, czyli punkt końcowy zero. Kolejne cztery deskryptory interfejsów specyficznych dla klasy opisują strukturę urządzenia z rysunku 3.1. Format tych deskryptorów jest identyczny z tym z tabeli 3.9 dla klasy CDC. Pierwszy z tych deskryptorów jest nagłówkiem całej struktury. Jego pole bDescriptorSubtype zawiera wartość 1. Pole bcdADC zawiera numer wersji specyfikacji klasy audio, z którą w zamierzeniu ma być zgodne urządzenie opisywane tym deskryptorem. Pole wTotalLength zawiera łączny rozmiar w bajtach czterech omawianych poniżej deskryptorów interfejsów specyficznych dla klasy audio. Pole bInCollection zawiera liczbę interfejsów dla transmisji danych i interfejsów MIDI obsługiwanych przez interfejs sterujący opisywany tym deskryptorem. Po tym polu następuje lista pól baInterfaceNr, w których wymienia się numery tych interfejsów. Po nagłówku umieszcza się deskryptory bloków, z których składa się urządzenie. Ich kolejność jest dowolna, ale najlepsza jest kolejność zgodna z kierunkiem przepływu sygnałów przez urządzenie.

Tab. 3.17. Hierarchia deskryptorów opisujących konfigurację przykładowego urządzenia audio

Pole	Rozmiar	Wartość	Opis
bLength	1	9	Rozmiar deskryptora konfiguracji
bDescriptorType	1	2	Deskryptor konfiguracji
wTotalLength	2	109	Całkowity rozmiar hierarchii deskryptorów
bNumInterfaces	1	2	Liczba interfejsów
bConfigurationValue	1	1	Numer konfiguracji
iConfiguration	1	0	Numer deskryptora tekstowego opisującego konfigurację
bmAttributes	1	0xC0	Atrybuty konfiguracji związane z zasilaniem
bMaxPower	1	1	Maksymalny prąd pobierany przez urządzenie z szyny
bLength	1	9	Rozmiar deskryptora
bDescriptorType	1	4	Deskryptor interfejsu
bInterfaceNumber	1	0	Numer interfejsu
bAlternateSetting	1	0	Wariant ustawień
bNumEndpoints	1	0	Liczba punktów końcowych
bInterfaceClass	1	1	Klasa audio
bInterfaceSubClass	1	1	Interfejs sterujący
bInterfaceProtocol	1	0	Protokół niezdefiniowany
iInterface	1	0	Numer deskryptora tekstowego opisującego interfejs
bLength	1	9	Rozmiar deskryptora
bDescriptorType	1	36	Deskryptor interfejsu specyficzny dla klasy
bDescriptorSubtype	1	1	Nagłówek
bcdADC	2	0x0100	Wersja specyfikacji audio
wTotalLength	2	39	Łączny rozmiar deskryptorów interfejsów specyficznych dla klasy umieszczonej w tej strukturze
bInCollection	1	1	Łączna liczba interfejsów dla transmisji danych i MIDI związanych z tym interfejsem sterującym
baInterfaceNr	1	1	Numer interfejsu dla transmisji danych lub MIDI związanej z tym interfejsem sterującym
bLength	1	12	Rozmiar deskryptora
bDescriptorType	1	36	Deskryptor interfejsu specyficzny dla klasy
bDescriptorSubtype	1	2	Terminal wejściowy
bTerminalID	1	1	Identyfikator terminalu
wTerminalType	2	0x0101	Typ terminalu
bAssocTerminal	1	0	Numer terminalu wyjściowego, z którym powiązany jest ten terminal wejściowy
bNrChannels	1	1	Liczba kanałów
wChannelConfig	2	0x0004	Położenie w przestrzeni
iChannelNames	1	0	Numer deskryptora tekstowego opisującego kanały
iTerminal	1	0	Numer deskryptora tekstowego opisującego terminal
bLength	1	9	Rozmiar deskryptora
bDescriptorType	1	36	Deskryptor interfejsu specyficzny dla klasy
bDescriptorSubtype	1	6	Regulator
bUnitID	1	2	Identyfikator regulatora
bSourceID	1	1	Identyfikator bloku, do którego wyjścia podłączone jest wejście tego regulatora
bControlSize	1	2	Rozmiar następnego pola

Pole	Rozmiar	Wartość	Opis
bmaControls0	2	0x0003	Dostępne elementy regulacyjne
iFeature	1	0	Numer deskryptora tekstowego opisującego regulator
bLength	1	9	Rozmiar deskryptora
bDescriptorType	1	36	Deskryptor interfejsu specyficzny dla klasy
bDescriptorSubtype	1	3	Terminal wyjściowy
bTerminalID	1	3	Identyfikator terminalu
wTerminalType	2	0x0304	Typ terminalu
bAssocTerminal	1	0	Numer terminalu wejściowego, z którym powiązany jest ten terminal wyjściowy
bSourceID	1	2	Identyfikator bloku, do którego wyjścia podłączone jest wejście tego terminalu
iTerminal	1	0	Numer deskryptora tekstowego opisującego terminal
bLength	1	9	Rozmiar deskryptora
bDescriptorType	1	4	Deskryptor interfejsu
bInterfaceNumber	1	1	Numer interfejsu
bAlternateSetting	1	0	Wariant ustawień
bNumEndpoints	1	0	Liczba punktów końcowych
bInterfaceClass	1	1	Klasa audio
bInterfaceSubClass	1	2	Interfejs dla transmisji danych
bInterfaceProtocol	1	0	Protokół niezdefiniowany
iInterface	1	0	Numer deskryptora tekstowego opisującego interfejs
bLength	1	9	Rozmiar deskryptora
bDescriptorType	1	4	Deskryptor interfejsu
bInterfaceNumber	1	1	Numer interfejsu
bAlternateSetting	1	1	Wariant ustawień
bNumEndpoints	1	1	Liczba punktów końcowych
bInterfaceClass	1	1	Klasa audio
bInterfaceSubClass	1	2	Interfejs dla transmisji danych
bInterfaceProtocol	1	0	Protokół niezdefiniowany
iInterface	1	0	Numer deskryptora tekstowego opisującego interfejs
bLength	1	7	Rozmiar deskryptora
bDescriptorType	1	36	Deskryptor interfejsu specyficzny dla klasy
bDescriptorSubtype	1	1	Ogólny
bTerminalLink	1	1	Identyfikator terminalu powiązanego z tym interfejsem
bDelay	1	2	Opóźnienie w ramkach wprowadzane przez ten interfejs
wFormatTag	2	1	Format danych
bLength	1	11	Rozmiar deskryptora
bDescriptorType	1	36	Deskryptor interfejsu specyficzny dla klasy
bDescriptorSubtype	1	2	Format danych
bFormatType	1	1	Typ formatu
bNrChannels	1	1	Liczba kanałów
bSubframeSize	1	2	Liczba bajtów zajmowanych przez jedną próbkę
bBitResolution	1	16	Liczba bitów faktycznie wykorzystywanych w próbce
bSamFreqType	1	1	Sposób określenia częstotliwości próbkowania
tSamFreq	3	48000	Częstotliwość próbkowania w hercach
bLength	1	9	Rozmiar deskryptora

Pole	Rozmiar	Wartość	Opis
bDescriptorType	1	5	Deskryptor punktu końcowego
bEndpointAddress	1	0x01	Adres punktu końcowego
bmAttributes	1	9	Rodzaj przesyłanych danych oraz dodatkowe cechy i atrybuty punktu końcowego
wMaxPacketSize	2	96	Maksymalny rozmiar danych przesyłanych w jednym pakiecie
bInterval	1	1	Okres odpytywania punktu końcowego
bRefresh	1	0	Zawsze wartość zero
bSynchAddress	1	0	Adres punktu końcowego do przesyłania danych synchronizacyjnych
bLength	1	7	Rozmiar deskryptora
bDescriptorType	1	37	Deskryptor punktu końcowego specyficzny dla klasy
bDescriptorSubtype	1	1	Ogólny
bmAttributes	1	0x00	Dodatkowe atrybuty punktu końcowego
bLockDelayUnits	1	2	Czas synchronizacji zegara – jednostka miary
wLockDelay	2	2400	Czas synchronizacji zegara – wartość



Rys. 3.1. Przykładowe urządzenie audio

Zatem pierwszym opisywanym blokiem jest terminal wejściowy. Pole `bDescriptorSubtype` jego deskryptora ma wartość 2. Pole `bTerminalID` zawiera unikalny identyfikator bloku terminalu wejściowego. Pole `wTerminalType` określa typ terminalu. W prezentowanym przykładzie ma ono wartość 0x0101, co oznacza wyjściowy punkt końcowy dla danych izochronicznych. Innymi słowy, źródłem dźwięku jest interfejs USB (ang. *USB streaming*). Wszystkie wartości, jakie mogą wystąpić w tym polu, opisane są w [29]. Jeśli deskryptor opisująły terminal dwukierunkowy, to w polu `bAssocTerminal` należałoby podać numer terminalu wyjściowego związanego z tym terminalem. Dla terminali jednokierunkowych w polu tym wpisuje się zero. Pole `bNrChannels` określa liczbę kanałów dźwięku. Pole `wChannelConfig` określa położenie tych kanałów w przestrzeni. W przykładzie mamy jeden kanał położony centralnie z przodu (ang. *center front*). Sposób kodowania wartości w tym polu opisany jest szczegółowo w [26]. Pola `iChannelNames` i `iTerminal` zawierają numery opcjonalnych deskryptorów tekstowych, w których można umieścić dodatkowe informacje odpowiednio o kanałach dźwiękowych i samym terminalu. Zero oznacza brak deskryptora tekstowego.

Drugim opisywanym blokiem jest regulator. Pole `bDescriptorSubtype` jego deskryptora ma wartość 6. Pole `bUnitID` zawiera unikalny identyfikator bloku regulatora. Pole `bSourceID` zawiera identyfikator bloku, którego wyjście jest podłączone do wejścia regulatora. Pole `bControlSize` zawiera łączny rozmiar w bajtach nastę-

pujących po nim pól bitowych `bmaControls` opisujących, jakie elementy regulacyjne udostępnia regulator definiowany tym deskryptorem. Wartość 0x0003 oznacza układ wyciszający i regulator głośności. Format tych pól bitowych jest dokładnie omówiony w [26]. Pole bitowe `bmaControls0` opisuje elementy regulacyjne wspólne dla wszystkich kanałów dźwiękowych (ang. *master channel*). Opcjonalnie po tym polu mogą wystąpić kolejne pola bitowe `bmaControls1`, `bmaControls2` itd. dla poszczególnych kanałów dźwiękowych. Numer w nazwie pola jest numerem kanału. Ostatnim polem jest `iFeature`. Zawiera ono opcjonalny numer deskryptora tekstowego opisującego regulator. Zero oznacza brak deskryptora tekstowego.

Trzecim opisanym blokiem jest terminal wyjściowy. Pole `bDescriptorSubtype` jego deskryptora zawiera wartość 3. Pole `bTerminalID` zawiera unikalny identyfikator bloku terminalu wyjściowego. Pole `wTerminalType` określa typ terminalu. W przedstawionym przykładzie ma ono wartość 0x0304, co oznacza głośnik (ang. *desktop speaker*). Wszystkie wartości, jakie mogą wystąpić w tym polu, opisane są w [29]. Jeśli deskryptor opisywałby terminal dwukierunkowy, to w polu `bAssocTerminal` należałoby podać numer terminalu wejściowego związanego z tym terminalem. Dla terminali jednokierunkowych w polu tym wpisuje się zero. Pole `bSourceID` zawiera identyfikator bloku, którego wyjście jest podłączone do wejścia tego terminalu wyjściowego. Pole `iTerminal` zawiera numer opcjonalnego deskryptora tekstowego, w którym można umieścić dodatkowe informacje o terminalu. Zero oznacza brak deskryptora tekstowego.

Po opisie struktury urządzenia następują deskryptory interfejsów dla transmisji danych (wartość 2 w polu `bInterfaceSubClass`). W rozważanym przykładzie mamy tylko jeden taki interfejs, ale ma on dwa warianty ustawień – odpowiednio wartości 0 i 1 w polu `bAlternateSetting`. W wariantie zerowym nie ma punktów końcowych. Ten wariant reprezentuje interfejs wyłączony, gdy nie są przesyłane żadne dane. Wariant pierwszy reprezentuje interfejs włączony i ma jeden punkt końcowy. Bezpośrednio po standardowych deskryptorach interfejsów następują deskryptory interfejsów specyficzne dla klasy audio. W rozważanym przykładzie są dwa takie deskryptory.

Pierwszy specyficzny dla klasy audio deskryptor interfejsu danych ma wartość 1 w polu `bDescriptorSubtype` i opisuje ogólne własności interfejsu. Pole `bTerminalLink` zawiera identyfikator terminalu, z którym powiązany jest ten interfejs. Pole `bDelay` określa opóźnienie wnoszone przez całą ścieżkę danych od terminalu wejściowego do wyjściowego. Opóźnienie wyraża się w liczbie ramek, czyli w przypadku urządzenia FS w milisekundach. Informacja o wprowadzanym przez urządzenie opóźnieniu jest istotna, gdyż pozwala zachować prawidłowe relacje fazowe między sygnałami przetwarzanymi przez różne urządzenia. Pole `wFormatTag` określa format przesyłanych tym interfejsem danych. Może to być kodowanie z równomiernym próbkowaniem PCM, kodowanie zmiennoprzecinkowe, kodowanie z nierównomiernym próbkowaniem stosowane w telekomunikacji według specyfikacji A-Law lub μ-Law, różne warianty MPEG i inne. Szczegóły są opisane w [28]. W przedstawionym przykładzie wykorzystujemy kodowanie PCM.

Drugi deskryptor ma wartość 2 w polu `bDescriptorSubtype` i opisuje bardziej szczegółowo format kodowania. Pole `bFormatType` określa rodzinę formatów, do

której należy opisywany format. Szczegółowy opis formatów znajduje się w [28]. Pole `bNrChannels` określa liczbę kanałów. Pole `bSubframeSize` określa liczbę bajtów zajmowanych przez jedną próbkę. Próbka może zajmować 1, 2, 3 lub 4 bajty, ale nie wszystkie bity muszą być wykorzystane. Pole `bBitResolution` określa, ile bitów faktycznie koduje jedną próbkę. Pole `bSamFreqType` określa częstotliwość próbkowania. Jeśli ma wartość zero, to po nim umieszcza się dwa trzybajtowe pola określające odpowiednio minimalną i maksymalną dopuszczalną częstotliwość próbkowania. Jeśli pole `bSamFreqType` ma wartość niezerową, to określa liczbę dopuszczalnych dyskretnych wartości częstotliwości próbkowania. Listę tych częstotliwości umieszcza się po tym polu. Każdy element listy ma trzy bajty.

Interfejs musi jednoznacznie wyznaczać punkt końcowy, którym dostarczane są dane do wymienionego w nim terminalu (jeśli jest to terminal wejściowy) lub którym odbierane są dane z tego terminalu (jeśli jest to terminal wyjściowy). Dlatego po deskryptorach interfejsów umieszcza się deskryptory punktów końcowych. Najpierw standardowy deskryptor punktu końcowego. W porównaniu do formatu standardowego deskryptora punktu końcowego opisanego w rozdziale 1 deskryptor ten ma dwa dodatkowe pola. Pole `bRefresh` nie jest używane i ma ono zawsze wartość zero. Pole `bSynchAddress` zawiera adres punktu końcowego, który służy do przesyłania informacji potrzebnych do synchronizacji strumienia danych. Jeśli nie korzysta się z takiego punktu końcowego, pole to powinno mieć wartość zero.

Po standardowym deskryptorze punktu końcowego umieszcza się ogólny deskryptor punktu końcowego specyficzny dla klasy audio. Deskryptor ten ma w polu `bDescriptorType` wartość 37, a w polu `bDescriptorSubtype` – wartość 1. Pola `bLock-DelayUnits` i `wLockDelay` określają czas potrzebny do synchronizacji wewnętrznego zegara urządzenia. Pierwsze pole określa jednostkę miary czasu: wartość 1 oznacza milisekundę, a wartość 2 – okres próbkowania. Drugie z tych pól zawiera liczbę jednostek czasu. Jest to czas, przez który po rozpoczęciu transmisji należy przesyłać ciszę. Pole bitowe `bmAttributes` zawiera dodatkowe atrybuty:

- ustawiony bit 0 oznacza, że punkt końcowy umożliwia sterowanie częstotliwością próbkowania;
- ustawiony bit 1 oznacza, że punkt końcowy umożliwia sterowanie prędkością odtwarzania (ang. *pitch control*);
- ustawiony bit 7 oznacza, że dopuszczalne są tylko pakiety danych o rozmiarze podanym w polu `wMaxPacketSize` lub pakiety puste;
- wyzerowany bit 7 oznacza, że dopuszczalne są pakiety danych o rozmiarze nieprzekraczającym tego podanego w polu `wMaxPacketSize`;
- pozostałe bity są zarezerwowane i powinny mieć wartość zero.

3.3.2. Żądania

Urządzenie klasy audio musi obsługiwać, opisane w rozdziale 1, żądania standarde oraz żądania specyficzne dla tej klasy i przedstawione w tabeli 3.18. Żądania mogą być kierowane do interfejsu. Wtedy pole `bmRequestType` ma wartość 00100001 lub 10100001, zależnie od kierunku żądania. Młodszy bajt pola `wIndex` zawiera numer interfejsu, a starszy bajt identyfikator bloku. Żądania mogą być też kierowa-

ne do punktu końcowego. Wtedy pole `bmRequestType` ma wartość 00100010 lub 10100010, zależnie od kierunku żądania. Młodszy bajt pola `wIndex` zawiera adres punktu końcowego, a starszy ma wartość zero.

Tab. 3.18. Żądania specyficzne dla klasy audio

<code>bmRequestType</code>	<code>bRequest</code>	<code>wValue</code>	<code>wIndex</code>	<code>wLength</code>	<code>Dane</code>
00100001	SET_CUR 1 SET_MIN 2 SET_MAX 3 SET_RES 4	Identyfikator atrybutu	Identyfikator bloku i numer interfejsu	Rozmiar parametrów	Parametry
00100010			Adres punktu końcowego		
10100001	GET_CUR 129 GET_MIN 130 GET_MAX 131 GET_RES 132	Identyfikator atrybutu	Identyfikator bloku i numer interfejsu	Rozmiar parametrów	Parametry
10100010			Adres punktu końcowego		
00100001	SET_MEM 5	Przesunięcie	Identyfikator bloku i numer interfejsu	Rozmiar danych	Dane
00100010			Adres punktu końcowego		
10100001	GET_MEM 133	Przesunięcie	Identyfikator bloku i numer interfejsu	Rozmiar danych	Dane
10100010			Adres punktu końcowego		
10100001	GET_STAT 255	0	Identyfikator bloku i numer interfejsu	Rozmiar statusu	Status
10100010			Adres punktu końcowego		

Żądania `SET_CUR`, `SET_MIN`, `SET_MAX` i `SET_RES` służą do ustawiania wartości atrybutów. Za pomocą żądań `GET_CUR`, `GET_MIN`, `GET_MAX` i `GET_RES` odczytuje się wartości atrybutów. Atrybut identyfikuje się za pomocą pola `wValue`. Wszystkie wartości są opisane w [26]. Pole `wLength` zawiera rozmiar atrybutu w bajtach. Wartość atrybutu przesyła się w fazie danych obsługi żądania.

Urządzenie audio może udostępniać bardzo generyczny interfejs, pozwalając na bezpośredni odczyt i zapis fragmentu swojej pamięci lub odwzorowując pewne funkcje w udostępnianym obszarze pamięci. Żądanie `SET_MEM` zapisuje obszar pamięci. Żądanie `GET_MEM` odczytuje obszar pamięci. Pole `wValue` zawiera przesunięcie w bajtach względem początku udostępnianego obszaru pamięci. Pole `wLength`

zawiera rozmiar przesyłanego bloku danych w bajtach. Zawartość bloku danych przesyła się w fazie danych obsługi żądania.

Żądanie GET_STAT służy do odczytania statusu bloku funkcjonalnego lub punktu końcowego. To żądanie jest przewidziane do wykorzystania w przyszłości, co zapewne oznacza, że nigdy nie będzie używane. Format statusu nie został zdefiniowany. Specyfikacja nakazuje w odpowiedzi na to żądanie odsyłać pakiet danych o zerowej długości, niezależnie od wartości w polu wLength.

3.3.3. Synchronizacja

Prawidłowe odtwarzanie dźwięku wymaga dopasowania częstotliwości próbkowania źródła (ang. *source*) i ujścia (ang. *sink*) dźwięku. Ponieważ transmisji izochronicznych nie potwierdza się, nie można zastosować kontroli przepływu opartej na potwierdzeniach. Zamiast tego standard USB definiuje inne sposoby synchronizacji, które zostały zebrane w **tabeli 3.19**.

Przy transmisji asynchronicznej zegar domeny audio, czyli zegar odpowiedzialny za generowanie częstotliwości próbkowania, nie jest zsynchronizowany z zegarem USB. Asynchroniczne źródło wysyła w każdej ramce tyle próbek sygnału, ile wynika z częstotliwości próbkowania. Liczba próbek w ramce się zmienia. Odbiorca danych musi odtworzyć częstotliwość próbkowania na podstawie średniej liczby próbek w ramce. Asynchroniczne ujście stosuje jawne sprzężenia zwrotne. Sprzężenie zwrotne realizuje się za pomocą punktu końcowego przesyłającego do źródła sygnału informację o liczbie próbek oczekiwanych w kolejnych ramkach.

Tab. 3.19. Warianty synchronizacji danych izochronicznych

Transmisja	Źródło	Ujście
Asynchroniczna	Swobodna częstotliwość próbkowania Niejawne sprzężenie ze strumieniem danych	Swobodna częstotliwość próbkowania Jawne sprzężenie zwrotne
Synchroniczna	Częstotliwość próbkowania synchronizowana do SOF Jawne sprzężenie zwrotne	Częstotliwość próbkowania synchronizowana do SOF Jawne sprzężenie zwrotne
Adaptacyjna	Częstotliwość próbkowania synchronizowana do ujścia Jawne sprzężenie zwrotne	Częstotliwość próbkowania synchronizowana do strumienia danych Niejawne sprzężenie ze strumieniem danych

Przy transmisji synchronicznej częstotliwość próbkowania jest synchronizowana do częstotliwości ramek lub mikroramek USB, przy użyciu jawnego sprzężenia zwrotnego, czyli na przykład pętli fazowej. Przypomnijmy, że kontroler sygnalizuje początek każdej ramki lub mikroramki, wysyłając pakiet SOF.

Przy transmisji adaptacyjnej częstotliwość próbkowania nie jest zsynchronizowana z zegarem USB, podobnie jak przy transmisji asynchronicznej. Adaptacyjne źródło stosuje jawnie sprzężenia zwrotne. Jest to punkt końcowy, za pomocą którego źródło sygnału odbiera informację o liczbie próbek, które mają być wygenerowane w kolejnych ramkach. Adaptacyjne ujście dopasowuje się do szybkości napływu danych i odtwarza częstotliwość próbkowania na podstawie średniej liczby próbek w ramce.

Z powyższego opisu należy wysnuć następujące wnioski. Można uzyskać synchroniczną transmisję i zapewnić, że sygnał dźwiękowy nie ulegnie degeneracji w trzech przypadkach:

- adaptacyjne źródło jest połączone z dowolnym wariantem ujścia;
- adaptacyjne ujście jest połączone z dowolnym wariantem źródła;
- synchroniczne źródło jest połączone z synchronicznym ujściem.

Jeśli nie można uzyskać synchronizacji i zapobiec degeneracji sygnału dźwiękowego, to pewne próbki sygnału muszą być pomijane (ang. *sample slip*) lub powtarzane (ang. *sample stuff*). Sytuacja taka zachodzi, gdy:

- asynchroniczne źródło jest połączone z asynchronicznym ujściem;
- asynchroniczne źródło jest połączone z synchronicznym ujściem;
- synchroniczne źródło jest połączone z asynchronicznym ujściem.

Przy przesyłaniu dźwięku, oprócz synchronizacji częstotliwości próbkowania, aby zapewnić poprawne odtwarzanie efektów przestrzennych, ważne jest zachowanie prawidłowych relacji fazowych między kanałami. Dlatego w deskryptorze interfejsu dla transmisji danych podaje się opóźnienie, jakie wprowadza ten interfejs. Opóźnienia tego nie można uniknąć, gdyż dane generuje się i odtwarza w sposób ciągły, a transmituje w pakietach. Zatem w praktyce dla zapewnienia ciągłości odtwarzania każde urządzenie musi mieć bufor mieszczący co najmniej dwie ramki danych. W przypadku źródła dane z jednej ramki są wysyłane, a dane potrzebne do wygenerowania kolejnej ramki są gromadzone. W przypadku ujścia dane z jednej ramki są odtwarzane, a jednocześnie oczekuje się na odebranie kolejnej ramki. Opóźnienie wprowadzane przez interfejs deklaruje się w wielokrotnościach ramki. Jeśli wynosi ono δ ramek, to dla zachowania zgodności fazy z dokładnością do okresu próbkowania należy zapewnić, aby pierwsza próbka odebrana w ramce n została odtworzona jako pierwsza w ramce $n + \delta$.

3.3.4. Przetwornik cyfrowo-analogowy

dac.h

W przykładowej aplikacji dwa bloki urządzenia z rysunku 3.1, czyli regulator głośności z możliwością wyciszenia dźwięku oraz terminal wyjściowy, są zintegrowane w module jednokanałowego (monofonicznego) przetwornika cyfrowo-analogowego. Jego interfejs zadeklarowany jest w pliku *dac.h* w sposób na tyle ogólny, aby możliwe było wiele jego implementacji. Wszystkie wartości dotyczące poziomu głośności podaje się w jednostkach odpowiadających 1/256 dB, czyli np. wartości 256, 0 i -192 oznaczają odpowiednio 1 dB, 0 dB i -1,5 dB.

```
int DACconfigure(unsigned samplingFrequency);
```

Funkcja *DACconfigure* inicjuje i uaktywnia przetwornik cyfrowo-analogowy. Parametr *samplingFrequency* określa częstotliwość próbkowania w hercach. Funkcja zwraca zero, jeśli konfigurowanie przetwornika zakończyło się powodzeniem, a wartość ujemną, gdy wystąpił błąd.

```
void DACreset();
```

Bezparametrowa funkcja `DACreset` zeruje bufor przetwornika i przywraca początkowe ustawienia, czyli domyślny poziom głośności z wyłączonem wyciszaniem. Powinna być wywołana raz przed uaktywnieniem przetwornika, a po jego uaktywnieniu może być wywoływaną wielokrotnie. Funkcja ta nie zwraca żadnej wartości.

```
uint8_t DACgetMute();
```

Bezparametrowa funkcja `DACgetMute` zwraca zero, gdy wyciszczenie nie jest aktywne, a jedynkę, gdy jest uaktywnione.

```
void DACsetMute(uint8_t new_mute);
```

Funkcja `DACsetMute` steruje wyciszaniem. Jeśli parametr `new_mute` ma wartość zero, wyciszczenie zostaje wyłączone, a jeśli ma wartość jeden, to jest uaktywniane. Funkcja ta nie zwraca żadnej wartości.

```
int16_t DACgetVolume();
```

Bezparametrowa funkcja `DACgetVolume` zwraca aktualnie ustawiony poziom głośności.

```
int16_t DACgetVolumeMin();
```

Bezparametrowa funkcja `DACgetVolumeMin` zwraca minimalny dopuszczalny poziom głośności.

```
int16_t DACgetVolumeMax();
```

Bezparametrowa funkcja `DACgetVolumeMax` zwraca maksymalny dopuszczalny poziom głośności.

```
void DACsetVolume(int16_t new_voulume);
```

Funkcja `DACsetVolume` ustawia nowy poziom głośności podany w parametrze `new_voulume`. Nie zwraca żadnej wartości.

```
uint16_t DACgetResolution();
```

Bezparametrowa funkcja `DACgetResolution` zwraca ustawioną rozdzielcość regulacji poziomu głośności, czyli wartość, której wielokrotnością muszą być aktualny, minimalny i maksymalny poziom głośności. Przykładowo, jeśli funkcja ta zwróci 256, to wartości te muszą być wyrażone całkowitą liczbą decybeli.

```
void DACsetResolution(uint16_t new_resolution);
```

Funkcja `DACsetResolution` ustawia rozdzielcość regulacji poziomu głośności. Nową rozdzielcość podaje się w parametrze `new_resolution`. Przykładowa implementacja ustawia nową wartość, ale jej nie używa. Ze względu na niezbyt wysoką rozdzielcość zastosowanego przetwornika cyfrowo-analogowego poziom głośności jest zaokrąglany do całkowitej liczby decybeli. Funkcja ta nie zwraca żadnej wartości.

```
void DACputSamples(int16_t const *samples, uint32_t count);
```

Funkcja `DACputSamples` wstawia próbki do bufora przetwornika cyfrowo-analogowego. Parametr `samples` wskazuje na początek tablicy z próbками. Za pomocą parametru `count` przekazuje się liczbę próbek do wstawienia. Wartości próbek są kwantowane liniowo i zapisywane jako 16-bitowa liczba ze znakiem w kodzie uzupełnieniowym do dwójki. Funkcja ta nie zwraca żadnej wartości.

`dac_buffer.h`

`dac_buffer.c`

Przykładowa implementacja modułu przetwornika cyfrowo-analogowego zawarta jest w plikach `dac_buffer.h` i `dac_buffer.c`. Nazwy tych plików sugerują, że implementacja ta korzysta z bufora. Jest to niezbędne, gdyż próbki przychodzą w pakietach. W każdej ramce USB odbierany jest jeden pakiet. Natomiast odtwarzanie musi być ciągłe. Zatem odebrany pakiet musi być gdzieś przez chwilę przechowyany. Jak widzimy w tabeli 3.17, w przykładowej implementacji dźwięk jest próbkowany z częstotliwością 48 kHz i z rozdzielcością 16 bitów, czyli pakiet ma rozmiar 96 bajtów i przesyła się w nim 48 próbek. Przetwornik cyfrowo-analogowy jest taktowany zegarem mikrokontrolera i nie możemy zapewnić jego synchronizacji z zegarem kontrolera USB, dlatego (posługując się nomenklaturą z poprzedniego podrozdziału) w przykładowej implementacji stosujemy użycie z adaptacyjną synchronizacją. Jak widzimy w tabeli 3.17, parametr `bmAttributes` standardowego deskryptora punktu końcowego ma wartość 9, co oznacza, że jest to izochroniczny punkt końcowy z adaptacyjną synchronizacją.

W celu uzyskania synchronizacji częstotliwości próbkowania regulujemy okres próbkowania przetwornika. Okres ten wyznaczany jest za pomocą jednego z dostępnych w mikrokontrolerze liczników. Dla ustalenia uwagi przyjmijmy, że licznik taktowany jest z częstotliwością 72 MHz. Żeby uzyskać żądaną okres próbkowania, czyli 1/48 ms, licznik powinien zgłaszać przerwanie inicjujące kolejny cykl przetwornika co 1500 taktów zegara. Tak byłoby, gdyby wszystkie zegary pracowały z częstotliwością nominalną, co w praktyce nigdy nie ma miejsca. Stosujemy bufor mogący pomieścić 191 próbki sygnału. Obserwując wypełnienie bufora, modyfikujemy okres próbkowania. Jeśli w buforze jest od 0 do 63 próbek, to wskazuje, że przetwornik zbyt szybko czyta z bufora. Zwalniamy go, ustawiając okres próbkowania na 1501 taktów zegara. Jeśli w buforze jest od 64 do 127 próbek, to wypełnienie bufora jest optymalne i ustawiamy okres próbkowania na nominalne 1500 taktów zegara. Jeśli w buforze jest od 128 do 191 próbek, to oznacza, że

przetwornik pracuje za wolno. Przyspieszamy go, ustawiając okres próbkowania na 1499 taktów zegara.

Powyższa strategia doskonale sprawdza się, gdy względna różnica częstotliwości próbkowania źródła i ujścia jest mniejsza niż odwrotność liczby taktów zegara na jedną próbkę, czyli w tym przypadku 1/1500. W praktyce, gdy stosuje się kwarcową stabilizację zegarów, warunek ten jest zawsze spełniony.

Przedstawione rozwiązanie ma wadę. Okres próbkowania zmienia się skokowo, co wprowadza nierównomierność częstotliwości próbkowania (ang. *jitter*). Z pewnością nie jest to rozwiązanie, które zadowoliłoby audiofilów. Jednak sam przetwornik cyfrowo-analogowy zastosowany w mikrokontrolerze jest tylko 12-bitowy. Dodatkowe zniekształcenia wprowadza, włączony domyślnie, bufor (wzmacniacz) wyjściowy. Można go wyłączyć, co zmniejsza jednak dopuszczalne maksymalne obciążenie przetwornika. Szczegółów należy poszukać w nacie katalogowej mikrokontrolera. Wbudowany przetwornik nie jest więc wysokiej klasy. Natomiast doskonale nadaje się do odtwarzania sygnału mowy czy różnego rodzaju melodyjek i z pewnością sprawdzi się w wielu aplikacjach. Do odtwarzania muzyki z wysoką jakością należy zastosować wysokiej klasy zewnętrzny przetwornik cyfrowo-analogowy.

```
dac_12bits_10x.c  
dac_12bits_15x.c  
dac_12bits_2xx.c  
dac_12bits_4xx.c
```

Funkcja inicjująca przetwornik cyfrowo-analogowy i procedura obsługi jego przerwania, które są zależne od modelu mikrokontrolera, zostały wydzielone i znajdują się w plikach *dac_12bits_10x.c*, *dac_12bits_15x.c*, *dac_12bits_2xx.c* i *dac_12bits_4xx.c*. Końcówka nazwy pliku identyfikuje rodzinę układów, dla których przeznaczony jest ten plik. W procedurze obsługi przerwania próbki z bufora przesyłane są do przetwornika. Przy czym następuje dodanie składowej stałej, co jest potrzebne, gdyż przetwornik nie może wytworzyć na wyjściu napięcia ujemnego – wytwarza sygnał o wartościach z przedziału od zera do napięcia zasilania. Ponadto musi nastąpić redukcja rozdzielczości. Przetwornik jest 12-bitowy i cztery najmniej znaczące bity każdej próbki są po prostu ignorowane. W procedurze obsługi przerwania wyznaczany jest też okres próbkowania na podstawie wypełnienia bufora, według wyżej opisanego algorytmu. Sygnał analogowy jest dostępny na wyjściu PA4.

Żeby zapewnić spójność danych w buforze, zapis do niego musi się odbywać przy zablokowanym przerwaniu przetwornika. Czas blokowania musi być możliwie krótki, aby próbki były przetwarzane w regularnych odstępach czasu. Alternatywnym rozwiązaniem mogłoby być zastosowanie DMA do przesyłania próbek z bufora do przetwornika, ale to wymagałoby zmiany algorytmu działania bufora i dodania obsługi przerwania DMA.

```
dac_pwm_10x.c
```

W modelach mikrokontrolerów, które nie mają przetwornika cyfrowo-analogowego, do wytworzenia sygnału analogowego można wykorzystać modulację czasu trwa-

nia impulsu – PWM (ang. *Pulse Width Modulation*). Przykładowa implementacja dla modeli STM32F10x znajduje się w pliku *dac_pwm_10x.c*. Przy częstotliwości taktowania rdzenia 48 MHz można uzyskać rozdzielcość 9-bitową, a przy częstotliwości 72 MHz – 10-bitową. Aby skonfigurować ten rodzaj przetwornika, w pliku *board_def.h* należy zdefiniować następujące stałe:

- `PWM_TIM_N` – numer licznika;
- `PWM_TIM_CH` – numer kanału licznika wykorzystywany do generowania przebiegu PWM;
- `PWM_OUT_GPIO_N` – literowe oznaczenie portu, na który wyprowadzony jest przebieg PWM;
- `PWM_OUT_PIN_N` – numer wyjścia, na które wyprowadzony jest przebieg PWM.

Przykładowo, jeśli korzystamy z pierwszego kanału licznika TIM3, to sygnał PWM jest dostępny na wyjściu PA6 i definicje te powinny wyglądać następująco:

```
#define PWM_TIM_N          3
#define PWM_TIM_CH         1
#define PWM_OUT_GPIO_N     A
#define PWM_OUT_PIN_N      6
```

Na koniec rozważań o przetworniku cyfrowo-analogowym należy wspomnieć, że do jego wyjścia można bezpośrednio podłączyć przetwornik piezoelektryczny. Jeśli chcemy sygnał z wyjścia przetwornika podać na wejście analogowego wzmacniacza mocy, to należy go bezwzględnie przepuścić przez filtr dolnoprzepustowy o częstotliwości granicznej mniejszej niż połowa częstotliwości próbkowania.

<i>cs43l22.h</i>
<i>cs43l22.c</i>
<i>cs43l22_322xg.c</i>
<i>cs43l22_disco_4xx.c</i>
<i>ioe_stm322xg.h</i>
<i>ioe_stm322xg.c</i>

W zestawie STM3220G-EVAL i module STM32F4-Discovery można wyprowadzić sygnał analogowy na zainstalowane w nich gniazda słuchawkowe, używając jako wzmacniacza układu CS43L22 [3]. Przy czym korzystamy tylko z jego części analogowej. Wyprowadzenie zerujące układ CS43L22 w STM3220G-EVAL podłączone jest do eksplandera wejścia-wyjścia, a eksplander do mikrokontrolera podłączony jest za pomocą szyny I²S. Dane cyfrowe z mikrokontrolera do układu CS43L22 przesyła się za pomocą szyny I²S. Nawet jeśli korzysta się tylko z analogowej części tego układu, interfejs I²S musi być uruchomiony i trzeba nim przesyłać ciszę. Mając powyższe na uwadze, należy do projektu dołączyć pliki:

- *cs43l22.c* – implementacja obsługi układu CS43L22;
- *cs43l22_322xg.c* – specyficzna dla STM3220G-EVAL część implementacji obsługi CS43L22;

- *cs43l22_disco_4xx.c* – specyficzna dla STM32F4-Discovery część implementacji obsługi CS43L22;
- *ioe_stm322xg.c* – obsługa ekspandera wejśc-wyjśc (tylko dla STM3220G-EVAL);
- *i2c.c* – implementacja obsługi szyny I²C (tylko dla STM3220G-EVAL);
- *i2s.c* – implementacja obsługi szyny I²S.

Ponadto potrzebne są pliki nagłówkowe:

- *cs43l32.h* – deklaracja interfejsu dla układu CS43L32;
- *ioe_stm322xg.h* – deklaracja interfejsu dla ekspandera wejśc-wyjśc (tylko dla STM3220G-EVAL);
- *i2c.h* – deklaracja interfejsu szyny I²C (tylko dla STM3220G-EVAL);
- *i2s.h* – deklaracja interfejsu szyny I²S.

Moduły *i2c* i *i2s* są szczegółowo opisane w rozdziale 2.

3.3.5. Implementacja

ex_audio_dev.c

Implementacja odtwarzacza audio znajduje się w pliku *ex_audio_dev.c*. Jest ona w wielu miejscach bardzo podobna do implementacji zaprezentowanych już w tym rozdziale dwóch urządzeń, czyli myszy i wirtualnego portu szeregowego. Dlatego pomijam opis deskryptora urządzenia, deskryptorów tekstowych oraz funkcji USB-DgetApplicationCallbacks, Reset, GetDescriptor, GetConfiguration. W celu opisania konfiguracji urządzenia musimy zdefiniować strukturę deskryptorów i wypełnić ją zgodnie z tabelą 3.17.

```
typedef struct {
    usb_configuration_descriptor_t           cnf_descr;
    usb_interface_descriptor_t                if0_descr;
    usb_ac_header_descriptor_t               ac_h_descr;
    usb_ac_input_terminal_descriptor_t       ac_it_descr;
    usb_ac_feature_unit_descriptor_t        ac_fu_descr;
    usb_ac_output_terminal_descriptor_t     ac_ot_descr;
    usb_interface_descriptor_t               if10_descr;
    usb_interface_descriptor_t               if11_descr;
    usb_as_general_descriptor_t             as_g_descr;
    usb_as_format_type_descriptor_t         as_ft_descr;
    usb_std_audio_data_endpoint_descriptor_t std_eplin_descr;
    usb_cs_audio_data_endpoint_descriptor_t cs_eplin_descr;
} __packed usb_audio_configuration_t;
static usb_audio_configuration_t const audio_configuration = {
    ...
};
```

Pole `cnf_descr` zawiera deskryptor konfiguracji. Pole `if0_descr` zawiera standardowy deskryptor interfejsu numer 0, który pełni funkcję interfejsu sterującego. Pole `ac_h_descr` zawiera deskryptor interfejsu specyficzny dla klasy audio – jest to nagłówek opisu struktury urządzenia. Pole `ac_it_descr` zawiera deskryptor interfejsu specyficzny dla klasy audio opisujący terminal wejściowy. Pole `ac_fu_descr` zawiera deskryptor interfejsu specyficzny dla klasy audio opisujący regulator. Pole `ac_ot_descr` zawiera deskryptor interfejsu specyficzny dla klasy audio opisujący terminal wyjściowy. Pole `if10_descr` zawiera standardowy deskryptor interfejsu numer 1, opisujący jego wariant numer 0, czyli interfejs dla transmisji danych będący w stanie wyłączonej. Pole `if11_descr` zawiera standardowy deskryptor interfejsu numer 1, opisujący jego wariant numer 1, czyli włączony interfejs dla transmisji danych. Pola `as_g_descr` i `as_ft_descr` zawierają deskryptory interfejsu specyficzne dla klasy audio i opisują szczegółowo format danych przesyłanych tym interfejsem. Pola `std_eplin_descr` i `cs_eplin_descr` zawierają odpowiednio standardowy i specyficzny dla klasy audio deskryptor punktu końcowego.

Stan aplikacji opisują cztery zmienne globalne. Zmienna `progress` jest zwiększana o jeden (modulo rozmiar typu `unsigned`) po odebraniu każdego pakietu i służy do wyświetlania postępu transmisji danych. Zmienna `configuration` przechowuje numer wybranej konfiguracji. Ponieważ jest tylko jedna konfiguracja, to ma ona wartość zero, gdy konfiguracja ta nie została wybrana, a wartość jeden, gdy została już wybrana. Zmienna `interface1` przechowuje numer wybranego wariantu ustawień interfejsu dla transmisji danych (interfejsu numer 1). Ma ona wartość jeden, gdy transmitowane są dane, a zero, gdy nie są transmitowane żadne dane. Pomożnicza zmienna `refresh` przyjmuje wartość jeden, gdy trzeba odświeżyć zawartość ekranu wyświetlacza ciekłokrystalicznego.

```
static unsigned progress;
static uint8_t configuration, interface1, refresh;
```

Odświeżaniem zawartości ekranu zajmuje się funkcja zwrotna `LCDrefresh`. Jest ona wywoływaną cyklicznie w funkcji `main` programu. Jeśli zmienna `refresh` ma niezerową wartość, to jest ona zerowana, a następnie wyświetlana jest aktualnie ustalona głośność w decybelach oraz informacja o aktywacji wyciszenia. Jeśli interfejs dla transmisji danych jest aktywny, to dodatkowo wyświetlany jest postęp transmisji danych.

```
#define LCD_REFRESH 8

static void LCDrefresh(void) {
    static const char fan[4] = {'|', '/', '-', '\\'};
    char buffer[20];

    if (refresh) {
        refresh = 0;
        sprintf(buffer, "%3hd dB %s %c",
                DACgetVolume() >> 8, DACgetMute() ? "MUTE" : "     ",
                interface1 ? fan[(progress >> LCD_REFRESH) & 3] : ' ');
    }
}
```

```

LCDgoto(0, 0);
LCDwrite(buffer);
}
}

```

Pomocnicza funkcja ResetState przywraca domyślne ustawienia przetwornika cyfrowo-analogowego i inicjuje wszystkie zmienne globalne. Jest ona wywoływana raz na początku działania aplikacji w funkcji Configure oraz w funkcji Reset za każdym razem, gdy kontroler USB zainicjuje procedurę zerowania szyny.

```

static void ResetState(void) {
    DACreset();
    progress = 0;
    configuration = 0;
    interface1 = 0;
    refresh = 1;
}

```

Funkcja Configure konfiguruje urządzenie. Jest ona wywoływana raz po włączeniu zasilania lub wyzerowaniu mikrokontrolera. Funkcja ta inicjuje zmienne globalne i bufor przetwornika. Następnie rejestruje funkcję zwrotną LCDrefresh, żeby była cyklicznie wywoływana w funkcji main. Na koniec uaktywnia przetwornik cyfrowo-analogowy z częstotliwością próbkowania 48 kHz.

```

int Configure() {
    ResetState();
    LCDsetRefresh(LCDrefresh);
    return DACconfigure(48000);
}

```

Funkcja zwrotna SetConfiguration uaktywnia konfigurację urządzenia i konfiguruje przynależne do niej punkty końcowe. W naszym przypadku jest to tylko jeden wyjściowy punkt końcowy dla danych izochronicznych. Rozmiar pakietu ISO_BUFF_SIZE deklarujemy na 96 bajtów, gdyż w każdym pakiecie są przesyłane 48 próbki sygnału, a każda próbka zajmuje 2 bajty.

```

usb_result_t SetConfiguration(uint16_t confValue) {
    if (confValue > device_descriptor.bNumConfigurations)
        return REQUEST_ERROR;
    configuration = confValue;
    USBDdisableAllNonControlEndPoints();
    if (confValue == audio_configuration.cnf_descr.bConfigurationValue)
        return USBDEndPointConfigure(ENDP1, ISOCHRONOUS_TRANSFER,
                                      ISO_BUFF_SIZE, 0);
    return REQUEST_SUCCESS; /* confValue == 0 */
}

```

Funkcja zwrotna GetInterface jest wywoływana, gdy kontroler USB zażąda przesłania wybranego wariantu ustawień dla interfejsu. Parametr interface zawiera numer interfejsu, którego dotyczy żądanie. Parametr setting jest wskaźnikiem na zmienną, której trzeba przypisać żądaną wartość. Interfejs numer 0 jest interfejsem sterującym i ma tylko jeden wariant ustawień, czyli wariant zerowy. Interfejs numer 1 jest interfejsem dla transmisji danych i ma dwa warianty ustawień. Jak już to zostało napisane wyżej, aktualnie wybrany wariant ustawień interfejsu numer 1 przechowywany jest w zmiennej interface1.

```
usb_result_t GetInterface(uint16_t interface, uint8_t *setting) {
    if (interface == 0) {
        *setting = 0;
        return REQUEST_SUCCESS;
    }
    else if (interface == 1) {
        *setting = interface1;
        return REQUEST_SUCCESS;
    }
    else {
        *setting = 0;
        return REQUEST_ERROR;
    }
}
```

Funkcja zwrotna SetInterface jest wywoływana, gdy kontroler USB zażąda aktywowania wybranego wariantu ustawień dla interfejsu. Parametr interface zawiera numer interfejsu, którego dotyczy żądanie. Parametr setting zawiera numer wariantu, który ma być ustawiony.

```
usb_result_t SetInterface(uint16_t interface, uint16_t setting) {
    if (interface == 0 && setting == 0)
        return REQUEST_SUCCESS;
    else if (interface == 1 && setting <= 1) {
        interface1 = setting;
        refresh = 1;
        return REQUEST_SUCCESS;
    }
    else
        return REQUEST_ERROR;
}
```

Do obsługi żądań specyficznych dla klasy audio potrzebne są dwa statyczne bufore: buffer8 dla danych 8-bitowych i buffer16 dla 16-bitowych.

```
static uint8_t buffer8;
static uint16_t buffer16;
```

Kolejne trzy omawiane funkcje obsługują żądania specyficzne dla klasy audio. Funkcja zwrotna `ClassInDataSetup` jest wywoływana dla obsługi żądań, w których kontroler USB chce otrzymać jakieś dane. Są to po kolei: aktualne ustawienie wyliczania, aktualne ustawienie głośności, minimalna głośność, maksymalna głośność, rozdzielcość regulacji głośności.

```

usb_result_t ClassInDataSetup(usb_setup_packet_t const *setup,
                           uint8_t const **data,
                           uint16_t *length) {
    if (setup->bmRequestType == (DEVICE_TO_HOST |
                                  CLASS_REQUEST |
                                  INTERFACE_RECIPIENT)) {
        if (setup->bRequest == GET_CUR &&
            setup->wValue == MUTE_CONTROL << 8 &&
            setup->wIndex == 0x0200 &&
            setup->wLength == 1) {
            buffer8 = DACgetMute();
            *data = &buffer8;
            *length = 1;
            return REQUEST_SUCCESS;
        }
        else if (setup->bRequest == GET_CUR &&
                  setup->wValue == VOLUME_CONTROL << 8 &&
                  setup->wIndex == 0x0200 &&
                  setup->wLength == 2) {
            buffer16 = HTOUSBS(DACgetVolume());
            *data = (uint8_t const *)&buffer16;
            *length = 2;
            return REQUEST_SUCCESS;
        }
        else if (setup->bRequest == GET_MIN &&
                  setup->wValue == VOLUME_CONTROL << 8 &&
                  setup->wIndex == 0x0200 &&
                  setup->wLength == 2) {
            buffer16 = HTOUSBS(DACgetVolumeMin());
            *data = (uint8_t const *)&buffer16;
            *length = 2;
            return REQUEST_SUCCESS;
        }
        else if (setup->bRequest == GET_MAX &&
                  setup->wValue == VOLUME_CONTROL << 8 &&
                  setup->wIndex == 0x0200 &&

```

```

        setup->wLength == 2) {
    buffer16 = HTOUSBS(DACgetVolumeMax());
    *data = (uint8_t const *)&buffer16;
    *length = 2;
    return REQUEST_SUCCESS;
}
else if (setup->bRequest == GET_RES &&
         setup->wValue == VOLUME_CONTROL << 8 &&
         setup->wIndex == 0x0200 &&
         setup->wLength == 2) {
    buffer16 = HTOUSBS(DACgetResolution());
    *data = (uint8_t const *)&buffer16;
    *length = 2;
    return REQUEST_SUCCESS;
}
return REQUEST_ERROR;
}

```

Funkcja zwrotna ClassOutDataSetup jest wywoływana dla obsługi żądań, w których kontroler USB chce wysłać jakieś dane do urządzenia. Są to po kolei: aktywacja bądź dezaktywacja wyciszania, ustawienie głośności, ustawienie rozdzielczości regulacji głośności. Głównym zadaniem tej funkcji jest przekazanie za pomocą parametru data wskaźnika na statyczny bufor, w którym mają być umieszczone wysyłane przez kontroler dane.

```

usb_result_t ClassOutDataSetup(usb_setup_packet_t const *setup,
                               uint8_t **data) {
if (setup->bmRequestType == (HOST_TO_DEVICE |
                                CLASS_REQUEST |
                                INTERFACE_RECIPIENT)) {
    if (setup->bRequest == SET_CUR &&
        setup->wValue == MUTE_CONTROL << 8 &&
        setup->wIndex == 0x0200 &&
        setup->wLength == 1) {
        *data = &buffer8;
        return REQUEST_SUCCESS;
    }
    else if (setup->bRequest == SET_CUR &&
             setup->wValue == VOLUME_CONTROL << 8 &&
             setup->wIndex == 0x0200 &&
             setup->wLength == 2) {
        *data = (uint8_t *)&buffer16;
        return REQUEST_SUCCESS;
    }
}
}

```

```

    }

    else if (setup->bRequest == SET_RES &&
              setup->wValue == VOLUME_CONTROL << 8 &&
              setup->wIndex == 0x0200 &&
              setup->wLength == 2) {
        *data = (uint8_t *)&buffer16;
        return REQUEST_SUCCESS;
    }
}

return REQUEST_ERROR;
}

```

Funkcja zwrotna ClassStatusIn jest wywoływana po zakończeniu obsługi żądania, w którym kontroler USB chce wysłać jakieś dane do urządzenia, czyli gdy dane te zostały już przesłane i można z nich korzystać. Obsługiwane muszą być dokładnie te same żądania, co w funkcji ClassOutDataSetup. Dane odczytuje się ze statycznego bufora przekazanego przez tę funkcję.

```

void ClassStatusIn(usb_setup_packet_t const *setup) {
    if (setup->bmRequestType == (HOST_TO_DEVICE |
                                    CLASS_REQUEST |
                                    INTERFACE_RECIPIENT)) {
        if (setup->bRequest == SET_CUR &&
            setup->wValue == MUTE_CONTROL << 8 &&
            setup->wIndex == 0x0200 &&
            setup->wLength == 1) {
            DACsetMute(buffer8);
            refresh = 1;
        }
        else if (setup->bRequest == SET_CUR &&
                  setup->wValue == VOLUME_CONTROL << 8 &&
                  setup->wIndex == 0x0200 &&
                  setup->wLength == 2) {
            DACsetVolume(USBTOHS(buffer16));
            refresh = 1;
        }
        else if (setup->bRequest == SET_RES &&
                  setup->wValue == VOLUME_CONTROL << 8 &&
                  setup->wIndex == 0x0200 &&
                  setup->wLength == 2) {
            DACsetResolution(USBTOHS(buffer16));
            refresh = 1;
        }
    }
}

```

Funkcja zwrotna EP1OUT jest wywoływana po odebraniu każdego pakietu z danymi. Najpierw kopujemy te dane do lokalnego bufora buffer. Następnie w pętli for za pomocą makra USBTOHS zmieniamy porządek bajtów na obowiązujący lokalnie. Potem skonwertowane próbki kopujemy do bufora przetwornika cyfrowo-analogowego. Na koniec modyfikujemy zmienne odpowiadające za wyświetlanie informacji o postępie transmisji. Pewien niepokój powinno wzbudzić dwukrotne kopiowanie danych. Można by tego uniknąć kosztem skomplikowania tekstu źródłowego i zmniejszenia separacji między poszczególnymi modułami programu – aplikacja musiałaby znać strukturę wewnętrznych buforów USB i bufora przetwornika oraz mieć do nich bezpośredni dostęp. Wewnętrzna struktura tych buforów jest ukrywana odpowiednio przez funkcje USBDread i DACputSamples.

```
void EP1OUT() {  
    int16_t buffer[(ISO_BUFF_SIZE + 1) >> 1];  
    uint32_t received, i;  
  
    received = USBDread(ENDP1, (uint8_t *)buffer, ISO_BUFF_SIZE);  
    received >>= 1;  
    for (i = 0; i < received; ++i)  
        buffer[i] = USBTOHS(buffer[i]);  
    DACputSamples(buffer, received);  
    ++progress;  
    if ((progress & ((1U << LCD_REFRESH) - 1)) == 0)  
        refresh = 1;  
}
```

Na zakończenie jeszcze mała dygresja. Porządek bajtów stosowany w USB jest identyczny z porządkiem bajtów w architekturze ARM Cortex-M. Zatem makrodefinicja USBTOHS jest pusta i pętla for zostanie zoptymalizowana (usunięta) przez kompilator.

3.3.6. Kompilowanie i testowanie

Archiwum z przykładami zawiera kilka wersji projektu odtwarzacza audio. W katalogu *./make* znajdują się podkatalogi, których nazwy rozpoczynają się przedrostkiem *usb3_audio_device*. W tych podkatalogach umieszczone są pliki *makefile* umożliwiające skompilowanie tego projektu dla przykładowych wariantów sprzętu. Można je oczywiście łatwo dostosować do innego sprzętu, posługując się wskazówkami z rozdziału 2. Pliki *makefile* są również przydatne dla tych, którzy nie chcą korzystać bezpośrednio z programu *make*, gdyż zawierają listę plików źródłowych niezbędnych do skompilowania programu. Należy przy tym pamiętać, aby dołączyć plik *startup_stm32.c* i właściwą wersję biblioteki STM32.

Urządzenie po podłączeniu do gniazda USB jest rozpoznawane jako głośnik, który może być używany przez dowolny program odtwarzający dźwięk. Na ekranie komputera pojawia się ikona głośnika. Jej kliknięcie otwiera panel z kontrolkami regulacji głośności i wyciszania. Jeśli komputer wyposażony jest w kartę dźwiękową,

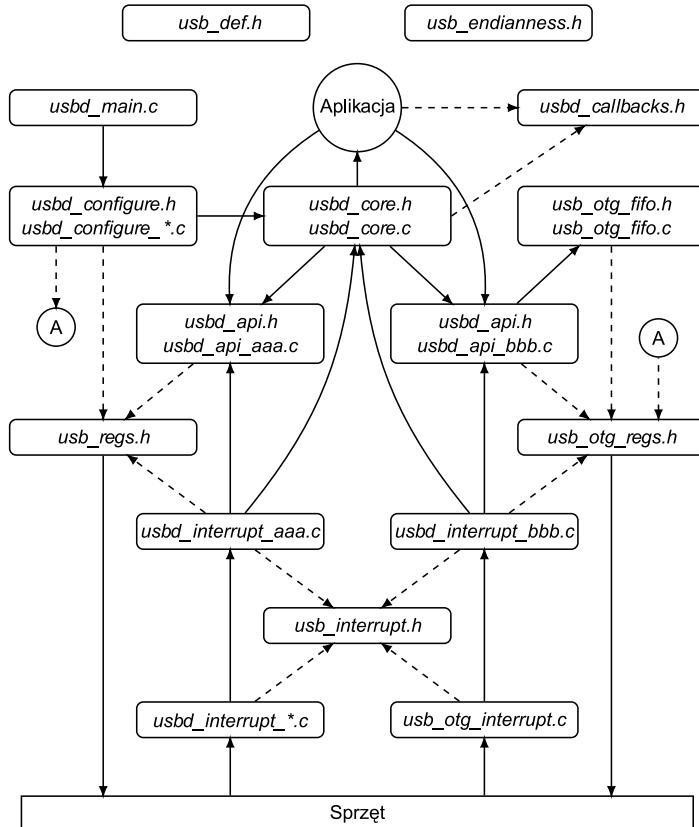
a tak zwykle jest, to najprawdopodobniej jest ona skonfigurowana jako domyślne urządzenie odtwarzające dźwięk. Wtedy w aplikacji miksera (zależnej od wersji systemu operacyjnego) można zmienić to ustawienie. Dodatkowo, jeśli płytka prototypowa wyposażona jest w LCD, wyświetlane są na nim informacje diagnostyczne: poziom głośności w decybelach, informacja o aktywacji wyciszania, informacja o aktywnej transmisji danych.

W celu uproszczenia korzystania z USB i uniezależnienia warstwy aplikacji od warstwy sprzętu stworzyłem prostą bibliotekę. Bazując na niej wszystkie prezentowane w książce przykłady. W zamyśle jest ona przeznaczona na mikrokontrolery STM32, ale starałem się ją napisać bardzo ogólnie z wyraźnym podziałem na moduły, aby jej przeniesienie na inny sprzęt nie było trudne. Biblioteka ta nie jest dziełem skończonym i ma pewne ograniczenia, na przykład może obsługiwać jednocześnie tylko jeden interfejs USB, ale dzięki modułowej strukturze może być podstawą, punktem wyjścia do opracowania własnego, bardziej rozbudowanego oprogramowania. W pierwszym podrozdziale tego rozdziału przedstawiam pierwszą część tej biblioteki, dotyczącą urządzeń USB. Omawiam też pliki wspólne dla implementacji urządzenia i kontrolera. Drugą część biblioteki, dotyczącą tylko kontrolera USB, omawiam w rozdziale 7, który jest w całości poświęcony właśnie przykładowej implementacji kontrolera. Drugi podrozdział traktuje o bibliotece `libusb`, która ułatwia napisanie aplikacji komunikującej się z urządzeniem USB. W ostatnim podrozdziale przedstawiam projekt korzystający z obu bibliotek i pokazujący, jak zaimplementować urządzenie USB własnej klasy (ang. *vendor specific*). Przykład ten demonstruje przesyłanie danych sterujących, pilnych, masowych i izochronicznych, w obu kierunkach: kontroler-urządzenie i urządzenie-kontroler.

4.1. Biblioteka urządzenia USB dla STM32

Nazwy wszystkich plików należących do biblioteki rozpoczynają się przedrostkiem `usb`. Pliki, których nazwa rozpoczyna się od `usbd`, dotyczą tylko urządzenia USB. Natomiast pliki, których nazwa rozpoczyna się przedrostkiem `usbh`, dotyczą tylko kontrolera USB i ich omówienie odkładam do rozdziału 7. W tym podrozdziale opisuję dość szczegółowo interfejs programistyczny (ang. *API*) biblioteki. Przedstawiam też wybrane kluczowe fragmenty jej implementacji. Zamieszczenie pełnych wydruków tekstu źródłowego biblioteki nie jest możliwe, głównie z braku miejsca i nie miałoby większego sensu. Jak łatwo jest się domyślić, implementacja biblioteki w wielu miejscach po prostu odwołuje się, bezpośrednio lub pośrednio poprzez bibliotekę STM32, do rejestrów konfiguracyjnych mikrokontrolera. Czytelnika zainteresowanego szczegółami technicznymi proszę o zajrzenie do plików źródłowych. Jest tam trochę komentarzy, a nazwy zmiennych i funkcji są samoopisujące.

Zależności między plikami biblioteki zaprezentowane są na **rysunku 4.1**. Strzałka narysowana linią przerywaną oznacza włączanie pliku dyrektywą `#include` przez plik z implementacją (z rozszerzeniem *c*). Strzałka narysowana linią ciągłą oznacza wywoływanie funkcji lub bezpośrednie odwołania do rejestrów mikrokontrolera. Dla uproszczenia rysunku część plików nagłówkowych (z rozszerzeniem *h*) wymieniona jest łącznie z odpowiadającymi im plikami z implementacją (z rozszerzeniem *c*). Oznacza to, że ten plik z implementacją oraz wszystkie pliki wywołujące zaimplementowane w nim funkcje włączają podany plik nagłówkowy. Dwa pliki nagłówkowe umieszczone na samej górze rysunku zawierają definicje potrzebne w wielu miejscach biblioteki i aplikacji. Pliki te są bezpośrednio lub pośrednio włączane przez większość pozostałych plików, co nie zostało zaznaczone, aby nie zaciemniać



Rys. 4.1. Struktura biblioteki urządzenia USB

rysunku. Gwiazdka w nazwie pliku oznacza, że dany plik występuje w kilku wersjach specyficznych dla poszczególnych modeli mikrokontrolerów STM32.

Funkcje biblioteki nie są współużywalne (ang. *reentrant*). W omawianych projektach unikamy tego problemu, gdyż konfigurowanie urządzenia odbywa się przy wyłączonych przerwaniach, które są włączane dopiero w momencie, gdy mogą już być bezpiecznie obsłużone. Wszystkie pozostałe wywołania biblioteki następują w kontekście procedury obsługi przerwania USB. Jeśli wystąpi przerwanie o wyższym priorytecie wywłaszczenia, to w procedurze jego obsługi nie są wywoływanie żadne funkcje biblioteki.

4.1.1. Makra, stałe i struktury danych

usb_def.h

Plik `usb_def.h` zawiera definicje, które są niezbędne niemal we wszystkich plikach źródłowych implementujących jakieś aspekty USB. Są to przede wszystkim definicje struktur i stałych opisanych w standardzie USB, ale też wszelkie globalne parametry implementacji.

usb_endianness.h

Plik *usb_endianness.h* zawiera cztery makrodefinicje zamieniające porządek bajtów:

- HTOUSBS(x) zamienia wartość dwubajtową z kolejności obowiązującej lokalnie na kolejność obowiązującą w USB;
- USBTOHS(x) zamienia wartość dwubajtową z kolejności obowiązującej w USB na kolejność obowiązującą lokalnie;
- HTOUSBL(x) zamienia wartość czterobajtową z kolejności obowiązującej lokalnie na kolejność obowiązującą w USB;
- USBTOHL(x) zamienia wartość czterobajtową z kolejności obowiązującej w USB na kolejność obowiązującą lokalnie.

Porządek bajtów stosowany w USB jest identyczny z porządkiem bajtów w architekturze ARM Cortex-M – jest to porządek cienkokońcowkowy (ang. *little-endian*). Zatem wyżej wymienione makrodefinicje są puste i zostaną zoptymalizowane (usunięte) przez kompilator. Architektura ARM nie jest jednak na sztywno przywiązana do porządku cienkokońcowkowego. Nie jest wykluczone, że ktoś kiedyś zechce zastosować rdzeń z porządkiem grubokońcowkowym (ang. *big-endian*). Nie powinno się więc zapominać o problemie kolejności bajtów i należy konsekwentnie używać makr dopasowujących ten porządek. Dzięki temu tekst źródłowy będzie można łatwo przenieść na dowolną architekturę, nie tylko ARM. W razie potrzeby omawiane makra można zdefiniować jako funkcje rozwijane w miejscu wywoływania.

4.1.2. Konfigurowanie urządzenia

usbd_configure.h usbd_configure_103.c usbd_configure_107.c usbd_configure_152.c usbd_configure_207.c

W pliku *usbd_configure.h* zadeklarowane są funkcje konfigurujące układ peryferyjny USB w celu użycia go jako urządzenia USB. Sygnatury tych funkcji są niezależne od wykorzystywanego sprzętu. Natomiast ich implementacja zależy od modelu mikrokontrolera. Biblioteka zawiera cztery implementacje. Plik *usbd_configure_103.c* przeznaczony jest dla STM32F102 i STM32F103, plik *usbd_configure_107.c* dla STM32F105 i STM32F107, plik *usbd_configure_152.c* dla STM32Lxxx, a plik *usbd_configure_207.c* dla STM32F2xx i STM32F4xx. Żeby skonfigurować układ peryferyjny USB, trzeba wywołać dwie funkcje. Taki podział powodowany jest tym, że specyfikacja USB wymaga dość krótkiego czasu między rozpoznaniem podłączenia urządzenia do szyny a uzyskaniem przez niego gotowości do działania. Może się jednak okazać, że przed uruchomieniem interfejsu USB trzeba skonfigurować wiele innych peryferii, co zajmuje sporo czasu.

```
int USBDpreConfigure(usb_speed_t speed, usb_phy_t phy);
```

Funkcja `USBDpreConfigure` musi być wywołana możliwie szybko po włączeniu zasilania. Jej głównym zadaniem jest skonfigurowanie rezystora podciągającego i pozostawienie go w stanie odłączonym, aby kontroler nie wykrył podłączenia urządzenia. Niektóre wersje sprzętu zapewniają, że rezystor podciągający jest domyślnie odłączony. Wtedy funkcja `USBDpreConfigure` nie jest potrzebna, ale pozostawiłem ją, aby zachować jednolity interfejs programistyczny biblioteki dla wszystkich wersji sprzętu. Parametr `speed` określa, z jaką szybkością ma działać interfejs USB. Może on przyjmować wartości: `LOW_SPEED`, `FULL_SPEED`, `HIGH_SPEED`. Należy jednak pamiętać, że konkretne układy peryferyjne USB nie pozwalają na pracę ze wszystkimi szybkościami. Parametr `phy` określa, jaki nadajnik-odbiornik ma zostać użyty. Dla STM32 może on przyjmować wartości:

- `USB_PHY_A` – wbudowany nadajnik-odbiornik FS z wyprowadzeniami na porcie PA,
- `USB_PHY_B` – wbudowany nadajnik-odbiornik FS z wyprowadzeniami na porcie PB,
- `USB_PHY_ULPI` – zewnętrzny nadajnik-odbiornik HS podłączony za pomocą interfejsu ULPI,
- `USB_PHY_I2C` – zewnętrzny nadajnik-odbiornik FS podłączony za pomocą interfejsu I²C.

Należy pamiętać, że konkretne układy peryferyjne USB współpracują tylko z wybranymi nadajnikami-odbiornikami. Funkcja `USBDpreConfigure` zwraca zero, gdy konfigurowanie się powiodło, a wartość ujemną, gdy wystąpił błąd, na przykład podano kombinację parametrów niedopuszczalną dla zastosowanego wariantu sprzętu.

```
int USBDconfigure(unsigned prio, unsigned subprio, int clk);
```

Funkcja `USBDconfigure` realizuje zasadniczy etap konfigurowania układu peryferyjnego USB, kończący się włączeniem rezystora podciągającego, co spowoduje wykrycie urządzenia przez kontroler. Parametry `prio` i `subprio` określają odpowiednio priorytet wywieszania i podpriorytet, z jakimi ma być zgłasiane przerwanie USB. Priorytety te muszą być starannie dobrane, gdyż wszystkie funkcje zwrotne, opisane poniżej, są wywoływanie w procedurze obsługi tego przerwania. Parametr `clk` określa częstotliwość taktowania rdzenia mikrokontrolera w megahercach. Parametr ten jest niezbędny do prawidłowego dobrania ustawień układu peryferyjnego USB. Choć w przypadku STM32 wartość tej częstotliwości można wywnioskować z ustawień rejestrów konfiguracyjnych mikrokontrolera, to jest ona podawana jawnie do funkcji konfigurującej, aby interfejs programistyczny biblioteki nie był zależny od wersji zastosowanego sprzętu. Konkretna implementacja może po prostu ignorować wartość tego parametru i odczytywać częstotliwość taktowania z rejestrów konfiguracyjnych. Funkcja `USBDconfigure` zwraca zero, gdy konfigurowanie się powiodło, a wartość ujemną, gdy wystąpił błąd.

4.1.3. Interfejs programistyczny

`usbd_callbacks.h`

W pliku `usbd_callbacks.h` zdefiniowano strukturę typu `usbd_callback_list_t` przechowującą listę wskaźników na funkcje zwrotne, które są wywoływanie w celu powiadomienia aplikacji o zdarzeniach zachodzących na szynie USB. Źródłem zdarzeń są przede wszystkim transakcje i żądania inicjowane przez kontroler. W aplikacji należy zdefiniować odpowiednie funkcje zwrotne, zadeklarować wspomnianą strukturę i wypełnić ją wskaźnikami na te funkcje. Jeśli nie potrzebujemy jakiejś funkcji zwrotnej, gdyż nie chcemy obsługiwać związanego z nią zdarzenia, to nie musimy tej funkcji definiować – wystarczy w odpowiednim miejscu struktury wpisać zero. Funkcje zwrotne nie muszą być bezpośrednio widoczne na zewnątrz jednostki translacji i mogą być zadeklarowane z atrybutem `static`.

```
typedef struct {
    int          (*Configure) (void);
    uint8_t      (*Reset) (usb_speed_t speed);
    void         (*SoF) (uint16_t frameNumber);
    usb_result_t (*GetDescriptor) (uint16_t wValue, uint16_t wIndex,
                                  uint8_t const **data,
                                  uint16_t *length);
    usb_result_t (*SetDescriptor) (uint16_t wValue, uint16_t wIndex,
                                  uint16_t wLength, uint8_t **data);
    uint8_t      (*GetConfiguration) (void);
    usb_result_t (*SetConfiguration) (uint16_t configurationValue);
    uint16_t      (*GetStatus) (void);
    usb_result_t (*GetInterface) (uint16_t interface, uint8_t *setting);
    usb_result_t (*SetInterface) (uint16_t interface, uint16_t setting);
    usb_result_t (*ClearDeviceFeature) (uint16_t featureSelector);
    usb_result_t (*SetDeviceFeature) (uint16_t featureSelector);
    usb_result_t (*ClassNoDataSetup) (usb_setup_packet_t const *setup);
    usb_result_t (*ClassInDataSetup) (usb_setup_packet_t const *setup,
                                    uint8_t const **data,
                                    uint16_t *length);
    usb_result_t (*ClassOutDataSetup) (usb_setup_packet_t const *setup,
                                    uint8_t **data);
    void         (*ClassStatusIn) (usb_setup_packet_t const *setup);
    void         (*EPin[15]) (void);
    void         (*EPout[15]) (void);
    void         (*Suspend) (void);
    void         (*Wakeup) (void);
} usbd_callback_list_t;
```


Poniżej omawiam szczegółowo wszystkie funkcje zwrotne. Kolejność bajtów w parametrach tych funkcji jest w porządku lokalnym – nie trzeba do nich stosować makra USBTOHS.

```
int Configure(void);
```

Bezparametrowa funkcja `Configure` jest wywoływana tylko raz podczas uruchamiania układu. Jej zadaniem jest skonfigurowanie sprzętu specyficznego dla aplikacji. Funkcja ta powinna zwrócić zero, gdy zakończyła się powodzeniem, a wartość ujemną, gdy wystąpił jakiś błąd.

```
uint8_t Reset(usb_speed_t speed);
```

Funkcja `Reset` jest wywoływana za każdym razem, gdy kontroler wykona procedurę zerowania szyny USB. Parametr `speed` określa szybkość, z jaką ma pracować urządzenie i może przyjmować wartości: `LOW_SPEED`, `FULL_SPEED`, `HIGH_SPEED`. W funkcji tej należy skonfigurować dwukierunkowy zerowy punkt końcowy, który jest domyślnym punktem końcowym dla danych sterujących. Funkcja ta powinna zwrócić wartość z pola `bMaxPacketSize0` deskryptora urządzenia, czyli maksymalny rozmiar pola danych dla pakietów sterujących. Z różnych przyczyn (szybko następujące po sobie podłączenia i rozłączenia złącza USB, permanentne zgłaszań przez sprzęt przerwania aż do zakończenia trwania sygnału zerującego) funkcja ta może zostać wywołana więcej niż jeden raz. Dlatego musi być idempotentna – wielokrotne wywoływanie ma dokładnie taki sam efekt jak wywołanie jednorazowe.

```
void SoF(uint16_t frameNumber);
```

Funkcja `SoF` jest wywoływana na początku każdej ramki lub mikroramki. Parametr `frameNumber` zawiera numer ramki przesłany w pakiecie SOF. Funkcja ta nie zwraca żadnej wartości.

```
usb_result_t GetDescriptor(uint16_t wValue, uint16_t wIndex,
                           uint8_t const **data, uint16_t *length);
```

Funkcja `GetDescriptor` jest wywoływana, gdy zostanie odebrane żądanie `GET_DESCRIPTOR`. Pierwsze dwa parametry `wValue` i `wIndex` to wartości ze struktury `setup` przesłanej w fazie ustanowienia żądania. Identyfikują one deskryptor, którego żąda kontroler – patrz rozdział 1. Za pomocą parametru `data` zwraca się wskaźnik na deskryptor do przesłania, a za pomocą parametru `length` przekazuje się rozmiar deskryptora. Przekazany wskaźnik musi być ważny również po zakończeniu wykonywania tej funkcji. Musi zatem wskazywać na strukturę, która jest zadeklarowana jako statyczna lub globalna. Funkcja ta powinna zwrócić wartość `REQUEST_SUCCESS`, gdy żądanie jest poprawne i zostało zrealizowane, a `REQUEST_ERROR`, gdy kontroler zażądał deskryptora, który nie istnieje.

```
usb_result_t SetDescriptor(uint16_t wValue, uint16_t wIndex,  
                           uint16_t wLength, uint8_t **data);
```

Funkcja `SetDescriptor` jest wywoływaną, gdy zostanie odebrane żądanie `SET_DESCRIPTOR`. Początkowe trzy parametry `wValue`, `wIndex` i `wLength` to wartości ze struktury `setup` przesłanej w fazie ustanowienia żądania. Identyfikują one wysyłany przez kontroler deskryptor oraz jego rozmiar. Za pomocą parametru `data` zwraca się wskaźnik na bufor w pamięci, gdzie ma zostać skopiowany deskryptor po jego przesłaniu. Bufor musi być globalny lub statyczny, gdyż przekazany wskaźnik musi być ważny również po zakończeniu wykonywania tej funkcji. Bufor powinien też mieć dostateczny rozmiar, aby zmieścić cały deskryptor, czyli co najmniej `wLength` bajtów. Funkcja ta powinna zwrócić wartość `REQUEST_SUCCESS`, gdy żądanie jest poprawne i zostanie zrealizowane, a `REQUEST_ERROR`, gdy żądanie ma zostać odrzucone.

```
uint8_t GetConfiguration(void);
```

Bezparametrowa funkcja `GetConfiguration` jest wywoływaną po odebraniu żądania `GET_CONFIGURATION`. Funkcja ta zwraca numer aktualnie ustawionej konfiguracji, a zero, gdy żadna konfiguracja nie została wybrana lub konfiguracja została dezaktywowana.

```
usb_result_t SetConfiguration(uint16_t configurationValue);
```

Funkcja `SetConfiguration` jest wywoływaną po odebraniu żądania `SET_CONFIGURATION`. Parametr `configurationValue` jest numerem konfiguracji, która ma być ustawiona. Jeśli parametr ten ma wartość dodatnią i istnieje konfiguracja o wskazanym numerze, to należy aktywować tę konfigurację i skonfigurować wszystkie punkt końcowe z nią związane. Jeśli parametr ma wartość zero, to należy dezaktywować aktualnie ustawioną konfigurację i wszystkie związane z nią punkty końcowe. Jeśli parametr ma wartość dodatnią, ale nie istnieje konfiguracja o wskazanym numerze, to żądanie jest niepoprawne i powinno zostać odrzucone. Funkcja ta powinna zwrócić wartość `REQUEST_SUCCESS`, gdy parametr ma poprawną wartość i żądanie zostanie zrealizowane, a `REQUEST_ERROR`, gdy żądanie ma zostać odrzucone lub wystąpił błąd podczas aktywowania konfiguracji, na przykład podczas konfigurowania jej punktów końcowych.

```
uint16_t GetStatus(void);
```

Bezparametrowa funkcja `GetStatus` jest wywoływaną po odebraniu żądania `GET_STATUS` kierowanego do urządzenia. Powinna ona zwrócić 16-bitowy status urządzenia informujący o jego aktualnym źródle zasilania. Status ma ustawiony bit 0, gdy urządzenie w danym momencie nie pobiera prądu z linii VBUS i jest zasilane autonomiczne. Gdy urządzenie jest całkowicie lub częściowo zasilane z szyny VBUS (pobiera z niej prąd), bit 0 statusu urządzenia musi być wyzerowany. Bit 1 statusu

informuje, czy aktywna jest funkcja zdalnego budzenia kontrolera. Pozostałe bity statusu są zarezerwowane i powinny być wyzerowane.

```
usb_result_t GetInterface(uint16_t interface, uint8_t *setting);
```

Funkcja GetInterface jest wywoływana, gdy kontroler prosi o przesłanie wybranego wariantu ustawień interfejsu za pomocą żądania GET_INTERFACE. Parametr interface zawiera numer interfejsu, którego dotyczy żądanie. Parametr setting jest wskaźnikiem na zmienną, do której trzeba przypisać żądany wariant ustawień. Funkcja ta powinna zwrócić wartość REQUEST_SUCCESS, gdy parametry mają poprawne wartości i żądanie zostanie zrealizowane, a REQUEST_ERROR, gdy nie istnieje interfejs o podanym numerze i żądanie ma zostać odrzucone.

```
usb_result_t SetInterface(uint16_t interface, uint16_t setting);
```

Funkcja SetInterface jest wywoływana, gdy kontroler prosi o aktywowanie wariantu ustawień interfejsu za pomocą żądania SET_INTERFACE. Parametr interface zawiera numer interfejsu, którego dotyczy żądanie. Parametr setting zawiera numer wariantu, której ma być ustawiony. Funkcja ta powinna zwrócić wartość REQUEST_SUCCESS, gdy parametry mają poprawne wartości i żądanie zostanie zrealizowane, a REQUEST_ERROR, gdy żądanie ma zostać odrzucone, ponieważ nie istnieje interfejs o podanym numerze lub interfejs nie ma podanego wariantu ustawień.

```
usb_result_t ClearDeviceFeature(uint16_t featureSelector);
```

Funkcja ClearDeviceFeature jest wywoływana po odebraniu żądania CLEAR_FEATURE kierowanego do urządzenia. Parametr featureSelector może mieć tylko wartość 1, co zgodnie z tabelą 1.16 oznacza wyłączenie zdalnego budzenia kontrolera. Funkcja ta powinna zwrócić wartość REQUEST_SUCCESS, gdy parametr ma poprawną wartość i żądanie zostanie zrealizowane, a REQUEST_ERROR, gdy żądanie ma zostać odrzucone.

```
usb_result_t SetDeviceFeature(uint16_t featureSelector);
```

Funkcja SetDeviceFeature jest wywoywana po odebraniu żądania SET_FEATURE kierowanego do urządzenia. Parametr featureSelector przyjmuje tylko wartość 1, co zgodnie z tabelą 1.16 oznacza włączenie zdalnego budzenia kontrolera. Funkcja ta powinna zwrócić wartość REQUEST_SUCCESS, gdy parametr ma poprawną wartość i żądanie zostanie zrealizowane, a REQUEST_ERROR, gdy żądanie ma zostać odrzucone.

```
usb_result_t ClassNoDataSetup(usb_setup_packet_t const *setup);
```

Funkcja ClassNoDataSetup obsługuje niestandardowe żądania (specyficzne dla konkretnej klasy lub konkretnego dostawcy) i niewymagające przesyłania danych. Parametr setup jest wskaźnikiem do struktury identyfikującej żądanie i przesłanej w fazie jego ustanowienia w transakcji SETUP. Na podstawie zawartości tej struk-

tury funkcja powinna zdekodować żądanie i jego parametry. Funkcja ta powinna zwrócić wartość REQUEST_SUCCESS, gdy żądanie jest poprawne i zostanie zrealizowane, a REQUEST_ERROR, gdy wystąpił błąd lub żądanie nie jest obsługiwane.

```
usb_result_t ClassInDataSetup(usb_setup_packet_t const *setup,  
                             uint8_t const **data, uint16_t *length);
```

Funkcja ClassInDataSetup obsługuje niestandardowe żądania (specyficzne dla konkretnej klasy lub konkretnego dostawcy) i wymagające przesyłania danych z urządzenia do kontrolera. Parametr setup jest wskaźnikiem do struktury identyfikującej żądanie i przesłanej w jego fazie ustanowienia, w transakcji SETUP. Na podstawie zawartości tej struktury funkcja powinna zdekodować żądanie i jego parametry. Za pomocą parametru data zwraca się wskaźnik na dane do przesyłania, a za pomocą parametru length przekazuje się ich rozmiar. Przekazany wskaźnik powinien wskazywać na dane statyczne lub globalne, gdyż musi być ważny również po zakończeniu funkcji aż do zakończenia fazy transmisji danych. Funkcja ta powinna zwrócić wartość REQUEST_SUCCESS, gdy żądanie jest poprawne i zostanie zrealizowane, a REQUEST_ERROR, gdy wystąpił błąd lub żądanie nie jest obsługiwane.

```
usb_result_t ClassOutDataSetup(usb_setup_packet_t const *setup,  
                             uint8_t **data);
```

Funkcja ClassOutDataSetup obsługuje niestandardowe żądania (specyficzne dla konkretnej klasy lub konkretnego dostawcy) i wymagające przesyłania danych z kontrolera do urządzenia. Parametr setup jest wskaźnikiem do struktury identyfikującej żądanie i przesłanej w jego fazie ustanowienia, w transakcji SETUP. Na podstawie zawartości tej struktury funkcja powinna zdekodować żądanie i jego parametry. Parametr data służy do przekazania wskaźnika do statycznego bufora, do którego mają być skopiowane dane wysłane przez kontroler. Jednak z bufora tego będzie można czytać dopiero po zakończeniu obsługi żądania, o czym zostaniemy poinformowani za pomocą funkcji ClassStatusIn. Funkcja ClassOutDataSetup powinna zwrócić wartość REQUEST_SUCCESS, gdy żądanie jest poprawne i zostanie zrealizowane, a REQUEST_ERROR, gdy wystąpił błąd lub żądanie nie jest obsługiwane.

```
void ClassStatusIn(usb_setup_packet_t const *setup);
```

Funkcja ClassStatusIn informuje o zakończeniu obsługi żądania, w którym zostały przesłane dane z kontrolera do urządzenia. Jest ona wywoływana po zakończeniu fazy statusu tego żądania. Oznacza to, że dane zostały odebrane i znajdują się w buforze, do którego wskaźnik przekazaliśmy za pomocą funkcji ClassOutDataSetup. Parametr setup jest wskaźnikiem do struktury identyfikującej żądanie, które zostało zrealizowane. Funkcja ta nie zwraca żadnej wartości.

```
void (*EPin[15])(void);
```

Element EPin[i] jest wskaźnikiem do bezparametrowej funkcji, która jest wywoływana, gdy zakończy się wysyłanie danych przez punkt końcowy i+1. Funkcja ta

nie zwraca żadnej wartości. Dane do wysłania przekazuje się za pomocą opisanej poniżej funkcji `USBWrite`.

```
void (*EPout[15])(void);
```

Element `EPout[i]` jest wskaźnikiem do bezparametrowej funkcji, która jest wywoływana, gdy zostaną odebrane dane do punktu końcowego `i+1`. Funkcja ta nie zwraca żadnej wartości. Odebrane dane odczytuje się za pomocą opisanej poniżej funkcji `USBRead`.

```
void Suspend(void);
```

Bezparametrowa funkcja `Suspend` jest wywoływana, gdy urządzenie przechodzi w stan wstrzymania i powinno ograniczyć pobór prądu z linii VBUS, ale jeszcze zanim mikrokontroler przejdzie w stan uśpienia. W funkcji tej należy zaimplementować ograniczenie poboru prądu przez układy peryferyjne. Funkcja ta nie zwraca żadnej wartości.

```
void Wakeup(void);
```

Bezparametrowa funkcja `Wakeup` jest wywoywana, gdy urządzenie wychodzi ze stanu wstrzymania, po obudzeniu mikrokontrolera ze stanu uśpienia. W funkcji tej można przywrócić nominalny pobór prądu przez układy peryferyjne i nominalne warunki pracy urządzenia. Funkcja ta nie zwraca żadnej wartości.

```
usbd_callback_list_t const * USBDgetApplicationCallbacks(void);
```

Bezparametrowa funkcja `USBDgetApplicationCallbacks` zwraca wskaźnik do struktury przechowującej wskaźniki do zaimplementowanych funkcji zwrotnych. Jest to jedyna funkcja, którą musimy wyeksportować, gdyż musi być widoczna na zewnątrz modułu.

<code>usbd_api.h</code>
<code>usbd_api_aaa.c</code>
<code>usbd_api_bbb.c</code>

W pliku `usbd_api.h` znajdują się deklaracje funkcji tworzących niskopoziomowy interfejs programistyczny urządzenia USB. Realizacja tych funkcji bardzo zależy od sprzętu. Dostępne są dwie implementacje. W pliku `usbd_api_aaa.c` znajduje się implementacja dla układu peryferyjnego DEV-FS, a w pliku `usbd_api_bbb.c` dla układów OTG-FS i OTG-HS (patrz tabela 2.1). Interfejs programistyczny jest podzielony na dwie grupy funkcji, ze względu na miejsce ich wywoływanego: warstwa aplikacji i rdzeń protokołu USB. Zaczniemy od funkcji przeznaczonych do wywoływania w warstwie aplikacji.

```
void USBDisableAllNonControlEndPoints(void);
```

Bezparametrowa funkcja `USBDisableAllNonControlEndPoints` dezaktywuje wszystkie punkty końcowe z wyjątkiem zerowego punktu końcowego, który jest domyślnym punktem końcowym dla danych sterujących. Funkcję tę należy wywoływać wewnątrz funkcji zwrotnej `SetConfiguration`, gdy kontroler zażąda zmiany aktualnej konfiguracji. Funkcja ta nie zwraca żadnej wartości.

```
usb_result_t USBDisableAllNonControlEndPoints(usb_transfer_t type,
```

```
                                              uint16_t rxBufferSize,
```

```
                                              uint16_t txBufferSize);
```

Funkcja `USBEndPointConfigure` konfiguruje punkt końcowy i zasadniczo jest przeznaczona do wywoływanego w funkcjach zwrotnych `Reset` i `SetConfiguration`. Parametr `endPoint` jest numerem (bez bitu kierunku) konfigurowanego punktu końcowego. Parametr `type` określa rodzaj przesyłanych tym punktem końcowym danych i może przyjmować wartości: `CONTROL_TRANSFER`, `ISOCHRONOUS_TRANSFER`, `BULK_TRANSFER`, `INTERRUPT_TRANSFER`. Parametr `rxBufferSize` określa maksymalny rozmiar pola danych pakietu wysyłanego przez kontroler. Innymi słowy, określa on rozmiar pakietu dla wyjściowego punktu końcowego. Parametr `txBufferSize` określa maksymalny rozmiar pola danych pakietu wysyłanego przez urządzenie. Inaczej, określa on rozmiar pakietu dla wejściowego punktu końcowego. Jeśli oba parametry mają niezerową wartość, to zostanie skonfigurowany dwukierunkowy punkt końcowy (dwa jednokierunkowe punkty końcowe). Jeśli tylko parametr `rxBufferSize` ma niezerową wartość, to zostanie skonfigurowany jednokierunkowy wyjściowy punkt końcowy. Jeśli tylko parametr `txBufferSize` ma niezerową wartość, to zostanie skonfigurowany jednokierunkowy wejściowy punkt końcowy. Parametry `rxBufferSize` i `txBufferSize` są potrzebne do skonfigurowania rozmiarów buforów odbiorczych i nadawczych. Wartości tych parametrów muszą pochodzić z pól `wMaxPacketSize` odpowiednich deskryptorów punktów końcowych. Przy czym obowiązują ograniczenia narzucone przez standard USB, zebrańskie w tabeli 1.10. Należy też uwzględnić dodatkowe ograniczenia sprzętu, o których piszę poniżej. Jako pierwszy (w funkcji `Reset`) musi zostać skonfigurowany punkt końcowy zero dla danych sterujących. Ponowne wywołanie konfiguracji tego punktu końcowego nie zmienia – jest to wywołanie idempotentne. Po skonfigurowaniu domyślnego punktu końcowego dla danych sterujących i wybraniu przez kontroler konfiguracji (w funkcji `SetConfiguration`) można konfigurować punkty końcowe opisane w deskryptorach. Te wywołania nie są już idempotentne – każde skutkuje skonfigurowaniem następnego punktu końcowego. Kolejne konfigurowane punkty końcowe muszą mieć niepowtarzające się numery. Funkcja ta zwraca `REQUEST_SUCCESS`, gdy wszystko przebiegło poprawnie, a `REQUEST_ERROR`, gdy wystąpił błąd.

Układ peryferyjny DEV-FS ma 8 fizycznych punktów końcowych. Intencjonalnie użyłem tu przyniemiarka „fizyczny”, aby wyraźnie odróżnić je od (logicznych) punktów końcowych, o których mówi standard USB. Ponadto DEV-FS ma 512 bajtów pamięci PMA (ang. *packet memory area*), w której konfiguruje się bufore nadawcze i odbiorcze punktów końcowych. Jeden fizyczny punkt końcowy może realizować jednokierunkowy lub dwukierunkowy punkt końcowy z pojedynczym buforowa-

niem albo jednokierunkowy punkt końcowy z podwójnym buforowaniem. Numer punktu końcowego nie musi odpowiadać numerowi fizycznego punktu końcowego, który zostanie użyty. Przy konfigurowaniu domyślnego punktu końcowego dla danych sterujących parametr endPoint musi mieć wartość zero (ENDP0), parametr type – wartość CONTROL_TRANSFER, a parametry rxBuffSize i txBuffSize muszą mieć jednakowe wartości zgodne z wartością bMaxPacketSize0 podaną w deskryptorze urządzenia. Dla tego punktu końcowego biblioteka wykorzystuje fizyczny punkt końcowy zero, a w przestrzeni PMA alokuje dwa bufory o podanym rozmiarze: jeden odbiorczy i jeden nadawczy. Przy konfigurowaniu pozostałych punktów końcowych, opisanych za pomocą deskryptorów, parametr endPoint może przyjmować wartość z przedziału 1...15 (ENDP1 do ENDP15). Każde wywołanie omawianej funkcji z parametrem type równym BULK_TRANSFER lub INTERRUPT_TRANSFER alokuje pierwszy wolny fizyczny punkt końcowy i po jednym buforze o rozmiarze podanym w parametrze rxBuffSize i txBuffSize (jeśli któryś parametr ma wartość zero, to odpowiadający mu bufor nie jest alokowany). Wywołanie omawianej funkcji z parametrem type równym ISOCRONOUS_TRANSFER działa nieco inaczej. Dla danych izochronicznych stosowane jest podwójne buforowanie. Jeśli parametr rxBuffSize ma niezerową wartość, to alokowany jest fizyczny punkt końcowy i dwa bufory odbiorcze o podanym rozmiarze. Podobnie, jeśli parametr txBuffSize ma niezerową wartość, to alokowany jest fizyczny punkt końcowy i dwa bufory nadawcze o podanym rozmiarze. Zatem dwukierunkowy punkt końcowy dla danych izochronicznych zajmuje dwa fizyczne punkty końcowe. Podwójne bufory dla izochronicznego punktu końcowego tworzą dwuelementową kolejkę FIFO (ang. *first in first out*). Podczas konfigurowania wejściowego izochronicznego punktu końcowego do jego kolejki nadawczej jest wstawiany pusty pakiet (bez danych). Z przyczyn technicznych, niewynikających bynajmniej ze standardu USB, kolejka nadawcza nie może być pusta. Skutek jest taki, że dla pierwszej zainicjowanej przez kontroler izochronicznej transakcji IN zostanie wysłany pusty pakiet. Nie zakłoca to jednak strumienia danych izochronicznych – po prostu rozpoczęcie wysyłania danych opóźnia się o jedną ramkę. Dla układu peryferyjnego DEV-FS parametr rxBuffSize może efektywnie przyjmować tylko wartości 2, 4, 6, 8, ..., 60, 62, 64, 96, 128, 192, 256, 320, 384, 448, 512. Podanie innej wartości spowoduje wybranie najbliższej większej. Wywołanie omawianej funkcji kończy się błędem, jeśli podano błędную kombinację parametrów, nie ma wolnego fizycznego punktu końcowego lub w PMA zabrakło miejsca dla jakiegoś bufora.

Układ peryferyjny OTG-FS ma 4 wejściowe i 4 wyjściowe punkty końcowe oraz 1280 bajtów pamięci przeznaczonej na kolejki odbiorcze i nadawcze FIFO. Numer punktu końcowego w układzie peryferyjnym odpowiada numerowi podanemu w odpowiednim deskryptorze. Biblioteka rezerwuje 1024 bajty pamięci układu peryferyjnego dla kolejki odbiorczej wspólnej dla wszystkich punktów końcowych, a pozostałe 256 bajty dla kolejek nadawczych. Ponadto rezerwuje 1024 bajty RAM na bufory odbiorcze. Kolejki FIFO i bufory pracują w porcjach (słowa) czterobajtowych, czyli wszystkie rozmiary wyrównywane są do najbliższej nie mniejszej wielokrotności czterech bajtów. Przy konfigurowaniu domyślnego punktu końcowego dla danych sterujących parametr endPoint musi mieć wartość zero (ENDP0), parametr type – wartość CONTROL_TRANSFER, a parametry rxBuffSize i txBuffSize

ze muszą mieć jednakowe wartości zgodne z wartością `bMaxPacketSize0` podaną w deskryptorze urządzenia. Dla tego punktu końcowego biblioteka alokuje bufor odbiorczy i kolejkę nadawczą o podanym rozmiarze. Przy konfigurowaniu pozostałych punktów końcowych, opisanych za pomocą deskryptorów, parametr `endPoint` może przyjmować wartość z przedziału 1...3 (ENDP1 do ENDP3), a parametr `type` musi być różny od `CONTROL_TRANSFER`. Jeśli parametr `rxBuffSize` ma wartość dodatnią, to alokowany jest bufor odbiorczy o podanym rozmiarze i konfigurowany jest odpowiedni wyjściowy punkt końcowy. Jeśli parametr `txBuffSize` ma wartość dodatnią, to alokowana jest kolejka nadawcza o podanym rozmiarze i konfigurowany jest odpowiedni wejściowy punkt końcowy. Aby zachować jednolitą semantykę dla wszystkich wariantów sprzętu, podczas konfigurowania wejściowego punktu końcowego dla danych izochronicznych do jego kolejki nadawczej wstawiany jest pusty pakiet (bez danych). Zatem dla pierwszej zainicjowanej przez kontroler transakcji izochronicznej IN zostanie wysłany pusty pakiet. Wywołanie omawianej funkcji kończy się błędem, jeśli podano błędną kombinację parametrów albo zabrakło miejsca dla bufora lub kolejki.

Z punktu widzenia omawianej tu funkcji układ peryferyjny OTG-HS różni się od OTG-FS w następujących aspektach. Ma 6 punktów końcowych – parametr `endPoint` dla punktów końcowych niesterujących może przyjmować wartość z przedziału 1...5 (ENDP1 do ENDP5). Ma 4096 bajtów na kolejki FIFO, z czego biblioteka rezerwuje 1024 bajty na kolejkę odbiorczą i przeznacza 3072 bajty na kolejki nadawcze.

```
uint16_t USBDwrite(uint8_t ep, uint8_t const *buff, uint16_t count);
```

Funkcja `USBDwrite` wstawia pakiet do wysłania przez punkt końcowy. Parametr `ep` jest numerem tego punktu końcowego. Parametr `buff` jest wskaźnikiem na bufor z danymi do wysłania. Parametr `count` zawiera rozmiar danych do wysłania. Funkcja kopiuje dane z przekazanego bufora do wewnętrznego bufora nadawczego lub kolejki nadawczej (semantyka kopiowania). Dzięki temu bufor może być ponownie wykorzystany przez aplikację bezpośrednio po zakończeniu tej funkcji, mimo że dane mogły jeszcze nie zostać wysłane. Funkcja zwraca liczbę rzeczywiście skopiowanych bajtów. Zwrócona wartość będzie mniejsza niż przekazana w parametrze `count`, jeśli jego wartość przekracza maksymalny dopuszczalny dla podanego punktu końcowego rozmiar pola danych pakietu. Gdy podano błędny numer punktu końcowego lub wskaźnik `buff` ma wartość zero (`NULL`), funkcja zwraca zero.

Dla DEV-FS omawiana funkcja kopiuje dane do bufora nadawczego w PMA. Natomiast dla OTG-FS i OTG-HS wstawia je do kolejki nadawczej. W obu przypadkach aplikacja musi dbać o to, żeby kolejny pakiet do wysłania został wstawiony dopiero po zakończeniu wysyłania poprzedniego, o czym aplikacja jest informowana za pomocą wywołania odpowiedniej funkcji zwrotnej z tablicy `EPin`.

```
uint32_t USBDwriteEx(uint8_t ep, uint8_t const *buff,
                      uint32_t count);
```

Funkcja `USBDwriteEx` wysyła dane za pomocą podanego punktu końcowego. Parametr `ep` jest numerem tego punktu końcowego. Parametr `buff` jest wskaźnikiem na bufor z danymi do wysłania. Parametr `count` zawiera rozmiar danych do wysłania. Jeśli rozmiar danych do wysłania jest większy niż maksymalny dopuszczalny rozmiar pola danych pakietu dla podanego punktu końcowego, to dane zostaną podzielone na stosowną liczbę pakietów i wysłane za pomocą wielu kolejnych transakcji. Zajmuje się tym pomocnicza funkcja `USBDcontinueInTransfer`. Funkcja `USBDwriteEx` nie gwarantuje, że dane z przekazanego bufora zostaną gdzieś skopiowane przed zakończeniem jej wykonywania (semantyka bez kopiowania). Bufor nie może więc być ponownie wykorzystany, dopóki nie upewnimy się, że wszystkie dane z niego zostały wysłane – musi zadbać o to protokół aplikacji. Funkcja zwraca liczbę bajtów, które będą wysłane. Funkcja zwraca zero, gdy poprzednia transmisja zainicjowana dla podanego punktu końcowego za pomocą tej funkcji jeszcze się nie skończyła lub gdy podano błędny numer punktu końcowego. Nie wolno dla tego samego punktu końcowego wywoływać funkcji `USBDwrite`, jeśli trwa transmisja zainicjowana za pomocą funkcji `USBDwriteEx`.

Dla układu DEV-FS podział danych na pakiety realizowany jest programowo. Układy peryferyjne OTG-FS i OTG-HS wspierają sprzętowo podział ciągu danych na pakiety. W tym celu należy skonfigurować odpowiednio duży rozmiar kolejki nadawczej, aby zmieścić w niej dane dla więcej niż jednego pakietu. Można to zrobić za pomocą funkcji `USBDDepointConfigureEx`, którą wywołuje się tak samo jak funkcję `USBDDepointConfigure`, ale ma ona jeden dodatkowy parametr `txFifoSize` określający rozmiar kolejki nadawczej. Parametr `txBuffSize` określa w tym przypadku rozmiar pola danych pakietu. Rozwiążanie to może okazać się bardziej efektywne niż programowe dzielenie danych na pakiety, ale jest nieprzenośne.

```
uint16_t USBDread(uint8_t ep, uint8_t *buff, uint16_t buffer_size);
```

Funkcja `USBDread` odczytuje zawartość pola danych odebranego pakietu. Zasadniczo jest ona przewidziana do wywoływania w funkcjach zwrotnych wskazywanych przez elementy tablicy `EPout`, czyli funkcjach informujących o odebraniu danych. Parametr `buff` jest wskaźnikiem na bufor, do którego mają być skopiowane odebrane dane. Parametr `buffer_size` określa rozmiar tego bufora. Funkcja ta może być wywołana tylko raz dla każdego odebranego pakietu. Jej zachowanie przy powtórny wywołaniu jest niezdefiniowane. Funkcja zwraca liczbę rzeczywiście skopiowanych bajtów.

Dla DEV-FS omawiana funkcja kopiuje dane z bufora odbiorczego w PMA do bufora aplikacji. Dla OTG-FS i OTG-HS sytuacja jest bardziej skomplikowana. Po pojawienniu się danych w ogólnej kolejce odbiorczej zgłaszane jest przerwanie, w którego procedurze obsługa pomocnicza funkcja `USBDdataReceived` kopiuje dane do bufora odbiorczego właściwego punktu końcowego. Po zakończeniu obsługi transakcji zgłaszane jest kolejne przerwanie, w którego procedurze obsługa wywoływana jest funkcja `USBDtransfer`, w wyniku czego zostanie wywołana odpowiednia funkcja zwrotna z tablicy `EPout` informująca, że dane mogą być przeczytane za pomocą funkcji `USBDread`. W tym przypadku wielokrotne kopiowanie jest ceną, którą płacimy za podział biblioteki na moduły i niezależność jej interfejsu.

programistycznego od sprzętu. W każdym przypadku aplikacja musi zadbać o to, aby przeczytać dane z bufora odbiorczego, zanim zostanie odebrany kolejny pakiet i nadpisze ten bufor.

```
usb_result_t USBDgetEndPointStatus(uint8_t epaddr, uint16_t *status);
```

Funkcja `USBDgetEndPointStatus` służy do odczytania stanu punktu końcowego. Parametr `epaddr` jest adresem punktu końcowego (numer punktu końcowego łącznie z bitem kierunku). Parametr `status` jest wskaźnikiem na zmienną, w której ma być zapisany stan tego punktu końcowego. Zmiennej tej przypisywana jest wartość 0, gdy punkt końcowy jest aktywny, a wartość 1, gdy został wstrzymany (co oznacza, że odrzuca transakcje transmisji danych, wysyłając pakiet STALL). Przypominam, że punkt końcowy dla danych izochronicznych nie może być wstrzymany i nigdy nie odpowiada pakietem STALL. Patrz też rysunki 1.10, 1.11 i 1.25. Gdy podany punkt końcowy jest skonfigurowany, funkcja zwraca `REQUEST_SUCCESS`. W przeciwnym przypadku funkcja zwraca `REQUEST_ERROR`.

```
usb_result_t USBDsetEndPointHalt(uint8_t epaddr);
```

Funkcja `USBDsetEndPointHalt` wstrzymuje podany punkt końcowy. Parametr `epaddr` jest adresem punktu końcowego. Funkcja zwraca `REQUEST_SUCCESS`, gdy podany punkt końcowy jest skonfigurowany i nie jest to punkt końcowy dla danych izochronicznych. W przeciwnym przypadku funkcja zwraca `REQUEST_ERROR`.

```
usb_result_t USBDclearEndPointHalt(uint8_t epaddr);
```

Funkcja `USBDclearEndPointHalt` uaktywnia podany punkt końcowy. Parametr `epaddr` jest adresem punktu końcowego. Funkcja zwraca `REQUEST_SUCCESS`, gdy podany punkt końcowy jest skonfigurowany i nie jest to punkt końcowy dla danych izochronicznych. W przeciwnym przypadku funkcja zwraca `REQUEST_ERROR`.

Druga grupa funkcji zadeklarowanych w pliku `usbd_api.h` jest przeznaczona do wywoływania przez rdzeń protokołu USB. Głównym powodem stworzenia tych funkcji jest potrzeba odseparowania warstwy protokołu od warstwy sprzętu. Funkcje te ukrywają szczegóły działania układów peryferyjnych USB.

```
void USBDsetDeviceAddress(set_address_t token, uint8_t address);
```

Funkcja `USBDsetDeviceAddress` konfiguruje adres urządzenia. Parametr `address` zawiera adres, który ma być ustawiony. Parametr `token` określa kontekst, w którym jest ona wywoływana i przyjmuje trzy wartości:

- `SET_ADDRESS_RESET` – po wyzerowaniu szyny USB, gdy ma być ustawiony adres domyślny;
- `SET_ADDRESS_REQUEST` – po zakończeniu fazy ustanowienia żądania `SET_ADDRESS`;
- `SET_ADDRESS_STATUS` – po wysłaniu statusu kończącego żądanie `SET_ADDRESS`.

W pierwszym przypadku parametr `address` powinien mieć wartość zero, a w dwóch ostatnich powinien zawierać ustawiany adres docelowy. Taki dość skomplikowany sposób wywoływania tej funkcji wynika z konieczności poradzenia sobie z pewnymi niuansami w działaniu układów peryferyjnych USB. Standard USB nakazuje dokończenie żądania `SET_ADDRESS` z ustawionym adresem domyślnym i ustawienie adresu docelowego dopiero po zakończeniu fazy statusu tego żądania. W praktyce okazało się jednak, że niektóre układy peryferyjne (w przypadku STM32 są to OTG-FS i OTG-HS) muszą mieć skonfigurowany rejestr adresowy już wcześniej, po zakończeniu fazy ustanowienia żądania `SET_ADDRESS`. Dlatego funkcję tę należy wywoływać w obu przypadkach, a implementacja dla DEV-FS ignoruje wywołanie z parametrem `SET_ADDRESS_REQUEST`, natomiast implementacja dla OTG-FS i OTG-HS ignoruje wywołanie z parametrem `SET_ADDRESS_STATUS`. Funkcja ta nie zwraca żadnej wartości.

Ponieważ rdzeń protokołu USB intensywnie używa punktu końcowego zero dla danych sterujących, to kolejne funkcje są wersjami funkcji przedstawionych już powyżej, ale zoptymalizowanymi dla tego punktu końcowego.

```
uint16_t USBDwrite0(uint8_t const *buffer, uint16_t count);
```

Funkcja `USBDwrite0` wstawia pakiet do wysłania przez punkt końcowy zero. Parametr `buffer` jest wskaźnikiem na bufor z danymi do wysłania. Parametr `count` zawiera rozmiar danych do wysłania. Funkcja kopiuje dane z przekazanego bufora do wewnętrznego bufora nadawczego lub kolejki nadawczej (semantyka kopiowania). Dzięki temu bufor może być ponownie wykorzystany bezpośrednio po zakończeniu tej funkcji, mimo że dane mogły jeszcze nie zostać wysłane. Funkcja zwraca liczbę rzeczywiście skopiowanych bajtów. Zwrócona wartość będzie mniejsza niż przekazana w parametrze `count`, jeśli jego wartość przekracza maksymalny dopuszczalny dla punktu końcowego zero rozmiar pakietu lub brakuje miejsca w kolejce nadawczej.

```
uint16_t USBDread0(uint8_t *buffer, uint16_t count);
```

Funkcja `USBDread0` odczytuje dane odebrane przez punkt końcowy zero. Parametr `buffer` jest wskaźnikiem na bufor, do którego mają być skopowane odebrane dane. Parametr `count` określa rozmiar tego bufora. Funkcja zwraca liczbę rzeczywiście skopiowanych bajtów.

```
void USBDendPoint0TxVALID(void);
```

Bezparametrowa funkcja `USBDendPoint0TxVALID` aktywuje wejściowy punkt końcowy zero. Od tego momentu jest on gotowy do wysyłania danych, czyli do realizacji transakcji IN. Funkcja ta nie zwraca żadnej wartości.

```
void USBDendPoint0RxVALID(void);
```

Bezparametrowa funkcja `USBxDendPoint0RxVALID` aktywuje wyjściowy punkt końcowy zero. Od tego momentu jest on gotowy do odbierania danych, czyli do realizacji transakcji OUT. Funkcja ta nie zwraca żadnej wartości.

```
void USBxDendPoint0TxSTALL(void);
```

Bezparametrowa funkcja `USBxDendPoint0TxVALID` wstrzymuje wejściowy punkt końcowy zero. Od tego momentu odpowiada on pakietem STALL na inicjowane przez kontroler transakcje IN – patrz rysunek 1.10. Funkcja ta nie zwraca żadnej wartości.

```
void USBxDendPoint0RxSTALL(void);
```

Bezparametrowa funkcja `USBxDendPoint0TxVALID` wstrzymuje wyjściowy punkt końcowy zero. Od tego momentu odpowiada on pakietem STALL na pakiety danych w transakcjach OUT – patrz rysunek 1.10. Funkcja ta nie zwraca żadnej wartości.

4.1.4. Rdzeń protokołu

```
usbd_core.h  
usbd_core.c
```

Plik `usbd_core.c` zawiera implementację rdzenia protokołu USB opisanego w rozdziale 1. Implementacja ta jest praktycznie niezależna od sprzętu. Zadaniem tego modułu jest obsługa wszelkich zdarzeń, jakie zachodzą na szynie USB, w tym głównie żądań standardowych, oraz wywoływanie odpowiednich funkcji zwrotnych. Plik `usbd_core.h` zawiera deklaracje publicznych funkcji tego modułu. Powinny być one wywoływane po zaistnieniu odpowiednich zdarzeń, zwykle w procedurze obsługi przerwania związanego z danym zdarzeniem.

```
int USBDcoreConfigure(void);
```

Bezparametrowa funkcja `USBDcoreConfigure` inicjuje strukturę danych przechowującą stan urządzenia oraz wywołuje funkcję `USBDgetApplicationCallbacks`, aby uzyskać listę wskaźników na funkcje zwrotne, a potem funkcję zwrotną `Configure`, jeśli wskaźnik do niej został podany. Funkcja ta zwraca zero, gdy wszystko przebiegło poprawnie, a wartość ujemną, gdy wystąpił błąd.

```
void USBDreset(usb_speed_t speed);
```

Funkcja `USBDreset` jest wywoływana w reakcji na wyzerowanie szyny USB przez kontroler (w procedurze obsługi tego zdarzenia). Nie zwraca ona żadnej wartości. Wykonuje wszelkie niezbędne akcje związane z ponownym zainicjowaniem urządzenia i przypisaniem mu domyślnego adresu zerowego. Wywołuje funkcję zwrotną `Reset` w celu zainicjowania warstwy aplikacji. Parametr `speed` określa szybkość, z jaką ma być uruchomione urządzenie i jest przekazywany do funkcji zwrotnej `Reset`.

```
void USBDsuspend(void);
```

Bezparametrowa funkcja `USBDsuspend` jest wywoływana, gdy urządzenie przechodzi w stan wstrzymania (w procedurze obsługi tego zdarzenia), ale jeszcze przed uśpieniem mikrokontrolera. Modyfikuje ona stan urządzenia i wywołuje funkcję zwrotną `Suspend`. Nie zwraca żadnej wartości.

```
void USBDwakeu(p void);
```

Bezparametrowa funkcja `USBDwakeu(p)` jest wywoływana, gdy szyna USB powraca do stanu aktywnego (w procedurze obsługi tego zdarzenia), oczywiście po obudzeniu mikrokontrolera. Modyfikuje ona stan urządzenia i wywołuje funkcję zwrotną `Wakeu(p)`. Nie zwraca żadnej wartości.

```
void USBDsof(uint16_t frameNumber);
```

Funkcja `USBDsof` jest wywoływana po odebraniu pakietu SOF (w procedurze obsługi tego zdarzenia). Funkcja ta wywołuje funkcję zwrotną `SoF`, do której przekazuje parametr `frameNumber` zawierający numer ramki z pakietu SOF. Nie zwraca żadnej wartości.

```
void USBDtransfer(uint8_t ep, usb_pid_t token);
```

Funkcja `USBDtransfer` jest wywoływana dla każdej transakcji związanej z przesaniem danych (w procedurze obsługi tego zdarzenia). Parametr `ep` jest numerem punktu końcowego, który uczestniczy w transakcji, a parametr `token` zawiera PID transakcji. Jeśli transakcja kierowana jest do punktu końcowego zero, czyli dotyczy danych sterujących, to wywoływane są funkcje obsługujące tę transakcję, w wyniku czego mogą być wywołane również funkcje zwrotne, jeśli rozpoznane zostanie żądanie, którego obsługa tego wymaga. Jeśli transakcja nie jest kierowana do punktu końcowego zero, to wywoływana jest funkcja zwrotna wskazywana przez tablicę `EPout` lub `Epin`. Całkowita odpowiedzialność za obsługę takiej transakcji spoczywa wtedy na warstwie aplikacji.

Wspomniane wyżej funkcje zwrotne są wywoływane jedynie, gdy zostały skonfigurowane. Podanie zerowego wskaźnika jako adresu funkcji zwrotnej powoduje, że ta funkcja zwrotna nie będzie wywoływana.

4.1.5. Przerwania

Jak to zwykle bywa w świecie mikrokontrolerów, zdarzenia zachodzące na szynie USB są źródłem przerwań. Chcąc uczynić bibliotekę jak najbardziej ogólną i elastyczną, a mając na uwadze, że ze względu na różnorodność sprzętu obsługa przerwań jest najtrudniejszym do standaryzowania fragmentem oprogramowania, podzieliłem obsługę przerwań na trzy warstwy:

- pierwsza – zależna tylko od wersji mikrokontrolera,
- druga – zależna tylko od układu peryferyjnego USB,
- trzecia – niezależna od sprzętu.

Niewątpliwą wadą takiego podziału jest dodatkowy narzut związany z wywoływaniem funkcji, a powodowany koniecznością przekazywania sterowania do kolejnych warstw w celu ich wzajemnego odseparowania. Jednak kolosalną zaletą takiego podejścia jest łatwość dostosowywania biblioteki do różnych wariantów sprzętu. Dla urządzenia USB trzecia warstwa to rdzeń protokołu USB zaimplementowany w opisanych wyżej plikach *usbd_core.h* i *usbd_core.c*.

```
usbd_interrupt_103.c  
usbd_interrupt_152.c  
usb_otg_interrupt.c
```

Biblioteka zawiera trzy warianty implementacji pierwszej warstwy. W pliku *usbd_interrupt_103.c* znajdują się procedury obsługi przerwań dla STM32F102 i STM32F103. W pliku *usbd_interrupt_152.c* znajdują się procedury obsługi przerwań dla STM32Lxxx. W pliku *usb_otg_interrupt.c* znajdują się procedury obsługi przerwań dla STM32F105, STM32F107, STM32F2xx i STM32F4xx. Jeśli przerwanie wymaga obsługi innej niż skasowanie jego przyczyny, to sterowanie przekazywane jest do warstwy drugiej.

```
usb_interrupt.h  
usbd_interrupt_aaa.c  
usbd_interrupt_bbb.c
```

W pliku *usb_interrupt.h* zadeklarowana jest bezparametrowa funkcja *USBglobal-InterruptHandler*, będąca głównym punktem wejścia do drugiej warstwy obsługi przerwań USB. Nie zwraca ona żadnej wartości. Jej implementacja dla układu peryferyjnego DEV-FS znajduje się w pliku *usbd_interrupt_aaa.c*, a implementacja dla urządzenia realizowanego na układach peryferyjnych OTG-FS i OTG-HS jest w pliku *usbd_interrupt_bbb.c*.

```
void USBglobalInterruptHandler(void);
```

Funkcja *USBglobalInterruptHandler* rozpoznaje zdarzenie, które jest przyczyną przerwania. Obsługuje to zdarzenie, ewentualnie przekazując sterowanie do warstwy trzeciej lub do opisanego poniżej modułu odpowiedzialnego za zarządzanie energią interfejsu USB.

4.1.6. Abstrakcja sprzętu

```
usb_regs.h  
usb_otg_regs.h
```

Układy peryferyjne USB, podobnie jak wszystkie inne peryferie, widziane są jako szereg rejestrów umieszczonych w przestrzeni adresowej mikrokontrolera. Żeby ułatwić posługивание się tymi rejestrami, tworzy się dla każdego układu peryferyjnego pliki nagłówkowe zawierające różnego rodzaju makrodefinicje. Plik *usb_regs.h* zawiera definicje dla układu DEV-FS. Plik *usb_otg_regs.h* zawiera definicje dla

układów OTG-FS i OTG-HS, w tym również definicje potrzebne do zaimplementowania kontrolera. Oba pliki powstały na podstawie odpowiednich plików dostarczanych przez STMicroelectronics. Zostały jednak przeze mnie nieco zmodyfikowane i rozbudowane. Poprawiłem w nich też różne drobne błędy.

W pliku *usb_regs.h* zastosowano tradycyjne podejście wykorzystujące instrukcję `#define` preprocesora. Dla przykładu, aby ustawić adres urządzenia, należy wywołać makro `_SetDADDR`. Stała `DADDR` jest adresem względnym odpowiedniego rejestru konfiguracyjnego. Adresy rejestrów określane są względem początku obszaru pamięci zajmowanego przez układ peryferyjny DEV-FS i zdefiniowanego jako stała `RegBase`. Odpowiednie definicje przedstawia poniższy wydruk.

```
#define RegBase (0x40005C00L)
#define DADDR ((__IO unsigned *) (RegBase + 0x4C))
#define _SetDADDR(wRegValue) *DADDR = (uint16_t) (wRegValue)
```

Zobaczmy inny, bardziej rozbudowany przykład. Żeby skonfigurować rodzaj danych dla punktu końcowego, należy wywołać makro `_SetEPType`. Pierwszy parametr tego makra jest numerem fizycznego punktu końcowego. Drugi parametr określa rodzaj danych i może przyjmować jedną z czterech wartości: `EP_BULK`, `EP_CONTROL`, `EP_ISOCHRONOUS`, `EP_INTERRUPT`. Stała `EP0REG` definiuje początkowy adres przestrzeni adresowej rejestrów konfiguracyjnych punktów końcowych. Rejestry dla kolejnych punktów końcowych znajdują się pod kolejnymi adresami. Stałe rozpoczynające się przedrostkiem `EP` definiują pola bitowe rejestru konfiguracyjnego. Odpowiednie definicje przedstawia poniższy wydruk.

```
#define EP0REG          (__IO unsigned *) (RegBase)
#define EP_CTR_RX        (0x8000)
#define EP_DTOG_RX        (0x4000)
#define EPRX_STAT         (0x3000)
#define EP_SETUP           (0x0800)
#define EP_T_FIELD         (0x0600)
#define EP_KIND            (0x0100)
#define EP_CTR_TX          (0x0080)
#define EP_DTOG_TX          (0x0040)
#define EPTX_STAT          (0x0030)
#define EPADDR_FIELD       (0x000F)
#define EP_BULK             (0x0000)
#define EP_CONTROL          (0x0200)
#define EP_ISOCHRONOUS     (0x0400)
#define EP_INTERRUPT        (0x0600)
#define EPREG_MASK          (EP_CTR_RX | EP_SETUP | EP_T_FIELD | \
                           EP_KIND | EP_CTR_TX | EPADDR_FIELD)
#define EP_T_MASK            (~EP_T_FIELD & EPREG_MASK)
#define _GetENDPOINT(bEpNum) \
    ((uint16_t) (* (EP0REG + (bEpNum))))
```

```
#define _SetENDPOINT(bEpNum, wRegValue) \
    *(EP0REG + (bEpNum)) = (uint16_t)(wRegValue)
#define _SetEPTYPE(bEpNum, wType) \
    _SetENDPOINT(bEpNum, ((_GetENDPOINT(bEpNum) & EP_T_MASK) | (wType)))
```

Poważną wadą stosowania instrukcji `#define` jest brak kontroli typów argumentów, a co z tego wynika – brak jakiekolwiek ochrony przed błędnym użyciem makra. Zobaczmy przykład typowego błędu, który jest dość trudny do wykrycia. W pliku `usb_def.h` zdefiniowano stałą `CONTROL_TRANSFER` o wartości 0. Użycie tej stałej jako drugiego parametru makra `_SetEPTYPE`, zamiast stałej `EP_CONTROL` sprawi, że punkt końcowy, zamiast dla danych sterujących, zostanie skonfigurowany dla danych masowych. Program się skompiluje i nawet będzie wykazywał jakieś objawy działania. Zaletą stosowania makrodefinicji jest niewątpliwie efektywność tłumaczenia tekstu źródłowego na kod wykonywalny.

W pliku `usb_otg_regs.h` zastosowano zupełnie inne podejście. Układy OTG-FS i OTG-HS mają znacznie więcej rejestrów konfiguracyjnych niż układ DEV-FS. Bloki rejestrów pogrupowano w struktury danych. Przykładowo rejesty odpowiadające za wyjściowe punkty końcowe tworzą tablicę elementów typu `USB_OTG_DOUTEPS`.

```
typedef struct {
    volatile uint32_t DOEPCRTLx;
    volatile uint32_t reserved04;
    volatile uint32_t DOEPINTx;
    volatile uint32_t reserved0C;
    volatile uint32_t DOEPTSIZx;
    volatile uint32_t DOEPDMAX;
    volatile uint32_t reserved18;
    volatile uint32_t reserved1C;
} USB_OTG_DOUTEPS;
```

Poszczególne rejesty zdefiniowano jako unie z polami bitowymi. Przykładowo rejestr `DOEPCRTLx` zdefiniowano jako typ `USB_OTG_DEPCTLx_TypeDef`.

```
typedef union {
    uint32_t d32;
    struct {
        uint32_t mpsiz : 11;
        uint32_t reserved11_14 : 4;
        uint32_t usbaep : 1;
        uint32_t dpid_eonum : 1;
        uint32_t naksts : 1;
        uint32_t eptyp : 2;
        uint32_t snpm : 1;
        uint32_t stall : 1;
        uint32_t txfnum : 4;
    } DEPCTLx;
}
```

```

        uint32_t cnak           : 1;
        uint32_t snak           : 1;
        uint32_t sd0pid_sevnfrm : 1;
        uint32_t sd1pid_soddfrm : 1;
        uint32_t epdis          : 1;
        uint32_t epena          : 1;
    } b;
} USB_OTG_DEPCTLx_TypeDef;

```

Położenie tablicy bloków rejestrów w przestrzeni adresowej definiują makra, których przykładem jest P_USB_OTG_DOUTEPS.

```

#define USB_OTG_BASE_ADDR           0x50000000
#define USB_OTG_DEV_OUT_EP_REG_OFFSET 0x0B00
#define P_USB_OTG_DOUTEPS \
    ((USB_OTG_DOUTEPS *) (USB_OTG_BASE_ADDR + \
                           USB_OTG_DEV_OUT_EP_REG_OFFSET))

```

Skonfigurowanie punktu końcowego wymaga zadeklarowania zmiennej odpowiedniego typu, przypisania do jej pól właściwych wartości i zapisania wartości tej zmiennej do odpowiedniego rejestru. W poniższym przykładzie konfigurujemy punkt wyjściowy o numerze endPoint dla danych masowych. Przypisanie zera do pola d32 ma na celu wyzerowanie wszystkich pól, w tym również tych zarezerwowanych, jak np. pole reserved11_14.

```

USB_OTG_DEPCTLx_TypeDef depctl;
depctl.d32 = 0;
...
depctl.b.eptyp = BULK_TRANSFER;
...
P_USB_OTG_DOUTEPS[endPoint].DOEPCTLx = depctl.d32;

```

Zaletą takiego podejścia w stosunku do poprzedniego jest scislesza kontrola typów. Trudniej jest też pomylić się i zmodyfikować niewłaściwe bity rejestru. Natomiast wadą jest narzut przy tłumaczeniu tekstu źródłowego na kod wykonywalny. Eksperymenty pokazują, że w przypadku stosowania struktur z polami bitowymi kompilator emittuje nieco większą liczbę instrukcji maszynowych niż przy zastosowaniu podejścia z instrukcjami #define. Dodatkową wadą jest samo stosowanie dwóch różnych konwencji. Postanowiłem jednak pozostawić oba podejścia, aby nie utrudniać zbytnio korzystania z przykładów dostarczanych przez STMicroelectronics.

usb_otg_fifo.h
usb_otg_fifo.c

W układach peryferyjnych OTG-FS i OTG-HS dane wysyła i odbiera się za pomocą kolejek FIFO. Interfejs obsługi tych kolejek (w znacznej części wspólny dla urządzenia i kontrolera) jest zadeklarowany w pliku *usb_otg_fifo.h*, a jego implementacja znajduje się w pliku *usb_otg_fifo.c*.

Jeśli układ peryferyjny jest skonfigurowany jako urządzenie, to mamy jedną wspólną dla wszystkich punktów końcowych kolejkę odbiorczą i po jednej kolejce nadawczej dla każdego punktu końcowego. Pakiety z kolejki odbiorczej są czytane w kolejności, w jakiej zostały wysłane przez kontroler. Dane wstawione do określonej kolejki nadawczej są wysyłane w kolejności ich wstawienia do tej kolejki. Natomiast wybór kolejki nadawczej spośród wszystkich kolejek nadawczych odbywa się na podstawie kolejności inicjowania transakcji przez kontroler.

Jeśli układ peryferyjny jest skonfigurowany jako kontroler, to mamy jedną kolejkę odbiorczą i dwie kolejki nadawcze: dla danych periodycznych (pilnych lub izochronicznych) i nieperiodycznych (sterujących i masowych). Transmisje odbywają się w kolejności ich inicjowania przez kontroler.

```
int FlushTxFifo(int num);
```

Funkcja FlushTxFifo czyści kolejkę nadawczą. Żadne dane nie są wysyłane. Parametr num jest numerem punktu końcowego (w przypadku urządzenia) lub kanału (w przypadku kontrolera). Czyszczona jest kolejka związana z tym punktem końcowym lub kanałem. Jeśli parametr ten ma wartość ALL_TX_FIFOS, to czyszczone są wszystkie kolejki nadawcze. Funkcja zwraca zero, gdy zakończyła się powodzeniem, a wartość ujemną, gdy wystąpił błąd.

```
int FlushRxFifo(void);
```

Bezparametrowa funkcja FlushRxFifo czyści kolejkę odbiorczą. Funkcja zwraca zero, gdy zakończyła się powodzeniem, a wartość ujemną, gdy wystąpił błąd. Dla zachowania symetrii z funkcją czyszczącą kolejki nadawcze możliwe jest również wywołanie FlushRxFifo(ALL_RX_FIFOS).

```
void WriteFifo8(int num, uint8_t const *src, uint32_t len);
```

Funkcja WriteFifo8 zapisuje dane do kolejki nadawczej. Parametr num jest numerem punktu końcowego lub kanału. Dane zapisywane są do kolejki związanej z tym punktem końcowym lub kanałem. Parametr src jest wskaźnikiem na dane. Parametr len zawiera liczbę bajtów do wysłania. Przed wywołaniem tej funkcji należy się upewnić, że w kolejce nadawczej jest wystarczająco dużo miejsca. Funkcja ta nie zwraca żadnej wartości.

```
void WriteFifo32(int num, uint32_t const *src, uint32_t len);
```

Funkcja WriteFifo32 zapisuje dane do kolejki nadawczej. Parametr num jest numerem punktu końcowego lub kanału. Dane zapisywane są do kolejki związanej z tym punktem końcowym lub kanałem. Parametr src jest wskaźnikiem na dane. Adres ten musi być wyrównany do wielokrotności 4 bajtów. Parametr len zawiera liczbę bajtów do wysłania. Liczba skopiowanych bajtów jest zaokrąglana w górę do najbliższej wielokrotności 4. Funkcja ta kopiuje dane porcjami po 4 bajty, dzięki czemu na architekturze ARM jest efektywniejsza niż funkcja WriteFifo8, która

kopiuje je bajt po bajcie. Przed jej wywołaniem należy się upewnić, że w kolejce nadawczej jest wystarczająco dużo miejsca. Funkcja ta nie zwraca żadnej wartości.

```
void ReadFifo8(int num, uint8_t *dst, uint32_t len);
```

Funkcja ReadFifo8 odczytuje dane z kolejki odbiorczej. Parametr num jest numerem punktu końcowego lub kanału. W aktualnej implementacji jest on zbędny – jest tylko jedna kolejka odbiorcza, ale jest podawany dla zachowania symetrii z funkcją zapisującą dane do kolejki nadawczej. Parametr dst jest wskaźnikiem na bufor, do którego mają zostać skopowane dane. Parametr len zawiera liczbę bajtów, które mają być odczytane z kolejki. Liczba ta nie może być większa od liczby bajtów znajdujących się w kolejce – w aktualnej implementacji informację tę uzyskuje się przez odczytanie rejestru statusu w procedurze obsługi przerwania, które zostało wyzwolone w wyniku odebrania danych. Funkcja ta nie zwraca żadnej wartości.

```
void ReadFifo32(int num, uint32_t *dst, uint32_t len);
```

Funkcja ReadFifo32 odczytuje dane z kolejki odbiorczej. Parametr num jest numerem punktu końcowego lub kanału. W aktualnej implementacji jest on zbędny – jest tylko jedna kolejka odbiorcza, ale jest podawany dla zachowania symetrii z funkcją zapisującą dane do kolejki nadawczej. Parametr dst jest wskaźnikiem na miejsce w pamięci, gdzie mają zostać skopowane dane. Adres ten musi być wyrownany do wielokrotności 4 bajtów. Parametr len zawiera liczbę bajtów do odczytania. Liczba ta nie może być większa od liczby bajtów znajdujących się w kolejce – w aktualnej implementacji informację tę uzyskuje się przez odczytanie rejestru statusu w procedurze obsługi przerwania, którego przyczyną jest odebranie danych. Liczba czytanych bajtów jest zaokrąglana w górę do najbliższej wielokrotności 4. Funkcja ta kopiuje dane porcjami po 4 bajty, dzięki czemu na architekturze ARM jest efektywniejsza niż funkcja ReadFifo8, która kopiuje bajt po bajcie. Funkcja ta nie zwraca żadnej wartości.

```
uint32_t GetDeviceFreeTxFifoSpace(int ep);
```

Funkcja GetDeviceFreeTxFifoSpace zwraca rozmiar w bajtach pustego miejsca w kolejce nadawczej urządzenia. Parametr ep jest numerem punktu końcowego.

```
uint32_t GetHostFreeTxFifoSpace(int periodic);
```

Funkcja GetHostFreeTxFifoSpace zwraca rozmiar w bajtach pustego miejsca w kolejce nadawczej kontrolera. Parametr periodic ma wartość niezerową, gdy wywołanie dotyczy kolejki dla danych periodycznych, a wartość zero, gdy dotyczy kolejki dla danych nieperiodycznych.

Wiele informacji na temat układów peryferyjnych USB można znaleźć w dokumentach [9], [10], [11], [13]. Tam w szczególności należy szukać opisów wszystkich rejestrów konfiguracyjnych. Najlepszym sposobem poznania sposobu użycia tych rejestrów wydaje się przestudiowanie tekstu źródłowego biblioteki.

4.1.7. Główna funkcja programu

usbd_main.c

Plik *usbd_main.c* zawiera funkcję *main* wykorzystywaną we wszystkich przykładowych projektach urządzeń prezentowanych w tej książce. Funkcja ta została już omówiona w rozdziale 3.1.6. Wszystkie wywoływanie przez nią funkcje zostały opisane powyżej lub w rozdziale 2. Niepowodzenia podczas wykonywania funkcji *main* sygnalizowane są za pomocą migania czerwonej diody świecącej. Zadanie to realizuje funkcja *ErrorResetable*, która z kolei wywołuje funkcję *Error*. Wszystkie błędy zgłasiane przez urządzenia zostały dla porządku zebrane w **tabeli 4.1**. Numer błędu odpowiada liczbie mignięć w sekwencji. Należy zauważyć, że ponieważ w aktualnej implementacji funkcja *PWRconfigure* zawsze zwraca wartość zero, błęd numer 4 nigdy nie jest sygnalizowany.

Tab. 4.1. Błędy sygnalizowane przez urządzenie

Numer błędu	Opis błędu
1	Funkcja <i>USBpreConfigure</i> , niepoprawna wartość parametru
2	Funkcja <i>ClockConfigure</i> , niepoprawna lub niemożliwa do ustawienia częstotliwości takowania
3	Funkcja <i>LCDconfigure</i> , błąd podczas uruchamiania wyświetlacza, brak wyświetlacza
4	Funkcja <i>PWRconfigure</i> , błąd nigdy niesygnalizowany
5	Funkcja <i>USBConfigure</i> , niepoprawna wartość parametru, błąd podczas konfigurowania układu peryferyjnego USB w trybie urządzenia
6	Funkcja zwrotna <i>Reset</i> , nieakceptowana przez urządzenie szybkość
7	Funkcja zwrotna <i>Reset</i> , błąd podczas konfigurowania domyślnego punktu końcowego dla danych sterujących
8	Układ peryferyjny OTG-FS lub OTG-HS w trybie kontrolera, plik <i>usbd_interrupt_bbb.c</i>
9	Problem ze skonfigurowaniem dżojstika, plik <i>joystick_322xg.c</i>

4.1.8. Wybrane fragmenty implementacji

W tym podrozdziale opisuję kluczowe fragmenty implementacji rdzenia protokołu USB po stronie urządzenia, czyli obsługę żądań, które przesyła się za pomocą danych sterujących.

usb_def.c

W pliku *usb_def.h* zdefiniowany jest typ strukturalny *usb_setup_packet_t*, który reprezentuje format danych przesyłanych w transakcji SETUP. Makrodefinicja *_packed*, zdefiniowana na początku pliku *usb_def.h*, zależy od użytego kompilatora i sprawia, że pola struktury zostaną ulozone w pamięci ciasno, bez przerw.

```
typedef struct {
    uint8_t bmRequestType;
    uint8_t bRequest;
    uint16_t wValue;
    uint16_t wIndex;
    uint16_t wLength;
} __packed usb_setup_packet_t;
```

Dla ułatwienia posługiwania się polem bitowym `bmRequestType` w pliku `usb_def.h` zdefiniowane są stałe zamieszczone na poniższym wydruku.

```
#define REQUEST_DIRECTION          0x80
#define DEVICE_TO_HOST             0x80
#define HOST_TO_DEVICE             0x00
#define REQUEST_TYPE               0x60
#define STANDARD_REQUEST           0x00
#define CLASS_REQUEST              0x20
#define VENDOR_REQUEST             0x40
#define REQUEST_RECIPIENT          0x1f
#define DEVICE_RECIPIENT            0x00
#define INTERFACE_RECIPIENT         0x01
#define ENDPOINT_RECIPIENT          0x02
```

Wartości standardowych żądań, jakie mogą się pojawić w polu `bRequest`, zdefiniowane są jako typ wyliczeniowy `usb_standard_request_t`.

```
typedef enum {
    GET_STATUS      = 0,
    CLEAR_FEATURE   = 1,
    SET_FEATURE     = 3,
    SET_ADDRESS     = 5,
    GET_DESCRIPTOR  = 6,
    SET_DESCRIPTOR  = 7,
    GET_CONFIGURATION = 8,
    SET_CONFIGURATION = 9,
    GET_INTERFACE   = 10,
    SET_INTERFACE   = 11,
    SYNCH_FRAME    = 12
} usb_standard_request_t;
```

usbd_core.c

Implementacja rdzenia protokołu USB dla urządzenia znajduje się w pliku `usbd_core.c`. Najpierw wprowadzamy kilka makrodefinicji, które skracają zapis instrukcji.

```
#define IN_REQ      (DEVICE_TO_HOST)
#define STD_REQ     (STANDARD_REQUEST)
#define IN_STD_REQ_DEV \
    (DEVICE_TO_HOST | STANDARD_REQUEST | DEVICE_RECIPIENT)
#define IN_STD_REQ_IF \
    (DEVICE_TO_HOST | STANDARD_REQUEST | INTERFACE_RECIPIENT)
#define IN_STD_REQ_EP \
    (DEVICE_TO_HOST | STANDARD_REQUEST | ENDPOINT_RECIPIENT)
#define OUT_STD_REQ_DEV \
    (HOST_TO_DEVICE | STANDARD_REQUEST | DEVICE_RECIPIENT)
```

Protokół realizujący transmisję danych sterujących zaimplementowany jest jako automat stanowy. Jego stany zdefiniowane są jako typ wyliczeniowy `usb_control_state_t`. Stan `IDLE` jest stanem spoczynkowym, w którym automat czeka na transakcję `SETUP`. W fazie transmisji danych automat znajduje się w jednym z trzech stanów (zależnie od kierunku transmisji): `DATA_IN`, `LAST_DATA_IN` lub `DATA_OUT`. W stanie `DATA_IN` automat czeka na zakończenie transakcji IN. W stanie `LAST_DATA_IN` automat czeka na zakończenie ostatniej transakcji IN. W stanie `DATA_OUT` automat czeka na transakcję OUT. W stanie `WAIT_STATUS_IN` lub `WAIT_STATUS_OUT` automat czeka na zakończenie fazy statusu, czyli odpowiednio na transakcję IN lub OUT.

```
typedef enum {
    IDLE, DATA_IN, DATA_OUT, LAST_DATA_IN,
    WAIT_STATUS_IN, WAIT_STATUS_OUT
} usb_control_state_t;
```

Stan urządzenia przechowywany jest w strukturze typu `usb_device_state_t`.

```
typedef struct {
    usb_visible_state_t      visibleState;
    usb_control_state_t     controlState;
    usb_setup_packet_t      setup;
    usbd_callback_list_t    callback;
    uint8_t                  const *txdata;
    uint8_t                  *rxdata;
    uint16_t                 length;
    uint8_t                  maxPacketSize0;
} usb_device_state_t;
```

Składowa `visibleState` zawiera stan urządzenia. Może ona przyjmować wartości:

- `POWERED` – urządzenie jest podłączone i zasilane;
- `DEFAULT` – zakończona została procedura zerowania szyny, urządzenie ma adres domyślny (zero);
- `ADDRESS` – urządzenie ma przyznany docelowy adres;
- `CONFIGURED` – została wybrana konfiguracja.

Ponadto w każdym z powyższych stanów składowa ta może mieć ustawiony bit `SUSPENDED`, co oznacza, że szyna USB jest wstrzymana – nie odbywają się żadne transmisje i urządzenie ogranicza pobór prądu z szyny.

Składowa `controlState` zawiera stan automatu. Składowa `setup` przechowuje strukturę opisującą żądanie. Składowa `callback` zawiera wskaźnik na strukturę przechowującą adresy funkcji zwrotnych, wywoływanych w celu obsłużenia żądań, które nie mogą być obsłużone przez rdzeń. Składowa `txdata` jest wskaźnikiem na bufor z danymi do wysłania. Składowa `rxdata` jest wskaźnikiem na bufor, do którego mają być skopiowane odebrane dane. Składowa `length` zawiera rozmiar danych, jakie pozostały jeszcze do przesłania. Składowa `maxPacketSize0` zawiera maksymalny rozmiar pola danych pakietów.

Implementacja automatu stanowego podzielona jest między kilka poniższych funkcji. Wszystkie one przyjmują argument, który jest wskaźnikiem na strukturę przechowującą stan urządzenia. Funkcje te nie zwracają żadnej wartości.

```
static void Setup0(usb_device_state_t * );
static void NoDataSetup0(usb_device_state_t * );
static void InDataSetup0(usb_device_state_t * );
static void OutDataSetup0(usb_device_state_t * );
static void In0(usb_device_state_t * );
static void DataStageIn0(usb_device_state_t * );
static void Out0(usb_device_state_t * );
static void DataStageOut0(usb_device_state_t * );
```

Funkcja Setup0 jest wywoływana po rozpoznaniu przez urządzenie transakcji SETUP skierowanej do punktu końcowego zero. Zaczyna się ona od skopiowania do struktury setup zawartości odebranego w tej transakcji pakietu z danymi. Następnie sprawdzamy, czy automat znajduje się w stanie spoczynkowym i czy odebrano 8 bajtów. W przypadku błędu automat wraca do stanu spoczynkowego oraz blokuje realizację ewentualnych transakcji IN i OUT. Nie blokuje to możliwości obsłużenia następnej transakcji SETUP.

```
void Setup0(usb_device_state_t *ds) {
    uint16_t len;
    len = USBDRead0((uint8_t*)&(ds->setup), sizeof(usb_setup_packet_t));
    if (ds->controlState != IDLE || len != sizeof(usb_setup_packet_t))
    {
        ds->controlState = IDLE;
        USBDendPoint0RxSTALL();
        USBDendPoint0TxSTALL();
        return;
    }
    ds->setup.wValue    = USBTOHS(ds->setup.wValue);
    ds->setup.wIndex    = USBTOHS(ds->setup.wIndex);
    ds->setup.wLength   = USBTOHS(ds->setup.wLength);
    if (ds->setup.wLength == 0) {
        ds->setup.bmRequestType &= ~REQUEST_DIRECTION;
        NoDataSetup0(ds);
    }
    else if ((ds->setup.bmRequestType & REQUEST_DIRECTION) == IN_REQ)
    {
        InDataSetup0(ds);
    }
    else {
        OutDataSetup0(ds);
    }
}
```

Jeśli automat jest w stanie spoczynkowym i odebrał całą strukturę setup, to najpierw aplikujemy do jej pól makro USBTOHS, gdyż pola tej struktury mogą wymagać odwrócenia kolejności bajtów (jeśli lokalny porządek bajtów nie jest zgodny z po-rządkiem obowiązującym w USB). Następnie, zależnie od wartości pola wLength i bitu kierunku w polu bmRequestType, delegujemy dalszą obsługę żądania do jednej z trzech funkcji. Funkcja NoDataSetup0 obsługuje żądania niewymagające przesłania danych. W tym przypadku bit kierunku może być zignorowany. Funkcje InDataSetup0 i OutDataSetup0 obsługują odpowiednio żądania wymagające prze-słania danych z urządzenia do kontrolera lub z kontrolera do urządzenia. Na poniż-szych wydrukach zamieszczam tylko ogólną strukturę tych funkcji. Pominięte frag-menty zaznaczam trzema kropkami. Pełny tekst źródłowy tej funkcji znajduje się w archiwum z przykładami. W celu rozpoznania żądania i jego obsłużenia wykonu-jemy szereg sprawdzeń (za pomocą instrukcji if) wartości pól struktury setup oraz stanu, w którym znajduje się aktualnie urządzenie. Poprawność żądania może za-leżeć od tego stanu. Na kolejnych wydrukach pomijam większość tych sprawdzeń, ograniczając się tylko do pokazania obsługi żądań SET_ADDRESS i GET_DESCRIPTOR. Obsługa pozostałych żądań wygląda analogicznie.

Zacznijmy od funkcji NoDataSetup0. Zmienna lokalna result przyjmuje wartość REQUEST_SUCCESS, jeśli żądanie zostało poprawnie rozpoznane i będzie obsłużone, a wartość REQUEST_ERROR, gdy żądanie jest niepoprawne lub ma być odrzucone.

```
void NoDataSetup0(usb_device_state_t *ds) {
    usb_result_t result;
    uint8_t recipient;
    result = REQUEST_ERROR;
    if ((ds->setup.bmRequestType & REQUEST_TYPE) == STANDARD_REQUEST)
    {
        recipient = ds->setup.bmRequestType & REQUEST_RECIPIENT;
        if (recipient == DEVICE_RECIPIENT && ds->setup.wIndex == 0) {
            if (ds->setup.bRequest == SET_ADDRESS &&
                ds->setup.wValue <= 127 &&
                (ds->visibleState == DEFAULT ||
                 ds->visibleState == ADDRESS)) {
                USBDsetDeviceAddress(SET_ADDRESS_REQUEST, ds->setup.wValue);
                result = REQUEST_SUCCESS;
            }
            . . .
        }
    }
    else if ((ds->setup.bmRequestType & REQUEST_TYPE) != STD_REQ) {
        if (ds->callback->ClassNoDataSetup)
            result = ds->callback->ClassNoDataSetup(&ds->setup);
    }
    if (result == REQUEST_SUCCESS) {
```

```

        ds->controlState = WAIT_STATUS_IN;
        USBDwrite0(0, 0);
        USBDendPoint0RxSTALL();
        USBDendPoint0TxVALID();
    }
    else {
        ds->controlState = IDLE;
        USBDendPoint0RxSTALL();
        USBDendPoint0TxSTALL();
    }
}

```

Po rozpoznaniu żądania SET_ADDRESS wywołujemy funkcję `USBDsetDeviceAddress` w celu ustawienia adresu urządzenia. Jeśli rozpoznamy żądanie niestandardowe i jest zainstalowana funkcja zwrotna `ClassNoDataSetup` obsługująca takie żądania, to delegujemy obsługę żądania do tej funkcji. Na zakończenie, jeśli żądanie ma być obsłużone, przechodzimy do fazy statusu, czyli przechodzimy do stanu `WAIT_STATUS_IN`, wstawiamy do wysłania pusty pakiet za pomocą wywołania funkcji `USBDwrite0` z zerowymi argumentami, blokujemy wszystkie transakcje OUT (wywołanie `USBDendPoint0RxSTALL`) i zezwalamy na transakcję IN (wywołanie `USBDendPoint0TxVALID`). Jeśli natomiast żądanie ma być odrzucone, wracamy do stanu `IDLE` i blokujemy wszystkie transakcje OUT i IN.

Przejdźmy teraz do funkcji `InDataSetup0`. Zmienna lokalna `result` odgrywa tę samą rolę jak w funkcji `NoDataSetup0`. Po rozpoznaniu żądania `GET_DESCRIPTOR`, jeśli jest zainstalowana funkcja zwrotna `GetDescriptor`, delegujemy obsługę żądania do tej funkcji. Jeśli rozpoznamy żądanie niestandardowe i jest zainstalowana funkcja zwrotna `ClassInDataSetup`, to delegujemy obsługę żądania do tej funkcji.

```

void InDataSetup0(usb_device_state_t *ds) {
    static uint16_t buffer16;
    static uint8_t buffer8;
    usb_result_t result;
    result = REQUEST_ERROR;
    if ((ds->setup.bmRequestType == IN_STD_REQ_DEV ||
         ds->setup.bmRequestType == IN_STD_REQ_IF) &&
        ds->setup.bRequest == GET_DESCRIPTOR) {
        if (ds->callback->GetDescriptor)
            result = ds->callback->GetDescriptor(ds->setup.wValue,
                                                   ds->setup.wIndex,
                                                   &ds->txdata,
                                                   &ds->length);
    }
    . . .
    else if ((ds->setup.bmRequestType & REQUEST_TYPE) != STD_REQ) {

```

```

    if (ds->callback->ClassInDataSetup)
        result = ds->callback->ClassInDataSetup(&ds->setup,
                                                &ds->txdata,
                                                &ds->length);
    }
    if (result == REQUEST_SUCCESS) {
        ds->controlState = DATA_IN;
        if (ds->length > ds->setup.wLength)
            ds->length = ds->setup.wLength;
        DataStageIn0(ds);
    }
    else {
        ds->controlState = IDLE;
        USBDendPoint0TxSTALL();
        USBDendPoint0RxSTALL();
    }
}

```

Przed zakończeniem funkcji InDataSetup0, jeśli żądanie ma być obsłużone, przechodzimy do fazy transmisji danych, czyli przechodzimy do stanu DATA_IN. Jeśli urządzenie ma do wysłania więcej danych, niż zażądał kontroler (np. kontroler zażądał tylko początkowego fragmentu deskryptora), to ograniczamy rozmiar danych do wysłania. W celu obsłużenia fazy transmisji danych wywołujemy funkcję DataStageIn0. Jeśli natomiast żądanie ma być odrzucone, wracamy do stanu IDLE i blokujemy wszystkie transakcje.

Przejdźmy teraz do funkcji OutDataSetup0. Zmienna lokalna result odgrywa tę samą rolę co poprzednio. Jeśli rozpoznamy żądanie niestandardowe i jest zainstalowana funkcja zwrotna ClassOutDataSetup, to delegujemy obsługę żądania do tej funkcji.

```

void OutDataSetup0(usb_device_state_t *ds) {
    usb_result_t result;
    result = REQUEST_ERROR;
    . . .
    if ((ds->setup.bmRequestType & REQUEST_TYPE) != STD_REQ) {
        if (ds->callback->ClassOutDataSetup)
            result = ds->callback->ClassOutDataSetup(&ds->setup,
                                                        &ds->rxdata);
    }
    if (result == REQUEST_SUCCESS) {
        ds->controlState = DATA_OUT;
        ds->length = ds->setup.wLength;
        USBDwrite0(0, 0);
        USBDendPoint0TxVALID();
    }
}

```

```

        USBDendPoint0RxVALID() ;
    }
    else {
        ds->controlState = IDLE;
        USBDendPoint0TxSTALL();
        USBDendPoint0RxSTALL();
    }
}

```

Przed zakończeniem funkcji OutDataSetup0, jeśli żądanie ma być obsłużone, przechodzimy do fazy transmisji danych, czyli przechodzimy do stanu DATA_OUT. Ustawiamy rozmiar danych, które mają być odebrane przez urządzenie, na podstawie wartości pola wLength struktury setup. Następnie, wywołując funkcję USBDwrite0 z zerowymi argumentami, wstawiamy do wysłania pusty pakiet. W ten sposób przygotowujemy się do obsłużenia transakcji IN fazy statusu. Kontroler kończy transmisję danych, inicjując tę transakcję, ale może ją też zainicjować wcześniej, przed wysłaniem do urządzenia wszystkich danych, aby przerwać transmisję. Na koniec zezwalamy na transakcje IN i OUT. Jeśli żądanie ma być odrzucone, wracamy do stanu IDLE i blokujemy wszystkie transakcje.

Funkcja In0 jest wywoływana po zakończeniu transakcji IN skierowanej do punktu końcowego zero. Jeśli jesteśmy w trakcie wysyłania danych, czyli automat jest w stanie DATA_IN lub LAST_DATA_IN, to delegujemy obsługę tego zdarzenia do funkcji DataStageIn0. Jeśli jesteśmy w trakcie odbierania danych, czyli automat jest w stanie DATA_OUT, to oznacza, że kontroler właśnie chce przerwać tę transmisję i zakończyć żądanie. W tym przypadku wracamy do stanu spoczynkowego IDLE i blokujemy dalsze transakcje IN, natomiast zezwalamy na kolejną transakcję SETUP (oraz OUT, ale tej transakcji się nie spodziewamy).

```

void In0(usb_device_state_t *ds) {
    switch (ds->controlState) {
        case DATA_IN:
        case LAST_DATA_IN:
            DataStageIn0(ds);
            return;
        case DATA_OUT:
            ds->controlState = IDLE;
            USBDendPoint0TxSTALL();
            USBDendPoint0RxVALID();
            return;
        case WAIT_STATUS_IN:
            if (ds->setup.bmRequestType == OUT_STD_REQ_DEV &&
                ds->setup.bRequest == SET_ADDRESS &&
                ds->setup.wValue <= 127 &&
                ds->setup.wIndex == 0 &&

```

```

        ds->setup.wLength == 0 &&
        (ds->visibleState == DEFAULT ||
         ds->visibleState == ADDRESS)) {
    USBDsetDeviceAddress(SET_ADDRESS_STATUS, ds->setup.wValue);
    if (ds->visibleState == DEFAULT && ds->setup.wValue != 0)
        ds->visibleState = ADDRESS;
    else if (ds->visibleState == ADDRESS && ds->setup.wValue == 0)
        ds->visibleState = DEFAULT;
}
else if ((ds->setup.bmRequestType & REQUEST_TYPE) != STD_REQ) {
    if (ds->callback->ClassStatusIn)
        ds->callback->ClassStatusIn(&ds->setup);
}
ds->controlState = IDLE;
USBDendPoint0TxSTALL();
USBDendPoint0RxVALID();
return;
default:
    ds->controlState = IDLE;
    USBDendPoint0TxSTALL();
    USBDendPoint0RxSTALL();
}
}

```

Jeśli funkcja `In0` zostanie wywołana, gdy automat jest w stanie `WAIT_STATUS_IN`, oznacza to zakończenie żądania bez transmisji danych lub żądania, w którym były przesyłane dane z kontrolera do urządzenia. W tym przypadku również wracamy do stanu spoczynkowego `IDLE` i blokujemy dalsze transakcje IN, natomiast zezwalamy na transakcję SETUP (oraz OUT). Dodatkowo musimy w tym stanie obsłużyć dwie sytuacje: zakończenie żądania `SET_ADDRESS` i zakończenie żądania niestandardowego. Standard stanowi, że nowy adres należy przypisać urządzeniu dopiero po zakończeniu fazy statusu żądania `SET_ADDRESS`. Dlatego, jeśli zachodzi ta sytuacja, wywołujemy funkcję `USBDsetDeviceAddress` i ustawiamy odpowiednio stan urządzenia. Dla żądania niestandardowego wywołujemy funkcję `ClassStatusIn`, aby poinformować aplikację, że zakończyło się odbieranie danych.

Jeśli funkcja `In0` zostanie wywołana, gdy automat nie znajduje się w żadnym z wymienionych powyżej stanów, to uznajemy, że jest to błąd – wracamy do stanu `IDLE` i blokujemy wszystkie transakcje.

Funkcja `DataStageIn0` obsługuje wysyłanie danych sterujących przez urządzenie. Jeśli została wywołana w stanie `DATA_IN`, to obliczamy rozmiar danych, jakie mają być wysłane w kolejnej transakcji IN. Jeśli całkowity rozmiar wszystkich wysyłanych danych jest mniejszy niż określony w polu `wLength` struktury `setup` żądania, to urządzenie powinno zakończyć transmisję pakietem krótszym od maksymalnego. W szczególnym przypadku, gdy rozmiar wysyłanych danych jest wielokrotnością

tego maksymalnego rozmiaru, należy wysłać pusty pakiet. Jeśli wysyłany jest ostatni pakiet, to nastepnym stanem automatu będzie LAST_DATA_IN. W przeciwnym przypadku automat pozostaje w stanie DATA_IN. Za pomocą funkcji USBDwrite0 wstawiamy pakiet do wysłania, a następnie zezwalamy na transakcję IN. Zezwalamy też na transakcję OUT, gdyż kontroler może w ten sposób chcieć przerwać transmisję kolejnych pakietów. Na koniec zmniejszamy odpowiednio rozmiar danych do wysłania w składowej length i przesuwamy wskaźnik w składowej txdata, aby wskaływał na kolejną porcję do wysłania.

```
void DataStageIn0(usb_device_state_t *ds) {
    if (ds->controlState == DATA_IN) {
        uint16_t length;
        if (ds->length < ds->maxPacketSize0 ||
            (ds->length == ds->maxPacketSize0 &&
            ds->setup.wLength % ds->maxPacketSize0 == 0)) {
            ds->controlState = LAST_DATA_IN;
            length = ds->length;
        }
        else
            length = ds->maxPacketSize0;
        USBDwrite0(ds->txdata, length);
        USBDDendPoint0TxVALID();
        USBDDendPoint0RxVALID();
        ds->length -= length;
        ds->txdata += length;
    }
    else if (ds->controlState == LAST_DATA_IN) {
        ds->controlState = WAIT_STATUS_OUT;
        USBDDendPoint0TxSTALL();
        USBDDendPoint0RxVALID();
    }
}
```

Jeśli funkcja DataStageIn0 została wywołana, gdy automat znajduje się w stanie LAST_DATA_IN, to wysłane zostały wszystkie dane (zrealizowana została ostatnia transakcja IN). Przechodzimy do fazy statusu, czyli do stanu WAIT_STATUS_OUT. Blokujemy dalsze transakcje IN i zezwalamy na transakcję OUT.

Funkcja Out0 jest wywoływana po rozpoznaniu transakcji OUT skierowanej do punktu końcowego zero. Jeśli jesteśmy w trakcie odbierania danych, czyli automat jest w stanie DATA_OUT, to delegujemy obsługę tego zdarzenia do funkcji DataStageOut0. Jeśli automat jest w stanie WAIT_STATUS_OUT, to oznacza zakończenie żądania, w którym dane były przesyłane z urządzenia do kontrolera. Jeśli automat jest w stanie DATA_IN lub LAST_DATA_IN, to oznacza, że kontroler przerwał fazę odbierania danych. W każdym z tych trzech przypadków wracamy do stanu

IDLE, blokujemy transakcje IN i zezwalamy na transakcję SETUP (oraz OUT, ale tej transakcji się nie spodziewamy). Jeśli automat nie znajduje się w żadnym z wymienionych wyżej stanów, to traktujemy to jako błąd, wracamy do stanu IDLE i blokujemy wszystkie transakcje.

```
void Out0(usb_device_state_t *ds) {
    switch (ds->controlState) {
        case DATA_OUT:
            DataStageOut0(ds);
            return;
        case WAIT_STATUS_OUT:
        case DATA_IN:
        case LAST_DATA_IN:
            ds->controlState = IDLE;
            USBDendPoint0TxSTALL();
            USBDendPoint0RxVALID();
            return;
        default:
            ds->controlState = IDLE;
            USBDendPoint0TxSTALL();
            USBDendPoint0RxSTALL();
    }
}
```

Funkcja DataStageOut0 obsługuje odbieranie danych sterujących przez urządzenie. Zaczynamy od przeczytania odebranego pakietu za pomocą funkcji USBDread0. Następnie odpowiednio modyfikujemy w składowej length rozmiar danych, jakie pozostały jeszcze do odebrania, oraz przesuwamy wskaźnik rxdata, aby wskazywał na miejsce w buforze, gdzie ma być umieszczona następna porcja danych. Jeśli już odebraliśmy wszystkie dane (bufor odbiorczy jest pełny), to przechodzimy do fazy statusu, czyli stanu WAIT_STATUS_IN. W tym miejscu powinniśmy wstawić do wysłania pusty pakiet, ale zrobiliśmy to już w funkcji OutDataSetup0. Blokujemy transakcje OUT i zezwalamy na transakcję IN. Jeśli pozostały jeszcze jakieś dane do odebrania, to pozostajemy w stanie DATA_OUT i zezwalamy na wszystkie transakcje, gdyż czekamy na kolejne dane (transakcja IN) albo kontroler może zasygnalizować koniec danych (inicjując transakcję OUT).

```
void DataStageOut0(usb_device_state_t *ds) {
    uint16_t length;
    length = USBDread0(ds->rxdata, ds->length);
    ds->length -= length;
    ds->rxdata += length;
    if (ds->length == 0) {
        ds->controlState = WAIT_STATUS_IN;
        USBDendPoint0RxSTALL();
    }
}
```

```

        USBDendPoint0TxVALID() ;
    }
    else {
        USBDendPoint0RxVALID() ;
        USBDendPoint0TxVALID() ;
    }
}

```

4.2. Biblioteka libusb

Biblioteka libusb pozwala na komunikowanie się z urządzeniem USB z poziomu aplikacji użytkownika. Nie trzeba implementować sterownika, który musiałby być uruchamiany w przestrzeni jądra systemu operacyjnego, a co z tym związane musiałby działać z uprawnieniami administratora. Biblioteka jest dostępna dla wszystkich popularnych systemów operacyjnych: Linux, Darwin (Mac OS X), Windows, OpenBSD i NetBSD, a jej tekst źródłowy jest otwarty (ang. *open source*). We wszystkich systemach zachowano ten sam interfejs programistyczny, co bardzo ułatwia przenoszenie oprogramowania. Wiele popularnych programów bazuje na tej bibliotece, w tym m.in. wspomniany w rozdziale 2 program lsusb. Prezentowany tu opis nie jest wyczerpującym podręcznikiem. Skupiam się na libusb w wersji 1.0, której interfejs zdefiniowany jest w pliku nagłówkowym *libusb-1.0/libusb.h*. Omawiam przede wszystkim funkcje potrzebne w projekcie, którym kończy się ten rozdział. Dalszych informacji należy szukać na stronach <http://www.libusb.org> i <http://libusb.sourceforge.net/api-1.0>. Dla systemów Windows dostępne są też biblioteki o funkcjonalności bardzo podobnej do libusb. Są to na przykład libusb-win32 bazująca na libusb w wersji 0.1 (API nie jest w pełni kompatybilne z wersją 1.0) oraz libusbK (zupełnie inne API).

Tab. 4.2. Kody błędów biblioteki libusb

Stała	Wartość	Opis
LIBUSB_SUCCESS	0	Poprawne zakończenie, brak błędu
LIBUSB_ERROR_IO	-1	Błąd operacji wejścia-wyjścia
LIBUSB_ERROR_INVALID_PARAM	-2	Błędna wartość parametru
LIBUSB_ERROR_ACCESS	-3	Brak dostępu, niedostateczne uprawnienia
LIBUSB_ERROR_NO_DEVICE	-4	Brak urządzenia, urządzenie odłączone
LIBUSB_ERROR_NOT_FOUND	-5	Podano błędny numer konfiguracji, interfejsu, punktu końcowego itp.
LIBUSB_ERROR_BUSY	-6	Zasób zajęty
LIBUSB_ERROR_TIMEOUT	-7	Przekroczony limit czasu
LIBUSB_ERROR_OVERFLOW	-8	Przepelenie
LIBUSB_ERROR_PIPE	-9	Nieobsługiwane żądanie lub punkt końcowy wstrzymany
LIBUSB_ERROR_INTERRUPTED	-10	Wywołanie systemowe przerwane (prawdopodobnie przez sygnał)
LIBUSB_ERROR_NO_MEM	-11	Brak pamięci
LIBUSB_ERROR_NOT_SUPPORTED	-12	Operacja niewspierana lub niezaimplementowana w tym systemie operacyjnym
LIBUSB_ERROR_OTHER	-99	Żaden z wyżej wymienionych błędów

Większość funkcji biblioteki libusb zwraca wartość zero, gdy wywołanie zakończyło się powodzeniem lub kod błędu mający wartość ujemną. Należy dokładnie sprawdzać wartości zwracane przez wszystkie funkcje. Urządzenie USB może zostać wyjęte przez użytkownika z gniazda w każdej chwili. Nie mamy zatem żadnej gwarancji, że jeśli jakaś funkcja wykonała się poprawnie, to za chwilę ta sama funkcja lub inna też wykona się poprawnie. Przyczyną niepowodzenia wywołania funkcji libusb może być zajęcie urządzenia przez inny program. Użytkownik może też nie mieć dostatecznych uprawnień do urządzenia – patrz opis programu udev w rozdziale 2.7.1. Kody błędów libusb zebrane są w **tabeli 4.2**.

4.2.1. Inicjowanie i zwalnianie biblioteki

Jest dość oczywiste, że wiele programów może chcieć równocześnie używać biblioteki libusb. Każde wywołanie biblioteki wiąże się z potencjalną zmianą jej stanu wewnętrznego. Zatem każde wywołanie odbywa się w pewnym kontekście, który wynika z historii poprzednich wywołań funkcji bibliotecznych. Kontekst ten jest przechowywany w strukturze typu `libusb_context`. Wskaźnik do tej struktury podaje się jako parametr wywołania funkcji, które wymagają kontekstu. Podanie zerowego wskaźnika oznacza chcę użycia domyślnego kontekstu, który jest wspólny dla wszystkich aplikacji korzystających z libusb. Używanie domyślnego kontekstu ma sens tylko wtedy, gdy w danym momencie tylko jedna aplikacja korzysta z biblioteki. Warunek ten jest trudny do spełnienia w środowisku, w którym można jednocześnie uruchomić wiele programów, czyli praktycznie we wszystkich współczesnych systemach operacyjnych. Dlatego raczej nie zaleca się korzystania z domyślnego kontekstu.

```
int libusb_init(libusb_context **pctx);
```

Funkcja `libusb_init` otwiera nową sesję biblioteki libusb i inicjuje kontekst wywołań tej biblioteki. Musi ona być wywołana przed wywołaniem jakiejkolwiek innej funkcji z libusb. Parametr `pctx` jest adresem wskaźnika na kontekst. Aplikacja przechowuje tylko wskaźnik do struktury z kontekstem. Sama struktura jest alokowana przez bibliotekę. Jeżeli podano adres zerowy, to zostanie utworzony i zainicjowany domyślny kontekst. Jeśli domyślny kontekst już istnieje, to zostanie on użyty bez ponownego inicjowania. Funkcja zwraca zero, gdy zakończyła się powodzeniem. W przeciwnym przypadku zwraca kod błędu i wtedy nie wolno korzystać ze wskaźnika, którego adres podano jako parametr.

```
void libusb_exit(libusb_context *ctx);
```

Funkcja `libusb_exit` zamyka sesję biblioteki libusb oraz zwalnia wszystkie zasoby alokowane przez aplikację i związane z kontekstem wywołań tej biblioteki. Parametr `ctx` jest wskaźnikiem na strukturę przechowującą kontekst. Musi to być wskaźnik, którego adres przekazano w funkcji `libusb_init`. Podanie wskaźnika zerowego oznacza chcę użycia domyślnego kontekstu. Funkcja ta nie zwraca żadnej wartości. Poniższy wydruk przedstawia prosty przykład użycia omówionych w tym podrzędiale funkcji. Jeśli inicjowanie biblioteki się nie powiedzie, to zostanie wywo-

łana funkcja `error`, która wypisuje na standardowe wyjście komunikat o błędzie. W przypadku poprawnego zainicjowania biblioteki wywoływana jest funkcja `find_device`, której zadaniem jest znalezienie interesującego nas urządzenia i która zostanie omówiona w kolejnym podrozdziale. Pełny tekst źródłowy tego przykładu znajduje się w pliku `ex_libusb.c`, który jest w archiwum z przykładami w katalogu `./make/linux`.

```
int init_usb() {
    libusb_context *context;
    int result;
    result = libusb_init(&context);
    if (result) {
        error("libusb_init", result);
    }
    else {
        result = find_device(context);
        libusb_exit(context);
    }
    return result;
}
```

4.2.2. Wyszukiwanie i otwieranie urządzenia

Po zainicjowaniu biblioteki libusb trzeba odszukać urządzenie, z którym chcemy się komunikować. Urządzenie reprezentowane jest za pomocą struktury typu `libusb_device`. Struktura ta przechowuje licznik referencji. Struktura przestaje być potrzebna i pamięć przez nią zajmowana jest zwalniana, gdy licznik referencji zmniejszy się do zera, co oznacza, że żaden program nie korzysta już z urządzenia, które jest reprezentowane przez tę strukturę.

```
ssize_t libusb_get_device_list(libusb_context *ctx,
                               libusb_device ***plist);
```

Funkcja `libusb_get_device_list` odczytuje listę aktualnie podłączonych urządzeń. Parametr `ctx` jest wskaźnikiem na kontekst lub wskaźnikiem zerowym, jeśli ma być użyty domyślny kontekst. Parametr `plist` jest adresem wskaźnika, do którego, w przypadku pomyślnego zakończenia funkcji, zostanie przypisany adres listy (reprezentowanej jako tablica języka C) struktur typu `libusb_device`. Dla każdego urządzenia na liście zwiększany jest licznik referencji. Funkcja ta zwraca liczbę urządzeń na liście lub kod błędu.

```
void libusb_free_device_list(libusb_device **list,
                             int unref_devices);
```

Lista urządzeń jest alokowana przez bibliotekę i musi zostać zwolniona za pomocą funkcji `libusb_free_device_list`. Parametr `list` jest wskaźnikiem na zwalnianą

listę. Parametr `unref_devices` określa, czy ma być przy tym zmniejszony licznik referencji dla każdego z urządzeń znajdujących się na liście. Funkcja ta nie zwraca żadnej wartości.

Po znalezieniu urządzenia na liście, aby móc się komunikować z jego punktami końcowymi, należy go otworzyć, podobnie jak otwiera się plik, czyli uzyskać jego uchwyt, który w przypadku urządzenia USB jest wskaźnikiem na strukturę typu `libusb_device_handle`. Uchwyt urządzenia podaje się jako parametr funkcji przesyłających dane.

```
int libusb_open(libusb_device *dev,  
                libusb_device_handle **phandle);
```

Uchwyt urządzenia uzyskuje się za pomocą funkcji `libusb_open`. Parametr `dev` jest wskaźnikiem na strukturę reprezentującą urządzenie. Parametr `phandle` jest adresem wskaźnika, do którego ma być przypisany uchwyt urządzenia. Funkcja ta zwiększa licznik referencji urządzenia. Jest to operacja nieblokująca – nie odbywa się żadna komunikacja z urządzeniem. Funkcja zwraca zero, gdy zakończyła się powodzeniem. W przeciwnym przypadku zwraca kod błędu i wtedy nie wolno korzystać ze wskaźnika, którego adres podano jako drugi parametr.

```
void libusb_close(libusb_device_handle *handle);
```

Funkcja `libusb_close` zwalnia uchwyt urządzenia. Powinna być wywołana przed zakończeniem aplikacji na wszystkich uchwytych utworzonych za pomocą funkcji `libusb_open` lub opisanej poniżej funkcji `libusb_open_device_with_vid_pid`. Parametr `handle` jest uchwytem, który należy zwolnić. Funkcja ta zmniejsza licznik referencji urządzenia. Jest nieblokująca. Nie zwraca żadnej wartości.

Wyszukiwanie urządzeń najlepiej jest jednak zademonstrować na przykładzie. W funkcji `find_device` odczytujemy listę podłączonych urządzeń. Dla każdego urządzenia wywołujemy funkcję `check_device`, która sprawdza, czy jest to urządzenie, które chcemy obsługiwać. Jeśli takie urządzenie zostało znalezione, to wskaźnik `device` przyjmuje wartość niezerową i próbujemy uzyskać uchwyt tego urządzenia. Po otwarciu interesującego nas urządzenia możemy zwolnić listę urządzeń. Następnie wywołujemy funkcję `use_device`, w której można wybrać konfigurację urządzenia, uaktywnić interfejs i komunikować się z punktami końcowymi. Na koniec zwalniamy uchwyt urządzenia.

```
int find_device(libusb_context *context) {  
    libusb_device **device_list, *device;  
    libusb_device_handle *handle;  
    ssize_t count, i;  
    int result;  
    count = libusb_get_device_list(context, &device_list);  
    if (count < 0) {  
        error("libusb_get_device_list", count);  
    } else {  
        for (i = 0; i < count; i++) {  
            device = device_list[i];  
            if (check_device(device)) {  
                handle = libusb_open(device, &result);  
                if (handle == NULL) {  
                    error("libusb_open", result);  
                } else {  
                    use_device(handle);  
                }  
            }  
        }  
    }  
}
```

```

        return count;
    }
    for (device = NULL, i = 0; i < count; i++) {
        if (check_device(device_list[i])) {
            device = device_list[i];
            break;
        }
    }
    if (device)
        result = libusb_open(device, &handle);
    else
        result = 1;
    if (result < 0)
        error("libusb_open", result);
    libusb_free_device_list(device_list, 1);
    if (result == 0) {
        result = use_device(handle);
        libusb_close(handle);
    }
    return result;
}

```

W funkcji `check_device` odczytujemy deskryptor urządzenia za pomocą funkcji `libusb_get_device_descriptor`. Zawartość tego deskryptora pozwala nam zidentyfikować urządzenie. W tym konkretnym przykładzie poszukujemy urządzenia z identyfikatorem producenta 0x0483, czyli identyfikatorem, którym posługuje się firma STMicroelectronics. Funkcja zwraca jedynkę, gdy urządzenie zostało znalezione, a zero w przeciwnym przypadku.

```

static struct libusb_device_descriptor device_descriptor;

int check_device(libusb_device *device) {
    int result;
    result = libusb_get_device_descriptor(device, &device_descriptor);
    if (result) {
        error("libusb_get_device_descriptor", result);
        return 0;
    }
    else if (device_descriptor.idVendor == 0x0483)
        return 1;
    else
        return 0;
}

```

Wyżej zaprezentowana procedura znajdowania urządzenia i uzyskiwania jego uchwytu jest bardzo ogólna. Umożliwia zastosowanie wielu kryteriów wyszukiwania. Można ją rozbudować, aby znajdowała nie tylko pierwsze, ale wszystkie urządzenia spełniające zadane kryteria. Jest natomiast dość skomplikowana. W niektórych sytuacjach można ją zastąpić wywołaniem tylko jednej funkcji.

```
libusb_device_handle *  
libusb_open_device_with_vid_pid(libusb_context *ctx,  
                                 uint16_t vendor_id,  
                                 uint16_t product_id);
```

Funkcja `libusb_open_device_with_vid_pid` pozwala łatwo uzyskać uchwyt urządzenia, gdy się zna jego identyfikator producenta `vendor_id` i produktu `product_id`. Parametr `ctx` jest wskaźnikiem na kontekst lub wskaźnikiem zerowym, jeśli ma być użyty domyślny kontekst. Funkcja zwraca uchwyt do urządzenia lub wartość zero (`NULL`), gdy żądane urządzenie nie zostało znalezione lub wystąpił inny błąd. Funkcja ta ma pewną wadę. Jeśli podłączono więcej niż jedno urządzenie o tych samych wartościach identyfikatorów, zostanie znalezione tylko jedno z nich. Rozwiązanie pozbawione tej wady, ale bardziej pracochłonne, zostało omówione wyżej i polega na wywołaniu funkcji `libusb_get_device_list` w celu otrzymania listy wszystkich urządzeń, a następnie odszukaniu na tej liście urządzenia, które nas interesuje i uzyskaniu uchwytu za pomocą funkcji `libusb_open`.

4.2.3. Wybieranie konfiguracji i rezerwowanie interfejsu

Jak pamiętamy, urządzenie może mieć wiele konfiguracji, które z kolei mogą udostępniać wiele interfejsów. Biblioteka `libusb` dostarcza kilku funkcji operujących na konfiguracjach i interfejsach.

```
int libusb_set_configuration(libusb_device_handle *handle,  
                           int configuration);
```

Funkcja `libusb_set_configuration` ustawia konfigurację urządzenia. Parametr `handle` jest uchwytem urządzenia. Parametr `configuration` zawiera numer konfiguracji, która ma być aktywowana, czyli wartość z pola `bConfigurationValue` odpowiedniego deskryptora konfiguracji. Gdy aktualna konfiguracja ma zostać zdezaktywowana, parametr ten, zgodnie ze standardem USB, powinien przyjąć wartość zero. Dokumentacja `libusb` nakazuje jednak podawać w tym przypadku wartość minus jeden. Testy pokazują, że obie wartości działają tak samo. Funkcja ta zwraca zero, gdy zakończyła się powodzeniem, w przeciwnym przypadku zwraca odpowiedni kod błędu. Jest to operacja blokująca – wysyłane jest żądanie `SET_CONFIGURATION` i funkcja kończy działanie dopiero po zakończeniu obsługi tego żądania. Nie można zmienić aktualnej konfiguracji, jeśli jakiś interfejs związany z tą konfiguracją został zarezerwowany przez inny program. Jeśli interfejs został zarezerwowany przez ten sam program, który usiłuje zmienić konfigurację, to najpierw trzeba zwolnić rezerwację interfejsu za pomocą funkcji `libusb_release_interface`.

```
int libusb_claim_interface(libusb_device_handle *handle, int iface);
```

Funkcja `libusb_claim_interface` rezerwuje interfejs urządzenia dla operacji wejścia-wyjścia i musi zostać wywołana, zanim zaczniemy wywoływać funkcje przesyłające dane do lub z punktów końcowych tego interfejsu. Parametr `handle` jest uchwytem urządzenia. Parametr `iface` jest numerem interfejsu, który ma być zarezerwowany. Jest to wartość umieszczona w polu `bInterfaceNumber` deskryptora interfejsu. Rezerwacja interfejsu ma na celu poinformowanie systemu operacyjnego, że aplikacja chce przejąć kontrolę nad wskazanym interfejsem. Jest to operacja nieblokującą – nie wymaga żadnej komunikacji z urządzeniem. Funkcja zwraca zero, gdy rezerwacja powiodła się, albo odpowiedni kod błędu, gdy nie można zarezerwować interfejsu, bo wskazane urządzenie zostało odłączone, podano zły numer interfejsu, inny program zarezerwował już ten interfejs lub wystąpił inny błąd. Legalne jest ponowne zarezerwowanie już zarezerwowanego interfejsu przez ten sam program. Funkcja nic wtedy nie robi i zwraca zero.

```
int libusb_release_interface(libusb_device_handle *handle,
                             int iface);
```

Funkcja `libusb_release_interface` zwalnia rezerwację interfejsu. Parametr `handle` jest uchwytem urządzenia. Parametr `iface` jest numerem interfejsu, którego rezerwacja ma zostać zwolniona. Jest to wartość umieszczona w polu `bInterfaceNumber` deskryptora interfejsu. Należy zwolnić wszystkie zarezerwowane interfejsy przed zwolnieniem uchwytu urządzenia. Jest to operacja blokująca – do interfejsu wysyłane jest żądanie `SET_INTERFACE` ustawiające pierwszy (zwykle o numerze zero) wariant ustawień i funkcja kończy działanie dopiero po zakończeniu obsługi tego żądania. Funkcja ta zwraca zero, gdy zakończyła się sukcesem, albo odpowiedni kod błędu, gdy wskazany interfejs nie jest zarezerwowany, wskazane urządzenie zostało odłączone lub wystąpił inny błąd.

```
int libusb_set_interface_alt_setting(libusb_device_handle *handle,
                                      int iface, int setting);
```

Funkcja `libusb_set_interface_alt_setting` aktywuje wariant ustawień interfejsu. Parametr `handle` jest uchwytem urządzenia. Parametr `iface` jest numerem interfejsu. Parametr `setting` zawiera ustawiany wariant. Aby skutecznie wywołać tę funkcję, interfejs musi być najpierw zarezerwowany. Jest to operacja blokująca – do interfejsu wysyłane jest żądanie `SET_INTERFACE` i funkcja kończy działanie dopiero po zakończeniu obsługi tego żądania. Funkcja ta zwraca zero, gdy zakończyła się sukcesem, albo odpowiedni kod błędu, gdy wskazany interfejs nie jest zarezerwowany, podany wariant ustawień nie istnieje, wskazane urządzenie zostało odłączone lub wystąpił inny błąd.

4.2.4. Przesyłanie blokujące

Biblioteka libusb oferuje dwa rodzaje funkcji realizujących przesyłanie danych między kontrolerem a punktem końcowym: blokujące i nieblokujące. Omówienie

przesyłania nieblokującego odkładam do następnego podrozdziału. W dokumentacji libusb funkcje blokujące są nazywane synchronicznym wejściem-wyjściem. Po wywołaniu takiej funkcji wykonywanie programu jest blokowane, a powrót z funkcji następuje dopiero po zakończeniu obsługi transakcji lub całego żądania. Jest to bardzo proste i wygodne z punktu widzenia programisty, ale wadą takiego podejścia jest marnotrawienie czasu. Zwykle tylko niewielki ułamek mocy obliczeniowej procesora potrzebny jest do obsługi interfejsu USB. Zamiast czekać, program mógłby wykonywać jakieś użyteczne obliczenia. Oczywiście czas ten nie zostanie całkowicie zmarnowany – gdy nasz program czeka, system operacyjny przydzieli procesor innemu programowi. Przesyłanie blokujące możliwe jest tylko dla danych sterujących, masowych i pilnych. Biblioteka udostępnia w tym celu trzy funkcje.

```
int libusb_control_transfer(libusb_device_handle *handle,
                           uint8_t bmRequestType, uint8_t bRequest,
                           uint16_t wValue, uint16_t wIndex,
                           unsigned char *data, uint16_t wLength,
                           unsigned int timeout);
```

Funkcja `libusb_control_transfer` realizuje żądanie skierowane do domyślnego punktu końcowego dla danych sterujących. Parametr `handle` jest uchwytem urządzenia. Parametry `bmRequestType`, `bRequest`, `wValue`, `wIndex` i `wLength` zawierają wartości, które mają być umieszczone w polu danych transakcji SETUP – patrz tabela 1.14. Wartości parametrów podaje się w lokalnym porządku bajtów – ewentualny obowiązek zmiany kolejności bajtów w celu dopasowania do porządku obowiązującego w USB spoczywa na bibliotece libusb. Parametr `data` jest wskaźnikiem na bufor z danymi do wysłania lub bufor, do którego mają zostać zapisane odebrane dane. Kierunek transmisji ustala się na podstawie bitu kierunku w parametrze `bmRequestType`. Parametr `timeout` określa czas oczekiwania na zakończenie obsługi żądania w milisekundach. Wartość zero oznacza brak ograniczenia czasowego, czyli nieskończone czekanie. Funkcja ta zwraca zero, gdy zakończyła się sukcesem, albo odpowiedni kod błędu, gdy został przekroczony czas oczekiwania na zakończenie realizacji żądania, żądanie nie jest obsługiwane przez urządzenie, urządzenie zostało odłączone lub wystąpił inny błąd.

```
int libusb_bulk_transfer(struct libusb_device_handle *handle,
                        unsigned char endpoint, unsigned char *data,
                        int length, int *transferred,
                        unsigned int timeout);
```

Funkcja `libusb_bulk_transfer` realizuje przesłanie danych masowych. Parametr `handle` jest uchwytem urządzenia. Parametr `endpoint` jest adresem punktu końcowego. Parametr `data` jest wskaźnikiem na bufor z danymi do wysłania lub bufor, do którego mają zostać zapisane odebrane dane. Kierunek transmisji ustala się na podstawie bitu kierunku w adresie punktu końcowego. Parametr `length` określa liczbę bajtów do wysłania lub maksymalną liczbę bajtów, które chcemy odebrać (rozmiar bufora). Za pomocą jednego wywołania tej funkcji można przesyłać więcej danych,

niż wynosi maksymalny rozmiar pola danych pakietu dla danego punktu końcowego. Dane zostaną podzielone na pakiety i zostanie zainicjowana odpowiednia liczba transakcji. Parametr `timeout` określa czas oczekiwania na zakończenie całego transferu w milisekundach. Wartość zero oznacza brak ograniczenia czasowego, czyli nieskończone czekanie. Funkcja ta zwraca zero, gdy zakończyła się sukcesem, albo odpowiedni kod błędu, gdy został przekroczyony czas oczekiwania na zakończenie transferu, punkt końcowy jest wstrzymany, urządzenie zaoferowało więcej danych niż wartość parametru `length`, urządzenie zostało odłączone lub wystąpił inny błąd. Ponieważ przesłanie całości danych może być podzielone na wiele transakcji, wystąpienie błędu przekroczenia czasu może oznaczać, że niektóre transakcje zostały zrealizowane i część danych została przesłana. Parametr `transferred` jest wskaźnikiem na zmienną, w której zostanie zapisana liczba bajtów, które rzeczywiście zostały przesłane.

```
int libusb_interrupt_transfer(libusb_device_handle *handle,
                             unsigned char endpoint,
                             unsigned char *data,
                             int length, int *transferred,
                             unsigned int timeout);
```

Funkcja `libusb_interrupt_transfer` realizuje przesłanie danych pilnych. Parametr `handle` jest uchwytem urządzenia. Parametr `endpoint` jest adresem punktu końcowego. Parametr `data` jest wskaźnikiem na bufor z danymi do wysłania lub bufor, do którego mają zostać zapisane odebrane dane. Kierunek transmisji ustala się na podstawie bitu kierunku w adresie punktu końcowego. Parametr `length` określa liczbę bajtów do wysłania lub maksymalną liczbę bajtów, które chcemy odebrać (rozmiar bufora). Za pomocą jednego wywołania tej funkcji można przesyłać więcej danych, niż wynosi maksymalny rozmiar pola danych pakietu dla danego punktu końcowego. Dane zostaną podzielone na pakiety i zostanie zainicjowana odpowiednia liczba transakcji. Parametr `timeout` określa czas oczekiwania na zakończenie całego transferu w milisekundach. Wartość zero oznacza brak ograniczenia czasowego, czyli nieskończone czekanie. Funkcja ta zwraca zero, gdy zakończyła się sukcesem, albo odpowiedni kod błędu, gdy został przekroczyony czas oczekiwania na zakończenie transferu, punkt końcowy jest wstrzymany, urządzenie zaoferowało więcej danych niż wartość parametru `length`, urządzenie zostało odłączone lub wystąpił inny błąd. Ponieważ przesłanie całości danych może być podzielone na wiele transakcji, wystąpienie błędu przekroczenia czasu może oznaczać, że niektóre transakcje zostały zrealizowane i część danych została przesłana. Parametr `transferred` jest wskaźnikiem na zmienną, w której zostanie zapisana liczba bajtów, które rzeczywiście zostały przesłane.

4.2.5. Przesyłanie nieblokujące

Idea przesyłania nieblokującego polega na tym, że wywołujemy funkcję, która tylko inicjuje transmisję danych. Funkcja ta nie czeka na zakończenie transmisi i sterowanie wraca do naszego programu, więc może on kontynuować inne obliczenia, a sama transmisja odbywa się niejako w tle. Inna funkcja pozwala sprawdzić,

czy transmisja się zakończyła. Biblioteka `libusb` nie uruchamia żadnych wątków. Dodatkowy wątek wykonujący transmisję w tle musi być uruchomiony jawnie przez programistę. Wątki wykonują się asynchronicznie i poważnym wyzwaniem jest zapewnienie ich poprawnej synchronizacji. Innym rozwiązaniem, niepotrzebującym dodatkowego wątku, jest programowanie sterowane zdarzeniami, co pozwala na symultaniczne wykonywanie kilku zadań. Program w pętli czeka na zdarzenia i gdy pojawi się jakieś zdarzenie (niekoniecznie związane z USB), to je obsługuje. Zdarzenia pojawiają się często, a ich obsługa musi trwać krótko. Dzięki temu wydaje się, że wszystkie zadania wykonują się równolegle. Programowanie wielowątkowe i programowanie sterowane zdarzeniami nie są przedmiotem tej książki, więc nie będę dalej rozwijał tego tematu.

W dokumentacji `libusb` przesyłanie nieblokujące jest nazywane asynchronicznym wejściem-wyjściem. Głównym powodem jego omawiania jest to, że jest ono możliwe dla wszystkich rodzajów danych, w szczególności dla danych izochronicznych, dla których nie można użyć przesyłania blokującego. Dlatego właśnie omówię interfejs programistyczny przesyłania nieblokującego na przykładzie transmisji izochronicznej. Transmisja podzielona jest na pięć kroków:

- przydzielenie potrzebnej pamięci (ang. *allocation*),
- zainicjowanie struktur danych (ang. *filling*),
- zainicjowanie transmisji (ang. *submission*),
- obsłużenie zakończenia transmisji (ang. *completion handling*),
- zwolnienie przydzielonych zasobów (ang. *deallocation*).

Transmisję opisuje się za pomocą struktury typu `libusb_transfer`, a identyfikuje ją się za pomocą wskaźnika na strukturę tego typu. Większość składowych tej struktury jest ustawiana przez `libusb` i nie należy ich modyfikować bezpośrednio – najlepiej jest posłużyć się opisanymi niżej funkcjami. Do kilku składowych trzeba jednak odwoływać się bezpośrednio. Składowa `status` zawiera kod zakończenia transmisji. Jest modyfikowana przez `libusb`, a przez aplikację może być tylko odczytywana. Zdefiniowane kody zebrane są w **tablicy 4.3**. Składowa `user_data` jest wskaźnikiem, który może być w dowolny sposób wykorzystywany przez aplikację. Zwykle wskazuje na prywatne dane aplikacji związane z daną transmisją. Obowiązek przydzielenia i zwolnienia pamięci dla takich danych spoczywa na aplikacji. Do składowej `flags` przypisuje się znaczniki modyfikujące zachowanie

Tab. 4.3. Kody zakończenia transmisji

Stan	Opis
<code>LIBUSB_TRANSFER_COMPLETED</code>	Transmisja zakończona pomyślnie
<code>LIBUSB_TRANSFER_ERROR</code>	Błąd transmisji
<code>LIBUSB_TRANSFER_TIMED_OUT</code>	Przekroczony czas oczekiwania na zakończenie transmisji
<code>LIBUSB_TRANSFER_CANCELLED</code>	Transmisja anulowana
<code>LIBUSB_TRANSFER_STALL</code>	Nieobsługiwane żądanie (dane sterujące) albo punkt końcowy (dane pilne lub masowe) wstrzymany (ang. <i>stall</i>)
<code>LIBUSB_TRANSFER_NO_DEVICE</code>	Brak urządzenia, urządzenie odłączone
<code>LIBUSB_TRANSFER_OVERFLOW</code>	Przepełnienie, zaoferowanie przez urządzenie większej ilości danych, niż oczekiwano

Funkcja `libusb_fill_iso_transfer` wypełnia strukturę opisującą transmisję izochroniczną. Parametr `transfer` jest wskaźnikiem na tę strukturę. Parametr `handle` jest uchwytem urządzenia. Parametr `endpoint` jest adresem punktu końcowego. Parametr `buffer` jest wskaźnikiem na bufor, w którym są dane do wysłania lub do którego mają być zapisane odebrane dane. Kierunek transmisji jest determinowany bitem kierunku adresu punktu końcowego. Parametr `length` zawiera rozmiar bufora. Parametr `num_iso_packets` zawiera liczbę pakietów, które mają być przesłane. Parametr `callback` jest adresem funkcji zwrotnej, która ma zostać wywołana po zakończeniu transmisji lub wystąpieniu błędu. Parametr `user_data` jest wskaźnikiem, który jest przypisywany do mającej tę samą nazwę składowej struktury opisującej transmisję, a omówionej we wstępie do tego podrozdziału. W zamierzeniu autorów biblioteki parametr ten służy do przekazywania danych aplikacji do funkcji zwrotnej. Parametr `timeout` określa czas oczekiwania na zakończenie całej transmisji w milisekundach. Funkcja ta nie zwraca żadnej wartości. Wspomniana funkcja zwrotna ma sygnaturę

```
void transfer_call_back(struct libusb_transfer *transfer);
```

Jej argumentem jest wskaźnik na strukturę opisującą transmisję. Dzięki temu funkcja zwrotna ma dostęp do wszystkich potrzebnych jej danych.

```
void libusb_set_iso_packet_lengths(struct libusb_transfer *transfer,  
                                    unsigned int length);
```

Funkcja `libusb_set_iso_packet_lengths` ustawia rozmiar pola danych wszystkich pakietów izochronicznych. Parametr `transfer` jest wskaźnikiem na strukturę opisującą transmisję. Parametr `length` zawiera rozmiar, który ma być ustawiony. Liczba pakietów musi być ustalona wcześniej. Funkcja ta nie zwraca żadnej wartości.

```
int libusb_submit_transfer(struct libusb_transfer *transfer);
```

Funkcja `libusb_submit_transfer` inicjuje transmisję. Parametr `transfer` jest wskaźnikiem na strukturę opisującą transmisję. Funkcja ta zwraca zero, gdy zakończyła się sukcesem, albo kod błędu, gdy urządzenie zostało odłączone, transmisja już została zainicjowana lub wystąpił inny błąd.

```
int libusb_handle_events(libusb_context *ctx);
```

Żeby kod biblioteki `libusb` obsługujący zdarzenia związane z transmisją był wywoływany, biblioteka musi być regularnie pobudzana do działania. Jak już wspomniałem, konieczne jest wywoływanie w pętli jakiejś funkcji odbierającej zdarzenia i ich obsługiwanie. W najprostszym przypadku całą pracę wykona za nas funkcja `libusb_handle_events`, która obsługuje wykonanie i zakończenie transmisji. Nie jest ona szczególnie zalecana, gdyż jest to funkcja blokująca, ale dla prostej aplikacji korzystającej z transmisji izochronicznej jest to zupełnie zadowalające rozwiązanie. Parametr `ctx` jest wskaźnikiem na kontekst lub wskaźnikiem zerowym, jeśli ma być użyty domyślny kontekst. Funkcja ta zwraca zero, gdy zakończyła się powodzeniem, lub odpowiedni kod błędu.

```
void libusb_free_transfer(struct libusb_transfer *transfer);
```

Funkcja `libusb_free_transfer` zwalnia zasoby związane z transmisją. Parametr `transfer` jest wskaźnikiem na strukturę opisującą transmisję. Jeśli parametr ten ma wartość zero, funkcja nic nie robi. Nie wolno zwalniać transmisji, która jest aktywna, czyli została zainicjowana, ale się nie skończyła. Funkcji tej nie wolno też wywoływać, jeśli w składowej `flags` struktury wskaazywanej przez parametr `transfer` został ustawiony bit `LIBUSB_TRANSFER_FREE_TRANSFER`. Wtedy zasoby zostaną zwolnione automatycznie przez `libusb` po zakończeniu transmisji. Funkcja ta nie zwraca żadnej wartości.

4.2.6. Pozostałe funkcje

Dotychczas omówione funkcje nie wyczerpują możliwości biblioteki `libusb`. W tym podrozdziale zamieszczam skróty przegląd pozostałych funkcji, ale już bez szczegółowego omawiania ich argumentów – potrzebne informacje można znaleźć w dokumentacji biblioteki.

Adres urządzenia możemy uzyskać za pomocą funkcji `libusb_get_device_address`. Szybkość urządzenia zwraca funkcja `libusb_get_device_speed`. Aby uzyskać maksymalny rozmiar pakietu zadeklarowany w deskryptorze punktu końcowego, należy wywołać funkcję `libusb_get_max_packet_size`. Funkcja `libusb_get_max_iso_packet_size` oblicza, ile bajtów danych można przesyłać w jednej ramce za pomocą izochronicznego punktu końcowego.

Funkcja `libusb_ref_device` zwiększa, a funkcja `libusb_unref_device` zmniejsza licznik referencji urządzenia. Funkcja `libusb_get_device` zwraca wskaźnik do struktury opisującej urządzenie na podstawie uchwytu tego urządzenia. Funkcja `libusb_get_configuration` odczytuje aktualnie ustawioną konfigurację urządzenia. Funkcja `libusb_reset_device` zeruje port USB, do którego podłączone jest podane urządzenie.

Dla wielu urządzeń system operacyjny ma odpowiednie sterowniki, które automatycznie przejmują ich obsługę. Za pomocą funkcji `libusb_kernel_driver_active` można sprawdzić, czy sterownik systemowy jest aktywny. Jeśli jest aktywny, to aby komunikować się z urządzeniem za pomocą `libusb`, trzeba go odłączyć. Służy do tego funkcja `libusb_detach_kernel_driver`. Możliwe jest też ponowne podłączenie sterownika systemowego za pomocą funkcji `libusb_attach_kernel_driver`. Każda z trzech wspomnianych funkcji ma dwa parametry: uchwyt urządzenia i numer interfejsu. Urządzenie będące tematem projektu prezentowanego na końcu tego rozdziału nie należy do żadnej standardowej klasy i nie ma sterownika systemowego, więc nie trzeba go odłączać.

Dowolny deskryptor można odczytać za pomocą funkcji `libusb_get_descriptor`. Ponadto do odczytania ważnych i najczęściej potrzebnych deskryptorów biblioteka ma specyficzne funkcje:

- `libusb_get_device_descriptor` – deskryptor urządzenia,
- `libusb_get_config_descriptor` – deskryptor konfiguracji o podanym indeksie,

- libusb_get_config_descriptor_by_value – deskryptor konfiguracji o podanej wartości z pola bConfigurationValue,
- libusb_get_active_config_descriptor – deskryptor aktualnie wybranej konfiguracji,
- libusb_get_string_descriptor – deskryptor tekstowy,
- libusb_get_string_descriptor_ascii – deskryptor tekstowy jako napis języka C.

Makro libusb_cpu_to_le16 zamienia kolejność bajtów w liczbie 16-bitowej z porządku obowiązującego na lokalnym procesorze (ang. *cpu*) na porządek cienkokoncówkowy (ang. *little-endian*) obowiązujący w USB. Makro libusb_le16_to_cpu wykonuje tę samą operację, ale dla konwencji powinno być używane przy zamianie wartości z porządku obowiązującego w USB na porządek obowiązujący na lokalnym procesorze.

Biblioteka libusb ma też zestaw funkcji do programowania sterowanego zdarzeniami (ale tylko dla systemów uniksowych). Funkcje te współpracują z asynchronicznym wejściem-wyjściem i umożliwiają integrację libusb z innymi interfejsami programistycznymi, które oferują ten paradygmat programowania, na przykład z gniazdami (ang. *sockets*) sieciowymi. Więcej na ten temat można przeczytać w dokumentacji biblioteki.

4.3. Projekt urządzenia własnej klasy

Tematem projektu kończącego ten rozdział jest urządzenie własnej klasy, które ma cztery dwukierunkowe punkty końcowe: zerowy dla danych sterujących, pierwszy dla danych pilnych, drugi dla danych izochronicznych, trzeci dla danych masowych. W celu zademonstrowania działania urządzenia każdy z punktów końcowych realizuje funkcję echa – po prostu odsyła odebrane wcześniej dane.

4.3.1. Deskryptory

Urządzenie musi oczywiście mieć standardowy deskryptor urządzenia. Dobrym pomysłem jest wypełnienie pól bDeviceClass, bDeviceSubClass, bDeviceProtocol zerami i zdefiniowanie funkcji urządzenia na poziomie interfejsu. W polach bInterfaceClass, bInterfaceSubClass, bInterfaceProtocol deskryptora interfejsu wpisuje się wartość 255 oznaczającą klasę, podklasę i protokół dostawcy. Dzięki temu urządzenie może udostępniać też interfejsy innych klas. Prezentowane urządzenie ma jedną konfigurację składającą się z deskryptora konfiguracji, jednego deskryptora interfejsu i sześciu deskryptorów punktów końcowych, po dwa deskryptory jednokierunkowych punktów końcowych dla danych pilnych, izochronicznych i masowych. Domyślny zerowy punkt końcowy dla danych sterujących nie potrzebuje deskryptora.

4.3.2. Żądania

Urządzenie musi obsługiwać standardowe żądania. Oprócz nich można też zdefiniować własne żądania, specyficzne dla dostawcy czy producenta urządzenia. Należy przy tym przestrzegać ogólnych reguł i wzorować się na żądaniach zdefiniowanych

Tab. 4.5. Własne żądania

bmRequestType	bRequest	wValue	wIndex	wLength	Dane
01000001	SET_VALUE 3	Ustawiana wartość	Numer interfejsu	0	Brak
11000001	GET_DATA 4	Indeks	Numer interfejsu	Rozmiar danych	Przesyłane dane
01000001	SET_DATA 5	Indeks	Numer interfejsu	Rozmiar danych	Przesyłane dane

w standardzie. Przykładowe urządzenie obsługuje żądania zamieszczone w **tabeli 4.5**. Parametr `bmRequestType` ma ustawiony bit 6 informujący, że są to własne żądania dostawcy. Ponieważ funkcja urządzenia została zdefiniowana w deskryptorach na poziomie interfejsu, żądania powinny być kierowane do interfejsu. Zatem parametr `bmRequestType` ma ustawiony najmłodszy bit, a parametr `wIndex` zawiera numer interfejsu, do którego kierowane jest żądanie. Konwencja nakazuje również, aby żądania przesyłania danych z urządzenia do kontrolera miały ustawiony bit 7 parametru `bmRequestType`, a pozostałe miały ten bit wyzerowany. Parametr `wLength` powinien zawierać rozmiar danych, które mają być przesłane. Parametr `bRequest` określa rodzaj żądania. Żądanie `SET_VALUE` pozwala przesyłać do urządzenia dwubajtową wartość, która znajduje się w parametrze `wValue`. Żądanie `GET_DATA` służy do przesyłania bloku danych z urządzenia do kontrolera, a żądanie `SET_DATA` – z kontrolera do urządzenia. W obu przypadkach parametr `wValue` określa numer bloku, który ma być przesłany.

4.3.2. Implementacja

ex_vendor_dev.c

Implementacja znajduje się w pliku *ex_vendor_dev.c*. Na początku definiujemy rozmiary buforów dla poszczególnych typów danych. Z wyjątkiem zerowego punktu końcowego dla danych sterujących będą to też maksymalne rozmiary pola danych odpowiednich pakietów. Maksymalny rozmiar pola danych dla zerowego punktu końcowego ustalamy w deskryptorze urządzenia na 32 bajty.

```
#define CNT_BUFF_SIZE 80
#define INT_BUFF_SIZE 48
#define ISO_BUFF_SIZE 40
#define BLK_BUFF_SIZE 64
```

Deskryptor urządzenia jest wypełniony standardowo. Stałe VID i PID zdefiniowane są w pliku *usb_vid_pid.h*. Składowe `idVendor` i `idProduct` mają odpowiednio wartości 0x0483 i 0x5754. Deskryptory tekstowe, których numery zapisane są w składowych `iManufacturer`, `iProduct` i `iSerialNumber`, są bardzo podobne do tych opisanych w poprzednim rozdziale, więc je tu pomijam.

```
static usb_device_descriptor_t const device_descriptor = {
    sizeof(usb_device_descriptor_t), /* bLength */
    DEVICE_DESCRIPTOR, /* bDescriptorType */
```

```

HTOUSBS(0x0200),
0x00,
0x00,
0x00,
32,
HTOUSBS(VID),
HTOUSBS(PID + 4),
HTOUSBS(0x0100),
1,
2,
3,
1
};

};
```

Urządzenie ma jedną konfigurację, która udostępnia jeden interfejs z sześcioma jednokierunkowymi punktami końcowymi. Konfiguracja ta zdefiniowana jest jako struktura `vendor_conf` typu `usb_vendor_conf_t`.

```
typedef struct {
    usb_configuration_descriptor_t    cnf_descr;
    usb_interface_descriptor_t        if_descr;
    usb_endpoint_descriptor_t         ep_descr[6];
} __packed usb_vendor_conf_t;
```

Definicja konfiguracji zamieszczona jest na poniższym wydruku. Stałe `USB_BM_ATTRIBUTES` i `USB_B_MAX_POWER` są zdefiniowane w pliku `board_usb_def.h`. Stała `VENDOR_SPECIFIC` ma wartość 255 i jest zdefiniowana w pliku `usb_def.h`, podobnie jak pozostałe stałe, których wartości wynikają wprost ze standardu USB.

```
static usb_vendor_conf_t const vendor_conf = {
{
    sizeof(usb_configuration_descriptor_t), /* bLength */
    CONFIGURATION_DESCRIPTOR,           /* bDescriptorType */
    HTOUSBS(sizeof(usb_vendor_conf_t)), /* wTotalLength */
    1,                                /* bNumInterfaces */
    1,                                /* bConfigurationValue */
    4,                                /* iConfiguration */
    USB_BM_ATTRIBUTES,                /* bmAttributes */
    USB_B_MAX_POWER                  /* bMaxPower */
},
{
    sizeof(usb_interface_descriptor_t), /* bLength */

```

```

    INTERFACE_DESCRIPTOR,
    0,
    0,
    6,
    VENDOR_SPECIFIC,
    VENDOR_SPECIFIC,
    VENDOR_SPECIFIC,
    4
        /* bDescriptorType */
        /* bInterfaceNumber */
        /* bAlternateSetting */
        /* bNumEndpoints */
        /* bInterfaceClass */
        /* bInterfaceSubClass */
        /* bInterfaceProtocol */
        /* iInterface */

},
{
{
    sizeof(usb_endpoint_descriptor_t), /* bLength */
    ENDPOINT_DESCRIPTOR,           /* bDescriptorType */
    ENDP1 | ENDP_IN,               /* bEndpointAddress */
    INTERRUPT_TRANSFER,           /* bmAttributes */
    HTOUSBS(INT_BUFF_SIZE),       /* wMaxPacketSize */
    1                             /* bInterval */

},
{
    sizeof(usb_endpoint_descriptor_t), /* bLength */
    ENDPOINT_DESCRIPTOR,           /* bDescriptorType */
    ENDP1 | ENDP_OUT,              /* bEndpointAddress */
    INTERRUPT_TRANSFER,           /* bmAttributes */
    HTOUSBS(INT_BUFF_SIZE),       /* wMaxPacketSize */
    1                             /* bInterval */

},
{
    sizeof(usb_endpoint_descriptor_t), /* bLength */
    ENDPOINT_DESCRIPTOR,           /* bDescriptorType */
    ENDP2 | ENDP_IN,               /* bEndpointAddress */
    ISOCHRONOUS_TRANSFER,          /* bmAttributes */
    HTOUSBS(ISO_BUFF_SIZE),        /* wMaxPacketSize */
    1                             /* bInterval */

},
{
    sizeof(usb_endpoint_descriptor_t), /* bLength */
    ENDPOINT_DESCRIPTOR,           /* bDescriptorType */
    ENDP2 | ENDP_OUT,              /* bEndpointAddress */

```

```

ISOCHRONOUS_TRANSFER, /* bmAttributes */
HTOUSBS(ISO_BUFF_SIZE), /* wMaxPacketSize */
1 /* bInterval */

},
{

sizeof(usb_endpoint_descriptor_t), /* bLength */
ENDPOINT_DESCRIPTOR, /* bDescriptorType */
ENDP3 | ENDP_IN, /* bEndpointAddress */
BULK_TRANSFER, /* bmAttributes */
HTOUSBS(BLK_BUFF_SIZE), /* wMaxPacketSize */
0 /* bInterval */

},
{

sizeof(usb_endpoint_descriptor_t), /* bLength */
ENDPOINT_DESCRIPTOR, /* bDescriptorType */
ENDP3 | ENDP_OUT, /* bEndpointAddress */
BULK_TRANSFER, /* bmAttributes */
HTOUSBS(BLK_BUFF_SIZE), /* wMaxPacketSize */
0 /* bInterval */

}
}

```

Definiujemy stałe określające typy własnych żądań, zgodnie z tabelą 4.5.

```
#define SET_VALUE 3  
#define GET_DATA 4  
#define SET DATA 5
```

Definiujemy liczbę buforów dla każdego z punktów końcowych. Stosujemy konwencję, że liczba kończąca nazwę stałej odpowiada numerowi punktu końcowego.

```
#define BUFFER_COUNT0    4  
#define BUFFER_COUNT1    4  
#define BUFFER_COUNT2   112  
#define BUFFER_COUNT3   112
```

Definiujemy bufory dla zerowego punktu końcowego, czyli dla danych sterujących. Tablica `data0` reprezentuje bloki danych, o których jest mowa w żądaniach `GET_DATA` i `SET_DATA`. Index, będący wartością parametru `wValue` tych żądań, jest pierwszym indeksem tej tablicy. Tablica `size0` określa, ile danych jest faktycznie zapamiętywanych w bloku o danym indeksie. Zmienna `value` to wartość, która jest ustawiana za pomocą żądania `SET_VALUE`.

```
static uint16_t size0[BUFFER_COUNT0];
static uint8_t  data0[BUFFER_COUNT0][CNT_BUFF_SIZE];
static uint16_t value;
```

Definiujemy cykliczną kolejkę buforów dla pierwszego punktu końcowego, czyli dla danych pilnych. Kolejkę tę tworzy tablica `data1`. Tablica `size1` określa, ile danych jest faktycznie zapamiętywanych w buforze o danym indeksie. Zmienna `bufFree1` mówi, ile buforów w kolejce jest zajętych. Zmienna `rxdx1` zawiera indeks pierwszego wolnego bufora, do którego zostaną zapisane dane z następnego odebranego pakietu. Zmienna `txidx1` zawiera indeks pierwszego zajętego bufora, który zawiera dane do wysłania w kolejnym pakiecie. Zmienna `txbusy1` przyjmuje wartość niezerową, gdy pakiet danych został wstawiony do wysłania, ale nie został jeszcze wysłany. Jeżeli żaden pakiet nie oczekuje na wysłanie, zmienna ta ma wartość zero.

```
static int          buffered1, rxidx1, txidx1, txbusy1;
static uint16_t    size1[BUFFER_COUNT1];
static uint8_t     data1[BUFFER_COUNT1][INT BUFF SIZE];
```

Definiujemy cykliczną kolejkę buforów dla drugiego punktu końcowego, czyli dla danych izochronicznych. Znaczenie poszczególnych zmiennych jest analogiczne jak odpowiednich zmiennych dla pierwszego punktu końcowego. Nie ma zmiennej informującej, czy jakiś pakiet oczekuje na wysłanie, gdyż dla danych izochronicznych zawsze jakiś pakiet (być może pusty) czeka na wysłanie – patrz też opis dotyczący konfigurowania izochronicznego punktu końcowego za pomocą funkcji `USBEndPointConfigure`.

```
static int          buffered2, rxidx2, txidx2;
static uint16_t    size2[BUFFER_COUNT2];
static uint8_t     data2[BUFFER_COUNT2][ISO_BUFF_SIZE];
```

Definiujemy cykliczną kolejkę buforów dla trzeciego punktu końcowego, czyli dla danych masowych. Znaczenie poszczególnych zmiennych jest analogiczne jak odpowiednich zmiennych dla pierwszego punktu końcowego.

```
static int buffered3, rxidx3, txidx3, txbusy3;  
static uint16_t size3[BUFFER_COUNT3];  
static uint8_t data3[BUFFER_COUNT3][BLK_BUFF_SIZE];
```

Funkcja zwrotna VendorNoDataSetup obsługuje żądanie SET_VALUE. Jej parametr setup jest wskaźnikiem na strukturę opisującą żądanie. Funkcja ta zwraca wartość REQUEST_SUCCESS, jeśli zakończyła się sukcesem, a wartość REQUEST_ERROR, jeśli zostało odebrane błędne żądanie. Otrzymana wartość jest zapisywana w zmiennej value, ale nie jest przez aplikacje nigdzie wykorzystywana.

```

        setup->bRequest == SET_VALUE &&
        setup->wIndex == 0 && /* Interface 0 */
        setup->wLength == 0) {
            value = setup->wValue;
            return REQUEST_SUCCESS;
        }
        return REQUEST_ERROR;
    }
}

```

Funkcja zwrotna `VendorInDataSetup` obsługuje żądanie `GET_DATA`. Jej parametr `setup` jest wskaźnikiem na strukturę opisującą żądanie. Za pomocą parametru `data` zwraca się wskaźnik na dane do przesłania, a za pomocą parametru `length` przekazuje się ich rozmiar. Przekazany wskaźnik jest adresem globalnego bufora, gdyż musi być ważny również po zakończeniu funkcji aż do zakończenia fazy transmisji danych. Funkcja ta zwraca wartość `REQUEST_SUCCESS`, jeśli zakończyła się sukcesem, a wartość `REQUEST_ERROR`, jeśli zostało odebrane błędne żądanie albo bufor o podanym indeksie jest pusty.

```

usb_result_t VendorInDataSetup(usb_setup_packet_t const *setup,
                               uint8_t const **data,
                               uint16_t *length) {
    if (setup->bmRequestType == (DEVICE_TO_HOST |
                                   VENDOR_REQUEST |
                                   INTERFACE_RECIPIENT) &&
        setup->bRequest == GET_DATA &&
        setup->wIndex == 0 && /* Interface 0 */
        setup->wValue < BUFFER_COUNT0 &&
        size0[setup->wValue] > 0) {
        *data = data0[setup->wValue];
        *length = size0[setup->wValue];
        return REQUEST_SUCCESS;
    }
    return REQUEST_ERROR;
}

```

Obsługa żądania `SET_DATA` podzielona jest między dwie funkcje zwrotne. Najpierw wywoływana jest funkcja `VendorOutDataSetup`. Jej parametr `setup` jest wskaźnikiem na strukturę opisującą żądanie. Za pomocą parametru `data` zwraca się wskaźnik na bufor, w którym mają być umieszczone przysłane dane. Wskaźnik ten jest adresem globalnego bufora, gdyż musi być ważny również po zakończeniu funkcji aż do zakończenia fazy transmisji danych. Wyzerowanie odpowiedniego wpisu w tablicy `size0` ma na celu unieważnienie danych przechowywanych w tym buforze. Funkcja zwraca wartość `REQUEST_SUCCESS`, jeśli zakończyła się sukcesem, a wartość `REQUEST_ERROR`, jeśli zostało odebrane błędne żądanie.

```

usb_result_t VendorOutDataSetup(usb_setup_packet_t const *setup,
                               uint8_t **data) {
    if (setup->bmRequestType == (HOST_TO_DEVICE |
                                  VENDOR_REQUEST |
                                  INTERFACE_RECIPIENT) &&
        setup->bRequest == SET_DATA &&
        setup->wIndex == 0 /* Interface 0 */ &&
        setup->wValue < BUFFER_COUNT0 &&
        setup->wLength > 0 &&
        setup->wLength <= CNT_BUFF_SIZE) {
        *data = data0[setup->wValue];
        size0[setup->wValue] = 0;
        return REQUEST_SUCCESS;
    }
    return REQUEST_ERROR;
}

```

Po zakończeniu fazy statusu żądania SET_DATA wywoływana jest funkcja zwrotna VendorStatusIn. Jej parametr setup jest wskaźnikiem na strukturę opisującą żądanie. Pozwala to sprawdzić, czy chodzi o to samo żądanie, dla którego została wywołana funkcja VendorOutDataSetup. Jeśli wszystko się zgadza, to rozmiar odebranych danych zapisujemy w odpowiednim miejscu tablicy size0, co oznacza, że dane w buforze są ważne i można z nich korzystać. Funkcja ta nie zwraca żadnej wartości.

```

void VendorStatusIn(usb_setup_packet_t const *setup) {
    if (setup->bmRequestType == (HOST_TO_DEVICE |
                                  VENDOR_REQUEST |
                                  INTERFACE_RECIPIENT) &&
        setup->bRequest == SET_DATA &&
        setup->wIndex == 0 /* Interface 0 */ &&
        setup->wValue < BUFFER_COUNT0 &&
        setup->wLength > 0 &&
        setup->wLength <= CNT_BUFF_SIZE) {
        size0[setup->wValue] = setup->wLength;
    }
}

```

Bezparametrowa funkcja zwrotna EPinterruptRx jest wywoływana, gdy został odebrany pakiet z danymi pilnymi. Jeśli cykliczna kolejka buforów nie jest pełna, to pakiet odczytujemy za pomocą funkcji USBRead i zapisujemy do pierwszego wolnego bufora w kolejce data1[rxidx1]. Rozmiar odczytanych danych zapisujemy w tablicy size1[rxidx1]. Następnie zwiększamy liczbę zajętych buforów buffe-

red1 i przesuwamy cyklicznie indeks rxidx1 wskazujący na pierwszy wolny bufor. Jeśli cykliczna kolejka buforów jest pełna, to wywołujemy tylko funkcję `USBDread` z zerowymi argumentami, aby opróżnić kolejkę odbiorczą i umożliwić odebranie kolejnego pakietu. Bieżący pakiet nigdzie nie jest zapisywany – zostaje porzucony. Na koniec, jeśli zmienna `txbusy1` ma wartość zero, czyli żaden pakiet nie oczekuje na wysłanie, to wywołujemy funkcję `EPinterruptTx` w celu wstawienia pakietu do kolejki nadawczej. Funkcja `EPinterruptRx` nie zwraca żadnej wartości.

```
void EPinterruptRx() {
    if (buffered1 < BUFFER_COUNT1) {
        size1[rxidx1] = USBDread(ENDP1, data1[rxidx1], INT_BUFF_SIZE);
        ++buffered1;
        rxidx1 = rxidx1 < BUFFER_COUNT1 - 1 ? rxidx1 + 1 : 0;
    }
    else
        USBDread(ENDP1, 0, 0);
    if (!txbusy1)
        EPinterruptTx();
}
```

Bezparametrowa funkcja zwrotna `EPinterruptTx` zasadniczo jest wywoływana, gdy pakiet z danymi pilnymi wstawiony do wysłania za pomocą funkcji `USBDwrite` został wysłany. Oprócz tego jest wywoływana wewnątrz funkcji `EPinterruptRx`. Jeśli cykliczna kolejka buforów nie jest pusta, to kolejny pakiet z tej kolejki, wskazywany przez indeks `txidx1`, wstawiamy do wysłania za pomocą funkcji `USBDwrite`. Następnie zmniejszamy liczbę zajętych buforów `buffered1` i przesuwamy cyklicznie indeks `txidx1` wskazujący na pierwszy zajęty bufor. Jeśli cykliczna kolejka buforów jest pusta, to zerujemy zmienną `txbusy1`, aby zaznaczyć, że żaden pakiet nie oczekuje na wysłanie. Funkcja ta nie zwraca żadnej wartości.

```
void EPinterruptTx() {
    if (buffered1 > 0) {
        USBDwrite(ENDP1, data1[txidx1], size1[txidx1]);
        -- buffered1;
        txidx1 = txidx1 < BUFFER_COUNT1 - 1 ? txidx1 + 1 : 0;
        txbusy1 = 1;
    }
    else
        txbusy1 = 0;
}
```

Bezparametrowa funkcja zwrotna `EPisochronousRx` jest wywoływana, gdy zostanie odebrany pakiet z danymi izochronicznymi. Jeśli cykliczna kolejka buforów nie jest pełna, to pakiet odczytujemy za pomocą funkcji `USBDread` i zapisujemy do pierwszego wolnego bufora w kolejce `data2[rxidx2]`. Rozmiar odczytanych danych zapisujemy w tablicy `size2[rxidx2]`. Następnie zwiększamy liczbę zajętych buforów

buffered2 i przesuwamy cyklicznie indeks rxidx2 wskazujący na pierwszy wolny bufor. Funkcja ta nie zwraca żadnej wartości.

```
void EPisochronousRx() {
    if (buffered2 < BUFFER_COUNT2) {
        size2[rxidx2] = USBDread(ENDP2, data2[rxidx2], ISO_BUFF_SIZE);
        ++buffered2;
        rxidx2 = rxidx2 < BUFFER_COUNT2 - 1 ? rxidx2 + 1 : 0;
    }
}
```

Bezparametrowa funkcja zwrotna EPisochronousTx jest wywoływana, gdy pakiet z danymi izochronicznymi wstawiony do wysłania za pomocą funkcji USBDwrite został wysłany. Jeśli cykliczna kolejka buforów nie jest pusta, to kolejny pakiet z tej kolejki, wskazywany przez indeks txidx2, wstawiamy do wysłania za pomocą funkcji USBDwrite. Następnie zmniejszamy liczbę zajętych buforów buffered2 i przesuwamy cyklicznie indeks txidx2 wskazujący na pierwszy zajęty bufor. Jeśli cykliczna kolejka buforów jest pusta, to wstawiamy do wysłania pakiet bez danych za pomocą wywołania funkcji USBDwrite z zerowymi parametrami, żeby zawsze jakiś pakiet izochroniczny czekał na wysłanie, nawet gdy w rzeczywistości nie ma żadnych danych do wysłania. Funkcja EPisochronousTx nie zwraca żadnej wartości.

```
void EPisochronousTx() {
    if (buffered2 > 0) {
        USBDwrite(ENDP2, data2[txidx2], size2[txidx2]);
        -- buffered2;
        txidx2 = txidx2 < BUFFER_COUNT2 - 1 ? txidx2 + 1 : 0;
    }
    else
        USBDwrite(ENDP2, 0, 0);
}
```

Bezparametrowa funkcja zwrotna EPbulkRx jest wywoywana, gdy zostanie odebrany pakiet z danymi masowymi. Algorytm jej działania jest taki sam jak funkcji EPinterruptRx. Funkcja ta nie zwraca żadnej wartości.

```
void EPbulkRx() {
    if (buffered3 < BUFFER_COUNT3) {
        size3[rxidx3] = USBDread(ENDP3, data3[rxidx3], BLK_BUFF_SIZE);
        ++buffered3;
        rxidx3 = rxidx3 < BUFFER_COUNT3 - 1 ? rxidx3 + 1 : 0;
    }
}
```

```
    else
        USBDread(ENDP3, 0, 0);
    if (!txbusy3)
        EPbulkTx();
}
```

Bezparametrowa funkcja zwrotna EPbulkTx zasadniczo jest wywoływana, gdy pakiet z danymi masowymi wstawiony do wysłania za pomocą funkcji USBDwrite został wysłany. Oprócz tego jest wywoływana wewnątrz funkcji EPbulkRx. Algorytm jej działania jest taki sam jak funkcji EPinterruptTx. Funkcja ta nie zwraca żadnej wartości.

```
void EPbulkTx() {
    if (buffered3 > 0) {
        USBDwrite(ENDP3, data3[txidx3], size3[txidx3]);
        -- buffered3;
        txidx3 = txidx3 < BUFFER_COUNT3 - 1 ? txidx3 + 1 : 0;
        txbusy3 = 1;
    }
    else
        txbusy3 = 0;
}
```

Globalna zmienna configuration przechowuje numer aktualnie ustawionej konfiguracji.

```
static uint8_t configuration;
```

Funkcja ResetState, wywoływana w funkcji zwrotnej Reset, inicjuje stan aplikacji, czyli opróżnia bufora dla zerowego punktu końcowego i cykliczne kolejki buforów dla pozostałych punktów końcowych. Dodatkowo inicjuje zmienną configuration.

```
static void ResetState() {
    int i;
    for (i = 0; i < BUFFER_COUNT0; ++i)
        size0[i] = 0;
    value = 0;
    buffered1 = rxidx1 = txidx1 = txbusy1 = 0;
    buffered2 = rxidx2 = txidx2 = 0;
    buffered3 = rxidx3 = txidx3 = txbusy3 = 0;
    configuration = 0;
}
```

Funkcja zwrotna SetConfiguration jest wywoływana po odebraniu żądania SET_CONFIGURATION. Jeśli parametr confValue ma wartość jeden, to aktywowana jest jedyna konfiguracja urządzenia – konfigurowane są wszystkie niedomyślne, opisane deskryptorami punkty końcowe. Jeśli parametr ten ma wartość zero, to konfiguracja jest dezaktywowana. Funkcja ta zwraca REQUEST_SUCCESS, gdy wszystko przebiegło pomyślnie, a wartość REQUEST_ERROR, gdy podany numer konfiguracji jest błędny lub nie udało się skonfigurować któregoś punktu końcowego.

```
usb_result_t SetConfiguration(uint16_t confValue) {
    if (confValue <= device_descriptor.bNumConfigurations) {
        configuration = confValue;
        USBDisableAllNonControlEndPoints();
        if (confValue == vendor_conf.cnf_descr.bConfigurationValue) {
            usb_result_t r1, r2, r3;
            r1 = USBDendPointConfigure(ENDP1, INTERRUPT_TRANSFER,
                                         INT_BUFF_SIZE, INT_BUFF_SIZE);
            r2 = USBDendPointConfigure(ENDP2, ISOCHRONOUS_TRANSFER,
                                         ISO_BUFF_SIZE, ISO_BUFF_SIZE);
            r3 = USBDendPointConfigure(ENDP3, BULK_TRANSFER,
                                         BLK_BUFF_SIZE, BLK_BUFF_SIZE);
            if (r1 == REQUEST_SUCCESS &&
                r2 == REQUEST_SUCCESS &&
                r3 == REQUEST_SUCCESS)
                return REQUEST_SUCCESS;
            else
                return REQUEST_ERROR;
        }
        return REQUEST_SUCCESS;
    }
    return REQUEST_ERROR;
}
```

Żeby implementacja urządzenia była kompletna, potrzebne są jeszcze funkcje Reset, GetDescriptor, GetConfiguration i GetStatus. Są one analogiczne jak w przypadku urządzeń opisywanych w poprzednim rozdziale, dlatego je pomijam. Adresy wszystkich funkcji zwrotnych są umieszczone w strukturze ApplicationCallbacks typu usbd_callback_list_t. Wskaźnik na tę strukturę zwracany jest przez funkcję USBGetApplicationCallbacks.

```
static usbd_callback_list_t const ApplicationCallbacks = {
    0, Reset, 0, GetDescriptor, 0, GetConfiguration, SetConfiguration,
    GetStatus, 0, 0, 0, VendorNoDataSetup,
    VendorInDataSetup, VendorOutDataSetup, VendorStatusIn,
```

```
{EPinterruptTx, EPisochronousTx, EPbulkTx,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  
{EPinterruptRx, EPisochronousRx, EPbulkRx,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  
0, 0  
};
```

4.3.3. Komplilowanie i testowanie

Archiwum z przykładami zawiera kilka wersji projektu. W katalogu *./make* znajdują się podkatalogi, których nazwy rozpoczynają się przedrostkiem *usb4_vendor_device*. W tych podkatalogach umieszczone są pliki *makefile* umożliwiające skompilowanie projektu dla różnych wariantów sprzętu. Można je dostosować do innego sprzętu, posługując się wskazówkami z rozdziału 2. Pliki *makefile* zawierają listę plików źródłowych niezbędnych do skompilowania programu, są zatem przydatne również dla tych, którzy nie chcą korzystać bezpośrednio z polecenia *make*. Wtedy należy pamiętać, aby dołączyć plik *startup_stm32.c* i właściwą wersję biblioteki STM32.

W katalogu *./make/linux* znajdują się dwa pliki z testami dla tego projektu: *ex_libusb.c* i *ex_vendor_test.c*. Są one przeznaczone dla systemu Linux. W katalogu tym znajduje się też plik *makefile* umożliwiający ich skompilowanie.

Plik *ex_libusb.c* zawiera prosty test biblioteki *libusb*. Fragmenty tego programu omówiłem w podrozdziale 4.2. Poszukuje on urządzenia o identyfikatorze producenta 0x0483. Dla pierwszego pasującego urządzenia odczytuje z jego deskryptora urządzenia liczbę konfiguracji i próbuje ustawić różne konfiguracje, wywołując funkcję *libusb_set_configuration*. Jeśli urządzenie jest podłączone i działa poprawnie, to po skompilowaniu tego programu i wpisaniu polecenia

```
./ex_libusb  
powinniśmy zobaczyć następujący wynik
```

```
Device 0483:5754 found.  
Configuration -1 set.  
Configuration 0 set.  
Configuration 1 set.
```

Program zawarty w pliku *ex_vendor_test.c* wykonuje kilka testów transmisji danych, demonstrując jednocześnie użycie biblioteki *libusb*. Program ma dwa parametry. Pierwszy parametr identyfikuje urządzenie – należy podać identyfikator producenta i identyfikator produktu oddzielone dwukropkiem. Drugim parametrem jest liczba testów, które mają być wykonane w celu obliczenia średniego czasu transmisji danych. W każdym teście program najpierw wysyła dane do urządzenia, a potem je odbiera i sprawdza, czy otrzymał te same dane. Po skompilowaniu możemy wywołać go w następujący sposób

```
./ex_vendor_test 0483:5754 200
```

Pierwszy test sprawdza działanie zerowego punktu końcowego dla danych sterujących. Wysyłanych jest pięć żądań SET_DATA, jedno żądanie SET_VALUE i sześć żądań GET_DATA. Jedno żądanie SET_DATA powinno zostać odrzucone z powodu błędnej wartości parametru wValue. Powinny też zostać odrzucone dwa żądania GET_DATA: jedno z powodu błędnej wartości parametru wValue, a drugie z powodu błędnej wartości parametru wLength. Wynik poprawnie zakończonego testu jest pokazany na poniższym wydruku. Dwa ostatnie wiersze zawierają statystyki odpowiednio dla transmisji kontroler-urządzenie (OUT) i urządzenie-kontroler (IN). Są to po kolej liczba poprawnych i niepoprawnych transmisji oraz minimalny, średni i maksymalny czas transmisji. Czasy podawane są w nanosekundach. Jak widać, żądania z danymi sterującymi realizowane są w ułamku milisekundy, czyli prawie natychmiast.

Control buffer test

Warning: one OUT and two IN tests should fail.

```
out libusb_control_transfer failed, result -9, value 4, length 7
in libusb_control_transfer failed, result -9, value 4, length 7
in libusb_control_transfer failed, result -9, value 5, length 0
OUT: 5 pass, 1 fail, time min/avg/max 184873/241836/284817 ns
IN: 4 pass, 2 fail, time min/avg/max 120408/199453/298996 ns
```

Drugi test sprawdza działanie cyklicznej kolejki buforów dla pierwszego punktu końcowego, który jest skonfigurowany dla danych pilnych. Wykonywanych jest pięć transmisji w każdym kierunku. Jedna z transmisji składa się z dwóch pakietów, a pozostałe z jednego. Zatem wysyłanych jest łącznie sześć pakietów. Cykliczna kolejka buforów ma rozmiar cztery. Pierwszy odebrany pakiet zostanie natychmiast wstawiony do wysłania. Zatem zbuforowanych może być maksymalnie pięć pakietów. Szósty odebrany przez urządzenie pakiet zostanie porzucony. Dlatego jedna transmisja urządzenie-kontroler (IN) nie powinna się udać. Jak widać na poniższym wydruku, minimalny czas przesłania danych wynosi ok. 1 ms, co odpowiada transmisji jednego pakietu pilnego, a maksymalny – ok. 2 ms, co odpowiada transmisji dwóch pakietów. Transakcje przesłania danych pilnych inicjowane są przez kontroler z okresem wynikającym z ustawienia parametru bInterval w deskryptorze punktu końcowego. Ponieważ w naszym przypadku ma on wartość 1, to transakcje powinny być inicjowane przez kontroler co 1 ms. Zatem wszystko się zgadza.

Interrupt buffer test

Warning: one IN test should fail.

```
in libusb_interrupt_transfer failed, result -7, length 3, transferred 0
OUT: 5 pass, 0 fail, time min/avg/max 977731/1314766/2030548 ns
IN: 4 pass, 1 fail, time min/avg/max 911172/1242819/1896163 ns
```

Kolejny test ma na celu sprawdzenia czasu odpowiedzi punktu końcowego dla danych pilnych na nieco większej próbce danych. Program wyznacza średni czas realizacji przesłania dla liczby transmisji podanych jako drugi argument programu.

Program najpierw wysyła, a potem odbiera pakiety zawierające wMaxPacketSize, czyli 48 bajtów danych. Jak widać na poniższym wydruku, średnie czasy wysyłania i odbierania pakietu są, zgodnie z oczekiwaniem, bardzo bliskie 1 ms.

```
Interrupt response time test, data size 48 bytes
OUT: 200 pass, 0 fail, time min/avg/max 798728/997611/1085498 ns
IN: 200 pass, 0 fail, time min/avg/max 927724/998522/1062728 ns
```

Następny test ma na celu sprawdzenie szybkości przesyłania danych izochronicznych przez drugi punkt końcowy. Dane wysypane i odbierane są w pakietach zawierających wMaxPacketSize, czyli 40 bajtów. Parametr bInterval ma wartość 1, czyli pakiety powinny być przesyłane co 1 ms. Najpierw program testuje przesłanie danych z kontrolera do urządzenia. W każdym teście program wysyła po 98 pakietów. Jak widać na poniższym wydruku, średni czas ich transmisji wynosi ok. 106 ms. Dodatkowe 8 ms wynika najprawdopodobniej z dość skomplikowanego procesu inicjowania transmisji izochronicznej przez libusb. Następnie program testuje przesyłanie danych z urządzenia do kontrolera. Program próbuje odebrać 100 pakietów, czyli więcej niż wysłał. Zgodnie z algorytmem działania urządzenia nadmiarowe pakiety powinny być puste. Jak widać na poniższym wydruku, średni czas transmisji wynosi teraz ok. 108 ms i znów obserwujemy czas dłuższy o ok. 8 ms, niż wynikałoby to z samej zasadły działania transmisji izochronicznej. Widać też, że transmisja jest rzeczywiście izochroniczna – różnica między minimalnym a maksymalnym czasem nie przekracza 0,61 ms.

```
Isochronous transfer rate test, data size 40 bytes, sent packets 98,
received packets 100
OUT: 200 pass, 0 fail, time min/avg/max 106016820/106176746/106557679 ns
IN: 200 pass, 0 fail, time min/avg/max 108423093/108627397/109028712 ns
```

Celem dwóch ostatnich testów jest sprawdzenie szybkości przesyłania danych masowych przez trzeci punkt końcowy. W pierwszym teście program wysyła i odbiera dane po jednym pakiecie zawierającym wMaxPacketSize, czyli 64 bajtów danych. Po podzieleniu rozmiaru danych przez średni czas ich przesyłania otrzymujemy, że dane są wysypane ze średnią szybkością 4,37 Mb/s, a odbierane ze średnią szybkością 4,23 Mb/s. Jest to wyraźnie mniej niż teoretyczna przepustowość 12 Mb/s. Test był wykonywany przy braku innych aktywności na szynie USB. Zatem za obserwowane spowolnienie odpowiada oprogramowanie i narzut wprowadzany przez protokół USB.

```
Bulk transfer rate test, data size 64 bytes
OUT: 200 pass, 0 fail, time min/avg/max 100364/117074/245568 ns
IN: 200 pass, 0 fail, time min/avg/max 105254/121059/248780 ns
```

Aby zminimalizować wpływ narzutu wprowadzanego przez oprogramowanie po stronie kontrolera (aplikację i libusb), w drugim teście program przekazuje do wysyłania znacznie większe porcje danych, mianowicie po 6400 bajtów. Choć oczywiście dane są nadal przesyłane w pakietach po 64 bajty. Test pokazuje, że teraz średnia szybkość wysyłania wynosi 7,64 Mb/s, a średnia szybkość odbierania wy-

nosi 7,23 Mb/s. Jest to trochę więcej niż połowa teoretycznej przepływności, co należy uznać za bardzo dobry wynik, zważywszy spory narzut wprowadzany przez mechanizm transakcji, nagłówki pakietów, sekwencje rozpoczynające i kończące każdy pakiet, nadziewanie bitami, pakiety SOF, rezerwację części pasma danych dla sterujących, izochronicznych i pilnych. Niepowiązany wpływ na wynik tego testu ma też narzut wprowadzany przez działające na mikrokontrolerze oprogramowanie urządzenia.

```
Bulk transfer rate test, data size 6400 bytes
OUT: 200 pass, 0 fail, time min/avg/max 6376514/6705265/6952299 ns
IN: 200 pass, 0 fail, time min/avg/max 6695079/7084261/7492536 ns
```


W dotychczasowych rozważaniach pomijałem prawie zupełnie kwestię zarządzania zasilaniem (ang. *power management*) w urządzeniu USB. Było to spowodowane tym, że jest to zagadnienie nielatwe. Świadczy o tym choćby to, że wcale nie jest łatwo znaleźć seryjnie produkowane urządzenie niespełniające pokładanych w nim oczekiwani związanych z zarządzaniem zasilaniem. Problem może polegać na niespełnianiu wszystkich wymagań standardu przez urządzenie, sterownik urządzenia po stronie kontrolera albo aplikację korzystającą z tego urządzenia. Przyczyną problemu może też być błąd w systemie operacyjnym lub jego konfiguracji. Trzeba jednak spróbować zmierzyć się z problemem zasilania urządzenia USB. Dlatego poświęcam temu cały, choć krótki, rozdział. Omawiam tu, zamieszczone w standardzie, wymagania dotyczące poboru prądu przez urządzenie USB. Opisuję praktycznie stosowane sposoby realizacji tych wymagań. Projekt kończący ten rozdział demonstruje omawiane zagadnienia.

5.1. Wymagania standardu i praktyczne sposoby ich realizacji

Celem zarządzania zasilaniem jest rozsądne gospodarowanie zasobami energetycznymi systemu komputerowego. Polega to na przełączaniu komponentów, które w danej chwili nie są wykorzystywane, w stan uśpienia, czyli stan o obniżonym poborze prądu i ograniczonej funkcjonalności, bądź wręcz ich całkowitemu wyłączeniu. Jeśli uśpiony komponent jest potrzebny, zostaje obudzony (lub włączony) i przywrócony do pełnej funkcjonalności.

5.1.1. Komentarz do zawartości standardu

Interfejs USB umożliwia zasilanie podłączonych do niego urządzeń. Bezpośrednio po podłączeniu urządzenie może pobierać z linii VBUS do 100 mA prądu. W deskryptorze konfiguracji w polu `bMaxPower` deklaruje się, ile maksymalnie prądu urządzenie chce pobierać w tej konfiguracji. Wartość w miliamperach otrzymuje się, mnożąc przez dwa wartość z pola `bMaxPower`. Nie może to być jednak więcej niż 500 mA. Mimo że wartość tego prądu można podać z dokładnością do 2 mA, powszechnie przyjęło się podawanie wartości zaokrąglonej do najbliższej nie mniejszej wielokrotności 100 mA, czyli wartości 100, 200, 300, 400 lub 500 mA. Po wybraniu przez kontroler konfiguracji (za pomocą żądania `SET_CONFIGURATION`) urządzenie może pobierać prąd zadeklarowany w tej konfiguracji. Po dezaktywacji konfiguracji urządzenie powinno ograniczyć pobór prądu do wartości nieprzekraczającej 100 mA.

Przy braku aktywności na szynie przez 3 ms urządzenie powinno przejść w stan wstrzymania (ang. *suspend*) i ograniczyć pobór prądu z linii VBUS. Urządzenie, które w deskryptorze konfiguracji deklaruje do 100 mA w stanie aktywnym, powinno w stanie wstrzymania ograniczyć pobierany prąd do 0,5 mA. Każde dodatkowo zadeklarowane 100 mA zwiększa ten limit o kolejne 0,5 mA. Po wznowieniu aktywności na szynie urządzenie powinno się wybudzić (ang. *wakeup*) i znów może pobierać z szyny prąd o zadeklarowanej wartości. Rozważa się dwa scenariusze ograniczania poboru prądu: globalne wstrzymanie (ang. *global suspend*) całej szy-

ny i selektywne wstrzymywanie (ang. *selective suspend*) pojedynczego portu USB. Globalne wstrzymanie stosuje się, gdy system komputerowy przechodzi w stan uśpienia lub hibernacji i trzeba wtedy również ograniczyć pobór prądu przez wszystkie urządzenia. Kontroler sygnalizuje stan globalnego wstrzymania, zaprzestając jakiejkolwiek aktywności na szynie, w tym przede wszystkim zaprzestając wysyłania pakietów SOF. Zadaniem selektywnego wstrzymywania jest ograniczenie poboru prądu przez urządzenie, które pozostaje bezczynne (ang. *idle*) przez ustalony czas. Urządzenie jest bezczynne, jeśli nie jest zajęte przez żaden program i nie przesyła żadnych danych. Koncentrator jest bezczynny, jeśli wszystkie podłączone do niego urządzenia są wstrzymane. W celu selektywnego wstrzymania urządzenia lub fragmentu szyny kontroler wysyła do koncentratora żądanie zaprzestania aktywności na określonym porcie. Selektywne wstrzymywanie bywa też nazywane dynamicznym wstrzymywaniem (ang. *dynamic suspend, runtime suspend*) lub samowstrzymywaniem (ang. *auto suspend*).

Wstrzymane urządzenie może samo wybudzić się na skutek jakiegoś zdarzenia i wybudzić cały system komputerowy. Nazywa się to zdalnym budzeniem (ang. *remote wakeup*). Typowym przykładem jest budzenie komputera przez naciśnięcie klawisza. Zdarzenie to najpierw wybudza klawiaturę, która z kolei inicjuje zdalne budzenie komputera. Urządzenie deklaruje zdolność do zdalnego budzenia przez ustawienie bitu 5 w polu `bmAttributes` deskryptora konfiguracji. Funkcja zdalnego budzenia powinna pozostać wyłączona po uruchomieniu urządzenia i po każdorazowym wyzerowaniu szyny. Funkcja ta jest włączana przez kontroler za pomocą żądania `SET_FEATURE`, a wyłączana za pomocą żądania `CLEAR_FEATURE` – patrz rozdział 1. Sygnał zdalnego budzenia musi być generowany przez co najmniej 1 ms, ale nie dłużej niż 15 ms.

Wymagane przez standard ograniczenie wartości prądu, jaki może być pobierany przez wstrzymane urządzenie, wcale nie wydaje się bardzo drastyczne, ale w praktyce okazuje się trudne do spełnienia. W stanie wstrzymania musi pozostać podłączony rezystor $1,5\text{ k}\Omega$ podciągający jedną z linii danych do napięcia 3,3 V, odpowiednio linię D- dla urządzenia LS, a linię D+ dla urządzenia FS. Dla zachowania wstępnej kompatybilności urządzenie HS w stanie wstrzymania przełącza się w tryb FS, tym samym podłączając rezystor podciągający do linii D+. Po stronie kontrolera obie linie danych są ściągnięte do masy rezystorami $15\text{ k}\Omega$. Oznacza to, że między zasilaniem a masą mamy włączoną rezystancję $16,5\text{ k}\Omega$, przez którą płynie prąd 0,2 mA. Prąd ten można nieco zmniejszyć, gdyż standard dopuszcza większe wartości rezystorów ściągających. Z drugiej strony układ trzeba jednak projektować na najgorszy przypadek. Napięcie, do którego jest przyłączany rezystor podciągający, może być z przedziału 3...3,6 V. Rezystor podciągający przy braku transmisji może być z przedziału $900\ldots 1575\text{ }\Omega$, a rezystory ściągające – z przedziału $14,25\ldots 24,80\text{ k}\Omega$. Zatem w pesymistycznym przypadku przez rezystor podciągający płynie prąd 0,24 mA, czyli prawie połowa dozwolonego limitu.

Po rezystorze podciągającym drugim poważnym odbiorcą prądu w stanie wstrzymania jest regulator napięcia zasilającego. Napięcie zasilające USB musi mieścić się w przedziale 4,75...5,25 V, ale urządzenie pobierające do 100 mA powinno prawidłowo pracować przy napięciu zasilającym obniżonym nawet do 4,4 V. Natomiast

mikrokontroler i wiele układów peryferyjnych zasila się napięciem 3,3 V lub nawet mniejszym. Należy zatem zastosować regulator napięcia, który ma możliwie mały prąd skrośny. Dostępne są regulatory, dla których prąd ten nie przekracza kilku do kilkudziesięciu mikroamperów. Przykłady takich regulatorów zostały wymienione przy przedstawianiu schematów płyt prototypowych w rozdziale 2.

W stanie wstrzymania trzeba koniecznie wprowadzić mikrokontroler w któryś z trybów o obniżonym zapotrzebowaniu na energię. Trzeba ograniczyć pobór prądu przez wbudowane w mikrokontroler układy peryferyjne. Trzeba również tak zaprojektować schemat urządzenia, aby było możliwe ograniczenie poboru prądu przez zewnętrzne układy peryferyjne lub wręcz ich całkowite wyłączenie. Ponadto bezpośrednio połączeniu zasilania wszystkie układy peryferyjne powinny być domyślnie wyłączone. Stan ich wejść aktywujących powinien być jawnie wymuszony na przykład za pomocą odpowiednich rezystorów. Chodzi o to, żeby po podłączeniu urządzenia do gniazda USB nie pobierało ono nadmiernego prądu, a aktywowanie peryferii następowało jawnie po aktywacji konfiguracji urządzenia przez kontroler. Nawet tak prozaiczne elementy jak rezystory podciągające wymagają przemyślenia. Wyobraźmy sobie rezistor podciągający o wartości $10\text{ k}\Omega$. Jeśli ktoś (użytkownik) lub coś (inny układ) poda niski poziom na wejście podciągane tym rezystorem, to popłynie prąd $0,33\text{ mA}$, co najpewniej spowoduje przekroczenie limitu dozwolonego w stanie wstrzymania. Dlatego w dobrze zaprojektowanym urządzeniu należy przewidzieć możliwość odłączania również rezystorów podciągających, a co z tego wynika, należy zapewnić inny sposób wymuszenia właściwego stanu na wejściu. Można to osiągnąć, stosując wewnętrzne rezystory podciągające, odpowiednio rekonfigurując porty wejściowe przed uśpieniem mikrokontrolera i przywracając ich pierwotną konfigurację po jego obudzeniu.

W praktyce układ zasilający port USB jest zwykle tylko zabezpieczony przed uszkodzeniem wynikającym z nadmiernego obciążenia, więc nie mierzy prądu pobieranego przez urządzenie, a nawet jeśli mierzy, to bardzo niedokładnie. Zatem cała odpowiedzialność za ograniczenie pobieranego prądu spoczywa na urządzeniu, które musi zachowywać się fair. Specyfikacja USB nakazuje implementowanie zarządzania zasilaniem we wszystkich urządzeniach. Jednak w praktyce często zdarza się, że urządzenie daje się wstrzymać, ale po wybudzeniu przestaje działać poprawnie lub w ogóle nie działa. Przyczyna może leżeć zarówno po stronie urządzenia, jak i po stronie systemu komputerowego (oprogramowanie kontrolera, sterownik urządzenia, aplikacja). Ponadto w przypadku zasilania baterijnego prąd pobierany przez szynę USB w stanie wstrzymania może okazać się nie do zaakceptowania. Dodatkowe problemy sprawia możliwość odłączenia wstrzymanego urządzenia, pojawienie się sygnału zdalnego budzenia, gdy system jeszcze nie zakończył usypiania itp. Niektóre BIOS-y zerują kontroler USB zarówno podczas startu, jak i wybudzania komputera. W większości systemów można pozostawić zasilanie szyny USB przy uśpieniu zachowującym zawartość RAM, ale nie można tego zrobić przy hibernacji zapisującej obraz systemu na dysku. Wszystko to sprawia, że prawidłowa obsługa wszystkich przypadków bywa trudna do zaimplementowania. Dlatego dobrą strategią wydaje się całkowite odłączanie zasilania szyny USB, gdy system chce zasnąć. Po obudzeniu przywraca się zasilanie i inicjuje wszystkie urządzenia USB. Działa to wyśmienicie w odniesieniu do urządzenia, które można bez obaw odłą-

czyć w każdym momencie i dla którego nie trzeba przechowywać stanu albo stan ten jest bardzo łatwo odtworzyć po obudzeniu, czyli np. dla klawiatury. Problem sprawiają natomiast np. pamięci masowe, gdzie trzeba uważać, aby nie doprowadzić do niespójności w systemie plików.

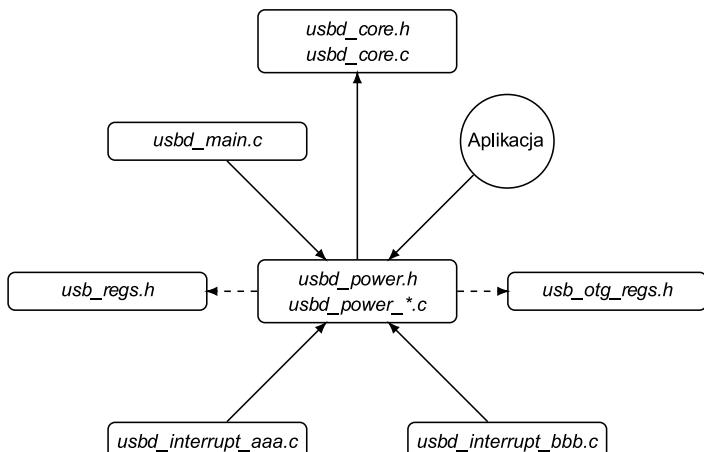
5.1.2. Rozszerzenie biblioteki urządzenia

```
usbd_power.h
usbd_power_dummy.c
usbd_power_103.c
usbd_power_152.c
usbd_power_x07.c
```

Zarządzanie energią zostało wydzielone do osobnego modułu biblioteki urządzenia USB, gdyż jest to fragment oprogramowania trudny do napisania, bardzo trudny do testowania, a wymagany tylko dla urządzeń pobierających prąd z szyny USB. Interfejs modułu zadeklarowany jest w pliku *usbd_power.h*. Implementacja umieszczona w pliku *usbd_power_dummy.c* zawiera tylko prototypy funkcji i jest przeznaczona dla urządzeń z całkowicie własnym zasilaniem, niepobierających prądu z szyny USB. Plik *usbd_power_103.c* zawiera implementację dla STM32F102 i STM32F103. Plik *usbd_power_152.c* zawiera implementację dla STM32Lxxx. Plik *usbd_power_x07.c* zawiera implementację dla STM32F105, STM32F107, STM32F2xx i STM32F4xx. Zależności między wymienionymi powyżej plikami a pozostałymi modułami biblioteki przedstawione są na **rysunku 5.1**.

```
int PWRconfigure(unsigned prio, unsigned subprio, int clk);
```

Funkcja *PWRconfigure* konfiguruje aspekty związane z zarządzaniem energią urządzenia USB. Parametry *prio* i *subprio* określają odpowiednio priorytet wywiesz-



Rys. 5.1. Powiązania modułu zarządzającego energią urządzenia z innymi plikami biblioteki

czania i podpriorytet, z jakimi ma być zgłoszane przerwanie, którego źródłem jest wybudzenie urządzenia ze stanu wstrzymania. Priorytet tego przerwania musi być wyższy niż priorytet przerwania USB podany w funkcji `USBDconfigure`, żeby wybudzić mikrokontroler, który został uśpiony podczas obsługi przerwania USB. Parametr `clk` określa częstotliwość taktowania rdzenia mikrokontrolera w megahercach. Znajomość tej częstotliwości jest niezbędna do przywrócenia prawidłowego taktowania mikrokontrolera i układu peryferyjnego USB po wybudzeniu ze stanu uśpienia. Choć w przypadku STM32 wartość częstotliwości taktowania rdzenia można wywnioskować z ustawień rejestrów konfiguracyjnych, to jest ona podawana jawnie jako parametr funkcji konfigurującej, aby interfejs programistyczny biblioteki nie był zależny od wersji zastosowanego sprzętu. Konkretna implementacja może po prostu ignorować wartość tego parametru. W aktualnej implementacji funkcja `PWRconfigure` zwraca zawsze zero. Możliwość zwrócenia innej (ujemnej) wartości jest pozostawiona dla przypadku, gdyby jej implementacja na jakimś sprzęcie potrzebowała sygnalizować jakiś błąd.

```
void PWRmanagementEnable(void);
```

Bezparametrowa funkcja `PWRmanagementEnable` uaktywnia aspekty związane z zarządzaniem energią urządzenia USB, czyli możliwość przejścia urządzenia w stan wstrzymania i powrotu z tego stanu. Nie zwraca ona żadnej wartości.

```
void PWRreduce(void);
```

Bezparametrowa funkcja `PWRreduce` jest wywoływana, gdy urządzenie przechodzi w stan wstrzymania (w procedurze obsługi tego zdarzenia), ale dopiero po wywołaniu funkcji `USBDsuspend` (patrz opis pliku `usbd_core.c`). Jej celem jest wprowadzenie układu peryferyjnego USB i całego mikrokontrolera w stan niskiego poboru energii, czyli ich uśpienie. Funkcja ta nie zwraca żadnej wartości.

```
void PWRresume(void);
```

Bezparametrowa funkcja `PWRresume` jest wywoywana, gdy urządzenie wychodzi ze stanu wstrzymania (w procedurze obsługi tego zdarzenia), już po obudzeniu mikrokontrolera, ale jeszcze przed wywołaniem funkcji `USBDwakeUp` (patrz opis pliku `usbd_core.c`). Jej celem jest dokończenie wybudzania mikrokontrolera i układu peryferyjnego USB ze stanu o niskim poborze energii. Funkcja ta nie zwraca żadnej wartości.

```
remote_wakeup_t PWRgetRemoteWakeUp(void);
```

Bezparametrowa funkcja `PWRgetRemoteWakeUp` zwraca wartość typu `remote_wakeup_t` informującą o aktualnym stanie zdalnego budzenia kontrolera. Jest to jedna z dwóch wartości: `RW_DISABLED` – zdalne budzenie kontrolera zablokowane, `RW_ENABLED` – zdalne budzenie kontrolera odblokowane.

```
void PWRsetRemoteWakeUp(remote_wakeup_t rw);
```

Funkcja `PWRsetRemoteWakeUp`, zależnie od wartości parametru `rw`, blokuje bądź od-blokowuje zdalne budzenia kontrolera. Nie zwraca żadnej wartości.

```
void PWRremoteWakeUp(void);
```

Bezparametrowa funkcja `PWRremoteWakeUp` inicjuje generowanie na szynie USB sygnału zdalnego budzenia kontrolera. Sygnał zacznie być generowany, jeśli zdalne budzenie kontrolera jest odblokowane i sygnał taki nie jest aktualnie generowany. Funkcja ta nie zwraca żadnej wartości.

5.2. Projekt wirtualnego portu szeregowego zasilanego z szyny

Zamieszczony w tym rozdziale projekt pokazuje, jak dostosować do zasilania z szyny USB urządzenie z podrozdziału 3.2 realizujące wirtualny port szeregowy. Wzorując się na tym opisie, można dostosować do zasilania z szyny każde inne urządzenie prezentowane w tej książce. Nie wymaga to istotnych modyfikacji deskryptorów. Wymaga natomiast napisania funkcji zwrotnych obsługujących dwa dodatkowe żądania: `SET_FEATURE` i `CLEAR_FEATURE`. Wymaga również napisania funkcji zwrotnych obsługujących zdarzenia związane z wykryciem braku aktywności na szynie i przywróceniem tej aktywności. Wymaga też drobnych zmian w kilku innych funkcjach zwrotnych. Przechodzę więc od razu do opisu implementacji, koncentrując się wyłącznie na różnicach w stosunku do implementacji przedstawionej w podrozdziale 3.2.

5.2.1. Implementacja

```
ex_com_bus_pwr_dev.c
```

Implementacja wirtualnego portu szeregowego zasilanego z szyny znajduje się w pliku `ex_com_bus_pwr_dev.c`. Deskryptor urządzenia mógłby pozostać bez zmian, ale ponieważ jest to piąty projekt, to dla porządku modyfikujemy identyfikator produktu `idProduct`, aby miał wartość `PID+5`, czyli `0x5755`. Nadajemy też naszemu urządzeniu nowy numer wersji 1.10, czyli w parametrze `bcdDevice` wpisujemy wartość `0x0110`. Ponadto nieco inny jest deskryptor tekstowy numer 2 identyfikujący produkt. W deskryptorze konfiguracji musimy zmodyfikować parametry związane z zasilaniem. W parametrze `bmAttributes` zamiast bitu `SELF_POWERED` ustawiamy bit `REMOTE_WAKEUP`. W parametrze `bMaxPower` deklarujemy, że nasze urządzenie będzie pobierało z szyny maksymalny dopuszczalny prąd, czyli 500 mA. Dla zwiększenia czytelności tekstu źródłowego definiujemy w tym celu stałe `USB_BM_ATTRIBUTES` i `USB_B_MAX_POWER`.

```
#define USB_BM_ATTRIBUTES  (REMOTE_WAKEUP | D7_RESERVED)  
#define USB_B_MAX_POWER   (500 / 2)
```

W funkcji zwrotnej `Configure`, wywoływanej w celu skonfigurowania urządzenia, jak poprzednio ustawiamy początkowe wartości zmiennych globalnych za pomocą funkcji `ResetState`, ustawiamy funkcję `LCDrefresh` odświeżającą ekran wyświet-

lacza ciekłokrystalicznego za pomocą funkcji LCDsetRefresh, konfiguruujemy wprowadzenie, do którego podłączona jest dioda świecąca mocy, za pomocą funkcji PowerLEDconfigure. Dioda ta symuluje pobór dużego prądu z szyny. Dodatkowo wywołujemy funkcję WakeupButtonConfigure (patrz podrozdział 3.2.4), aby skonfigurować przycisk, za pomocą którego będziemy budzić mikrokontroler i inicjować zdalne budzenie. Na koniec wywołujemy funkcję PWRsetRemoteWakeUp (patrz podrozdział 5.1.2) blokującą zdalne budzenie, które zgodnie ze standardem musi pozostawać wyłączone po uruchomieniu urządzenia i po wyzerowaniu szyny.

```
int Configure() {
    ResetState();
    LCDsetRefresh(LCDrefresh);
    PowerLEDconfigure();
    WakeupButtonConfigure();
    PWRsetRemoteWakeUp(RW_DISABLED);
    return 0;
}
```

W funkcji zwrotnej Reset, wywoływanej po wyzerowaniu szyny, jak poprzednio ustawiamy początkowe wartości zmiennych globalnych za pomocą funkcji ResetState i konfiguruujemy za pomocą funkcji USBDendPointConfigure domyślny punkt końcowy zero dla danych sterujących. Dodatkowo blokujemy zdalne budzenie za pomocą funkcji PWRsetRemoteWakeUp, a za pomocą funkcji PWRmanagementEnable uaktywniamy możliwość przejścia urządzenia w stan wstrzymania i powrót z niego. Z punktu widzenia tekstu źródłowego programu oznacza to włączenie przerwań związanych z wykryciem braku aktywności na szynie i ponowną aktywnością. W procedurach obsługi tych przerwań wykonywany jest kod odpowiadający za przejście urządzenia w stan wstrzymania i uśpienie mikrokontrolera, a potem jego wybudzenie i powrót urządzenia do normalnej aktywności. Standard USB stwierdza, że przejście w stan wstrzymania powinno być możliwe z każdego stanu, w którym urządzenie jest zasilane. Eksperymenty pokazują jednak, że między włączeniem zasilania a wyzerowaniem szyny jest zwykle zbyt mało czasu, aby udało się obsłużyć przejście do stanu wstrzymania i powrót z niego, co powoduje, że mikrokontroler może nie zdążyć obsłużyć zerowania szyny w przewidzianym na to czasie. Dlatego zezwalamy na przejście do stanu wstrzymania dopiero po zakończeniu procedury zerowania szyny. Funkcja Reset zwraca wartość pola bMaxPacketSize0 deskryptora urządzenia.

```
uint8_t Reset(usb_speed_t speed) {
    ErrorResetable(speed == FULL_SPEED ? 0 : -1, 6);
    PWRsetRemoteWakeUp(RW_DISABLED);
    ResetState();
    if (USBDendPointConfigure(ENDP0, CONTROL_TRANSFER,
                               device_descriptor.bMaxPacketSize0,
                               device_descriptor.bMaxPacketSize0) != REQUEST_SUCCESS)
```

```

        ErrorResetable(-1, 7);
        PWRmanagementEnable();
        return device_descriptor.bMaxPacketSize0;
    }
}

```

W funkcji zwrotnej SetConfiguration, wywoływanej po odebraniu żądania SET_CONFIGURATION, jak poprzednio konfigurujemy punkty końcowe opisane w deskryptorach. Dodatkowo, gdy konfiguracja jest dezaktywowana, czyli parametr confValue ma wartość zero, wyłączamy diodę świecącą mocy, aby ograniczyć pobór prądu z szyny do wartości nieprzekraczającej 100 mA. Funkcja PowerLEDOff nie robi nic, jeśli dioda jest akurat wyłączona.

```

usb_result_t SetConfiguration(uint16_t confValue) {
    if (confValue > device_descriptor.bNumConfigurations)
        return REQUEST_ERROR;
    configuration = confValue;
    USBDisableAllNonControlEndPoints();
    if (confValue == com_configuration.cnf_descr.bConfigurationValue) {
        usb_result_t r1, r2;
        r1 = USBDEndPointConfigure(ENDP1, BULK_TRANSFER,
                                     BLK_BUFF_SIZE, BLK_BUFF_SIZE);
        r2 = USBDEndPointConfigure(ENDP2, INTERRUPT_TRANSFER,
                                   0, INT_BUFF_SIZE);
        if (r1 == REQUEST_SUCCESS && r2 == REQUEST_SUCCESS)
            return REQUEST_SUCCESS;
        else
            return REQUEST_ERROR;
    }
    else { /* confValue == 0 */
        PowerLEDOff();
        return REQUEST_SUCCESS;
    }
}

```

Funkcja zwrotna GetStatus zwraca aktualne ustawienia urządzenia związane z zarządzaniem zasilaniem. Stan zdalnego budzenia sprawdza się za pomocą funkcji PWRgetRemoteWakeUp.

```

uint16_t GetStatus() {
    uint16_t result = 0;
    if (com_configuration.cnf_descr.bmAttributes & SELF_POWERED)
        result |= STATUS_SELF_POWERED;
    if ((com_configuration.cnf_descr.bmAttributes & REMOTE_WAKEUP) &&
        PWRgetRemoteWakeUp() == RW_ENABLED)
        result |= STATUS_REMOTE_WAKEUP;
    return result;
}

```

Funkcja zwrotna ClearDeviceFeature jest wywoływana po odebraniu żądania CLEAR_FEATURE kierowanego do urządzenia. Parametr featureSelector identyfikuje ustawienie, które ma zostać wyłączone. Jedyne aktualnie zdefiniowane w standardzie ustawienie, które dotyczy urządzenia i może być wyłączone, to zdalne budzenie (patrz tabela 1.16). Wyłącza się go za pomocą funkcji PWRsetRemoteWakeUp z argumentem RW_DISABLED.

```
usb_result_t ClearDeviceFeature(uint16_t featureSelector) {
    if (featureSelector == DEVICE_REMOTE_WAKEUP &&
        (com_configuration.cnf_descr.bmAttributes & REMOTE_WAKEUP)) {
        PWRsetRemoteWakeUp(RW_DISABLED);
        return REQUEST_SUCCESS;
    }
    return REQUEST_ERROR;
}
```

Funkcja zwrotna SetDeviceFeature jest wywoływana po odebraniu żądania SET_FEATURE kierowanego do urządzenia. Parametr featureSelector identyfikuje ustawienie, które ma zostać włączone. Jedyne zdefiniowane w standardzie ustawienia, które dotyczą urządzenia i mogą być włączone, to zdalne budzenie i tryb testowy (patrz tabela 1.16). Tryb testowy nie jest zaimplementowany w tej wersji biblioteki. Zdalne budzenie włącza się za pomocą funkcji PWRsetRemoteWakeUp z argumentem RW_ENABLED.

```
usb_result_t SetDeviceFeature(uint16_t featureSelector) {
    if (featureSelector == DEVICE_REMOTE_WAKEUP &&
        (com_configuration.cnf_descr.bmAttributes & REMOTE_WAKEUP)) {
        PWRsetRemoteWakeUp(RW_ENABLED);
        return REQUEST_SUCCESS;
    }
    return REQUEST_ERROR;
}
```

Demonstracja działania wirtualnego portu szeregowego polega na włączaniu i wyłączaniu diod świecących. W zmiennych globalnych green_led_state, red_led_state i power_led_state przechowujemy stan tych diod świecących.

```
static int green_led_state, red_led_state, power_led_state;
```

Funkcja zwrotna Suspend jest wywoływana, gdy urządzenia ma przejść w stan wstrzymania. W funkcji tej należy zaimplementować ograniczenie poboru prądu przez wszystkie układy peryferyjne. W naszym przypadku po prostu zapisujemy aktualny stan diod świecących i je wyłączamy.

```
void Suspend() {
    green_led_state = GreenLEDstate();
    red_led_state = RedLEDstate();
```

```

power_led_state = PowerLEDstate();
GreenLEDOff();
RedLEDOff();
PowerLEDOff();
}

```

Funkcja zwrotna `Wakeup` jest wywoływaną po wybudzeniu urządzenia, gdy powraca ono ze stanu wstrzymania do stanu normalnej aktywności. W funkcji tej należy zaimplementować przywrócenie nominalnego poboru prądu przez wszystkie układy peryferyjne. W naszym przypadku po prostu odtwarzamy stan diod świecących.

```

void Wakeup() {
    if (green_led_state)
        GreenLEDon();
    if (red_led_state)
        RedLEDon();
    if (power_led_state)
        PowerLEDon();
}

```

Prezentowane tu urządzenie korzysta w celach diagnostycznych z wyświetlacza ciekłokrystalicznego. Obsługa wyświetlacza zaimplementowana w module `lcd` nie jest przystosowana do spełnienia wymagań standardu związanych z zarządzaniem zasilaniem. Należałoby przystosować schemat płytka prototypowej w ten sposób, aby dało się wyświetlacz wyłączyć albo przynajmniej istotnie ograniczyć pobierany przez niego prąd. W module `lcd` należy w tym celu zaimplementować funkcję wyłączającą i włączającą wyświetlacz. W funkcji `Suspend` należy dodać jego wyłączenie, a w funkcji `Wakeup` ponowne włączenie. Alternatywnie można po prostu nie używać wyświetlacza, komplikując projekt z pustą implementacją modułu `lcd` zawartą w pliku `lcd_dummy.c`.

5.2.2. Zdalne budzenie

`pwr_periph_10x.c`
`pwr_periph_152.c`
`pwr_periph_2xx.c`

Implementacja zdalnego budzenia wymaga osobnego, nieco bardziej szczegółowego omówienia. W module `pwr_periph` znajduje się procedura `PUSH_BUTTON_IRQHandler` obsługująca przerwanie przycisku, którego naciśnięcie ma wybudzić mikrokontroler. Jej implementacja dla STM32F10x znajduje się w pliku `pwr_periph_10x.c`, dla STM32Lxxx w pliku `pwr_periph_152.c`, a dla STM32F2xx i STM32F4xx w pliku `pwr_periph_2xx.c`. Przerwanie przycisku powinno mieć wyższy priorytet wywieszczania niż przerwanie USB, aby wybudzić mikrokontroler, który zasnął w procedurze obsługi przerwania USB. W obsłudze przerwania przycisku wywoływana jest funkcja `PWRremoteWakeUp` inicjująca zdalne budzenie, a potem za pomocą funkcji `EXTI_ClearITPendingBit` zerowana jest przyczyna przerwania. Zerowanie

przyczyny przerwania odbywa się intencjonalnie na końcu, aby zostały zgubione wszelkie zgłoszenia tego przerwania, które nastąpiły podczas wykonywania funkcji PWRremoteWakeUp.

```
void PUSH_BUTTON_IRQHandler(void) {
    if (EXTI_GetITStatus(PUSH_BUTTON_EXTI_LINE)) {
        PWRremoteWakeUp();
        EXTI_ClearITPendingBit(PUSH_BUTTON_EXTI_LINE);
    }
}
```

*usbd_power_103.c
usbd_power_152.c*

Funkcja PWRremoteWakeUp dla STM32F102 i STM32F103 zaimplementowana jest w pliku *usbd_power_103.c*, a dla STM32Lxxx w pliku *usbd_power_152.c*. Globalna zmienna *remoteWakeUp* typu *remote_wakeup_t* przechowuje informację, czy zdalne budzenie jest włączone. Jest ona modyfikowana w funkcji PWRsetRemoteWakeUp, a odczytywana w funkcji PWRgetRemoteWakeUp. Globalna zmienna *state* przyjmuje dwie wartości: *RW_IDLE* – sygnał zdalnego budzenia nie jest generowany, *RW_STARTED* – sygnał zdalnego budzenia jest właśnie generowany. Wyzerowanie bitu *CNTR_LPMODE* budzi układ peryferyjny DEV-FS. Funkcja PWRresume przywraca takowanie mikrokontrolera. Żeby rozpoczęć generowanie sygnału zdalnego budzenia, wywołujemy funkcję StartRemoteHostWakeupsignalling. Funkcja TimerStart włącza licznik. Po upływie czasu zadeklarowanego za pomocą stałej *RW_TIME_MS*, czyli 6 ms, zostanie wywołana funkcja StopRemoteHostWakeupsignalling wstrzymująca generowanie sygnału zdalnego budzenia. Na koniec wywołujemy funkcję USBDwakeup, która z kolei wywołuje funkcję zwrotną *WakeUp*, żeby aplikacja mogła przywrócić zasilanie układów peryferyjnych.

```
void PWRremoteWakeUp() {
    if (remoteWakeUp == RW_ENABLED && state == RW_IDLE) {
        _SetCNTR(_GetCNTR() & ~CNTR_LPMODE);
        PWRresume();
        StartRemoteHostWakeupsignalling();
        TimerStart(1, StopRemoteHostWakeupsignalling, RW_TIME_MS);
        USBDwakeup();
    }
}
```

Funkcja StartRemoteHostWakeupsignalling ustawia globalną zmienną *state*, aby zaznaczyć, że rozpoczęła generowanie sygnału zdalnego budzenia. Funkcję RedLEDon wywołujemy tylko do celów diagnostycznych – w czasie generowania sygnału zdalnego budzenia świeci czerwona dioda. Ustawienie bitu *CNTR_RESUME* rozpoczęyna właściwe generowanie sygnału zdalnego budzenia.

```
static void StartRemoteHostWakeupSignalling(void) {
    state = RW_STARTED;
    RedLEDOn();
    _SetCNTR(_GetCNTR() | CNTR_RESUME);
}
```

Wyzerowanie bitu CNTR_RESUME przez funkcję StopRemoteHostWakeupSignalling kończy generowanie sygnału zdalnego budzenia. W celach diagnostycznych wywołujemy funkcję RedLEDOff wyłączającą czerwoną diodę świecącą. Zmienna state przyjmuje wartość oznaczającą, że zakończyło się generowanie sygnału zdalnego budzenia.

```
static void StopRemoteHostWakeupSignalling(void) {
    _SetCNTR(_GetCNTR() & ~CNTR_RESUME);
    RedLEDOff();
    state = RW_IDLE;
}
```

usbd_power_x07.c

Funkcja PWRremoteWakeUp dla STM32F105, STM32F107, STM32F2xx i STM32F4xx jest zaimplementowana w pliku *usbd_power_x07.c*. W tej implementacji ustawia ona tylko globalną zmienną state przyjmującą trzy wartości: RW_IDLE – sygnał zdalnego budzenia nie jest generowany, RW_INITIATED – sygnał zdalnego budzenia ma zostać wygenerowany, RW_STARTED – sygnał zdalnego budzenia jest właśnie generowany.

```
void PWRremoteWakeUp() {
    if (remoteWakeUp == RW_ENABLED && state == RW_IDLE)
        state = RW_INITIATED;
}
```

Mikrokontroler zasypia i budzi się w funkcji PWRreduce. Po obudzeniu, jeśli zmieniona state ma wartość RW_INITIATED, czyli przyczyną obudzenia jest wciśnięcie przycisku, to rozpoczynamy generowanie sygnału zdalnego budzenia za pomocą funkcji StartRemoteHostWakeupSignalling, a następnie włączamy licznik, aby po czasie określonym stał RW_TIME_MS zaprzestać generowania tego sygnału za pomocą funkcji StopRemoteHostWakeupSignalling. W tej implementacji funkcja PWRresume jest pusta. Poniższy wydruk zawiera odpowiedni fragment tekstu źródłowego funkcji PWRreduce.

```
if (state == RW_INITIATED) {
    StartRemoteHostWakeupSignalling();
    TimerStart(1, StopRemoteHostWakeupSignalling, RW_TIME_MS);
}
```

W funkcji StartRemoteHostWakeupSignalling zmiennej state nadajemy wartość oznaczającą, że rozpoczęło się generowanie sygnału zdalnego budzenia, w celach

diagnostycznych włączamy czerwoną diodę świecącą, a następnie ustawiamy bit `rwusig`, co rozpoczyna właściwe generowanie sygnału zdalnego budzenia.

```
static void StartRemoteHostWakeupSignalling(void) {
    USB_OTG_DCTL_TypeDef dctl;
    state = RW_STARTED;
    RedLEDOn();
    dctl.d32 = P_USB_OTG_DREGS->DCTL;
    dctl.b.rwusig = 1;
    P_USB_OTG_DREGS->DCTL = dctl.d32;
}
```

Funkcja `StopRemoteHostWakeupSignalling` zeruje bit `rwusig` i tym samym kończy generowanie sygnału zdalnego budzenia. Po czym wyłącza czerwoną diodę świeczącą i przywraca początkową wartość zmiennej `state`.

```
static void StopRemoteHostWakeupSignalling(void) {
    USB_OTG_DCTL_TypeDef dctl;
    dctl.d32 = P_USB_OTG_DREGS->DCTL;
    dctl.b.rwusig = 0;
    P_USB_OTG_DREGS->DCTL = dctl.d32;
    RedLEDOff();
    state = RW_IDLE;
}
```

5.2.3. Kompilowanie i testowanie

Archiwum z przykładami zawiera kilka wersji projektu wirtualnego portu szeregowego zasilanego z szyny. W katalogu `./make` znajdują się podkatalogi, których nazwy rozpoczynają się przedrostkiem `usb5_busPowered_com_device`. W tych podkatalogach umieszczone są pliki `makefile` umożliwiające skompilowanie projektu dla przykładowych wariantów sprzętu. Spośród wymienionych w punkcie 2.1.2 wariantów płyt prototypowych wymagania standardu w pełni spełniają tylko zmodyfikowany STM32L-Discovery, zmodyfikowany ZL31ARM oraz gadżet. Pozostałe płytki nie pozwalają w zadowalający dla tego projektu sposób sterować prądem pobieranym z szyny, ale ponieważ kontroler nie sprawdza, jaki prąd urządzenie rzeczywiście pobiera, również na nich można przeprowadzić wszystkie testy. Pliki `makefile` można oczywiście łatwo dostosować do innego sprzętu, posługując się wskazówkami z rozdziału 2. Są one również przydatne dla tych, którzy nie chcą korzystać bezpośrednio z programu `make`, gdyż zawierają listę plików źródłowych niezbędnych do skompilowania programu. Należy przy tym pamiętać, aby dołączyć plik `startup_stm32.c` i właściwą wersję biblioteki STM32. Testowanie funkcjonalności urządzenia niezwiązanej z zarządzaniem zasilaniem przebiega według punktu 3.2.5. Testowanie trybów o zmniejszonym poborze prądu przez mikrokontroler należy przeprowadzać przy odłączonym programatorze, gdyż aktywny interfejs JTAG lub SWD pobiera dodatkowy prąd. Układy z mikrokontrolerem STM32F103 nie zawsze wybudzają poprawnie interfejs USB, gdy są taktowane zegarem o częstot-

liwości 72 MHz. Dlatego zalecam taktowanie ich częstotliwością 48 MHz – patrz opisy plików *boot_10x.c* i *board_def.h*.

W systemie Linux zarządzanie interfejsami USB odbywa się poprzez system plików. Najpierw musimy odnaleźć interesujące nas urządzenie. Wpisujemy polecenie

```
lsusb -t
```

Powinniśmy zobaczyć podobny do poniższego wydruk, na którym odszukujemy nasz wirtualny port szeregowy *cdc_acm*.

```
/: Bus 02.Port 1: Dev 1, Class=root_hub, Driver=ehci_hcd/2p, 480M
|__ Port 1: Dev 2, If 0, Class=hub, Driver=hub/8p, 480M
    |__ Port 2: Dev 3, If 0, Class=HID, Driver=usbhid, 1.5M
    |__ Port 3: Dev 6, If 0, Class=comm., Driver=cdc_acm, 12M
    |__ Port 3: Dev 6, If 1, Class=data, Driver=cdc_acm, 12M
/: Bus 01.Port 1: Dev 1, Class=root_hub, Driver=ehci_hcd/2p, 480M
|__ Port 1: Dev 2, If 0, Class=hub, Driver=hub/6p, 480M
```

W tym konkretnym przypadku interesujące nas urządzenie jest podłączone do szyny USB2 (Bus 02). Na tej szynie do portu 1 głównego koncentratora (*root_hub*) podłączony jest kolejny koncentrator, do którego portu 3 podłączone jest z kolei nasze urządzenie (Dev 6). Te trzy liczby tworzą nazwę katalogu *2-1.3*, w którym znajdują się pliki związane z tym urządzeniem. Możemy zobaczyć nazwy plików, za pomocą których zarządza się zasilaniem tego urządzenia, wpisując poniższe polecenie.

```
ls /sys/bus/usb/devices/2-1.3/power
```

Powinniśmy zobaczyć tam plik *level* (starsze wersje jądra) lub *control* (najnowsze wersje jądra). Za pomocą tego pliku steruje się selektywnym wstrzymaniem urządzenia. Wpisując do niego wartość *auto*, włączamy selektywne wstrzymywanie. Wszystkie poniższe polecenia wymagają praw administratora.

```
echo auto > /sys/bus/usb/devices/2-1.3/power/control
```

Wpisując do niego wartość *on*, wyłączamy selektywne wstrzymywanie.

```
echo on > /sys/bus/usb/devices/2-1.3/power/control
```

Czas bezczynności ustawia się w pliku *autosuspend* (w sekundach, starsze wersje jądra) lub *autosuspend_delay_ms* (w milisekundach, najnowsze wersje jądra).

```
echo 5000 > /sys/bus/usb/devices/2-1.3/power/autosuspend_delay_ms
```

Aby aktywować zdalne budzenie, do czego konieczne jest pozostawienie zasilania szyny po uśpieniu systemu, trzeba wpisać odpowiednie wartości do plików *wakeup*. Zeby zadziałało zdalne budzenie, należy je też włączyć w BIOS-ie. Niestety nie wszystkie płyty główne umożliwiają obudzenie systemu przez USB.

```
echo enabled > /sys/bus/usb/devices/usb2/power/wakeup
```

```
echo enabled > /sys/bus/usb/devices/2-1/power/wakeup
```

```
echo enabled > /sys/bus/usb/devices/2-1.3/power/wakeup
```

```
echo USB2 > /proc/acpi/wakeup
```

Więcej informacji na temat działania USB w Linuksie można znaleźć w dokumentacji jądra systemu pod adresem <http://www.mjmwired.net/kernel/Documentation/usb>. Zarządzanie zasilaniem interfejsu USB opisane jest w dwóch umieszczonych tam dokumentach: *power-management.txt* i *persist.txt*.

W systemie Windows trzeba zainstalować sterownik urządzenia, na przykład według opisu z rozdziału 3.2.5. Przy czym należy w pliku *stmcdc.inf* poprawić identyfikator produktu, zamieniając `PID_5752` na `PID_5755`. Jeśli sterownik obsługuje zarządzanie zasilaniem interfejsu USB, to w jego właściwościach jest zakładka *zarządzanie energią*, w której są dwie opcje:

- zezwalaj komputerowi na wyłączanie tego urządzenia w celu oszczędzania energii (selektywne wstrzymywanie),
- zezwalaj temu urządzeniu na wznowianie pracy komputera (zdalne budzenie).

Takie same opcje są w zakładce *zarządzanie energią* głównego koncentratora USB i pozostałych koncentratorów. Sterowniki koncentratorów są w sekcji *kontrolery uniwersalnej magistrali szeregowej* w menedżerze urządzeń. Ponadto globalne opcje związane z zasilaniem USB są do znalezienia po wybraniu kolejno: *panel sterowania, opcje zasilania, edytowanie ustawień planu, zmień zaawansowane ustawienia zasilania, ustawienia USB*. Oczywiście zależnie od wersji systemu Windows nazwy poszczególnych menu mogą być nieco inne.

W tym rozdziale przedstawiam przykład urządzenia, które może pracować zarówno z pełną szybkością, jak i wysoką szybkością. Jest to urządzenie udostępniające interfejs MSC (ang. *Mass Storage Class*), czyli interfejs pamięci masowej. Interfejs tej klasy stosuje się do podłączania zewnętrznych dysków twardych, napędów dysków optycznych i dyskietek, przenośnych pamięci Flash (ang. *pendrive*), aparatów fotograficznych itp. Prezentowane urządzenie jest rozpoznawane przez system operacyjny jako wymienny dysk zewnętrzny. Jak łatwo się domyślić, z uwagi na charakter urządzenia, komunikacja z nim odbywa się z wykorzystaniem dwóch strumieni danych masowych.

6.1. Deskryptory i żądania

Jak każde urządzenie USB, urządzenie z interfejsem MSC musi udostępniać standardowe deskryptory i obsługiwać standardowe żądania, zgodnie z opisem zamieszczonym w poprzednich rozdziałach. W deskryptorze urządzenia pola określające klasę, podklasę i protokół, czyli pola `bDeviceClass`, `bDeviceSubClass`, `bDeviceProtocol` wypełnia się zerami. Pozostałe pola deskryptora urządzenia wypełnia się zgodnie z opisem zamieszczonym w rozdziale 1. Klasę, podklasę i protokół definiuje się w deskryptorze interfejsu, zgodnie z **tabelą 6.1**. W polu `bInterfaceClass` wpisuje się wartość 8, oznaczającą właśnie interfejs MSC.

Wartość w polu `bInterfaceSubClass` deskryptora interfejsu określa zestaw poleceń warstwy aplikacji stosowanych do sterowania pamięcią masową. Polecenia te nie są częścią standardu USB – są zaczerpnięte z innych protokołów i zdefiniowane w osobnych dokumentach normatywnych. Najbardziej uniwersalny i rozbudowany jest zestaw poleceń SCSI (ang. *Small Computer Systems Interface*). RBC (ang. *Reduced Block Commands*) zawiera ograniczony zestaw poleceń SCSI, wprowadzony w celu uproszczenia implementacji prostych urządzeń blokowych (jednostką danych jest blok nazywany też sektorem). Zestaw poleceń UFI (ang. *USB Floppy Interface*) zdefiniowano na potrzeby dyskietek i wzoruje się on również na SCSI. Protokół MMC (ang. *MultiMedia Commands*) zawiera zestaw poleceń przeznaczonych głównie do obsługi dysków optycznych. Jest on następcą protokołu ATAPI wprowadzonego w celu rozszerzenia równoległego interfejsu ATA o obsługę dys-

Tab. 6.1. Parametry deskryptora interfejsu

Pole	Rozmiar	Wartość	Znaczenie
<code>bInterfaceClass</code>	1	8	MSC
		1	RBC
		2	MMC (ATAPI)
		4	UFI
		6	SCSI
		7	LSDFS
		8	IEEE 1667
		0	CBI
<code>bInterfaceSubClass</code>	1	1	CBI
		80	BOT
		98	UAS

ków optycznych. Zarówno ATA, jak i ATAPI są obecnie uważane za przestarzałe. LSDFS jest protokołem negocjowania dostępu przed użyciem poleceń SCSI. IEEE 1667 jest standardem uwierzytelniania przenośnych pamięci podczas ich podłączania do komputera. Jak widać, dominuje standard SCSI i on też będzie przedmiotem dalszych rozważań.

Wartość w polu `bInterfaceProtocol` deskryptora interfejsu definiuje protokół warstwy transportowej. W protokole CBI (ang. *control, bulk, interrupt*) korzysta się z trzech punktów końcowych, odpowiednio dla danych sterujących (polecenia), masowych (właściwe dane) i pilnych (potwierdzenia wykonania poleceń). Są dwa warianty tego protokołu. Wariant identyfikowany wartością 0 potwierdza wykonanie poleceń, a wariant 1 ich nie potwierdza. W protokole BOT (ang. *bulk-only transport*), oznaczanym też skrótem BBB (ang. *bulk, bulk, bulk*), korzysta się tylko z jednego punktu końcowego dla danych masowych, za pomocą których przesyła się również polecenia i ich potwierdzenia. BOT jest bardzo prostym protokołem i nie pozwala na wykorzystanie wszystkich zalet SCSI. Protokół UAS (ang. *USB Attached SCSI*) usuwa poważne wady protokołu BOT. Umożliwia kolejkowanie poleceń i ich wykonywanie w zmienionej kolejności. Ma to istotne znaczenie dla zwiększenia efektywności, zwłaszcza w przypadku nośników mechanicznych, gdzie należy dostosować kolejność odczytów i zapisów do fizycznego rozmieszczenia sektorów na dysku. UAS eliminuje też narzut programowy związany z kapsułkowaniem poleceń SCSI w strumieniu danych masowych.

Urządzenie mogące pracować z dwoma szybkościami (np. FS i HS) musi mieć osobną konfigurację dla każdej szybkości. Powodem jest choćby to, że są różne dopuszczalne wartości pola `wMaxPacketSize` deskryptorów punktów końcowych – patrz tabela 1.10. W takim przypadku urządzenie powinno mieć też deskryptor kwalifikujący – patrz tabela 1.6.

Prezentowane urządzenie korzysta z najbliższego protokołu transportowego, czyli z protokołu BOT. Dlatego dalszy opis koncentruje się na tym protokole. W tym przypadku interfejs MSC ma jeden wejściowy i jeden wyjściowy punkt końcowy dla danych masowych. Ponadto tego typu interfejs MSC musi obsługiwać dwa specyficzne żądania, przedstawione w **tabeli 6.2**. W obu żądaniach parametr `wIndex` zawiera jak zwykle numer interfejsu, a parametr `wLength` rozmiar przesyłanych danych. Natomiast parametr `wValue` nie jest używany i musi mieć wartość zero.

Tab. 6.2. Żądania specyficzne dla MSC

<code>bmRequestType</code>	<code>bRequest</code>	<code>wValue</code>	<code>wIndex</code>	<code>wLength</code>	Dane
10100001	MSC_GET_MAX_LUN 254	0	Numer interfejsu	1	Maksymalny LUN
00100001	MSC_BULK_ONLY_RESET 255	0	Numer interfejsu	0	Brak

Interfejs MSC może udostępniać wiele jednostek (dysków) logicznych. Jednostka logiczna jest identyfikowana przez jej numer LUN (ang. *logical unit number*). Pierwotna specyfikacja SCSI przewiduje maksymalnie 8 jednostek logicznych, a najnowsza dopuszcza 16. Jednostki logiczne numeruje się od zera. Każde urządzenie powinno udostępniać jednostkę logiczną numer zero. Wartość LUN zapisuje

się w jednym bajcie. Nie należy mylić LUN z fizycznym identyfikatorem urządzenia na szynie SCSI. Jednostki logiczne stanowią dodatkowy poziom adresacji w obrębie urządzenia fizycznego. Za pomocą żądania `MSC_GET_MAX_LUN` kontroler dowiaduje się, jaki jest maksymalny numer jednostki logicznej dla danego urządzenia. Żądanie to zwraca zero, gdy jest tylko jedna jednostka logiczna.

Po wykryciu przez kontroler sytuacji, w której dalsza komunikacja w protokole BOT nie może być kontynuowana poprawnie, rozpoczyna on procedurę przywracania stanu początkowego (ang. *reset recovery*). Kontroler najpierw wysyła żądanie `MSC_BULK_ONLY_RESET`, a następnie dwa żądania `CLEAR_FEATURE`, które uruchamiają ponownie kolejno wejściowy i wyjściowy punkt końcowy dla danych masowych.

6.2. Protokoły pamięci masowej

Jak już wspomniałem, w prezentowanym projekcie komunikacja z pamięcią masową realizowana jest za pomocą protokołów BOT i SCSI. BOT jest protokołem transportowym. Jego zadaniem jest przesyłanie poleceń SCSI oraz danych, które zostały odczytane z pamięci lub mają być zapisane w pamięci. SCSI jest zestawem poleceń warstwy aplikacji. Polecenia te pozwalają m.in. na odczytanie pojemności pamięci oraz odczyt i zapis wskazanych bloków logicznych (sektorów) pamięci.

6.2.1. Protokół BOT

Protokół BOT multipleksuje polecenia SCSI i dane w strumieniu danych masowych. Pełny cykl komunikacji realizowanej przez protokół BOT składa się z trzech etapów. Najpierw, za pomocą wyjściowego punktu końcowego, przesyła się polecenie SCSI opakowane (kapsułkowane) w pakiecie CBW (ang. *command block wrapper*). Drugi etap jest opcjonalny i polega na przesłaniu danych za pomocą wejściowego lub wyjściowego punktu końcowego. Użyty punkt końcowy zależy od kierunku, w którym przesyła się dane. Kierunek jest zdeterminowany przez wykonywane polecenie. Jeśli polecenie nie wymaga przesłania żadnych danych, etap przesyłania danych się pomija. Na koniec, za pomocą wejściowego punktu końcowego, przesyła się wynik wykonania polecenia, czyli jego status, opakowany (kapsułkowany) w pakiecie CSW (ang. *command status wrapper*).

Pakiety CBW i CSW muszą rozpoczynać się na początku pakietu USB i kończyć się pakietem USB o długości mniejszej niż maksymalna. Podział między kilka pakietów USB jest konieczny, gdy maksymalny rozmiar danych w pakiecie USB jest zbyt mały, aby pomieścić cały pakiet CBW lub CSW. Pakiet CBW ma rozmiar 31 bajtów, a pakiet CSW ma 13 bajtów. Zatem w praktyce najprościej jest zadeklarować maksymalny rozmiar pola danych pakietu USB w trybie FS jako 32 lub 64 bajty. W trybie HS deklarujemy rozmiar 512 bajtów, gdyż jest to jedyna dopuszczalna wartość w tym trybie. Wtedy przesłanie zarówno pakietu CBW, jak i pakietu CSW wymaga przesyłania tylko pojedynczego pakietu USB, co istotnie upraszcza implementację protokołu.

Formaty pakietów CBW i CSW przedstawione są szczegółowo odpowiednio w **tabeli 6.3** i **tabeli 6.4**. Pola wielobajtowe w tych pakietach zapisuje się w porządku cienkokońcowkowym (ang. *little-endian*). Ponieważ w protokole BOT polecenia i dane przesyła się na przemian w tym samym strumieniu danych masowych, ist-

nieje ryzyko, że w wyniku błędu nastąpi rozsynchonizowanie strumienia danych, co może skutkować pomyleniem polecenia z danymi. Żeby zminimalizować prawdopodobieństwo takiej pomyłki, oba pakietы rozpoczynają się czterobajtową sygnaturą. Kolejne cztery bajty obu pakietów zajmuje znacznik, który pozwala powiązać polecenie z jego wynikiem. Urządzenie, odsyłając wynik polecenia, umieszcza w pakiecie CSW znacznik tego polecenia odczytany z pakietu CBW. Pole `dCBWDataTransferLength` określa, ile danych ma zostać przesłanych. Ma ono wartość zero, gdy polecenie nie wymaga przesyłania danych. Pole `bmCBWFlags` określa kierunek przesyłania danych. Jeśli najstarszy bit tego pola ma wartość zero, to dane są przesyłane z kontrolera do urządzenia. Jeśli ma wartość jeden, to z urządzenia do kontrolera. Jeśli rozmiar przesyłanych danych wynosi zero, to urządzenie powinno ignorować bit kierunku. Pozostałych bitów tego pola nie używa się i powinny być one wyzerowane. Pole `bCBWLUN` zawiera numer jednostki logicznej, której dotyczy polecenie. Używa się tylko czterech najmłodszych bitów tego pola. Pozostałe bity powinny być wyzerowane. Pole `bCBWCBLength` zawiera rzeczywisty rozmiar polecenia SCSI zawartego w polu `CBWCB`. Dopuszczalne wartości należą do przedziału 1...16. Pozostałe bajty pola `CBWCB` powinny być ignorowane przez urządzenie. W przypadku przesyłania danych z kontrolera do urządzenia pole `dCSWDataResidue` zawiera różnicę między rozmiarem danych deklarowanym w polu `dCBWDataTransferLength` a rozmiarem danych rzeczywiście przetworzonych (zaakceptowanych) przez urządzenie. W przypadku przesyłania danych z urządzenia do kontrolera pole `dCSWDataResidue` zawiera różnicę między rozmiarem danych oczekiwanych przez kontroler i deklarowanym w polu `dCBWDataTransferLength` a rozmiarem danych rzeczywiście wysłanych przez urządzenie. Wartość w polu `dCSWDataResidue` nie powinna przekraczać wartości z pola `dCBWDataTransferLength`. Pole `bCSWStatus` zawiera wynik wykonania polecenia. Przyjmuje ono trzy wartości:

- 0 – polecenie zakończone poprawnie,
- 1 – polecenie niepoprawne, błąd podczas wykonywania polecenia,
- 2 – rozsynchonizowany strumień danych.

Tab. 6.3. Pakiet CBW

Pole	Rozmiar	Opis
<code>dCBWSignature</code>	4	Sygnatura 0x43425355 identyfikująca początek pakietu
<code>dCBWTag</code>	4	Znacznik wiążący polecenie z jego wynikiem
<code>dCBWDataTransferLength</code>	4	Rozmiar przesyłanych danych
<code>bmCBWFlags</code>	1	Znaczniki
<code>bCBWLUN</code>	1	Numer jednostki logicznej
<code>bCBWCBLength</code>	1	Rozmiar polecenia
<code>CBWCB</code>	16	Kapsułkowane polecenie

Tab. 6.4. Pakiet CSW

Pole	Rozmiar	Opis
<code>dCSWSignature</code>	4	Sygnatura 0x53425355 identyfikująca początek pakietu
<code>dCSWTag</code>	4	Znacznik wiążący polecenie z jego wynikiem
<code>dCSWDataResidue</code>	4	Różnica między oczekiwaniem rozmiarem danych a rozmiarem danych przetworzonych lub wysłanych
<code>bCSWStatus</code>	1	Wynik wykonania polecenia

Właśnie z uwagi na możliwość rozsynchonizowania się protokołu urządzenie powinno bardzo dokładnie sprawdzać poprawność wszystkich pól pakietu CBW. Dalszych szczegółów na temat protokołu BOT, w szczególności opisu metody jego ponownej synchronizacji, należy szukać w [31].

6.2.2. Protokół SCSI

SCSI jest bardzo uniwersalnym interfejsem wykorzystywanym do komunikacji z napędami dysków magnetycznych, dysków optycznych, taśm magnetycznych czy skanerami. Choć obecnie stracił on na znaczeniu z uwagi na nowsze i bardziej wydajne interfejsy, np. SATA, to w dalszym ciągu zestaw poleceń SCSI jest powszechnie wykorzystywany w komunikacji z pamięciami masowymi. Specyfikacja SCSI jest bardzo rozbudowana i swoimi rozmiarami dorównuje specyfikacji USB. Na szczęście interfejs MSC korzysta tylko z niewielkiego podzbioru poleceń SCSI.

Każde polecenie SCSI rozpoczyna się jednobajtowym kodem operacji, który wyznacza rodzaj polecenia i sposób interpretacji jego parametrów umieszczonych w kolejnych bajtach. Parametry wielobajtowe zapisuje się w porządku grubokoniówkowym (ang. *big-endian*). Trzy najbardziej znaczące bity kodu operacji determinują rozmiar polecenia, zgodnie z **tabelą 6.5**. Kody z przedziału 0xC0...0xFF są zarezerwowane dla rozszerzeń protokołu specyficznych dla producenta urządzenia. Jedno polecenie może występować w kilku wariantach różniących się kodem operacji i rozmiarem polecenia. Na przykład jest pięć wariantów polecenia odczytu danych, oznaczanych w dokumentacji odpowiednio jako READ (6), READ (10), READ (12), READ (16), READ (32). Liczba w nawiasie oznacza rozmiar polecenia w bajtach. Poszczególne warianty tego polecenia różnią się parametrami, a przede wszystkim zakresem adresów bloków logicznych, jaki obsługują. Powiększające się rozmiary pamięci masowych wymagają zwiększania liczby bitów potrzebnych do zakodowania adresu i rozmiaru przesyłanych danych. Stąd wynika sukcesywne zwiększanie rozmiaru polecenia w kolejnych wersjach standardu. Polecenie READ (6) jest obecnie uznawane za przestarzałe i nie powinno być stosowane w nowych implementacjach.

Tab. 6.5. Kody operacji i rozmiary poleceń SCSI

Kod operacji	Rozmiar polecenia
0x00...0x1F	6
0x20...0x5F	10
0x60...0x7F	Zmienny, np. 32
0x80...0x9F	16
0xA0...0xBF	12
0xC0...0xFF	Definiowany przez producenta

Ogólna architektura SCSI opisana jest w [34]. Opis poleceń blokowych (ang. *block commands*) zamieszczono w [35], poleceń głównych (ang. *primary commands*) w [36]. Warto też zatrzymać się na dokumentu [32], gdzie opisano protokół UFI bazujący na transporcie CBI. Wybrane polecenia SCSI potrzebne do zaimplementowania

interfejsu MSC zebrane są w **tabeli 6.6** – patrz też rozdział 5.3 w [18]. Poniżej przedstawiam ich skrócony opis. Więcej szczegółów można znaleźć w wyżej wspomnianych dokumentach oraz studując implementację omawianą w następnym podrozdziale.

Polecenie TEST UNIT READY służy do sprawdzenia, czy jednostka logiczna jest gotowa akceptować polecenia dostępu do nośnika, na którym przechowywane są dane. Protokół BOT pozwala zwrócić tylko status informujący, czy polecenie zakończyło się poprawnie, czy wystąpił jakiś błąd. Szczegółowy opis błędu odczytuje się za pomocą polecenia REQUEST SENSE. Polecenie INQUIRY dostarcza ogólnych informacji o własnościach urządzenia (blokowe, wyjmowane itp.) oraz różnego rodzaju danych w postaci napisów: nazwa producenta, nazwa produktu, numer wersji, numer seryjny. Polecenia MODE SENSE (6) i MODE SENSE (10) dostarczają szczegółowych informacji o własnościach urządzenia. Za pomocą polecenia START STOP UNIT wysyła się do urządzenia żądanie zmiany stanu zasilania lub żądanie załadowania (ang. *load*) bądź wysunięcia (ang. *eject*) nośnika. Za pomocą polecenia PREVENT ALLOW MEDIUM REMOVAL wysyła się do urządzenia żądanie zablokowania bądź odblokowania wysuwania nośnika. Polecenie READ FORMAT CAPACITIES jest używane przez system Windows do odczytania informacji o pojemności dysku. Nie należy ono do standardowego zbioru poleceń SCSI. Zostało zaczerpnięte z protokołu UFI. Jego kod operacji opisany jest jako zarezerwowany dla rozszerzeń wprowadzanych przez producenta urządzenia (ang. *vendor specific*). System Linux odczytuje pojemność dysku polecienniem READ CAPACITY (10). W obu przypadkach pojemność dysku opisuje się dwoma wartościami: rozmiar bloku logicznego w bajtach i liczba bloków logicznych. Polecenie READ (10) pozwala odczytać ciąg kolejnych bloków logicznych. Natomiast polecenie WRITE (10) pozwala zapisać ciąg kolejnych bloków logicznych. Polecenie VERIFY (10) nakazuje sprawdzenie poprawności danych zapisanych we wskazanych sektorach.

Tab. 6.6. Wybrane polecenia SCSI

Polecenie	Kod operacji	Opis
TEST UNIT READY	0x00	Sprawdzenie gotowości
REQUEST SENSE	0x03	Szczegółowa informacja o błędzie
INQUIRY	0x12	Ogólne własności urządzenia
MODE SENSE (6)	0x1A	Szczegółowe parametry urządzenia
MODE SENSE (10)	0x5A	Szczegółowe parametry urządzenia
START STOP UNIT	0x1B	Zmiana stanu zasilania lub załadowanie bądź wysunięcie nośnika
PREVENT ALLOW MEDIUM REMOVAL	0x1E	Zablokowanie bądź odblokowanie wysuwania nośnika
READ FORMAT CAPACITIES	0x23	Informacja o pojemności dysku
READ CAPACITY (10)	0x25	Informacja o pojemności dysku
READ (10)	0x28	Odczyt sektorów
WRITE (10)	0x2A	Zapis sektorów
VERIFY (10)	0x2F	Weryfikacja zawartości sektorów

6.3. Implementacja

Implementacja została napisana i przetestowana z wykorzystaniem zestawu uruchomieniowego STM3220G-EVAL. Implementacja podzielona jest na trzy moduły (warstwy): obsługa pamięci zewnętrznej, obsługa protokołu SCSI, obsługa protokołu BOT i warstwy aplikacji protokołu USB. Omawiam je w kolejnych podrozdziałach.

6.3.1. Pamięć zewnętrzna

`stm32xg_extsram.h`
`stm32xg_extsram.c`

W zestawie STM3220G-EVAL zainstalowano 2 MiB zewnętrznej statycznej pamięci RAM, w której zorganizujemy dysk. Interfejs pamięci jest równoległy, co zapewnia szybki dostęp. Zgodnie z dokumentacją zestawu, żeby móc używać tej pamięci, zwory JP3 i JP10 powinny być zdjęte, a zwory JP1 i JP2 w pozycji 2–3 (ustawienie bliższe środka płytka). W mikrokontrolerze STM32 obsługą zewnętrznej pamięci zajmuje się układ peryferyjny FSMC (ang. *flexible static memory controller*). Niezbędne definicje znajdują się w pliku `stm32xg_extsram.h`. Implementacja jest umieszczona w pliku `stm32xg_extsram.c`.

```
void SRAMconfigure(void);
```

Bezparametrowa funkcja `SRAMconfigure` najpierw inicjuje porty wejścia-wyjścia, do których podłączona jest pamięć, a następnie konfiguruje układ FSMC. Funkcja ta nie zwraca żadnej wartości.

```
#define BANK1_SRAM2_ADDR ((uint32_t)0x64000000)
#define LB_SIZE    512
#define LB_COUNT   4096
```

Stała `BANK1_SRAM2_ADDR` definiuje początkowy adres banku pamięci. Stałe `LB_SIZE` i `LB_COUNT` określają odpowiednio rozmiar i liczbę bloków logicznych (sektorów). Rozmiar pamięci w bajtach jest iloczynem wartości tych dwóch stałych.

```
typedef uint8_t ram_disk_sector_t[LB_SIZE];
#define RAMDISK ((ram_disk_sector_t *) (BANK1_SRAM2_ADDR))
```

Żeby móc wygodnie odwoływać się do poszczególnych sektorów, zdefiniowano typ `ram_disk_sector_t` i stałą `RAMDISK`. Przy takich definicjach `RAMDISK[lba]` jest adresem sektora o numerze lba (numeracja rozpoczyna się od zera).

6.3.2. Protokół SCSI

`scsi.h`
`scsi.c`

Moduł `scsi` implementuje fragment protokołu SCSI potrzebny do zrealizowania urządzenia z interfejsem MSC. Plik `scsi.h` zawiera odpowiednie deklaracje, a w pliku `scsi.c` znajduje się implementacja.

```
void SCSIreset(void);
```

Bezparametrowa funkcja `SCSIreset` przywraca początkowy stan modułu. Powinna być wywołana w odpowiedzi na zgłoszenie przez kontroler żądania `MSC_BULK_ONLY_RESET`. Funkcja ta nie zwraca żadnej wartości.

```
uint8_t SCSIgetMaxLUN(void);
```

Bezparametrowa funkcja `SCSIgetMaxLUN` zwraca maksymalny numer obsługiwanej jednostki logicznej. Aktualna implementacja obsługuje tylko jedną jednostkę logiczną i zawsze zwraca wartość zero.

```
uint8_t SCSIcommand(uint8_t lun, uint8_t const *cb, uint8_t cb_len,
                     uint8_t **data, uint32_t *data_len,
                     scsi_direction_t *transfer_direction);
```

Funkcja `SCSIcommand` zajmuje się właściwą obsługą poleceń SCSI. Parametr `lun` zawiera numer jednostki logicznej, której dotyczy polecenie. Aktualna implementacja tej funkcji nie obsługuje wielu jednostek logicznych i ignoruje wartość tego parametru. Parametr `cb` jest wskaźnikiem na pierwszy bajt polecenia, czyli wskazuje na pole `CBWCB` pakietu CBW. Parametr `cb_len` zawiera rozmiar polecenia, czyli wartość przekazaną w polu `bCBWCBLenght` pakietu CBW. Parametr `data` jest wskaźnikiem na zmienną, za pomocą której funkcja zwraca wskaźnik na dane do wysłania lub wskaźnik na bufor w pamięci, gdzie mają być skopiowane odebrane dane. Parametr `data_len` jest wskaźnikiem na zmienną, za pomocą której funkcja zwraca rzeczywisty rozmiar danych, jakie mają być przesłane. Parametr `transfer_direction` jest wskaźnikiem na zmienną, za pomocą której funkcja zwraca kierunek przesyłania danych. Wartość ta jest istotna tylko wtedy, gdy wartość zwrócona za pomocą wskaźnika `data_len` jest niezerowa. Omawiana funkcja zwraca status wykonania polecenia, czyli wartość, która ma być umieszczona w polu `bCSWStatus` pakietu CSW.

```
typedef enum {SCSI_OUT, SCSI_IN} scsi_direction_t;
```

Pomocniczy typ wyliczeniowy `scsi_direction_t` przyjmuje dwie wartości, które określają, w jakim kierunku mają być przesyłane dane, o ile polecenie wymaga przesyłania jakichś danych. Wartość `SCSI_OUT` oznacza zapis danych, a wartość `SCSI_IN` odczyt danych przez kontroler. Jest to zgodne ze stosowaną w USB konwencją nazywania kierunków transmisji danych jako odpowiednio OUT i IN.

Przejdźmy teraz do nieco bardziej szczegółowego omówienia implementacji funkcji `SCSIcommand`. Trzy zmienne lokalne będą zawierały zdekodowane wartości parametrów polecenia. Niektóre polecenia wymagają udzielenia na nie odpowiedzi. Przy czym nie chodzi tu o dane z nośnika – do ich odczytu przewidziane jest polecenie READ. Zmienna `alloc_len` zawiera maksymalny oczekiwany rozmiar takiej odpowiedzi. Zmienna `lba` zawiera numer pierwszego bloku logicznego, który ma być odczytany z nośnika za pomocą polecenia READ bądź zapisany na nośniku za

pomocą polecenia WRITE. Zmienna `lbc` zawiera liczbę bloków logicznych, które mają uczestniczyć w tej operacji odczytu bądź zapisu. Zanim jednak rozpoczęniemy dekodowanie polecenia, sprawdzamy, czy ma ono poprawny rozmiar. Posiłkując się tabelą 6.5, sprawdzamy, czy rozmiar ten jest zgodny z kodem polecenia zapisanym w jego pierwszym bajcie `cb[0]`. Pierwsza instrukcja `if` dopuszcza wyjątek od reguły zawartej w tabeli 6.5. Jest to niezbędne, gdyż implementacja sterownika w systemach Windows zawiera błąd, który powoduje, że w pakiecie CBW dla polecenia INQUIRY podawany jest jego rozmiar większy niż 6, co jest niezgodne ze standardem. Instrukcja `else if` dokonuje właściwego sprawdzenia. Jeśli polecenie ma niepoprawny rozmiar, to wywołujemy makro `LOG`, wypisujące informację diagnostyczną na LCD, a opis błędu zapisujemy w strukturze `request_sense6`. Błąd kwalifikujemy jako niepoprawne żądanie (ang. *illegal request*). Kod `0x0e` oznacza nieprawidłowy komunikat (ang. *invalid information unit*). Następnie zapisujemy zera pod adresy wskazywane przez argumenty `data` i `data_len`, informując w ten sposób funkcję wywołującą, że nie będą przesyłane żadne dane. Na koniec zwracamy wartość `MSC_BOT_CSW_COMMAND_FAILED` oznaczającą, że wykonanie funkcji zakończyłło się błędem. Jeśli polecenie ma poprawny rozmiar, przechodzimy do jego dekodowania w instrukcji `switch`, którą omawiam nieco dalej.

```
uint8_t SCSIcommand(uint8_t lun, uint8_t const *cb, uint8_t cb_len,
                     uint8_t **data, uint32_t *data_len,
                     scsi_direction_t *transfer_direction) {
    uint32_t alloc_len, lba, lbc;

    if (cb[0] == SCSI_INQUIRY && cb_len >= 6) {
    }
    else if ((cb[0] >= 0x00 && cb[0] <= 0x1f && cb_len != 6) ||
              (cb[0] >= 0x20 && cb[0] <= 0x5f && cb_len != 10) ||
              (cb[0] >= 0x80 && cb[0] <= 0x9f && cb_len != 16) ||
              (cb[0] >= 0xa0 && cb[0] <= 0xbff && cb_len != 12)) {
        LOG("R", cb, cb_len);
        request_sense6.key = SCSI_ILLEGAL_REQUEST;
        request_sense6.code = 0x0e;
        *data = 0;
        *data_len = 0;
        return MSC_BOT_CSW_COMMAND_FAILED;
    }

    switch (cb[0]) {
        ...
    }
}
```

Makro `LOG` wyświetla na LCD napis podany jako pierwszy jego argument, a następnie wypisuje szesnastkowo początkowe `cb_len` bajtów polecenia wskazywane

nego przez cb. Okazało się ono bardzo pomocne podczas pisania i uruchamiania programu, a zostało pozostawione, gdyż implementacja wciąż ma charakter nieco eksperymentalny i nie jest pełna. Gdy zajdzie taka potrzeba, makro to można łatwo wyłączyć, zmieniając na początku pliku *scsi.c* odpowiednią instrukcję preprocesora z `#if 1` na `#if 0`.

```
typedef struct {
    uint8_t error_code;
    uint8_t obsolete;
    uint8_t key;
    uint8_t information[4];
    uint8_t add_length;
    uint8_t command_specific_information[4];
    uint8_t code;
    uint8_t qualifier;
    uint8_t specific[4];
} __packed scsi_request_sense6_data_t;
```

Opis błędu przechowujemy w zmiennej globalnej `request_sense6`, która jest strukturą typu `scsi_request_sense6_data_t`. W zasadzie korzystamy tylko z dwóch pól tej struktury. Rodzaj błędu zapisujemy w składowej `key`, a szczegółowy kod błędu w składowej `code`. Pozostałe pola mają ustalone wartości jak na poniższym wydruku.

```
static scsi_request_sense6_data_t request_sense6 = {
    0x70, 0, SCSI_NO_SENSE, {0, 0, 0, 0}, sizeof(request_sense6) - 8,
    {0, 0, 0, 0}, 0, 0, {0, 0, 0, 0}
};
```

Przejdźmy teraz do omówienia zawartości wspomnianej już instrukcji `switch` dekodującej polecenia. Obsługa polecenia TEST UNIT READY polega po prostu na jego bezwarunkowym zaakceptowaniu. Zawsze zwracamy wartość `MSC_BOT_CSW_COMMAND_PASSED` oznaczającą poprawne zakończenie funkcji. Specyfika zastosowanego w naszym urządzeniu nośnika, czyli pamięci SRAM, sprawia, że urządzenie zawsze jest gotowe do operacji odczytu i zapisu.

```
case SCSI_TEST_UNIT_READY:
    *data = 0;
    *data_len = 0;
    return MSC_BOT_CSW_COMMAND_PASSED;
```

Odpowiedzią na polecenie REQUEST SENSE powinno być odesłanie zawartości wspomnianej wyżej globalnej struktury `request_sense6`. Jej adres przypisujemy do zmiennej wskazywanej za pomocą argumentu `data`. Rozmiar odsyłanych danych wyznaczamy jako minimum z rozmiaru tej struktury i zażądanego maksymalnego rozmiaru odpowiedzi `alloc_len`. Funkcja `min` została opisana w punkcie 2.3.2. Rozmiar odsyłanych danych przypisujemy do zmiennej wskazywanej za pomocą

argumentu `data_len`. Za pomocą wskaźnika `transfer_direction` zwracamy kierunek transmisji danych. Potem wychodzimy z funkcji, zwracając wartość `MSC_BOT_CSW_COMMAND_PASSED`, czym informujemy o jej poprawnym zakończeniu.

```
case SCSI_REQUEST_SENSE:
    alloc_len = cb[4];
    *data = (uint8_t *)&request_sense6;
    *data_len = min(alloc_len, sizeof(request_sense6));
    *transfer_direction = SCSI_IN;
    return MSC_BOT_CSW_COMMAND_PASSED;
```

W odpowiedzi na polecenie INQUIRY, zależnie od jego parametrów, przekazujemy ogólne własności urządzenia zapisane w zmiennej globalnej `inquiry` lub jego numer seryjny zapisany w zmiennej globalnej `serial_number`. Jeśli polecenie zawiera nieobsługiwane wartości parametrów, to wypisujemy komunikat diagnostyczny na LCD, zapamiętujemy opis błędu w strukturze `request_sense6` i kończymy funkcję, sygnalizując błęd. Błąd kwalifikujemy jako niepoprawne żądanie z kodem `0x0e`.

```
case SCSI_INQUIRY:
    alloc_len = cb[3] << 8 | cb[4];
    if (cb[1] == 0 || cb[2] == 0) {
        *data = (uint8_t *)&inquiry;
        *data_len = min(alloc_len, sizeof(inquiry));
        *transfer_direction = SCSI_IN;
        return MSC_BOT_CSW_COMMAND_PASSED;
    }
    else if (cb[1] == 1 || cb[2] == 0x80) {
        *data = (uint8_t *)&serial_number;
        *data_len = min(alloc_len, sizeof(serial_number));
        *transfer_direction = SCSI_IN;
        return MSC_BOT_CSW_COMMAND_PASSED;
    }
    else {
        LOG("I", cb, cb_len);
        request_sense6.key = SCSI_ILLEGAL_REQUEST;
        request_sense6.code = 0x0e;
        *data = 0;
        *data_len = 0;
        return MSC_BOT_CSW_COMMAND_FAILED;
    }
```

Format danych zawierających ogólne własności urządzenia, odsyłanych w odpowiedzi na polecenie INQUIRY, deklarujemy jako strukturę typu `scsi_inquiry_data_t`.

```
typedef struct {
    uint8_t pq_pdt;
    uint8_t rmb;
    uint8_t version;
    uint8_t format;
    uint8_t add_length;
    uint8_t flags[3];
    char vendor[8];
    char product[16];
    char revision[4];
} __packed scsi_inquiry_data_t;
```

Zmienną inquiry przechowującą ogólne własności urządzenia deklarujemy jako globalną stałą strukturę powyższego typu i inicjujemy ją właściwymi wartościami jak na poniższym wydruku. Początkowe cztery pola określają typ dysku. Wartość wpisana w polu `pq_pdt` oznacza, że jest to nośnik bezpośrednio podłączony do jednostki logicznej, a dostęp do niego polega na bezpośrednim odczycie i zapisie bloków logicznych (jak to opisano w poleceniach READ i WRITE). Najbardziej znaczący bit pola `rmr` określa, czy jest to wyjmowalny nośnik danych (ang. *removable*). Pozostałe bity tego pola są nieużywane. Zerowa wartość w polu `version` oznacza brak zgody z jakimkolwiek standardem – prezentowana tu implementacja nie realizuje w pełni żadnego standardu. Wartość umieszczona w polu `format` oznacza, że jest to standardowy format odpowiedzi na polecenie INQUIRY. Pole `add_length` zawiera rozmiar danych znajdujących się za tym polem. Są to kolejno znaczniki określające dalsze własności urządzenia, nazwa producenta, nazwa produktu, numer wersji produktu. Ostatnie trzy pola są zakodowane jako napisy ASCII.

```
static const scsi_inquiry_data_t inquiry = {
    0x00, 0x80, 0x00, 0x02,
    sizeof(scsi_inquiry_data_t) - 5,
    {0, 0, 0},
    {'M', 'P', ' ', ' ', ' ', ' ', ' ', ' '},
    {'R', 'e', 'm', 'o', 'v', 'a', 'b', 'l',
     'e', ' ', 'm', 'e', 'm', 'o', 'r', 'y'},
    {'1', '.', '0', '0'}
};
```

Format danych z numerem seryjnym, odsyłanych w odpowiedzi na polecenie INQUIRY, deklarujemy jako strukturę typu `scsi_unit_serial_number_data_t`. Pole `pq_pdt` ma to samo znaczenie co poprzednio. Pole `page_code` określa rodzaj dalszej zawartości struktury. Pole `page_length` zawiera rozmiar dalszej zawartości struktury. W tym przypadku jest to numer seryjny zapisany w polu `serial_number` jako tekst ASCII.

```
typedef struct {
    uint8_t pq_pdt;
    uint8_t page_code;
    uint16_t page_length;
    char serial_number[8];
} __packed scsi_unit_serial_number_data_t;
```

Zmienną `serial_number` przechowującą numer seryjny definiujemy jako globalną stałą strukturę powyższego typu i inicjujemy ją właściwymi wartościami. Opis makra `REVERSE_UINT16` i powód jego zastosowania znajdują się w rozdziale 2.3.2.

```
static const scsi_unit_serial_number_data_t serial_number = {
    0x00, 0x80,
    REVERSE_UINT16(sizeof(scsi_unit_serial_number_data_t) - 4),
    {'0', '0', '0', '0', '0', '0', '0', '1'}
};
```

W odpowiedzi na polecenie MODE SENSE (6) powinniśmy odesłać szczegółowe informacje o urządzeniu. Informacje te zapisane są w globalnej zmiennej `mode_sense6`.

```
case SCSI_MODE_SENSE6:
    alloc_len = cb[4];
    *data = (uint8_t *)&mode_sense6;
    *data_len = min(alloc_len, sizeof(mode_sense6));
    *transfer_direction = SCSI_IN;
    return MSC_BOT_CS COMMAND_PASSED;
```

Format danych, które mają być odesłane w odpowiedzi na polecenie MODE SENSE (6), deklarujemy jako strukturę typu `scsi_mode_parameter_header6_t`.

```
typedef struct {
    uint8_t mode_data_length;
    uint8_t medium_type;
    uint8_t device_specific_parameter;
    uint8_t block_descriptor_length;
} __packed scsi_mode_parameter_header6_t;
```

Zmienną `mode_sense6` deklarujemy jako globalną stałą strukturę powyższego typu i inicjujemy ją właściwymi danymi. Pierwsze pole zawiera rozmiar następujących po nim danych, a pozostałe pola wypełniamy zerami.

```
static const scsi_mode_parameter_header6_t mode_sense6 = {
    sizeof(scsi_mode_parameter_header6_t) - 1, 0, 0, 0
};
```

Polecenie MODE SENSE (10) pełni analogczną funkcję jak polecenie MODE SENSE (6), ale ma inny format danych, które mają być odesłane.

```

case SCSI_MODE_SENSE10:
    alloc_len = cb[7] << 8 | cb[8];
    *data = (uint8_t *)&mode_sense10;
    *data_len = min(alloc_len, sizeof(mode_sense10));
    *transfer_direction = SCSI_IN;
    return MSC_BOT_CSW_COMMAND_PASSED;

```

Format danych, które mają być odesłane w odpowiedzi na polecenie MODE SENSE (10), deklarujemy jako strukturę typu `scsi_mode_parameter_header10_t`.

```

typedef struct {
    uint16_t mode_data_length;
    uint8_t medium_type;
    uint8_t device_specific_parameter;
    uint8_t long_lba;
    uint8_t reservrd;
    uint16_t block_descriptor_length;
} __packed scsi_mode_parameter_header10_t;

```

Zmienną `mode_sense10` deklarujemy jako globalną stałą strukturę powyższego typu i inicjujemy ją właściwymi danymi. Pierwsze pole zawiera rozmiar następujących po nim danych, a pozostałe pola wypełniamy zerami.

```

static const scsi_mode_parameter_header10_t mode_sense10 = {
    REVERSE_UINT16(sizeof(scsi_mode_parameter_header10_t) - 2),
    0, 0, 0, 0, 0
};

```

Polecenia START STOP UNIT i PREVENT ALLOW MEDIUM REMOVAL akceptujemy bezwarunkowo. Zależnie od parametrów tych polecień i zastosowanego nośnika powinno się tu znaleźć opróżnienie buforów, dokończenie wszystkich operacji zapisu i doprowadzenie nośnika do stanu, w którym może być on bezpiecznie odłączony bądź wyjęty.

```

case SCSI_START_STOP_UNIT:
case SCSI_PREVENT_ALLOW_MEDIUM_REMOVAL:
    *data = 0;
    *data_len = 0;
    return MSC_BOT_CSW_COMMAND_PASSED;

```

W odpowiedzi na polecenie READ FORMAT CAPACITIES należy odesłać informację o pojemności dysku. Potrzebne informacje zapisane są w zmiennej globalnej `format_capacity`.

```

case SCSI_READ_FORMAT_CAPACITIES:
    *data = (uint8_t *)&format_capacity;
    *data_len = sizeof(format_capacity);
    *transfer_direction = SCSI_IN;
    return MSC_BOT_CSW_COMMAND_PASSED;

```

Format danych, które mają być odesłane w odpowiedzi na polecenie READ FORMAT CAPACITIES, deklarujemy jako strukturę typu `scsi_format_capacity_data_t`. Składowa `lba_count` zawiera 32-bitową liczbę bloków logicznych. Można zatem zadeklarować maksymalnie $2^{32} - 1$ bloków logicznych. Składowe `block_length_msб` i `block_length` (traktowane łącznie) zawierają 24-bitowy rozmiar bloku logicznego. Pierwsza z nich zawiera najstarszy bajt, a druga dwa młodsze bajty (obowiązuje porządek grubokońcowkowy).

```
typedef struct {
    uint8_t reserved[3];
    uint8_t capacity_list_length;
    uint32_t lba_count;
    uint8_t descriptor_code;
    uint8_t block_length_msб;
    uint16_t block_length;
} __packed scsi_format_capacity_data_t;
```

Zmienną `format_capacity` deklarujemy jako globalną stałą strukturę powyższego typu i inicjujemy ją właściwymi danymi. Pole `capacity_list_length` zawiera rozmiar następujących po nim danych. W polu `descriptor_code` wpisujemy jedynkę oznaczającą, że jest to pojemność niesformatowanego dysku. Makro `REVERSE_UINT32` pełni analogiczną funkcję jak makro `REVERSE_UINT16`, ale dla danych 32-bitowych.

```
static const scsi_format_capacity_data_t format_capacity = {
    {0, 0, 0},
    sizeof(scsi_format_capacity_data_t) - 4,
    REVERSE_UINT32(LB_COUNT),
    1,
    0,
    REVERSE_UINT16(LB_SIZE)
};
```

W odpowiedzi na polecenie READ CAPACITY (10) odsyłamy informację o całkowitej pojemności dysku zapisaną w zmiennej globalnej `format_capacity`.

```
case SCSI_READ_CAPACITY10:
    *data = (uint8_t *)&capacity;
    *data_len = sizeof(capacity);
    *transfer_direction = SCSI_IN;
    return MSC_BOT_CSW_COMMAND_PASSED;
```

Format danych, które mają być odesłane w odpowiedzi na polecenie READ CAPACITY (10), deklarujemy jako strukturę typu `scsi_capacity_data_t`. Składowa `last_lba` zawiera numer ostatniego bloku logicznego. Bloki logiczne numeruje się od zera. Zatem składowa ta zawiera liczbę bloków logicznych pomniejszoną o jeden. Dzięki takiej konwencji można zadeklarować maksymalnie 2^{32}

bloków logicznych, choć prezentowana tu implementacja nie obsługuje poprawnie takiej wartości. Składowa `block_length` zawiera 32-bitowy rozmiar bloku logicznego.

```
typedef struct {
    uint32_t last_lba;
    uint32_t block_length;
} __packed scsi_capacity_data_t;
```

Zmienną `capacity` deklarujemy jako globalną stałą strukturę powyższego typu i inicjujemy ją właściwymi wartościami.

```
static const scsi_capacity_data_t capacity = {
    REVERSE_UINT32(LB_COUNT - 1),
    REVERSE_UINT32(LB_SIZE)
};
```

Obsługa poleceń READ (10) i WRITE (10) jest bardzo podobna, dlatego znaczna część implementacji realizującej te polecenia jest wspólna. Najpierw sprawdzamy, czy podany numer bloku logicznego `lba` i liczba bloków logicznych `lbc` mieszczą się w dopuszczalnym zakresie. Zastosowana postać warunku ma zapobiec sytuacji, gdy wartości `lba` i `lbc` nie są poprawne, a warunek mimo to jest fałszywy na skutek przepełnienia arytmetyki. Jeśli odczyt lub zapis miałby dotyczyć nieistniejącego bloku logicznego, wypisujemy na LCD komunikat diagnostyczny i wypełniamy strukturę `request_sense6` opisem błędu, który kwalifikujemy jako przepełnienie wolumenu (ang. *volume overflow*). Kod 0x21 oznacza zbyt duży numer bloku logicznego (ang. *logical block address out of range*). Opuszczamy funkcję, sygnalizując błęd. Jeśli wartości `lba` i `lbc` są poprawne, zwracamy właściwe wartości w zmiennych wskazywanych przez wskaźniki `data`, `data_len` i `transfer_direction`. Następnie wychodzimy z funkcji, sygnalizując jej poprawne zakończenie.

```
case SCSI_READ10:
case SCSI_WRITE10:
    lba = cb[2] << 24 | cb[3] << 16 | cb[4] << 8 | cb[5];
    lbc = cb[7] << 8 | cb[8];
    if (lbc == 0 || lbc > LB_COUNT || lba > LB_COUNT - lbc) {
        LOG(„T”, cb, cb_len);
        request_sense6.key = SCSI_VOLUME_OVERFLOW;
        request_sense6.code = 0x21;
        *data = 0;
        *data_len = 0;
        return MSC_BOT_CSW_COMMAND_FAILED;
    }
    else {
        *data = (uint8_t *)RAMDISK[lba];
        *data_len = lbc * LB_SIZE;
```

```

    if (cb[0] == SCSI_READ10)
        *transfer_direction = SCSI_IN;
    else
        *transfer_direction = SCSI_OUT;
    return MSC_BOT_CSW_COMMAND_PASSED;
}

```

Jeśli nie zostało rozpoznane żadne polecenie, to przechodzimy do sekcji `default`, w której wypisujemy komunikat diagnostyczny na LCD i zapamiętujemy opis błędu w strukturze `request_sense6`. Błąd kwalifikujemy jako niepoprawne żądanie. Kod 0x20 oznacza niepoprawny kod operacji polecenia (ang. *invalid command operation code*). Następnie opuszczamy funkcję, sygnalizując błąd.

```

default:
    LOG("C", cb, cb_len);
    request_sense6.key = SCSI_ILLEGAL_REQUEST;
    request_sense6.code = 0x20;
    *data = 0;
    *data_len = 0;
    return MSC_BOT_CSW_COMMAND_FAILED;
}

```

Przedstawiona w tym podrozdziale implementacja realizuje tylko najbardziej podstawowe funkcje pamięci masowej. Nie zostało zaimplementowane polecenia VERIFY, a implementacja wielu innych poleceń jest bardzo uproszczona. Rozszerzenie implementacji w kierunku pełnej obsługi poleceń SCSI pozostawiam Czytelnikowi jako dość ambitne ćwiczenie. Każdą nową implementację trzeba jednak móc przetestować, czyli znaleźć scenariusz, w którym aktualna implementacja zawodzi.

6.3.3. Protokoły USB i BOT

`ex_msc_hs_dev.c`

Implementacja warstwy aplikacji USB oraz protokołu BOT znajduje się w pliku `ex_msc_hs_dev.c`. Zaczynamy jak poprzednio od zdefiniowania wszystkich deskryptorów. Przy czym pomijam opis deskryptorów tekstowych, gdyż nie różnią się one istotnie od deskryptorów tekstowych urządzeń przedstawionych w poprzednich rozdziałach. Zauważmy, że deskryptory dla tego urządzenia, inne niż tekstowe, w odróżnieniu od dotychczas przedstawionych, nie są strukturami stałymi, a pola `bMaxPacketSize0` i `bMaxPacketSize` są początkowo wyzerowane. Pola te będą wypełniane dynamicznie, zależnie od szybkości, z jaką zostało uruchomione urządzenie. Na początek definiujemy deskryptor urządzenia, który określa parametry dla aktualnej szybkości pracy urządzenia i który jest wypełniony standardowo. Klasa urządzenia jest definiowana na poziomie interfejsu. Urządzeniu przypisujemy VID 0x0483 i PID 0x5756 – patrz plik `usb_vid_pid.h`.

```

static usb_device_descriptor_t device_descriptor = {
    sizeof(usb_device_descriptor_t), /* bLength */
    DEVICE_DESCRIPTOR, /* bDescriptorType */

```

```
HTOUSBS(0x0200), /* bcdUSB */
0x00, /* bDeviceClass */
0x00, /* bDeviceSubClass */
0x00, /* bDeviceProtocol */
0, /* bMaxPacketSize0 */
HTOUSBS(VID), /* idVendor */
HTOUSBS(PID + 6), /* idProduct */
HTOUSBS(0x0100), /* bcdDevice */
1, /* iManufacturer */
2, /* iProduct */
3, /* iSerialNumber */
1 /* bNumConfigurations */  
};
```

Urządzenie mogące pracować w trybach FS i HS musi mieć deskryptory kwalifikujące, który określają parametry urządzenia dla drugiej szybkości, czyli szybkości, z którą urządzenie aktualnie nie jest uruchomione. W naszym przypadku nie ma powodu, aby deskryptory kwalifikujące urządzenia zawierały inne wartości parametrów niż standardowy deskryptor urządzenia, być może z wyjątkiem wspomnianego już pola bMaxPacketSize0.

```
static usb_device_qualifier_descriptor_t device_qualifier = {  
    sizeof(usb_device_qualifier_descriptor_t), /* bLength */  
    DEVICE_QUALIFIER_DESCRIPTOR, /* bDescriptorType */  
    HTOUSBS(0x0200), /* bcdUSB */  
    0x00, /* bDeviceClass */  
    0x00, /* bDeviceSubClass */  
    0x00, /* bDeviceProtocol */  
    0, /* bMaxPacketSize0 */  
    1, /* bNumConfigurations */  
    0 /* bReserved */  
};
```

Z powyższych deskryptorów wynika, że urządzenie ma tylko jedną konfigurację, którą definiujemy jako strukturę typu `usb_configuration_t`. Zawiera ona deskryptor konfiguracji, deskryptor interfejsu i dwa deskryptory punktów końcowych dla danych masowych: wejściowego i wyjściowego.

```
typedef struct {
    usb_configuration_descriptor_t cnf_descr;
    usb_interface_descriptor_t      if_descr;
    usb_endpoint_descriptor_t      ep_descr[2];
} packed usb_configuration_t;
```

Konfigurację wypełniamy jak na poniższym wydruku. Wartość umieszczona w polu **bInterval** deskryptora punktu końcowego ma znaczenie tylko dla trybu HS i wyj-

ściowego punktu końcowego. Mówiąc ona, po ilu ramkach kontroler powinien powtórzyć transakcję OUT, gdy otrzyma z tego punktu końcowego negatywne potwierdzenie NAK. W pozostałych przypadkach wartość ta jest nieistotna i jest ignorowana przez kontroler.

```
static usb_configuration_t msc_config = {
{
    sizeof(usb_configuration_descriptor_t), /* bLength */
    CONFIGURATION_DESCRIPTOR, /* bDescriptorType */
    HTOUSBS(sizeof(usb_configuration_t)), /* wTotalLength */
    1, /* bNumInterfaces */
    1, /* bConfigurationValue */
    0, /* iConfiguration */
    USB_BM_ATTRIBUTES, /* bmAttributes */
    USB_B_MAX_POWER /* bMaxPower */
},
{
    sizeof(usb_interface_descriptor_t), /* bLength */
    INTERFACE_DESCRIPTOR, /* bDescriptorType */
    0, /* bInterfaceNumber */
    0, /* bAlternateSetting */
    2, /* bNumEndpoints */
    MASS_STORAGE_CLASS, /* bInterfaceClass */
    SCSI_TRANSPARENT_SUBCLASS, /* bInterfaceSubClass */
    BULK_ONLY_TRANSPORT_PROTOCOL, /* bInterfaceProtocol */
    0 /* iInterface */
},
{
    sizeof(usb_endpoint_descriptor_t), /* bLength */
    ENDPOINT_DESCRIPTOR, /* bDescriptorType */
    ENDP1 | ENDP_IN, /* bEndpointAddress */
    BULK_TRANSFER, /* bmAttributes */
    0, /* wMaxPacketSize */
    1 /* bInterval */
},
{
    sizeof(usb_endpoint_descriptor_t), /* bLength */
    ENDPOINT_DESCRIPTOR, /* bDescriptorType */
    ENDP1 | ENDP_OUT, /* bEndpointAddress */
    BULK_TRANSFER, /* bmAttributes */
    0, /* wMaxPacketSize */
    1 /* bInterval */
}
};
```

Większość stałych symbolicznych użytych do zainicjowania pól deskryptora konfiguracji, deskryptora interfejsu i deskryptorów punktów końcowych zdefiniowano w pliku *usb.def*. Wartości atrybutów związanych z zasilaniem zdefiniowane są w pliku *board_usb_def.h* w sposób przedstawiony poniżej – zgodnie z opisem w rozdziale 2.1.2 zestaw STM3220G-EVAL zasilamy za pomocą dostarczonego z nim zasilacza.

```
#define USB_BM_ATTRIBUTES    (SELF_POWERED | D7_RESERVED)
#define USB_B_MAX_POWER      (2 / 2)
```

W globalnej stałej strukturze *ApplicationCallBacks* typu *usbd_callback_list_t* umieszczamy adresy wszystkich wykorzystywanych przez urządzenie funkcji zwrotnych.

```
static usbd_callback_list_t const ApplicationCallBacks = {
    Configure, Reset, 0, GetDescriptor, 0, GetConfiguration,
    SetConfiguration, GetStatus, 0, 0, 0, 0, ClassNoDataSetup,
    ClassInDataSetup, 0, 0,
    {EP1IN, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {EP1OUT, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    0, 0
};
```

Funkcja *USBDgetApplicationCallbacks* zwraca adres struktury *ApplicationCallBacks*. Funkcja ta jest wywoływana podczas konfigurowania urządzenia. Jest to jedyna funkcja eksportowana (widoczna) na zewnątrz modułu *ex_msc_hs_dev.c*.

```
usbd_callback_list_t const * USBDgetApplicationCallbacks() {
    return &ApplicationCallBacks;
}
```

BOT jest protokołem stanowym. W dalszej części tego podrozdziału zapoznamy się z implementacją jego automatu stanowego. Najpierw jednak definiujemy typ wyliczeniowy *msc_bot_state_t* reprezentujący stany tego automatu. Początkowym stanem jest *MSC_BOT_READY*. Po odebraniu pakietu CBW automat przechodzi do jednego z trzech stanów: *MSC_BOT_DATA_OUT*, *MSC_BOT_DATA_IN*, *MSC_BOT_CSW*. Automat znajduje się w stanie *MSC_BOT_DATA_OUT*, gdy ma odebrać jakieś dane, a w stanie *MSC_BOT_DATA_IN*, gdy ma jakieś dane wysłać. Do stanu *MSC_BOT_CSW* przechodzi po wstawieniu do wysłania pakietu CSW – również bezpośrednio ze stanu *MSC_BOT_READY*, gdy nie ma żadnych danych do przesyłania. Automat powraca do stanu *MSC_BOT_READY* po zakończeniu wysyłania pakietu CSW. Automat przechodzi do stanu *MSC_BOT_ERROR*, gdy wystąpił błąd i czeka w tym stanie na żądanie *MSC_BULK_ONLY_RESET*, po którego obsłużeniu wraca do stanu początkowego *MSC_BOT_READY*.

```
typedef enum {
    MSC_BOT_READY,
    MSC_BOT_DATA_OUT,
    MSC_BOT_DATA_IN,
```

```

    MSC_BOT_CSW,
    MSC_BOT_ERROR
} msc_bot_state_t;

```

Stan urządzenia przechowujemy w kilku zmiennych globalnych. Zmienna TransferSize przechowuje aktualny maksymalny rozmiar pola danych pakietów dla danych masowych. Zauważmy, że rozmiar ten jest różny dla trybów FS i HS – patrz tabela 1.10. Zmienna CurrentSpeed przechowuje aktualną szybkość, z jaką pracuje urządzenie (FS lub HS). Zmienna BOTstate przechowuje stan protokołu BOT. Zmienna CurrentConfiguration przechowuje numer aktywnej konfiguracji. Z przedstawionych wyżej deskryptorów wynika, że urządzenie ma tylko jedną konfigurację o numerze jeden, więc zmienna ta przyjmuje wartość jeden, gdy konfiguracja ta została wybrana, lub zero, gdy nie została wybrana. Zmienna MaxLUN przechowuje maksymalny numer jednostki logicznej. Ponieważ aktualna implementacja nie obsługuje wielu jednostek logicznych, zmienna ta ma zawsze wartość zero. Została jednak zadeklarowana, aby ułatwić w przyszłości ewentualne rozszerzenie implementacji o obsługę wielu jednostek logicznych.

```

static uint32_t           TransferSize;
static usb_speed_t        CurrentSpeed;
static msc_bot_state_t   BOTstate;
static uint8_t             CurrentConfiguration;
static uint8_t             MaxLUN;

```

Z wyjątkiem dwóch pierwszych, pozostałe zmienne globalne są inicjowane w pomocniczej funkcji ResetState.

```

static void ResetState(void) {
    BOTstate = MSC_BOT_READY;
    CurrentConfiguration = 0;
    MaxLUN = SCSIgetMaxLUN();
}

```

Funkcja Configure wywoływana jest podczas konfigurowania urządzenia. Konfiguruje ona kontroler zewnętrznej pamięci, wywołując funkcję SRAMconfigure. Zeruje stan protokołu SCSI za pomocą funkcji SCSClreset. Następnie wywołuje funkcję ResetState inicjującą stan urządzenia. Na koniec za pomocą makra LOG wypisuje na LCD komunikat diagnostyczny informujący, że urządzenie zostało skonfigurowane. Funkcja ta nigdy nie zgłasza błędu – zawsze zwraca wartość zero.

```

int Configure() {
    SRAMconfigure();
    SCSClreset();
    ResetState();
    LOG("USB MSC SCSI RAM DISK\n");
    return 0;
}

```

Funkcja Reset wywoływana jest po każdorazowym wyzerowaniu szyny USB. Jej parametrem jest szybkość, z jaką zostało uruchomione urządzenie. Zależnie od tej szybkości inicjujemy zmienne CurrentSpeed i TransferSize oraz maksymalne rozmiary pakietów dla danych sterujących i masowych. Zmieniamy też odpowiednio indeks deskryptora tekstowego opisującego konfigurację. Wypisujemy na LCD komunikat diagnostyczny, informujący o szybkości urządzenia. Na koniec konfigurujemy domyślny punkt końcowy dla danych sterujących i zwracamy rozmiar pola danych pakietów przesyłanych przez ten punkt końcowy. Jeśli parametr speed ma niepoprawną wartość lub nie udało się skonfigurować punktu końcowego dla danych sterujących, to wykonujemy trzy sekwencje mignięć po odpowiednio 6 lub 7 razy czerwoną diodą świecącą i zerujemy mikrokontroler za pomocą funkcji ErrorResetable.

```
uint8_t Reset(usb_speed_t speed) {
    ResetState();
    CurrentSpeed = speed;
    if (speed == FULL_SPEED) {
        device_descriptor.bMaxPacketSize0 = MAX_FS_CONTROL_PACKET_SIZE;
        device_qualifier.bMaxPacketSize0 = MAX_HS_CONTROL_PACKET_SIZE;
        msc_config.cnf_descr.iConfiguration = 4;
        TransferSize = MAX_FS_BULK_PACKET_SIZE;
        msc_config.ep_descr[0].wMaxPacketSize = HTOUSBS(TransferSize);
        msc_config.ep_descr[1].wMaxPacketSize = HTOUSBS(TransferSize);
        LOG("FS ");
    }
    else if (speed == HIGH_SPEED) {
        device_descriptor.bMaxPacketSize0 = MAX_HS_CONTROL_PACKET_SIZE;
        device_qualifier.bMaxPacketSize0 = MAX_FS_CONTROL_PACKET_SIZE;
        TransferSize = MAX_HS_BULK_PACKET_SIZE;
        msc_config.cnf_descr.iConfiguration = 5;
        msc_config.ep_descr[0].wMaxPacketSize = HTOUSBS(TransferSize);
        msc_config.ep_descr[1].wMaxPacketSize = HTOUSBS(TransferSize);
        LOG("HS ");
    }
    else
        ErrorResetable(-1, 6);
    if (USBDDendPointConfigure(ENDP0, CONTROL_TRANSFER,
                                device_descriptor.bMaxPacketSize0,
                                device_descriptor.bMaxPacketSize0) != REQUEST_SUCCESS)
        ErrorResetable(-1, 7);
    return device_descriptor.bMaxPacketSize0;
}
```

Zauważmy, że rozmiar pola danych masowych dla punktów końcowych deklarujemy jako 64 lub 512 bajtów (odpowiednio stałe `MAX_FS_BULK_PACKET_SIZE` i `MAX_HS_BULK_PACKET_SIZE`). Jest to zawsze więcej niż rozmiar struktur danych odsyłanych przez polecenia SCSI inne niż READ. Dzięki temu każda z nich odsyłana jest jako jeden tak zwany krótki pakiet USB, czyli pakiet o rozmiarze mniejszym niż maksymalny dopuszczalny. Zakładamy też, że polecenie READ wymaga zawsze przesłania dokładnie tylu danych, ile zażądał kontroler, i ich rozmiar jest wielokrotnością rozmiaru sektora, czyli 512 bajtów. Zatem READ zawsze skutkuje przesaniem pełnych pakietów. Pozwala to uprościć implementację urządzenia. W sytuacji, gdy chcielibyśmy przesyłać do kontrolera mniej danych, niż zażądał i byłaby to wielokrotność rozmiaru pola danych pakietu, musielibyśmy wysłać na koniec dodatkowy pusty pakiet.

Funkcja `GetDescriptor` jest wywoływaną po odebraniu przez urządzenie żądania `GET_DESCRIPTOR`. Jej implementacja jest już znana z dotychczas omówionych urządzeń. Jedyną istotną różnicą jest obsługa deskryptora kwalifikującego urządzenia.

```
usb_result_t GetDescriptor(uint16_t wValue, uint16_t wIndex,
                           uint8_t const **data, uint16_t *length) {
    uint32_t index = wValue & 0xff;
    switch (wValue >> 8) {
        case DEVICE_DESCRIPTOR:
            if (index == 0 && wIndex == 0) {
                *data = (uint8_t const *)&device_descriptor;
                *length = sizeof(usb_device_descriptor_t);
                return REQUEST_SUCCESS;
            }
            return REQUEST_ERROR;
        case DEVICE_QUALIFIER_DESCRIPTOR:
            if (index == 0 && wIndex == 0) {
                *data = (uint8_t const *)&device_qualifier;
                *length = sizeof(usb_device_qualifier_descriptor_t);
                return REQUEST_SUCCESS;
            }
            return REQUEST_ERROR;
        case CONFIGURATION_DESCRIPTOR:
            if (index == 0 && wIndex == 0) {
                *data = (uint8_t const *)&msc_config;
                *length = sizeof(usb_configuration_t);
                return REQUEST_SUCCESS;
            }
            return REQUEST_ERROR;
        case STRING_DESCRIPTOR:
            if (index < stringCount) {
```

```

        *data = strings[index].data;
        *length = strings[index].length;
        return REQUEST_SUCCESS;
    }
    return REQUEST_ERROR;
default:
    return REQUEST_ERROR;
}
}

```

Funkcja GetConfiguration jest wywoływana po odebraniu przez urządzenie żądania GET_CONFIGURATION. Zwraca ona numer aktualnej konfiguracji, czyli wartość jeden, gdy jedyna konfiguracja naszego urządzenia została uaktywniona, a wartość zero, gdy nie została jeszcze uaktywniona lub została dezaktywowana.

```

uint8_t GetConfiguration() {
    return CurrentConfiguration;
}

```

Funkcja SetConfiguration jest wywoływana po odebraniu przez urządzenie żądania SET_CONFIGURATION. Parametr confValue zawiera numer żądanej konfiguracji lub zero, gdy aktualna konfiguracja ma zostać dezaktywowana. Najpierw sprawdzamy jego poprawność. Jeśli jest poprawny i jest to żądanie ustawienia konfiguracji, to konfigurujemy dwukierunkowy punkt końcowy dla danych masowych.

```

usb_result_t SetConfiguration(uint16_t confValue) {
    if (confValue > device_descriptor.bNumConfigurations)
        return REQUEST_ERROR;
    CurrentConfiguration = confValue;
    USBDisableAllNonControlEndPoints();
    if (confValue == msc_config.cnf_descr.bConfigurationValue)
        return USBDEndPointConfigure(ENDP1, BULK_TRANSFER,
                                      TransferSize, TransferSize);
    return REQUEST_SUCCESS;
}

```

Funkcja GetStatus jest wywoływana po odebraniu przez urządzenie żądania GET_STATUS. Zwraca ona aktualny status urządzenia związany z zasilaniem i zdalnym budzeniem. Ponieważ nasze urządzenie ma własne zasilanie, nie pobiera prądu z szyny USB i nie obsługuje funkcji zdalnego budzenia, zwracamy wartość STATUS_SELF_POWERED.

```

uint16_t GetStatus() {
    return STATUS_SELF_POWERED;
}

```

Funkcja ClassNoDataSetup obsługuje żądania specyficzne dla interfejsu MSC i niewymagające przesyłania danych. W naszym przypadku jest tylko jedno takie żądanie, a mianowicie `MSC_BULK_ONLY_RESET`. Realizacja tego żądania polega na ustawieniu początkowego stanu protokołów BOT i SCSI. Na koniec wypisujemy na LCD komunikat diagnostyczny.

```
usb_result_t ClassNoDataSetup(usb_setup_packet_t const *setup) {
    if (setup->bmRequestType == (HOST_TO_DEVICE |
                                  CLASS_REQUEST |
                                  INTERFACE_RECIPIENT) &&
        setup->bRequest == MSC_BULK_ONLY_RESET &&
        setup->wValue == 0 &&
        setup->wIndex == 0 /* interface number */ /
        setup->wLength == 0) {
        BOTstate = MSC_BOT_READY;
        SCSIreset();
        LOG("R ");
        return REQUEST_SUCCESS;
    }
    return REQUEST_ERROR;
}
```

Funkcja ClassInDataSetup obsługuje żądania specyficzne dla interfejsu MSC i wymagające przesyłania danych z urządzenia do kontrolera. W naszym przypadku jest tylko jedno takie żądanie, a mianowicie `MSC_GET_MAX_LUN`. Jego realizacja polega na wysłaniu wartości zmiennej globalnej `MaxLUN`.

```
usb_result_t ClassInDataSetup(usb_setup_packet_t const *setup,
                             uint8_t const **data,
                             uint16_t *length) {
    if (setup->bmRequestType == (DEVICE_TO_HOST |
                                  CLASS_REQUEST |
                                  INTERFACE_RECIPIENT) &&
        setup->bRequest == MSC_GET_MAX_LUN &&
        setup->wValue == 0 &&
        setup->wIndex == 0 /* interface number */ /
        setup->wLength == 1) {
        *data = &MaxLUN;
        *length = 1;
        return REQUEST_SUCCESS;
    }
    return REQUEST_ERROR;
}
```

Przejdźmy teraz do obiecanego omówienia automatu stanowego protokołu BOT. Korzystamy w nim z dwóch pomocniczych funkcji. Funkcja `BOTreadCBW` odczytuje pakiet CBW i jeśli liczba odczytanych bajtów jest poprawnym rozmiarem tego pakietu, to wykonuje niezbędne zamiany kolejności bajtów w jego wielobajtowych polach. Funkcja ta zwraca zero, gdy cały pakiet został odczytany poprawnie, a wartość ujemną, gdy wystąpił błąd.

```
static int BOTreadCBW(msc_bot_cbw_t *p) {
    uint32_t len;
    len = USBDread(ENDP1, (uint8_t *)p, sizeof(msc_bot_cbw_t));
    if (len == sizeof(msc_bot_cbw_t)) {
        p->dCBWSignature      = USBTOHL(p->dCBWSignature);
        p->dCBWTag            = USBTOHL(p->dCBWTag);
        p->dCBWDataTransferLength = USBTOHL(p->dCBWDataTransferLength);
        return 0;
    }
    return -1;
}
```

Funkcja `BOTwriteCSW` wstawia do wysłania pakiet CSW, wykonując wpierw niezbędne zamiany kolejności bajtów w jego wielobajtowych polach. Funkcja ta nie zwraca żadnej wartości.

```
static void BOTwriteCSW(msc_bot_csw_t *p) {
    p->dCSWSignature     = HTOUSBL(p->dCSWSignature);
    p->dCSWTag           = HTOUSBL(p->dCSWTag);
    p->dCSWDataResidue  = HTOUSBL(p->dCSWDataResidue);
    USBDwrite(ENDP1, (const uint8_t *)p, sizeof(msc_bot_csw_t));
}
```

Automat stanowy protokołu BOT przechowuje odczytany ostatnio pakiet CBW w globalnej zmiennej `cbw`. Pakiet CSW, który ma być wysłany w odpowiedzi, jest przechowywany w globalnej zmiennej `csw`. Ponadto automat ten korzysta z globalnego wskaźnika `data`, w którym przechowuje adres danych do wysłania lub adres w pamięci, gdzie mają być skopiowane odebrane dane.

```
static msc_bot_cbw_t      cbw;
static msc_bot_csw_t      csw;
static uint8_t             *data;
```

Automat pobudzany jest do działania w funkcjach zwrotnych informujących o odebraniu danych bądź zakończeniu wysyłania danych. Funkcja `EP1OUT` jest wywoływana, gdy urządzenie odebrało pakiet danych masowych. Jeśli automat (protokół) jest w stanie `MSC_BOT_READY`, podejmujemy próbę odczytania pakietu CBW za pomocą funkcji `BOTreadCBW`, a gdy to się powiedzie, weryfikujemy po-

prawność jego pól. Jeśli pakiet CBW jest poprawny, to inicjujemy pola pakietu CSW, a następnie wywołujemy funkcję `SCSIcommand` w celu wykonania polecenia SCSI. Jeśli wykonanie polecenia nie powiodło się (funkcja zwróciła wartość `MSC_BOT_CSW_COMMAND_FAILED`) lub nie ma żadnych danych do przesłania (pole `dCBWDataTransferLength` i zmienna `len` mają wartość zero), to przechodzimy od razu do stanu `MSC_BOT_CSW` i wstawiamy do wysłania pakiet CSW za pomocą funkcji `BOTwriteCSW`. Jeśli są jakieś dane do wysłania, to bardzo dokładnie sprawdzamy poprawność parametrów. Mianowicie sprawdzamy, czy pole `dCBWDataTransferLength` ma wartość dodatnią i czy jego wartość jest równa wartości zmiennej `len`. Za poprawną uznajemy również sytuację, w której wartości te nie są równe i do wysłania jest jeden krótki pakiet, czyli gdy zmienna `len` ma wartość mniejszą niż maksymalny rozmiar pola danych pakietu przechowywany w zmiennej `TransferSize`. Sprawdzamy też, czy kierunek transmisji zwrócony w zmiennej `transfer_direction` pokrywa się z bitem kierunku w pakiecie CBW. Jeśli wszystko się zgadza, to przechodzimy do stanu `MSC_BOT_DATA_IN`. Uaktualniamy rozmiar danych do wysłania w polu `dCBWDataTransferLength`. Wstawiamy do wysyłania jeden pakiet z danymi za pomocą funkcji `USBDwrite`. Przed jej wywołaniem zmienna `len` zawiera liczbę bajtów do wysłania. Po jej wywołaniu zapisujemy w tej zmiennej liczbę bajtów rzeczywiście wstawionych do wysłania. Przesuwamy wskaźnik `data` o liczbę wysłanych bajtów i zmniejszamy odpowiednio rozmiar danych pozostałych do wysłania `dCBWDataTransferLength` i wartość pola `dCSWDataResidue` pakietu CSW. Równie dokładnie sprawdzamy, czy są jakieś dane do odebrania. Sprawdzamy mianowicie, czy pole `dCBWDataTransferLength` ma wartość dodatnią i czy jego wartość jest równa wartości zmiennej `len`. Sprawdzamy też, czy kierunek transmisji zwrócony w zmiennej `transfer_direction` pokrywa się z bitem kierunku w pakiecie CBW. Jeśli wszystko się zgadza, to przechodzimy do stanu `MSC_BOT_DATA_OUT`, w którym będziemy oczekiwac na przesypane dane. Jeśli powyższe warunki nie są spełnione, to uznajemy, że doszło do utraty synchronizacji w protokole BOT. Kontroler oczekuje przesłania danych o innym rozmiarze lub w innym kierunku niż urządzenie. Przechodzimy do stanu `MSC_BOT_ERROR`. W polu komunikatu CSW wpisujemy `MSC_BOT_CSW_PHASE_ERROR` i wysyłamy komunikat CSW.

```
void EP1OUT() {
    uint32_t len;
    scsi_direction_t transfer_direction;
    if (BOTstate == MSC_BOT_READY &&
        BOTreadCBW(&cbw) == 0 &&
        cbw.dCBWSignature == MSC_BOT_CBW_SIGNATURE &&
        (cbw.bmCBWFlags & MSC_BOT_CBW_FLAGS_RESERVED_BITS) == 0 &&
        cbw.bCBWLUN <= MaxLUN &&
        cbw.bCBWCBLength > 0 &&
        cbw.bCBWCBLength <= MSC_CBWCB_LENGTH) {
        csw.dCSWSignature = MSC_BOT_CSW_SIGNATURE;
```

```
    csw.dCSWTag = cbw.dCBWTag;
    csw.dCSWDataResidue = cbw.dCBWDataTransferLength;
    csw.bCSWStatus = SCSIcommand(cbw.bCBWLUN, cbw.CBWCB,
                                  cbw.bCBWCBLlength, &data, &len,
                                  &transfer_direction);
    if (csw.bCSWStatus == MSC_BOT_CSW_COMMAND_FAILED ||
        (cbw.dCBWDataTransferLength == 0 && len == 0)) {
        BOTstate = MSC_BOT_CSW;
        BOTwriteCSW(&csw);
    }
    else if (cbw.dCBWDataTransferLength > 0 &&
              (cbw.dCBWDataTransferLength == len ||
               len < TransferSize) &&
               transfer_direction == SCSI_IN &&
               (cbw.bmCBWFlags & MSC_BOT_CBW_DATA_IN) != 0) {
        BOTstate = MSC_BOT_DATA_IN;
        cbw.dCBWDataTransferLength = len;
        len = min(cbw.dCBWDataTransferLength, TransferSize);
        len = USBDwrite(ENDP1, data, len);
        data += len;
        cbw.dCBWDataTransferLength -= len;
        csw.dCSWDataResidue -= len;
    }
    else if (cbw.dCBWDataTransferLength > 0 &&
              cbw.dCBWDataTransferLength == len &&
              transfer_direction == SCSI_OUT &&
              (cbw.bmCBWFlags & MSC_BOT_CBW_DATA_IN) == 0) {
        BOTstate = MSC_BOT_DATA_OUT;
    }
    else {
        BOTstate = MSC_BOT_ERROR;
        csw.bCSWStatus = MSC_BOT_CSW_PHASE_ERROR;
        BOTwriteCSW(&csw);
    }
}
else if (BOTstate == MSC_BOT_DATA_OUT) {
    len = min(cbw.dCBWDataTransferLength, TransferSize);
    len = USBDread(ENDP1, data, len);
```

```

        data += len;
        cbw.dCBWDataTransferLength -= len;
        csw.dCSWDataResidue -= len;
        if (cbw.dCBWDataTransferLength == 0) {
            BOTstate = MSC_BOT_CS;
            BOTwriteCSW(&csw);
        }
    }
    else {
        USBDread(ENDP1, 0, 0);
        BOTstate = MSC_BOT_ERROR;
        USBDsetEndPointHalt(ENDP1 | ENDP_IN);
        USBDsetEndPointHalt(ENDP1 | ENDP_OUT);
        LOG("EO ");
    }
}

```

Jeśli automat jest w stanie `MSC_BOT_DATA_OUT`, to interpretujemy odbierane pakietły jako przesypane dane, które kopujemy za pomocą funkcji `USBDread` w miejsce wskazywane przez zmienną `data`. Dane przesypane są w pakietach o maksymalnym rozmiarze określonym przez zmienną `TransferSize`. Przed odczytaniem danych zmienna `len` zawiera liczbę bajtów, które chcemy skopiować w tym wywołaniu funkcji `USBDread`. Pole `dCBWDataTransferLength` przechowuje rozmiar danych, jakie jeszcze pozostały do odebrania. Po odczytaniu kolejnego pakietu w zmiennej `len` znajduje się liczba rzeczywiście skopiowanych bajtów. Przesuwamy wskaźnik `data` o tę wartość i zmniejszamy odpowiednio rozmiar danych pozostałych do odebrania `dCBWDataTransferLength` oraz wartość pola `dCSWDataResidue`. Jeśli wartość pola `dCBWDataTransferLength` osiągnie zero, to oznacza, że wszystkie dane zostały odebrane. Przechodzimy do stanu `MSC_BOT_CS` i za pomocą funkcji `BOTwriteCSW` wstawiamy do wysłania komunikat `CSW`.

Jeśli funkcja `EP1OUT` zostanie wywołana (zostań odebrane dane) w stanie innym niż `MSC_BOT_READY` lub `MSC_BOT_DATA_OUT`, to wywołujemy funkcję `USBDread` z zerowymi parametrami w celu opróżnienia bufora odbiorczego i przechodzimy do stanu `MSC_BOT_ERROR`. Blokujemy też punkty końcowe za pomocą funkcji `USBDsetEndPointHalt`. Od tej pory będą one reagowały pakietem STALL na wszelkie próby zainicjowania transmisji przez kontroler. Na koniec wypisujemy na LCD komunikat diagnostyczny. W stanie `MSC_BOT_ERROR` pozostajemy aż do odebrania żądania `MSC_BULK_ONLY_RESET`, po którym powinny nastąpić żądania `CLEAR_FEATURE` uaktywniające punkty końcowe. Urządzenie z tego stanu wyprowadzi również odłączenie i ponowne podłączenia interfejsu USB (wyzerowanie szyny) bez odłączania zasilania urządzenia.

Funkcja `EP1IN` jest wywoływana, gdy zakończy się wysyłanie danych wstawionych do kolejki nadawczej za pomocą funkcji `USBDwrite`. Jeśli automat jest w stanie

MSC_BOT_DATA_IN i nie wszystkie dane zostały już wysłane (pole dCBWDataTransferLength ma niezerową wartość), to wstawiamy do wysłania kolejną porcję danych i odpowiednio modyfikujemy wartości data, dCBWDataTransferLength i dCSWDataResidue. W przeciwnym przypadku, czyli gdy wszystkie dane zostały już wysłane, przechodzimy do stanu MSC_BOT_CSW i wstawiamy do wysłania komunikat CSW.

```
void EP1IN() {
    uint32_t len;
    if (BOTstate == MSC_BOT_DATA_IN) {
        if (cbw.dCBWDataTransferLength > 0) {
            len = min(cbw.dCBWDataTransferLength, TransferSize);
            len = USBDwrite(ENDP1, data, len);
            data += len;
            cbw.dCBWDataTransferLength -= len;
            csw.dCSWDataResidue -= len;
        }
    } else {
        BOTstate = MSC_BOT_CSW;
        BOTwriteCSW(&csw);
    }
} else if (BOTstate == MSC_BOT_CSW) {
    BOTstate = MSC_BOT_READY;
}
else {
    BOTstate = MSC_BOT_ERROR;
    USBDsetEndPointHalt(ENDP1 | ENDP_IN);
    USBDsetEndPointHalt(ENDP1 | ENDP_OUT);
    LOG("EI ");
}
```

Jeśli automat jest w stanie MSC_BOT_CSW, to oznacza, że pakiet CSW został wysłany, cała sekwencja protokołu BOT zakończyła się poprawnie i można powrócić do stanu początkowego MSC_BOT_READY. W innych, nieprzewidzianych sytuacjach przechodzimy do stanu MSC_BOT_ERROR, w którym wstrzymujemy wszelkie transmisje. Po czym wypisujemy na LCD komunikat diagnostyczny. W stanie MSC_BOT_ERROR oczekujemy, jak to wyżej opisano, na reakcję kontrolera lub użytkownika.

Przedstawiona implementacja protokołu BOT jest nieco uproszczona. Opisana w [31] obsługa błędów jest bardziej wyrafinowana – rozróżnianych jest więcej przypadków. Mimo to powyższa implementacja wydaje się w pełni funkcjonalna.

6.4. Kompilowanie i testowanie

Archiwum z przykładami zawiera tylko jedną wersję tego projektu, mianowicie wersję dla zestawu STM3220G-EVAL. Spośród zaprezentowanych w rozdziale 2 płyt prototypowych tylko ten zestaw ma interfejs USB HS. W katalogu `./make` znajduje się podkatalog `usb6_msc_hs_device_207`, w którym umieszczony jest plik `makefile` umożliwiający skompilowanie projektu. Można go oczywiście dostosować do innego sprzętu, posługując się wskazówkami z rozdziału 2. W szczególności można go również przenieść na model mikrokontrolera z układem USB obsługującym tylko tryb FS, rezygnując z trybu HS. Plik `makefile` zawiera listę plików źródłowych niezbędnych do skompilowania programu. Jest zatem przydatny również dla tych, którzy nie chcą korzystać bezpośrednio z programu make. Wtedy należy pamiętać, aby dołączyć plik `startup_stm32.c` i właściwą wersję biblioteki STM32.

Zestaw STM3220G-EVAL ma dwa interfejsy USB: CN8 oznaczony też jako USB OTG FS oraz CN9 oznaczony jako USB OTG HS. Urządzenie może działać jako FS na obu interfejsach albo jako HS na interfejsie CN9. Wyboru dokonuje się za pomocą przełącznika BOOT2 i przycisku *Wakeups* zgodnie z opisem w tabeli 2.5. Jeśli urządzenie ma pracować jako HS, to trzeba ustawić przełącznik BOOT2 w pozycji 1 i wcisnąć przycisk *Wakeups* podczas uruchamiania urządzenia (np. przytrzymać ten przycisk podczas wciskania przycisku *Reset*).

Jeśli urządzenie działa poprawnie, to w systemie Linux po podłączeniu go do gniazda USB i wydaniu z konsoli polecenia `dmesg` powinniśmy zobaczyć podobny do poniższego wydruk.

```
[ 1660.401388] usb 2-1.3: new high speed USB device using ehci_hcd
                  and address 7
[ 1660.486946] usb 2-1.3: New USB device found, idVendor=0483, idPro-
                  duct=5756
[ 1660.486951] usb 2-1.3: New USB device strings: Mfr=1, Product=2,
                  SerialNumber=3
[ 1660.486955] usb 2-1.3: Product: USB full and high speed mass sto-
                  rage example
[ 1660.486958] usb 2-1.3: Manufacturer: Marcin Peczarski
[ 1660.486961] usb 2-1.3: SerialNumber: 0000000001
[ 1660.487762] scsi7 : usb-storage 2-1.3:1.0
[ 1661.488118] scsi 7:0:0:0: Direct-Access          MP             Removable
                  memory 1.00 PQ: 0 ANSI: 0
[ 1661.488390] sd 7:0:0:0: Attached scsi generic sg2 type 0
[ 1661.491338] sd 7:0:0:0: [sdb] 4096 512-byte logical blocks: (2.09
                  MB/2.00 MiB)
```

```
[ 1661.491899] sd 7:0:0:0: [sdb] Write Protect is off
[ 1661.491906] sd 7:0:0:0: [sdb] Mode Sense: 03 00 00 00
[ 1661.491910] sd 7:0:0:0: [sdb] Assuming drive cache: write through
[ 1661.494146] sd 7:0:0:0: [sdb] Assuming drive cache: write through
[ 1661.494154] sdb: unknown partition table
[ 1661.497764] sd 7:0:0:0: [sdb] Assuming drive cache: write through
[ 1661.497768] sd 7:0:0:0: [sdb] Attached SCSI removable disk
```

Na wydruku tym widzimy, że system wykrył urządzenie HS o identyfikatorze producenta 0483 i identyfikatorze produktu 5756. Niżej widzimy zawartość deskryptorów tekstowych opisujących produkt, producenta i numer seryjny. W ósmym wierszu widzimy informacje dostarczone przez polecenie INQUIRY. Kolejne linie informują, że jest to dysk SCSI, który ma 4096 bloków logicznych, a każdy blok logiczny ma 512 bajtów. Zatem całkowita pojemność dysku to 2 MiB. Istotna jest trzecia linia od końca, w której widzimy, że dysk ten ma nazwę `sdb` i system operacyjny nie wykrył na nim tablicy partycji. Jest tak dlatego, że po pierwszym podłączeniu pamięć urządzenia jest pusta. Zeby móc korzystać z pamięci urządzenia, musimy utworzyć w niej partycję i ją sformatować. Tablicę partycji i partycję możemy utworzyć poleciem `fdisk`, które wymaga jednak praw administratora. Oczywiście nazwa dysku w konkretnej instalacji systemu Linux może być inna – trzeba ją podać jako argument programu `fdisk`.

```
sudo /sbin/fdisk /dev/sdb
```

Po wydaniu powyższej komendy pojawia się zachęta do wprowadzania jednoliterowych poleceń. Każde polecenie kończymy, wciskając klawisz *Enter*. Polecenie *C* wyłącza tryb kompatybilności z DOS-em, który jest przestarzały. Polecenie *U* ustawia wyświetlanie rozmiarów w sektorach. Polecenie *M* wyświetla tekst pomocy z krótkim opisem wszystkich dostępnych poleceń. Nową, pustą tablicę partycji tworzy się poleciem *O*. Nową partycję dodaje się poleciem *N*. Po jego wydaniu pojawia się pytanie o rodzaj partycji. Wcisamy *P* i *Enter*, aby utworzyć partycję podstawową (główną). Pojawia się pytanie o numer partycji. Wpisujemy 1. Następnie pojawiają się pytania o numer pierwszego i ostatniego sektora. Najlepiej jest wciśnąć dwukrotnie *Enter*, wybierając proponowane wartości domyślne, czyli odpowiednio sektor 1 i sektor 4095. W ten sposób tworzona partycja zajmie całą dostępną przestrzeń na dysku. Wreszcie wydajemy polecenie *W*, aby zapisać tablicę partycji na dysku i zakończyć program `fdisk`.

Teraz po odłączeniu i ponownym podłączeniu urządzenia do gniazda USB (bez odłączania zasilania płytki) i wydaniu polecenia `dmesg` powinniśmy zobaczyć poniższy wydruk.

```
[ 2311.148234] usb 2-1.3: new high speed USB device using ehci_hcd
                  and address 8
[ 2311.234570] usb 2-1.3: New USB device found, idVendor=0483, idPro-
                  duct=5756
```

```
[ 2311.234575] usb 2-1.3: New USB device strings: Mfr=1, Product=2,
                  SerialNumber=3
[ 2311.234578] usb 2-1.3: Product: USB full and high speed mass storage example
[ 2311.234581] usb 2-1.3: Manufacturer: Marcin Peczarski
[ 2311.234583] usb 2-1.3: SerialNumber: 0000000001
[ 2311.235169] scsi8 : usb-storage 2-1.3:1.0
[ 2312.236510] scsi 8:0:0:0: Direct-Access           MP             Removable
                  memory 1.00 PQ: 0 ANSI: 0
[ 2312.236718] sd 8:0:0:0: Attached scsi generic sg2 type 0
[ 2312.240262] sd 8:0:0:0: [sdb] 4096 512-byte logical blocks: (2.09
                  MB/2.00 MiB)
[ 2312.241060] sd 8:0:0:0: [sdb] Write Protect is off
[ 2312.241067] sd 8:0:0:0: [sdb] Mode Sense: 03 00 00 00
[ 2312.241072] sd 8:0:0:0: [sdb] Assuming drive cache: write through
[ 2312.244850] sd 8:0:0:0: [sdb] Assuming drive cache: write through
[ 2312.244857] sdb: sdb1
[ 2312.248655] sd 8:0:0:0: [sdb] Assuming drive cache: write through
[ 2312.248659] sd 8:0:0:0: [sdb] Attached SCSI removable disk
```

W stosunku do poprzedniego uruchomienia różnica jest teraz taka, że urządzenie dostało adres 8, a poprzednio miało adres 7, ale najistotniejsza dla nas różnica jest widoczna w trzeciej linii od końca. System operacyjny rozpoznał na naszym dysku partycję o nazwie sdb1. Trzeba ją sformatować i utworzyć na niej system plików. System FAT tworzy się poleceniem mkdosfs (wymaga uprawnień administratora), na przykład tak

```
sudo /sbin/mkdosfs -nRAMDISK -s2 -S512 -f1 -F12 -r16 /dev/sdb1
lub
sudo /sbin/mkdosfs -nRAMDISK -s1 -S512 -f1 -F12 -r48 /dev/sdb1
lub
sudo /sbin/mkdosfs -nRAMDISK -s1 -S512 -f1 -F16 -r16 /dev/sdb1
```

Użyte parametry polecenia mkdosfs opisane są w **tabeli 6.7**. Dyskowi możemy nadać dowolną nazwę, która potem będzie go identyfikowała. W powyższych przykładach nadajemy mu nazwę RAMDISK. Podany rozmiar sektora musi być zgodny z fizycznym rozmiarem sektora na dysku (parametr -S512). Nasz dysk ma 4096 sektorów. Pierwszy (o numerze zero) zajmuje główny sektor startowy (ang. *master boot record*) z tablicą partycji. Partycja ma więc 4095 sektorów, z których pierwszy zawiera sektor początkowy. W kolejnych sektorach umieszczone są tablice FAT

Tab. 6.7. Parametry polecenia mkdosfs

Parametr	Opis
-n	Etykieta dysku, maksymalnie 11 znaków
-s	Liczba sektorów w jednostce alokacji plików (klastrze), potęga dwójką: 1, 2, 4, 8, ..., 128
-S	Rozmiar sektora w bajtach: 512, 1024, 2048, 4096, 8192, 16 384 lub 32 768
-f	Liczba kopii tablicy FAT
-F	Rozmiar wpisu w tablicy FAT: 12, 16 lub 32 bity
-r	Liczba pozycji w katalogu głównym
/dev/sdb1	Nazwa urządzenia reprezentującego partycję

i główny katalog. Z pozostałych sektorów tworzone są klastry, w których przechowywane są pliki i podkatalogi. FAT12 obsługuje maksymalnie 4077 do 4084 klastrów (zależnie od implementacji). Gdy zatem chcemy zastosować FAT12 (parametr -F12) i pozostało nam więcej sektorów do dyspozycji, to klaster musi mieć co najmniej dwa sektory (parametr -s2). Jeśli natomiast stosujemy FAT16 (parametr -F16), który obsługuje ponad 65 500 klastrów, to klaster może mieć jeden sektor (parametr -s1). Jest to pewien kompromis. W pierwszym przypadku tablica FAT zajmuje mniej miejsca, ale jednostka alokacji plików ma 1024 bajty. W drugim przypadku mamy mniejszą jednostkę alokacji plików o rozmiarze 512 bajtów, ale tablica FAT zajmuje więcej miejsca. Ponieważ stosujemy bardzo niezawodny nośnik, wystarczy tylko jedna kopia tablicy FAT (parametr -f1). Liczbę sektorów zajmowanych przez katalog główny możemy oszacować, wiedząc, że jeden wpis (pozycja) w tym katalogu zajmuje 32 bajty. Ostatnim parametrem jest nazwa urządzenia reprezentującego sformatowaną partycję.

Po utworzeniu systemu plików możemy utworzyć podkatalog w systemie plików i zamontować w nim dysk polecienniem `mount`.

```
sudo mkdir /media/usbdisk
sudo mount /dev/sdb1 /media/usbdisk
```

System plików jest buforowany w pamięci operacyjnej, więc aby mieć pewność, że wszystkie operacje zapisu na dysku zakończyły się, należy wydać z konsoli polecenie `sync`. Natomiast przed odłączeniem urządzenia od gniazda USB lub wyłączeniem jego zasilania trzeba go odmontować polecienniem `umount`.

```
sudo umount /dev/sdb1
```

Jak widać, większość powyższych operacji wymaga posiadania praw administratora. Ponieważ urządzenie jest rozpoznawane jako zewnętrzny dysk, podobnie jak pamięć Flash, w większości przypadków po odłączeniu i ponownym jego podłączeniu do gniazda USB zostanie ono zamontowane automatycznie albo do jego zamontowania można użyć jakiegoś menedżera plików ze środowiska graficznego X Windows (np. w KDE programu Dolphin). Aby zamontować dysk, wystarczy wtedy kliknąć reprezentującą go ikonę. W celu odmontowania należy wybrać z menu polecenie `usuń bezpiecznie` lub podobne. Dopóki nie wyłączymy zasilania płytka, urządzenie będzie utrzymywało zapisane w nim dane.

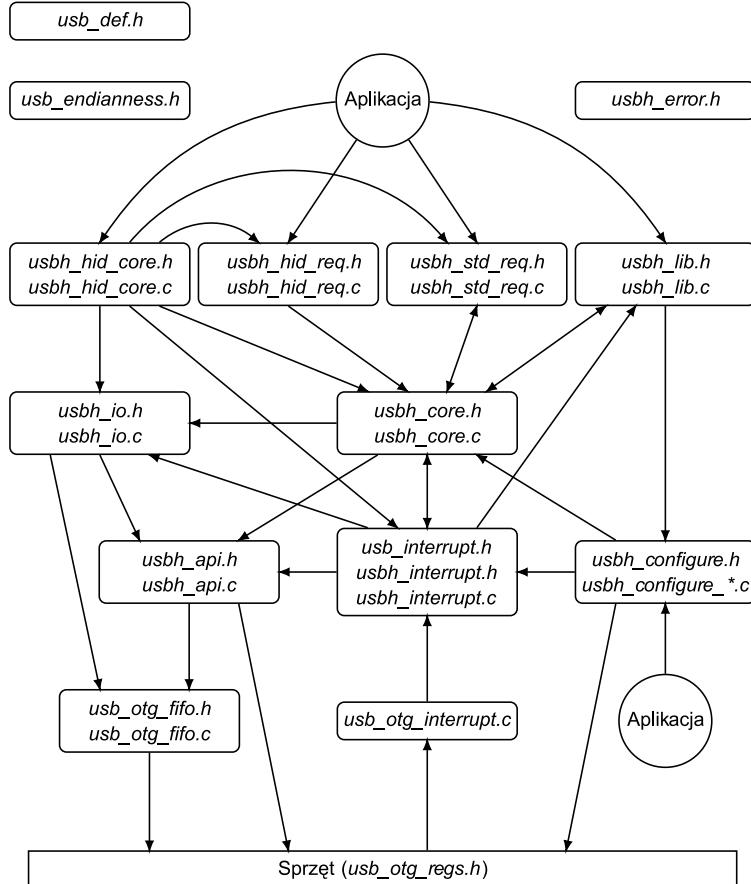
System Windows po podłączeniu do gniazda USB nowo zaprogramowanego urządzenia rozpoznaje go automatycznie i uruchamia właściwy dla niego sterownik. Podpowiada też, że przed użyciem trzeba dysk sformatować. Jest to bardzo proste – wystarczy podążyć za podpowiedziami systemu. Aby nie utracić zapisanych na dysku danych, przed odłączeniem go od gniazda USB trzeba z menu wybrać polecenie *wysuń* lub podobne. Dysk sformatowany z systemem plików FAT pod Windows może być bez problemu odczytany w systemie Linux i odwrotnie.

Po sporej dawce przykładów urządzeń USB przyszła pora na dwa projekty kontrolerów. W niniejszym rozdziale opisuję pierwszy z nich, obsługujący typowe urządzenia HID, czyli mysz i klawiaturę. Program ten wypisuje na wyświetlaczu ciekłokrystalicznym podstawowe informacje o podłączonym urządzeniu, a gdy rozpozna mysz lub klawiaturę, to umożliwia przetestowanie ich działania. Zacznę jednak od zaprezentowania obiecanej drugiej części biblioteki USB, dotyczącej właśnie kontrolera.

7.1. Biblioteka kontrolera USB dla STM32

Implementacja nawet najprostszego kontrolera jest dużo bardziej skomplikowana od implementacji odpowiedniego urządzenia. Mając to na uwadze, starałem się tę część biblioteki jak najbardziej uprościć. Nie jest to pełna implementacja kontrolera USB. Zawiera tylko niezbędne minimum funkcji potrzebnych do uruchomienia projektów. Moim zasadniczym celem jest zaprezentowanie idei działania kontrolera i pokazanie zasad implementacji protokołu USB po stronie kontrolera, unikając brnięcia w niepotrzebne szczegóły. Dlatego też prezentowane przykłady kontrolerów nie są gotowymi projektami urządzeń i nie zawierają żadnych efektownych wodotrysków (np. wyświetlanie grafiki czy nagrywanie dźwięku). Nie wydaje się to jednak poważną wadą, gdyż zasadniczym tematem tej książki jest USB. Natomiast pewną i być może istotną ceną płaconą za dokonane uproszczenie jest to, że biblioteka obsługuje jednocześnie tylko jeden interfejs USB i tylko jedno urządzenie. Symultaniczna obsługa wielu interfejsów i urządzeń wymagałaby znacznego rozbudowania kodu, co zmniejszałoby jego walory dydaktyczne. Ufam jednak, że prezentowana tu biblioteka może być podstawą do rozwijania własnych ciekawych projektów oraz że zrozumiawszy podstawy działania kontrolera, będzie można podjąć próbę rozszerzenia implementacji na więcej interfejsów i urządzeń albo przy najmniej znacznie łatwiej będzie korzystać z innych gotowych bibliotek.

Na **rysunku 7.1** przedstawione są zależności między poszczególnymi modułami biblioteki wykorzystanymi w projekcie kontrolera HID. Strzałka oznacza włączenie pliku nagłówkowego i odwołania do znajdujących się w tym pliku nagłówkowym definicji oraz do zadeklarowanych w nim funkcji lub zmiennych. Żeby nie zaciemniać schematu zależności, do trzech plików nagłówkowych umieszczonych na górze rysunku nie prowadzą żadne strzałki, gdyż pliki te są dość powszechnie włączane przez inne pliki biblioteki bezpośrednio lub pośrednio. Gwiazdka w nazwie pliku oznacza, że występuje on w wielu wariantach przeznaczonych na poszczególne modele mikrokontrolerów STM32. Kompilując program dla wybranego modelu, należy dodać właściwy plik. Biblioteka ma strukturę warstwową. Moduły *usb_otg_fifo*, *usb_otg_interrupt*, *usbh_api*, *usbh_interrupt* i *usbh_configure* (każdy moduł składa się z pliku nagłówkowego z rozszerzeniem *h* i pliku z implementacją z rozszerzeniem *c*) wraz z plikami nagłówkowymi *usb_interrupt.h* i *usb_otg_regs.h* tworzą warstwę abstrakcji sprzętu (ang. *hardware abstraction layer*). Warstwa ta uniezależnia wyższe warstwy od niskopoziomowych operacji odwołujących się bezpośrednio do sprzętu. Niezależny od sprzętu rdzeń protokołów USB tworzą moduły *usbh_io* i *usbh_core*. Moduły *usbh_std_req* i *usb_lib* zawierają funkcje wspomagające implementację warstwy aplikacji i wspólne dla wszystkich klas urządzeń. Moduł *usbh_hid_core* dodaje do rdzenia obsługę urządzeń HID, a mo-



Rys. 7.1. Zależności między modułami biblioteki dla projektu kontrolera HID

duł *usb_hid_req* zawiera najpotrzebniejsze funkcje wspomagające implementację warstwy aplikacji dla urządzeń klasy HID – opisuję je w podrozdziale 7.2. Tam też omawiam przykładową aplikację, która znajduje się w pliku *ex_hid_host.c*. Rysunek nie uwzględnia wszystkich modułów potrzebnych do skompilowania projektu. Pominąłem moduły pomocnicze zajmujące się inicjowaniem układu (*board_init_*.c*, *boot_*.c*, *startup_stm32.c*), odmierzaniem czasu (*delay.c* i *timer.c*), obsługą wyświetlacza ciekłokrystalicznego (*font5x8.c* lub *fonts.c*, *lcd_*.c*, *lcd_util.c*), sterowaniem diodami świecącymi (*led.c*) oraz wsparciem dla standardowej biblioteki języka C (*syscall_dummy.c*) – są one opisane w rozdziale 2.4.

7.1.1. Kody błędów

usbh_error.h

W pliku *usbh_error.h* znajdują się definicje kodów zwracanych przez niektóre funkcje biblioteczne dotyczące kontrolera. Błędy te są zebrane w **tabeli 7.1**. Ich nazwy i wartości są wzorowane na bibliotece *libusb*.

Tab. 7.1. Wartości zwracane przez funkcje kontrolera

Stała	Wartość	Opis
USBHLIB_SUCCESS	0	Poprawne zakończenie, brak błędu
USBHLIB_ERROR_IO	-1	Błąd operacji wejścia-wyjścia
USBHLIB_ERROR_INVALID_PARAM	-2	Błędna wartość parametru
USBHLIB_ERROR_NO_DEVICE	-4	Brak urządzenia, urządzenie odłączone
USBHLIB_ERROR_NOT_FOUND	-5	Podano błędny numer konfiguracji, interfejsu, punktu końcowego itp.
USBHLIB_ERROR_BUSY	-6	Zasób zajęty
USBHLIB_ERROR_TIMEOUT	-7	Przekroczony limit czasu
USBHLIB_ERROR_STALL	-9	Punkt końcowy wstrzymany lub nieobsługiwane żądanie
USBHLIB_ERROR_NO_MEM	-11	Brak pamięci
USBHLIB_ERROR_NOT_SUPPORTED	-12	Operacja niewspierana lub niezaimplementowana
USBHLIB_IN_PROGRESS	-50	Operacja w trakcie wykonywania, automat stanowy zajęty

7.1.2. Abstrakcja sprzętu

Żeby zapewnić przejrzystość, przenośność i łatwość pielęgnacji tekstu źródłowego biblioteki, dobrze jest wydzielić warstwę abstrakcji sprzętu, która separuje i uniezależnia implementację protokołu od niskopoziomowych operacji odwołujących się bezpośrednio do sprzętu, czyli do rejestrów układu peryferyjnego. W tym podrozdziale opisuję moduły zapewniające wymaganą separację, z wyjątkiem bezpośredniej obsługi przerwań, co zostanie omówione w jednym z dalszych podrozdziałów.

```
usbh_configure.h
usbh_configure_107.c
usbh_configure_207.c
```

Moduł *usbh_configure* odpowiada za uruchamianie układu peryferyjnego USB w trybie kontrolera. Plik *usbh_configure.h* zawiera deklaracje dwóch przedstawionych poniżej funkcji. Ich implementacja dla mikrokontrolerów STM32F105 i STM32F107 znajduje się w pliku *usbh_configure_107.c*, a dla modeli STM32F2xx i STM32F4xx w pliku *usbh_configure_207.c*.

```
int USBHconfigure(usb_phy_t phy, unsigned prio);
```

Funkcja *USBHconfigure* konfiguruje układ peryferyjny USB w trybie kontrolera. Zwraca ona wartość zero, gdy konfigurowanie zakończyło się poprawnie, a wartość ujemną, gdy wystąpił błąd.

Parametr *phy* determinuje układ peryferyjny i nadajnik-odbiornik, które mają być użyte. Należy zwrócić uwagę, że poszczególne podrodziny w obrębie STM32 są różnie wyposażone i nie wszystkie wartości są dopuszczalne dla poszczególnych modeli – patrz tabela 2.2. Oprócz wymienionych poniżej wariantów do układu OTG-HS istnieje jeszcze możliwość podłączenia zewnętrznego nadajnika-odbiornika FS za pomocą interfejsu I²C (wartość *USB_PHY_I2C* parametru *phy*), ale ta opcja

nie jest dostępna w aktualnej implementacji, ponieważ wydaje się mało atrakcyjna, gdy mamy do dyspozycji wbudowany nadajnik-odbiornik FS. Aktualna implementacja dopuszcza następujące wartości parametru phy:

- `USB_PHY_A` – układ OTG-FS z wewnętrznym nadajnikiem-odbiornikiem FS podłączonym do wyprowadzeń portu PA mikrokontrolera,
- `USB_PHY_B` – układ OTG-HS z wewnętrznym nadajnikiem-odbiornikiem FS podłączonym do wyprowadzeń portu PB mikrokontrolera,
- `USB_PHY_ULPI` – układ OTG-HS z zewnętrznym nadajnikiem-odbiornikiem HS podłączonym za pomocą interfejsu ULPI.

Parametr `prio` określa priorytet wywłaszczenia przerwań, z których korzysta implementacja kontrolera. Wszystkie przerwania mają ten sam priorytet wywłaszczenia, co zapewnia wzajemne wykluczanie wykonywania procedur ich obsługi i pozwala na bezpośrednie odwoływanie się w tych procedurach do globalnych zasobów (pamięć, rejestry konfiguracyjne). Są cztery źródła przerwań. Każdemu z nich przypisany jest inny podpriorytet, decydujący o kolejności ich obsługi. Najwyższy podpriorytet, czyli 0, ma przerwanie sygnalizujące pobór z linii VBUS zbyt dużego prądu. Główne przerwanie USB ma podpriorytet 2. Kontroler korzysta intensywnie z modułu *timer*. Przerwanie licznika milisekundowego ma przypisany podpriorytet 1, czyli wyższy niż główne przerwanie USB, gdyż licznik milisekundowy używany jest do zerowania szyny USB i restartowania urządzenia w przypadku problemu z jego uruchomieniem. Licznik mikrosekundowy jest wykorzystywany do pobudzania automatu stanowego kontrolera. Przerwanie licznika mikrosekundowego ma podpriorytet 3, czyli niższy niż główne przerwanie USB, aby komunikaty odbierane od urządzenia były obsługiwane przed komunikatami wysyłanymi.

Układy peryferyjne USB w mikrokontrolerach STM32 nie zasilają linii VBUS. Potrzebny jest zewnętrzny układ zasilający (patrz np. schematy płyt ZL29ARM i STM32F3220G-EVAL). Układ ten połączony jest z mikrokontrolerem za pomocą dwóch wyprowadzeń (patrz rysunek 2.3): wejście EN (ang. *enable*) służy do włączania napięcia, a wyjście OC (ang. *overcurrent*) – do sygnalizacji poboru zbyt dużego prądu. Moduł *usbh_configure* wymaga, aby w pliku *board_usb_def.h* zdefiniować następujące stałe:

- `HOST_VBUS_GPIO_N` – literowe oznaczenie portu, do którego podłączona jest linia EN;
- `HOST_VBUS_PIN_N` – numer wyprowadzenia, do którego podłączona jest linia EN;
- `HOST_VBUS_ON` – poziom logiczny, który trzeba podać na EN, aby włączyć zasilanie linii VBUS;
- `HOST_OVRCURR_GPIO_N` – literowe oznaczenie portu, do którego podłączona jest linia OC;
- `HOST_OVRCURR_PIN_N` – numer wyprowadzenia, do którego podłączona jest linia OC;
- `HOST_OVRCURR_IRQ_N` – nazwa źródła przerwania pochodzącego z linii OC;
- `HOST_OVRCURR_EDGE` – zbocze wyzwalające przerwanie OC.

Przykładowo, jeśli stanem aktywnym wejścia EN jest stan niski i wejście to podłączone jest do wyjścia PH5, a przekroczenie dopuszczalnego prądu sygnalizowane jest zbozem opadającym na wyjściu OC i wyjście to podłączone jest do wejścia PF11, to w pliku *board_usb_def.h* powinny się znaleźć następujące definicje:

```
#define HOST_VBUS_GPIO_N          H
#define HOST_VBUS_PIN_N           5
#define HOST_VBUS_ON              0
#define HOST_OVRCURR_GPIO_N       F
#define HOST_OVRCURR_PIN_N        11
#define HOST_OVRCURR_IRQ_N        EXTI15_10
#define HOST_OVRCURR_EDGE         EXTI_Trigger_Falling

void USBHvbus(int value);
```

Funkcja *USBHvbus* steruje zasilaniem linii VBUS, a tym samym ewentualnym zasilaniem urządzenia podłączonego do portu kontrolera. Jeśli parametr *value* ma wartość niezerową, zostaje włączone zasilanie tej linii, a gdy parametr ten ma wartość zero, napięcie na VBUS zostaje wyłączone. Funkcja ta nie zwraca żadnej wartości.

usb_otg_fifo.h
usb_otg_fifo.c

Od strony kontrolera transmisje danych odbywają się za pomocą kanałów. Kanał realizuje jednokierunkową transmisję danych między kontrolerem a punktem końcowym urządzenia. Kanał konfiguruje się, podając adres urządzenia, adres punktu końcowego (bit kierunku w adresie determinuje kierunek transmisji) i rodzaj przesyłanych danych (sterujące, pilne, masowe, izochroniczne). Obsługa kanałów jest zaszyta w spręciu. Przesyłanie danych do kanału odbywa się za pomocą kolejek nadawczych. Są dwie takie kolejki: dla danych nieperiodycznych (sterujących i masowych) i dla danych periodycznych (pilnych i izochronicznych). Odczytywanie danych z kanału odbywa się za pomocą kolejki odbiorczej, wspólnej dla wszystkich rodzajów danych. Interfejs programistyczny do obsługi kolejek nadawczo-odbiorczych jest zaimplementowany w module *usb_otg_fifo* i został przedstawiony w punkcie 4.1.6 przy okazji omawiania części biblioteki dotyczącej urządzenia. Układ OTG-FS ma 8 kanałów, a OTG-HS ma ich 12. Widać, że dostępnych kanałów jest znacznie mniej, niż wynosi iloczyn maksymalnej określonej przez standard liczby urządzeń i maksymalnej liczby punktów końcowych w każdym urządzeniu. Wynika stąd, że obsługa przez kontroler wielu urządzeń wymaga, aby kanały były przydzielane dynamicznie na okres transmisji i zwalniane po jej zakończeniu. Kanały numerowane są od zera. W bibliotece stosuję konwencję, że podanie jako numeru kanału wartości ujemnej oznacza kanał nieprzydzielony.

usbh_api.h
usbh_api.c

Moduł *usbh_api* dostarcza bardzo niskopoziomowego interfejsu dla wyższych warstw oprogramowania kontrolera. Są to funkcje obudowujące odwołania do re-

jestrów konfiguracyjnych układu peryferyjnego USB. Plik *usbh_api.h* zawiera ich deklaracje, a w pliku *usbh_api.c* znajduje się ich implementacja. Zaczynam od przedstawienia funkcji operujących na kanałach.

```
unsigned USBHgetChannelCount(void);
```

Bezparametrowa funkcja `USBHgetChannelCount` zwraca liczbę kanałów dostępnych w układzie peryferyjnym USB.

```
void USBHstopAllChannels(void);
```

Bezparametrowa funkcja `USBHstopAllChannels` opróżnia wszystkie kolejki nadawcze i odbiorcze, a następnie zatrzymuje transmisje we wszystkich kanałach oraz dezaktywuje wszystkie związane z nimi przerwania i zeruje wszystkie znaczniki zgłoszonych przerwań. Funkcja ta nie zwraca żadnej wartości.

```
void USBHinitChannel(int ch_num, uint8_t dev_addr, uint8_t ep_addr,
                      usb_speed_t dev_speed, usb_transfer_t ep_type,
                      uint16_t max_packet);
```

Funkcja `USBHinitChannel` konfiguruje kanał do komunikacji z wybranym punktem końcowym. Parametr `ch_num` jest numerem kanału sprzętowego, którego chcemy użyć. Parametr `dev_addr` zawiera adres urządzenia, z którym chcemy się komunikować. Parametr `ep_addr` zawiera adres punktu końcowego w tym urządzeniu. Bit kierunku w adresie punktu końcowego determinuje kierunek transmisji. Parametr `dev_speed` określa szybkość transmisji. Może on przyjmować wartość `LOW_SPEED`, `FULL_SPEED` lub `HIGH_SPEED`. Parametr `ep_type` określa rodzaj przesyłanych danych. Może on przyjmować wartość `CONTROL_TRANSFER`, `ISOCHRONOUS_TRANSFER`, `BULK_TRANSFER` lub `INTERRUPT_TRANSFER`. Parametr `max_packet` zawiera maksymalny rozmiar pola danych pakietu. Funkcja ta nie zwraca żadnej wartości.

```
int USBHprepareChannel(int ch_num, uint32_t xfer_len,
                       usb_pid_t data_pid);
```

Funkcja `USBHprepareChannel` przygotowuje kanał do transmisji danych. Parametr `ch_num` zawiera numer kanału, który ma być użyty. Parametr `xfer_len` zawiera rozmiar danych, które mają być przesłane. Na jego podstawie i ustawionego w funkcji `USBHinitChannel` maksymalnego rozmiaru pola danych wylicza się liczbę pakietów, na które muszą być podzielone dane. Parametr `data_pid` określa, jaki identyfikator pakietu PID ma być użyty w pierwszej transakcji. Może on przyjmować wartość `PID_SETUP`, `PID_DATA0`, `PID_DATA1`, `PID_DATA2` lub `PID_MDATA`. Jeśli transmisja wymaga przesyłania więcej niż jednego pakietu, to PID w kolejnych transakcjach jest automatycznie ustawiany przez sprzęt. Omawiana funkcja zwraca zero, gdy wszystko przebiegło poprawnie, a wartość ujemną, gdy wystąpił błąd. W aktualnej implementacji błąd może być spowodowany zbyt dużym rozmiarem danych przekazanych do wysłania.

```
void USBHhaltChannel(int ch_num);
```

Funkcja `USBHhaltChannel` zatrzymuje transmisję w kanale, którego numer została podany w parametrze `ch_num`. Funkcja ta nie zwraca żadnej wartości. W układach OTG-FS i OTG-HS kanał nie jest wstrzamywany natychmiast. Żądanie wstrzamymania kanału jeststawiane do kolejki razem z innymi żądaniami. Aktualnie realizowana transakcja nie jest przerywana. Gdy kanał zostanie wstrzymany, ustawiany jest bit zdarzenia, które może wyzwolić przerwanie.

Oprócz funkcji operujących na kanałach moduł `usbh_api` dostarcza także dwóch dodatkowych funkcji ułatwiających obsługę ramek.

```
uint16_t USBHgetCurrentFrame(void);
```

Bezparametrowa funkcja `USBHgetCurrentFrame` zwraca aktualny numer ramki (jeśli podłączone jest urządzenie LS lub FS) lub mikroramki (jeśli podłączone jest urządzenie HS). Układy OTG-FS i OTG-HS numerują ramki i mikroramki od 0 do 16 383 (0x3fff). Źeby zatem otrzymać 11-bitowy numer ramki, jak to definiuje standard USB, należy w przypadku urządzenia LS lub FS wziąć 11 młodszych bitów wartości zwróconej przez tę funkcję, a w przypadku urządzenia HS przesunąć tę wartość w prawo o 3 bity.

```
uint16_t USBHgetFrameClocksRemaining(void);
```

Bezparametrowa funkcja `USBHgetFrameTimeRemaining` zwraca liczbę taktów, które pozostały do końca bieżącej ramki (w trybie LS lub FS) lub mikroramki (w trybie HS). Jeśli podłączone jest urządzenie LS lub FS, to zegar ten ma częstotliwość 48 MHz i jedna ramka trwa odpowiednio 6000 lub 48 000 taktów. Jeśli natomiast podłączone jest urządzenie HS, to zegar ma częstotliwość 60 MHz i jedna mikroramka trwa 7500 taktów.

7.1.3. Niskopoziomowe wejście-wyjście

<code>usbh_io.h</code>
<code>usbh_io.c</code>

Moduł `usbh_io` udostępnia niezależne od sprzętu funkcje do obsługi kanałów i inicjowania transmisji danych. Deklaracje tych funkcji umieszczone są w pliku `usbh_io.h`. Ich implementacja znajduje się w pliku `usbh_io.c`. Tekst źródłowy tego modułu jest w dużym stopniu niezależny od sprzętu, gdyż odwołania do rejestrów sprzętowych odbywają się za pomocą funkcji z modułów `usbh_api` i `usb_otg_fifo`. Wewnętrznie moduł `usbh_io` reprezentuje kanały w tablicy struktur typu `usbh_channel_t`. Indeks w tej tablicy jest jednocześnie sprzętowym numerem kanału. Składowe wspomnianej struktury powielają pewne informacje, które są przechowywane w rejestrach sprzętowych układu peryferyjnego lub w warstwie aplikacji. Ten dodatkowy narzut jest ceną płaconą za niezależność od sprzętu i separację warstw oprogramowania.

```

typedef struct {
    uint8_t           used;
    uint8_t           dev_addr;
    uint8_t           ep_addr;
    usb_speed_t      dev_speed;
    usb_transfer_t   ep_type;
    uint16_t          max_packet;
    usbh_transaction_result_t tr_result;
    usb_pid_t         pid;
    uint8_t           *buffer;
    uint32_t          length;
    uint32_t          transferred;
} usbh_channel_t;

```

Składowa `used` struktury typu `usbh_channel_t` przyjmuje wartość jeden, gdy kanał został przydzielony, a wartość zero w przeciwnym przypadku. Składowe `dev_addr` i `ep_addr` przechowują odpowiednio adres urządzenia i punktu końcowego, dla których został skonfigurowany dany kanał. Składowa `dev_speed` struktury typu `usbh_channel_t` przechowuje szybkość, z jaką skonfigurowany jest ten kanał. Może ona przyjmować wartość `LOW_SPEED`, `FULL_SPEED` lub `HIGH_SPEED`. Składowa `ep_type` zawiera rodzaj przesyłanych danych. Przyjmuje ona wartość `CONTROL_TRANSFER`, `ISOCHRONOUS_TRANSFER`, `BULK_TRANSFER` lub `INTERRUPT_TRANSFER`. Składowa `max_packet` przechowuje maksymalny rozmiar pola danych pakietu. Składowa `buffer` jest wskaźnikiem na bufor z danymi do wysłania lub bufor, do którego mają zostać skopiowane odebrane dane. Składowa `length` zawiera rozmiar danych, jakie pozostały jeszcze do przesłania. Składowa `transferred` zawiera rozmiar dotychczas przesłanych danych.

Składowa `tr_result` struktury typu `usbh_channel_t` zawiera rezultat, z jakim zakończyła się ostatnia transakcja. Przyjmuje ona jedną z pięciu wartości:

- `TR_UNDEF` – stan nieokreślony, jednocześnie stan początkowy i spoczynkowy
– nie jest realizowana żadna transakcja lub transakcja została zainicjowana, ale jeszcze się nie zakończyła;
- `TR_DONE` – transakcja zakończona poprawnie;
- `TR_NAK` – transakcja odrzucona pakietem NAK;
- `TR_STALL` – transakcji odrzucona pakietem STALL;
- `TR_ERROR` – błąd transmisji: niepoprawne nadziewanie bitami (ang. *bit stuff error*), błędny znacznik końca pakietu EOP, niepoprawna wartość CRC, przekroczyły maksymalny czas oczekiwania na zakończenie transakcji (ang. *timeout*), więcej danych niż maksymalny rozmiar pakietu (ang. *babble error*), transakcja przekroczyła granicę ramki lub mikroramki (ang. *frame overrun*).

Składowa `data_pid` struktury typu `usbh_channel_t` określa rodzaj transakcji, która ma być zainicjowana. W aktualnej implementacji przyjmuje ona jedną z trzech wartości (wybór między transakcją IN i OUT następuje na podstawie bitu kierunku w adresie punktu końcowego – składowa `ep_addr`):

- `PID_SETUP` – transakcja SETUP z pakietem danych DATA0;

- PID_DATA0 – transakcja IN lub OUT z pakietem danych DATA0;
- PID_DATA1 – transakcja IN lub OUT z pakietem danych DATA1.

Moduł *usbh_io* udostępnia trzy grupy funkcji. Do pierwszej grupy należą funkcje przeznaczone do obsługi kanałów.

```
int USBHchannelsConfigure(void);
```

Bezparametrowa funkcja `USBHchannelsConfigure` przydziela pamięć dla tablicy struktur typu `usbh_channel_t` i zeruje wszystkie składowe `used`. Funkcja ta zwraca zero, gdy wszystko przebiegło poprawnie, a wartość ujemną, gdy wystąpił błąd – jest zbyt mało kanałów do dyspozycji albo nie powiodło się alokowanie pamięci. Do komunikacji z urządzeniem potrzebne są przynajmniej dwa kanały dla danych sterujących (jeden kanał do wysyłania, a drugi do odbierania danych). Funkcja ta jest wywoływana tylko raz podczas uruchamiania programu kontrolera, więc alokowanej przez nią pamięci nie trzeba zwalniać.

```
int USBHallocChannel(void);
```

Bezparametrowa funkcja `USBHallocChannel` przydziela kanał. Zwraca numer przydzielonego kanału lub wartość ujemną, gdy nie ma wolnego kanału.

```
void USBHfreeChannel(int ch_num);
```

Funkcja `USBHfreeChannel` zwalnia kanał przydzielony za pomocą funkcji `USBHallocChannel`. Parametr `ch_num` zawiera numer kanału, który ma być zwolniony. Funkcja ta nie zwraca żadnej wartości.

```
void USBHopenChannel(int ch_num, uint8_t dev_addr, uint8_t ep_addr,
                      usb_speed_t dev_speed, usb_transfer_t ep_type,
                      uint16_t max_packet);
```

Funkcja `USBHopenChannel` konfiguruje parametry kanału i przygotowuje go do użycia. Parametr `ch_num` jest numerem kanału. Parametry `dev_addr` i `ep_addr` zawierają odpowiednio adres urządzenia i punktu końcowego, z którymi ma się odbywać komunikacja za pomocą tego kanału. Parametr `dev_speed` zawiera szybkość transmisji. Parametr `ep_type` określa rodzaj danych. Parametr `max_packet` zawiera maksymalny rozmiar pola danych pakietu. Wartości parametrów są zapisywane w odpowiednich składowych struktury typu `usbh_channel_t`, po czym, w celu faktycznego zainicjowania kanału, wywoływana jest funkcja `USBHinitChannel`. Omawiana funkcja nie zwraca żadnej wartości.

```
void USBHmodifyChannel(int ch_num, uint8_t dev_addr,
                        uint16_t max_packet);
```

Zwykle po przydzieleniu urządzeniu adresu docelowego trzeba zmienić ustawienia kanałów komunikujących się z domyślnym punktem końcowym dla danych ste-

rujących. Funkcja `USBHmodifyChannel` modyfikuje ustawienia kanału o numerze `ch_num`. Zmienia adres urządzenia na nowy, podany za pomocą parametru `dev_addr`, oraz maksymalny rozmiar pola danych pakietu na ten podany w parametrze `max_packet`. Funkcja ta nie zwraca żadnej wartości. W celu wykonania właściwej modyfikacji wywoływana jest funkcja `USBHinitChannel`.

Drugą grupę funkcji udostępnianych przez moduł `usbh_io` tworzą funkcje wywoływane w odpowiedzi na pewne zdarzenia zachodzące na szynie. Funkcje te są wywoływane w procedurach obsługi przerwań wyzwalanych przez te zdarzenia.

```
void USBHpacketReceived(int ch_num, uint32_t length);
```

Funkcja `USBHpacketReceived` jest wywoływana, gdy w trakcie realizacji transakcji IN zostały odebrane dane, czyli po odebraniu pakietu DATAx (w procedurze obsługi przerwania związanego z tym zdarzeniem). Jej zadaniem jest odczytanie danych z kolejki odbiorczej i skopiowanie ich do bufora wskazywanego za pomocą składowej `buffer` struktury typu `usbh_channel_t`, a następnie takie zmodyfikowanie składowych `buffer`, `length`, `transferred` i `pid`, aby były gotowe do kolejnej transakcji IN. Parametr `ch_num` jest numerem kanału, którym dane zostały odebrane. Parametr `length` zawiera rozmiar odebranych danych. Funkcja ta nie zwraca żadnej wartości.

```
void USBHpacketSent(int ch_num);
```

Funkcja `USBHpacketSent` jest wywoywana, gdy w transakcji OUT lub SETUP zostały wysłane i potwierdzone dane, czyli po odebraniu pakietu ACK (w procedurze obsługi przerwania związanego z tym zdarzeniem). Jej zadaniem jest zmodyfikowanie składowych `buffer`, `length`, `transferred` i `pid` struktury typu `usbh_channel_t`, aby były gotowe dla kolejnej transakcji OUT. Parametr `ch_num` jest numerem kanału, którym dane zostały wysłane. Funkcja ta nie zwraca żadnej wartości.

```
void USBHtransferFinished(int ch_num, usbh_transaction_result_t tr);
```

Funkcja `USBHtransferFinished` jest wywoywana, gdy zakończy się transmisja danych (w procedurze obsługi przerwania związanego z tym zdarzeniem). Transmisja mogła składać się z jednej lub wielu kolejnych transakcji. Funkcja musi być wywołana po wszystkich wywołaniach funkcji `USBHpacketReceived` lub `USBHpacketSent` dotyczących kolejnych transakcji składających się na całą transmisję. Parametr `ch_num` jest numerem kanału, który uczestniczył w transmisji. Parametr `tr` zawiera rezultat ostatniej transakcji. Przyjmuje jedną z wartości wymienionych przy omawianiu składowej `tr_result` struktury typu `usbh_channel_t`. Zadaniem tej funkcji jest uaktualnienie wartości tej składowej oraz ewentualne wznowienie transmisji, gdy nie wszystkie dane zostały wysłane. Nie zwraca ona żadnej wartości.

Do trzeciej grupy funkcji udostępnianych przez moduł `usbh_io` należą funkcje inicjujące właściwą transmisję danych i przeznaczone do wywoływanego przez warstwę aplikacji protokołów.

```
int USBHstartTransaction(int ch_num, usb_pid_t pid,
                         uint8_t *buffer, uint32_t length);
```

Funkcja `USBHstartTransaction` inicjuje transakcję IN, OUT lub SETUP. Parametr `ch_num` zawiera numer kanału, który ma być użyty. Parametr `pid` określa rodzaj transakcji zgodnie z opisem składowej `pid` struktury typu `usbh_channel_t`. Parametr `buffer` jest wskaźnikiem na bufor z danymi do wysłania lub bufor, do którego mają być skopiowane odebrane dane. Kierunek transmisji jest zdeterminowany konfiguracją kanału (bitem kierunku w adresie związanego z nim punktu końcowego). Parametr `length` zawiera rozmiar danych, jakie mają być przesłane (rozmiar bufora). Funkcja ta zwraca `USBHLIB_SUCCESS` lub właściwy kod błędu. Funkcja ta zakończy się błędem, gdy wartość parametru `length` przekracza rozmiar pola danych pojedynczego pakietu. Inną przyczyną niepowodzenia zainicjowania transakcji OUT lub SETUP może być brak dostatecznej ilości miejsca w kolejce nadawczej. Jednak w praktyce, gdy do kontrolera podłączone jest tylko jedno urządzenie, nie generuje on aż tak dużego ruchu, aby zapchać tę kolejkę.

```
usbh_transaction_result_t USBHgetTransactionResult(int ch_num);
```

Funkcja `USBHgetTransactionResult` zwraca rezultat transakcji zainicjowanej za pomocą funkcji `USBHstartTransaction`, czyli wartość przechowywaną w składowej `tr_result` struktury typu `usbh_channel_t`. Parametr `ch_num` jest numerem kanału.

```
uint32_t USBHgetTransactionSize(int ch_num);
```

Funkcja `USBHgetTransactionSize` zwraca rozmiar rzeczywiście przesłanych danych przez transakcję zainicjonowaną za pomocą funkcji `USBHstartTransaction`. Parametr `ch_num` zawiera numer kanału. Funkcja ta zwraca wartość przechowywaną w składowej `transferred` struktury typu `usbh_channel_t`.

```
int USBHstartTransfer(int ch_num, uint8_t *buffer, uint32_t length);
```

Układy OTG-FS i OTG-HS wspomagają realizację przesyłania danych o rozmiarze przekraczającym rozmiar pola danych pojedynczego pakietu – sprzęt zajmuje się dzieleniem danych na pakiety i inicjowaniem kolejnych transakcji. Wysyłanie wielu pakietów realizuje funkcja `USBHstartTransfer`, która zależnie od potrzeby inicjuje jedną lub więcej kolejnych transakcji IN albo OUT. Parametr `ch_num` zawiera numer kanału, który ma być użyty. Parametr `buffer` jest wskaźnikiem na bufor z danymi do wysłania lub bufor, do którego mają być skopiowane odebrane dane. Kierunek transmisji jest zdeterminowany konfiguracją kanału (bitem kierunku w adresie związanego z nim punktu końcowego). Parametr `length` zawiera rozmiar danych, jakie mają być przesłane (rozmiar bufora). Funkcja ta zwraca `USBHLIB_SUCCESS` lub właściwy kod błędu. Funkcja ta zakończy się błędem, gdy liczba pakietów, na które miałyby być podzielone dane, jest zbyt duża (ograniczenie to wynika z maksymalnej wartości, jaką może przyjąć sprzętowy licznik transakcji).

Inną przyczyną niepowodzenia może być brak dostatecznej ilości miejsca w kolejce nadawczej. W układach OTG-FS i OTG-HS można uaktywnić przerwanie zgłasiane, gdy cała lub przynajmniej połowa kolejki nadawczej jest wolna. Wtedy w przypadku braku miejsca aktywujemy to przerwanie, a w jego procedurze obsługi wstawiamy do kolejki kolejne dane. Świadomie zrezygnowałem z takiego rozwiązania, gdyż bardzo skomplikowałoby to tekst źródłowy biblioteki i znacznie zmniejszyłoby jej walory dydaktyczne. Aktualna implementacja wstawia do kolejki maksymalną liczbę pakietów, jakie się zmieszcza. Rozmiar rzeczywiście przesyłanych danych należy sprawdzić po zakończeniu całej transmisji za pomocą funkcji `USBHgetTransferSize`. Należy również pamiętać, że transmisja zakończy się, gdy któraś transakcja IN lub OUT zostanie odrzucona przez urządzenie (pakietem NAK lub STALL) albo nie dojdzie do skutku z powodu błędu transmisyjnego.

```
int USBHrestartTransfer(int ch_num);
```

Jeśli nie wszystkie dane przekazane do funkcji `USBHstartTransfer` zostały przesłane (nie było dostatecznej ilości miejsca w kolejce nadawczej lub transakcja została odrzucona pakietem NAK), to za pomocą funkcji `USBHrestartTransfer` można wznowić transmisję pozostałych danych. Parametr `ch_num` zawiera numer kanału, w którym ma być wznowiona transmisja. Funkcja ta zwraca `USBHLIB_SUCCESS` lub właściwy kod błędu.

```
void USBHsetPID(int ch_num, usb_pid_t pid);
```

Jeśli użyjemy funkcji `USBHstartTransfer`, to biblioteka (w układach OTG-FS i OTG-HS jest to realizowane sprzętowo) dba o to, aby na początku oraz w kolejnych transakcjach był właściwie ustawiany PID pakietu danych (DATA0 bądź DATA1). Jednak w pewnych sytuacjach może zaistnieć potrzeba wymuszenia tego PID w pierwszej transakcji. Można to zrobić za pomocą funkcji `USBHsetPID`. Parametr `ch_num` zawiera numer kanału. Parametr `pid` określa typ transakcji zgodnie z opisem składowej `pid` struktury typu `usbh_channel_t`. Funkcja ta nie zwraca żadnej wartości.

```
usbh_transaction_result_t USBHgetTransferResult(int ch_num);
```

Funkcja `USBHgetTransferResult` zwraca rezultat ostatniej transakcji zainicjowanej za pomocą funkcji `USBHstartTransfer`, czyli wartość przechowywaną w składowej `tr_result` struktury typu `usbh_channel_t`. Parametr `ch_num` jest numerem kanału. Jej implementacja jest taka sama jak implementacja funkcji `USBHgetTransactionResult`.

```
uint32_t USBHgetTransferSize(int ch_num);
```

Funkcja `USBHgetTransferSize` zwraca rozmiar danych rzeczywiście przesyłanych przez wszystkie zrealizowane transakcje zainicjowane za pomocą funkcji `USBHstartTransfer`. Parametr `ch_num` jest numerem kanału. Funkcja ta zwraca war-

tość przechowywaną w składowej transferred struktury typu `usbh_channel_t`. Jej implementacja jest taka sama jak implementacja funkcji `USBHgetTransactionSize`.

7.1.4. Rdzeń protokołu

Zadaniem rdzenia protokołu USB po stronie kontrolera jest obsługa zdarzeń związanych z podłączaniem i odłączeniem urządzenia, ustanowienie dwukierunkowego kanału komunikacyjnego z zerowym punktem końcowym urządzenia, czyli domyślnym punktem końcowym dla danych sterujących, odczytanie deskryptora urządzenia i nadanie urządzeniu docelowego adresu, czyli wszystko to, co w dokumentacji określone jest terminem *enumeration*. Rdzeń realizuje te zadania za pomocą automatów stanowych.

```
usbh_core.h  
usbh_core.c
```

Najważniejsze funkcje rdzenia zaimplementowane są w module `usbh_core`. Jak zwykle plik `usbh_core.h` zawiera definicję interfejsu modułu, czyli nagłówki udostępnianych funkcji, a w pliku `usbh_core.c` znajduje się ich implementacja.

```
int USBHcoreConfigure(void);
```

Bezparametrowa funkcja `USBHcoreConfigure` inicjuje globalne struktury danych niezbędne do działania modułu `usbh_core`. Zwrócenie zera oznacza jej poprawne zakończenie, a zwrócenie wartości ujemnej sygnalizuje błąd. Funkcja ta powinna być wywołana tylko raz na początku działania programu. Nie musimy jej jednak wywoływać sami, gdyż jest wywoływana podczas konfigurowania interfejsu USB przez opisaną powyżej funkcję `USBHconfigure`.

Pozostałe funkcje udostępniane przez moduł `usbh_core` dzielą się na dwie grupy. Do pierwszej należą funkcje wywoływane w wyniku zajścia jakiegoś zdarzenia na szynie, wywoływane zawsze w procedurze obsługi przerwania wyzwolonego tym zdarzeniem (przerwania USB). Drugą grupę tworzą funkcje przeznaczone do wywoływania przez warstwę aplikacji. W opisie każdej z tych funkcji zaznaczam wyraźnie, jak może być ona wywoływana (w kontekście przerwania USB czy poza nim). Zaczynam po kolej od omówienia pierwszej grupy funkcji.

```
void USBHdeviceDisconnected(void);
```

Bezparametrowa funkcja `USBHdeviceDisconnected` jest wywoływana po wykryciu odłączenia urządzenia (w procedurze obsługi przerwania związanego z tym zdarzeniem). W tej funkcji kontroler powraca do stanu początkowego, w którym oczekuje na podłączenie urządzenia. Funkcja ta nie zwraca żadnej wartości.

```
void USBHdeviceAttached(void);
```

Bezparametrowa funkcja `USBHdeviceAttached` jest wywoływana po wykryciu podłączenia urządzenia (w procedurze obsługi przerwania związanego z tym zdarzeniem). Kontroler odnotowuje, że zostało podłączone urządzenie i że znajduje się ono w stanie szyna zasilana (ang. *powered*). Aktualna implementacja kontrolera nie

odróżnia tego stanu od stanu urządzenia szyna podłączona (ang. *attached*), gdyż nie ma możliwości wykrycia urządzenia, które nie jest zasilane. Funkcja ta nie zwraca żadnej wartości.

```
void USBHdeviceSpeed(usb_speed_t speed);
```

Funkcja `USBHdeviceSpeed` za pomocą parametru `speed` informuje o szybkości, z jaką będzie pracowało podłączone urządzenie. Jest ona wywoływana tuż po zakończeniu wysyłania przez kontroler sygnału zerowania szyny w procedurze obsługi przerwania związanego z tym zdarzeniem. Nie zwraca żadnej wartości.

```
void USBHdeviceResetDone(void);
```

Bezparametrowa funkcja `USBHdeviceResetDone` informuje, że zakończyła się procedura zerowania szyny i minął określony w standardzie czas, w którym urządzenie powinno zareagować na to zdarzenie. Jest ona wywoływana w procedurze obsługi przerwania związanego z zerowaniem szyny. Kontroler odnotowuje, że urządzenie jest teraz w stanie adres domyślny (ang. *default*). Następnie aktywnia komunikację z zerowym punktem końcowym urządzenia (aktywuje odpowiednie kanały komunikacyjne) i inicjuje procedurę nadania urządzeniu docelowego adresu. Funkcja ta nie zwraca żadnej wartości.

```
void USBHs0f(uint16_t frnum);
```

Funkcja `USBHs0f` jest wywoływana na początku każdej ramki lub mikroramki (w procedurze obsługi przerwania związanego z tym zdarzeniem). Nie zwraca żadnej wartości. Parametr `frnum` zawiera numer rozpoczynającej się właśnie ramki lub mikroramki. Do numeru tego odnoszą się wszystkie uwagi dotyczące wartości zwracanej przez funkcję `USBHgetCurrentFrame`.

```
void USBHcoreProcess(void);
```

Bezparametrowa funkcja `USBHcoreProcess` jest głównym punktem wejścia do automatu stanowego realizującego protokół USB. Nie zwraca żadnej wartości. Funkcja ta musi być wywoływana cyklicznie. Jej tekst źródłowy i sposób wywoływanego opisuję szczegółowo w rozdziale 7.1.7.

Przejdę teraz do omówienia drugiej grupy funkcji udostępnianych przez moduł `usbh_core`, czyli funkcji przeznaczonych do wywoływania przez warstwę aplikacji.

```
int USBHcontrolRequest(int synch,
                       usb_setup_packet_t const *setup,
                       uint8_t *buffer, uint32_t *length);
```

Funkcja `USBHcontrolRequest` inicjuje realizację żądania. Parametr `setup` jest wskaźnikiem na strukturę opisującą żądanie. Jej składowe mają lokalny porządek bajtów. Jeśli żądanie wymaga przesłania danych z kontrolera do urządzenia, to parametr `buff` jest wskaźnikiem na bufor z tymi danymi. Jeśli żądanie wymaga prze-

słania danych z urządzenia do kontrolera, to parametr `buffer` jest wskaźnikiem na bufor, w którym mają być umieszczone odebrane dane. Rozmiar bufora określony jest przez składową `wLength` struktury wskazywanej parametrem `setup`. Jeśli żądanie nie wymaga przesłania danych, to parametr `buff` powinieneć mieć wartość zero (`NULL`). Parametr `length` jest wskaźnikiem na zmienną, za pomocą której zwracany jest rzeczywisty rozmiar przesłanych danych. Może to być mniejsza, niż podano w składowej `wLength`. Jeśli nie potrzebujemy tej informacji (np. żądanie nie wymaga przesłania danych), to można podać wskaźnik zerowy (`NULL`).

Funkcja `USBHcontrolRequest` zachowuje się różnie zależnie od wartości parametru `synch`. Jeżeli ma on wartość zero, to jej wywołanie jest nieblokujące (asynchroniczne). Funkcja sprawdza, w jakim stanie jest automat realizujący żądania. Jeśli jest on w stanie spoczynkowym, to inicjuje działanie automatu i zwraca wartość `USBHLIB_IN_PROGRESS`. Tę samą wartość zwraca, gdy automat jest w trakcie realizacji żądania. Jeśli automat zakończył realizację żądania, to zwraca wartość `USBHLIB_SUCCESS`, gdy wszystko przebiegło poprawnie, albo odpowiedni kod błędu (patrz tabela 7.1). W każdym przypadku funkcja natychmiast kończy działanie. Nieblokujące wywołanie tej funkcji jest dopuszczalne jedynie w kontekście przerwania USB, aby wykluczyć współbieżne wykonanie z innymi funkcjami biblioteki.

Nieblokujące wywoływanie funkcji jest bardzo efektywne – nie tracimy czasu, czekając na zakończenie komunikacji. Jest to jednak dość trudne w użyciu. Dużo łatwiej programuje się za pomocą wywołań blokujących, szczególnie wtedy, gdy przed rozpoczęciem kolejnych działań i tak musimy poczekać na zakończenie komunikacji. Wywołanie funkcji `USBHcontrolRequest` z niezerową wartością parametru `synch` jest blokujące (synchroniczne). Funkcja czeka na zakończenie komunikacji i zwraca `USBHLIB_SUCCESS`, gdy wszystko przebiegło poprawnie, albo odpowiedni kod błędu. Wywołanie blokujące może się odbywać jedynie poza kontekstem przerwań – zablokowanie procedury obsługi przerwania zakończy się zwykle fatalnie. Przykłady zarówno wywołań blokujących, jak i nieblokujących zobaczymy w dalszej części tego rozdziału.

```
int USBHgetDevice(usb_speed_t *speed, uint8_t *dev_addr,  
                   usb_device_descriptor_t *dev_desc);
```

Funkcja `USBHgetDevice` sprawdza, czy do portu USB kontrolera podłączone jest urządzenie i czy jest ono gotowe do użycia (ma przyznany adres docelowy). Jeśli sprawdzenie wypadnie pozytywnie, to funkcja zwraca wartość `USBHLIB_SUCCESS`, a za pomocą wskaźników `speed`, `dev_addr` i `dev_desc` zwracane są odpowiednio szybkość pracy urządzenia, jego adres i deskryptor urządzenia. W przeciwnym przypadku funkcja zwraca `USBHLIB_ERROR_NO_DEVICE`. Wywołanie tej funkcji jest nieblokujące i może być ona wywoływana poza kontekstem obsługi przerwania.

```
usb_visible_state_t USBHgetVisibleDeviceState(void);
```

Bezparametrowa funkcja `USBHgetVisibleDeviceState` zwraca stan portu USB kontrolera. Jej wywołanie jest nieblokujące i może być ona wywoływana poza kontekstem przerwania. Aktualna implementacja zwraca jedną z czterech wartości:

- DISCONNECTED – do portu nie jest podłączone żadne urządzenie;
- POWERED – do portu jest podłączone urządzenie i jest ono zasilane;
- DEFAULT – do portu jest podłączone urządzenie i zakończyła się procedura zerowania szyny;
- ADDRESS – do portu jest podłączone urządzenie i ma ono przyznany docelowy adres.

```
int USBHsetClassMachine(int (*machine) (void *param),  
                        void (*at_sof) (void *param, uint16_t frnum),  
                        void (*at_disconnect) (void *param),  
                        void *param);
```

Automat stanowy zaimplementowany w module `usbh_core` obsługuje tylko minimalny fragment protokołu USB, potrzebny do uruchomienia urządzenia. Za pomocą funkcji `USBHsetClassMachine` można zainstalować automat stanowy specyficzny dla urządzeń konkretnej klasy. Parametr `machine` jest adresem funkcji zwrotnej realizującej ten automat. Musi to być funkcja przyjmująca jeden argument będący wskaźnikiem typu `void*` i zwracająca wartość typu `int`. Funkcja ta będzie wywoływana cyklicznie dopóty, dopóki będzie zwracać wartość zero. Zwrócenie wartości niezerowej oznacza, że automat zakończył swoje działanie i nie należy go więcej wywoływać. Parametr `at_sof` jest adresem funkcji zwrotnej, która będzie wywoływana na początku każdej ramki lub mikroramki. Funkcja wskazywana za pomocą parametru `at_sof` przyjmuje dwa argumenty: wskaźnik typu `void*` oraz liczbę typu `uint16_t` zawierającą numer właśnie rozpoczynającej się ramki lub mikroramki. Funkcja ta nie zwraca żadnej wartości. Parametr `at_disconnect` jest adresem funkcji zwrotnej, która ma zostać wywołana po wyjęciu urządzenia z gniazda. Musi to być funkcja przyjmująca jeden argument będący wskaźnikiem typu `void*` i niezwracająca żadnej wartości. Funkcja ta jest konieczna, gdyż po odłączeniu urządzenia może zaistnieć konieczność zwolnienia jakichś zasobów, a automat może nie zareagować właściwie w takiej sytuacji – w szczególności funkcja realizująca automat przestanie może być wywoływana. Parametr `param` jest wskaźnikiem, który będzie przekazywany jako argument `param` funkcji zwrotnych `machine`, `at_sof` i `at_disconnect`. Pozwala on przekazywać dane do tych funkcji. Zwykle będzie to wskaźnik na strukturę zawierającą dane specyficzne dla urządzeń konkretnej klasy. Opisywana funkcja zwraca wartość `USBHLIB_SUCCESS`, gdy automat został zainstalowany, a wartość `USBHLIB_ERROR_BUSY` w przeciwnym przypadku, gdy podstawowy automat nie zakončył swojego działania lub wciąż działa jakiś inny zainstalowany specyficzny automat i nie można zainstalować nowego automatu. Wywołanie omawianej funkcji jest nieblokujące i może być ona wywoływana poza kontekstem obsługi przerwania.

```
usbh_std_req.h
usbh_std_req.c
```

Wszystkie żądania można oczywiście zrealizować za pomocą funkcji USBHcontrolRequest. Jednak wiele żądań występuje tak często, że warto mieć dla nich odpowiedni zestaw specjalizowanych funkcji. Nagłówki potrzebnych funkcji znajdują się w pliku *usbh_std_req.h*, a ich implementacja – w pliku *usbh_std_req.c*. Wszystkie one obudowują wywołanie funkcji USBHcontrolRequest. Pierwszy ich argument synch ma takie samo znaczenie i jest do niej przekazywany. Wszystkie funkcje udostępniane przez ten moduł zwracają wartość przekazaną przez funkcję USBHcontrolRequest.

```
int USBHsetDeviceAddress(int synch, uint8_t addr);
```

Funkcja USBHsetDeviceAddress realizuje żądanie SET_ADDRESS. Parametr addr zawiera adres, który ma być ustawiony.

```
int USBHsetConfiguration(int synch, uint8_t conf);
```

Funkcja USBHsetConfiguration realizuje żądanie SET_CONFIGURATION. Parametr conf zawiera numer konfiguracji, która ma być ustawiona, albo zero, jeśli bieżąca konfiguracja ma być dezaktywowana.

```
int USBHclearEndpointHalt(int synch, uint8_t ep_addr);
```

Funkcja USBHclearEndpointHalt wysyła żądanie uaktywnienia punktu końcowego o adresie podanym w parametrze ep_addr, czyli żądanie CLEAR_FEATURE z parametrem wValue o wartości ENDPOINT_HALT.

```
int USBHgetDeviceDescriptor(int synch, uint8_t *desc,
                           uint16_t length);
```

Funkcja USBHgetDeviceDescriptor odczytuje deskryptor urządzenia. Parametr desc jest wskaźnikiem na bufor, do którego ma być skopiowany ten deskryptor. Parametr length zawiera maksymalną liczbę bajtów, które mają być odebrane. Nie może to być więcej, niż wynosi rozmiar bufora. Funkcja ta sprawdza, czy rozmiar odebranych danych zgadza się z wartością parametru length oraz czy otrzymany deskryptor jest rzeczywiście deskryptorem urządzenia. Jeśli sprawdzenia wypadną pozytywnie i lokalny porządek bajtów jest inny niż ten obowiązujący w USB, to zamienia kolejność bajtów w dwubajtowych polach deskryptora.

```
int USBHgetConfDescriptor(int synch, uint8_t idx, uint8_t *desc,
                          uint16_t length);
```

Funkcja USBHgetConfDescriptor odczytuje deskryptor konfiguracji. Parametr idx zawiera numer deskryptora. Parametr desc jest wskaźnikiem na bufor, do którego

ma być skopiowany ten deskryptor. Parametr `length` zawiera maksymalną liczbę bajtów, które mają być odczytane. Nie może to być więcej niż rozmiar bufora. Funkcja ta sprawdza rozmiar odebranych danych oraz poprawność typu otrzymanego deskryptora. Jeśli wszystko się zgadza i lokalny porządek bajtów jest inny niż ten obowiązujący w USB, to zamienia kolejność bajtów dwubajtowych pól deskryptora konfiguracji (jest tylko jedno takie pole, mianowicie `wTotalLength`). Funkcja ta może być również użyta do odczytania całej konfiguracji. Wtedy deskryptory znajdujące się za deskryptorem konfiguracji nie są analizowane, a więc nie są też modyfikowane ich wielobajtowe pola.

```
int USBHgetStringDescriptorASCII(int synch, uint8_t idx, char *desc,
                                 unsigned *length);
```

Funkcja `USBHgetStringDescriptorASCII` odczytuje deskryptor tekstowy. Parametr `idx` zawiera numer deskryptora. Wysyłane jest żądanie deskryptora w języku angielskim, a dokładniej w jego amerykańskiej wersji. Funkcja sprawdza, czy urządzenie przysłało poprawny deskryptor tekstowy. Jeśli sprawdzenie zakończyło się sukcesem, to otrzymany deskryptor jest konwertowany do postaci ASCII, a następnie kopiowany do bufora wskazywanego przez parametr `desc`. Ewentualne znaki nienależące do zbioru ASCII są zamieniane na znak zapytania. Parametr `length` jest wskaźnikiem na zmienną, która przed wywołaniem funkcji zawiera rozmiar bufora, a po odczytaniu deskryptora zawiera liczbę bajtów rzeczywiście skopiowanych do bufora. Tekst w buforze nie jest zakończony terminalnym zerem.

7.1.5. Funkcje pomocnicze

`usbh_lib.h`
`usbh_lib.c`

Moduł `usbh_lib` udostępnia kilka pomocniczych, ale dość przydatnych, funkcji, które nie znalazły swojego miejsca w innych modułach. Funkcje te są zadeklarowane w pliku `usbh_lib.h`. Plik `usbh_lib.c` zawiera ich implementację.

```
int USBHopenDevice(usb_speed_t *speed, uint8_t *dev_addr,
                    usb_device_descriptor_t *dev_desc, int timeout);
```

Funkcja `USBHopenDevice` oczekuje w pętli na podłączenie urządzenia. Parametr `timeout` określa liczbę iteracji, które zostaną wykonane. Jeśli w tym czasie urządzenie nie zostanie wykryte, to funkcja zwraca wartość `USBHLIB_ERROR_TIMEOUT`. Natomiast jeśli urządzenie zostanie wykryte, to zwraca wartość `USBHLIB_SUCCESS` i wtedy pozostałe parametry wywołania tej funkcji, będące wskaźnikami, służą do zwrócenia podstawowych informacji o tym urządzeniu. Za pomocą parametru `speed` zwracana jest szybkość, z jaką zostało ono uruchomione. Za pomocą parametru `dev_addr` zwracany jest przynajmniej mu adres. Za pomocą parametru `dev_desc` zwracany jest deskryptor urządzenia. Wywołanie tej funkcji jest blokujące i nie może ona być wywoływana w kontekście obsługi przerwania.

```
int USBHisDeviceReady(void);
```

Bezparametrowa funkcja `USBHisDeviceReady` zwraca jedynkę, gdy kontroler wykrył podłączone urządzenie i ma ono przypisany adres, czyli jest gotowe do komunikacji. W przeciwnym przypadku funkcja ta zwraca zero. Wywołanie tej funkcji jest nieblokujące i może ona być wywoływana poza kontekstem obsługi przerwania.

```
void USBHdeviceHardReset(unsigned time);
```

Funkcja `USBHdeviceHardReset` odłącza zasilanie linii VBUS portu USB na `time` milisekund. Jeśli podłączone urządzenie jest zasilane z portu, to w ten sposób można go całkowicie wyzerować i ponownie uruchomić. Funkcja ta nie zwraca żadnej wartości. Funkcja ta jest również przydatna, aby odłączyć zasilanie, gdy z linii VBUS pobierany jest zbyt duży prąd. Wywołanie tej funkcji jest nieblokujące i może ona być wywoływana zarówno w kontekście, jak i poza kontekstem obsługi przerwania.

7.1.6. Przerwania kontrolera

<code>usb_interrupt.h</code>
<code>usbh_interrupt.h</code>
<code>usbh_interrupt.c</code>

Obsługa przerwań kontrolera zaimplementowana jest w pliku `usbh_interrupt.c`. Zawiera on funkcje, które nie zależą od modelu mikrokontrolera, ale wciąż zależą od typu układu peryferyjnego USB.

```
void USBglobalInterruptHandler(void);
```

Bezparametrowa funkcja `USBglobalInterruptHandler` jest głównym punktem wejścia do obsługi wszystkich przerwań USB. Jest ona zadeklarowana w pliku `usb_interrupt.h` i jest czymś w rodzaju rozdzielnika. Rozpoznaje przyczynę przerwania i przekazuje sterowanie do właściwej funkcji, która ma zająć się jego obsługą. Nie zwraca żadnej wartości. Jest wywoływana przez zależną od modelu mikrokontrolera procedurę, która znajduje się w pliku `usb_otg_interrupt.c`.

```
void USBHovercurrentInterruptHandler(void);
```

Bezparametrowa funkcja `USBHovercurrentInterruptHandler` obsługuje przerwanie, które jest zgłasiane, gdy układ zasilający linię VBUS wykryje zbyt duży pobór prądu (np. zwarcie VBUS do masy). Zależnie od sposobu zasilania linii VBUS funkcja ta może być wywołana przez funkcję `USBglobalInterruptHandler` albo przez znajdująjącą się w pliku `usbh_configure_*.c` procedurę obsługującą przerwanie zewnętrzne. Funkcja ta odłącza zasilanie linii VBUS na około trzy sekundy w nadziei, że usunie to przyczynę zwarcia. Nie zwraca żadnej wartości. Funkcja ta jest zadeklarowana w pliku `usbh_interrupt.h`.

Ponadto w pliku *usbh_interrupt.h* zdefiniowane są następujące trzy pomocnicze funkcje.

```
static inline uint32_t USBHgetInterruptPriority(void);
```

Bezparametrowa funkcja `USBHgetInterruptPriority` zwraca priorytet wywieszczania wszystkich przerwań związanych z obsługą USB. Aktualnie jest to `MIDDLE_IRQ_PRIO` zdefiniowany w pliku *irq.h*.

```
static inline irq_level_t USBHprotectInterrupt(void);
```

Bezparametrowa funkcja `USBHprotectInterrupt` blokuje wszystkie przerwania o priorytecie równym lub niższym niż priorytet wywieszczania przerwań związanych z obsługą USB. Zwraca poprzedni stan blokowania przerwań. Funkcja ta korzysta z funkcji `IRQprotect` zdefiniowanej w pliku *irq.h*.

```
static inline void USBHunprotectInterrupt(irq_level_t level);
```

Funkcja `USBHunprotectInterrupt` przywraca stan blokowania przerwań zapisany w parametrze `level`, którego wartość została zwrocona przez funkcję `USBHprotectInterrupt`. Funkcja ta korzysta z funkcji `IRQunprotect` zdefiniowanej w pliku *irq.h*.

7.1.7. Wybrane fragmenty implementacji

usbh_interrupt.c

Przeanalizujmy teraz kluczowe fragmenty obsługi przerwań i implementacji rdzenia protokołu po stronie kontrolera. Zdarzenia związane z podłączaniem urządzenia do portu kontrolera sygnalizowane są przez ustawienie bitu `hprtint` w rejestrze `GINTSTS`. W odpowiedzi na przerwanie wyzwolone tym zdarzeniem wywołujemy funkcję `HostPortHandler`. Dokładną przyczynę zdarzenia rozpoznajemy, analizując zawartość rejestru `HPRT`.

```
static void HostPortHandler(void) {
    USB_OTG_HPRT_TypeDef hprt;
    USB_OTG_HCFG_TypeDef hcfg;
    hprt.d32 = P_USB_OTG_HREGS->HPRT;
    if (hprt.b.pcdet) {
        hcsg.d32 = 0;
        if (OTG_FS_REGS_USED)
            hcsg.b.fslspcs = hprt.b.pspd;
        else if (OTG_HS_REGS_USED)
            hcsg.b.fslspcs = 1;
        P_USB_OTG_HREGS->HCFG = hcsg.d32;
        USBHdeviceAttached();
        TimerStart(1, StartSignallingPortReset, STARTUP_TIME_MS);
        RedLEDOff();
```

```

        }
        if (hppt.b.penchng && hppt.b.pena && hppt.b.pcsts) {
            USBHdeviceSpeed(hppt.b.pspd);
            TimerStart(1, USBHdeviceResetDone, RECOVERY_TIME_MS);
        }
        hppt.b.pena = 0;
        P_USB_OTG_HREGS->HPRT = hppt.d32;
    }
}

```

Ustawiony bit pcdet w rejestrze HPRT oznacza, że zostało wykryte podłączenie urządzenia do portu USB. W polu fsllspcs rejestrów HCFG konfigurujemy częstotliwość taktowania układu peryferyjnego, gdy pracuje on w trybie LS lub FS. Konfiguracja ta jest nieco inna dla układów OTG-FS i OTG-HS. Stała OTG_FS_REGS_USED ma wartość niezerową, gdy używamy układu OTG-FS. Natomiast stała OTG_HS_REGS_USED jest niezerowa dla OTG-HS. Żeby poinformować rdzeń protokołu o podłączeniu nowego urządzenia, wywołujemy funkcję USBHdeviceAttached. Włączamy licznik, aby po upływie czasu określonego przez stałą STARTUP_TIME_MS wywołać funkcję StartSignallingPortReset, która rozpoczęcie zerowanie szyny. Wartość tej stałej jest zdefiniowana w pliku *usb_def.h* zgodnie ze standardem na 100 ms. W tym czasie urządzenie powinno się uruchomić i zainicjować. Na koniec w celach diagnostycznych sygnalizujemy podłączenie urządzenia, wyłączając czerwoną diodę świecącą.

Ustawiony bit penchng w rejestrze HPRT oznacza, że uległa zmianie wartość bitu pena w tym rejestrze. Bit pena informuje, czy port USB jest aktywny. Bit pena jest ustawiany sprzętowo po zakończeniu zerowania szyny. Ustawiony bit pcsts w rejestrze HPRT oznacza, że do portu kontrolera jest podłączone urządzenie. Podsumowując, jednoczesne ustawienie tych trzech bitów oznacza koniec zerowania szyny. W odpowiedzi na to zdarzenie informujemy rdzeń protokołu o szybkości podłączonego urządzenia za pomocą funkcji USBHdeviceSpeed. Pole pspd rejestrów HPRT zawiera szybkość, z jaką zostało uruchomione urządzenie (0 oznacza HS, 1 – FS, a 2 – LS). Następnie włączamy licznik, aby po upływie czasu zdefiniowanego za pomocą stałej RECOVERY_TIME_MS poinformować rdzeń protokołu o zakończeniu zerowania szyny. Wartość tej stałej jest zdefiniowana w pliku *usb_def.h* zgodnie ze standardem na 10 ms. W tym czasie urządzenie musi zakończyć wszystkie akcje związane z zerowaniem szyny.

Na koniec funkcji HostPortHandler zapisujemy ponownie odczytaną wartość rejestrów HPRT, aby wyzerować bity zdarzeń i pozostawić pozostałe bity niezmienione. Obowiązuje typowa konwencja, że bity sygnalizujące zdarzenia zerują się przez zapisanie do nich jedynki. Wartość bitu pena ustawiamy na zero, bo nie chcemy go wyzerować.

W funkcji StartSignallingPortReset ustawiamy bit prst rejestrów HPRT, aby rozpocząć generowanie sygnału zerującego szynę. Podczas zapisu do rejestrów HPRT bity sygnalizujące zdarzenia ustawiamy na zero, gdyż nie chcemy ich wyzerować – nie chcemy zgubić żadnego zdarzenia. Na koniec tej funkcji włączamy licznik, aby za pomocą funkcji StopSignallingPortReset zatrzymać generowanie sygnału zerującego po czasie określonym przez stałą RESET_TIME_MS. Czas ten jest zdefiniowany w pliku *usb_def.h* zgodnie ze standardem na 15 ms.

```

static void StartSignallingPortReset(void) {
    USB_OTG_HPRT_TypeDef hprt;
    hprt.d32 = P_USB_OTG_HREGS->HPRT;
    hprt.b.prst = 1;
    hprt.b.pocchng = 0;
    hprt.b.penchnng = 0;
    hprt.b.pena = 0;
    hprt.b.pcdet = 0;
    P_USB_OTG_HREGS->HPRT = hprt.d32;
    TimerStart(1, StopSignallingPortReset, RESET_TIME_MS);
}

```

W funkcji StopSignallingPortReset zerujemy bit prst rejestru HPRT, aby zatrzymać generowanie sygnału zerującego szynę. Podczas zapisu do rejestru HPRT bity sygnalizujące zdarzenia ustawiamy na zero, gdyż nie chcemy zgubić żadnego zdarzenia.

```

static void StopSignallingPortReset(void) {
    USB_OTG_HPRT_TypeDef hprt;
    hprt.d32 = P_USB_OTG_HREGS->HPRT;
    hprt.b.prst = 0;
    hprt.b.pocchng = 0;
    hprt.b.penchnng = 0;
    hprt.b.pena = 0;
    hprt.b.pcdet = 0;
    P_USB_OTG_HREGS->HPRT = hprt.d32;
}

```

Zdarzenie odłączenia urządzenia od portu kontrolera sygnalizowane jest przez ustawienie bitu discint w rejestrze GINTSTS. W odpowiedzi na przerwanie wyzwolone tym zdarzeniem wywołujemy funkcję HostDisconnectHandler. W funkcji tej zatrzymujemy używane liczniki. Na wszelki wypadek zatrzymujemy generowanie sygnału zerującego szynę – może się tak zdarzyć, że urządzenie zostanie odłączone podczas generowania tego sygnału. Informujemy o zdarzeniu rdzeń protokołu, wywołując funkcję USBHdeviceDisconnected. Na koniec w celach diagnostycznych sygnalizujemy odłączenie urządzenia, włączając zieloną diodę świecącą (za pomocą opisanej poniżej funkcji Blink) i włączając czerwoną.

```

static void HostDisconnectHandler(void) {
    FineTimerStop(1);
    TimerStop(1);
    StopSignallingPortReset();
    USBHdeviceDisconnected();
    Blink(0);
    RedLEDDon();
}

```

Przerwanie wyzwalane na początku każdej ramki lub mikroramki obsługujemy w funkcji HostSofHandler. Informujemy o tym zdarzeniu rdzeń protokołu za pomocą funkcji USBHsOf. Na koniec w celach diagnostycznych wywołujemy funkcję Blink. Funkcja ta, wywoływaną z parametrem o wartości jeden, migra zieloną diodą świeczącą z częstotliwością 1000 razy mniejszą, niż wynosi częstotliwość jej wywoływania. Realizuje to wewnętrzna zmienna statyczna zliczająca w dół od 999 do 0. W jednej sekundzie mamy 1000 ramek lub 8000 mikroramek. Zatem zielona dioda migra odpowiednio z częstotliwością 1 Hz lub 8 Hz. Funkcja Blink wywołana z parametrem o wartości zero wyłącza zieloną diodę świeczącą.

```
static void HostSofHandler(void) {
    USBHsOf(USBHGetCurrentFrame());
    Blink(1);
}
```

Każde przerwanie USB informuje o jakimś zdarzeniu na szynie, dlatego po obsłudze przerwania wypada sprawdzić, czy automat stanowy nie ma czegoś do zrobienia. Wszystkie przerwania USB są najpierw kierowane do funkcji USBglobalInterruptHandler, która deleguje ich obsługę do innych funkcji. Po zakończeniu obsługi przerwania włączamy licznik, który po 2 µs, więc praktycznie natychmiast, wywoła funkcję CoreProcessHandler. Za pomocą tej funkcji wywołujemy funkcję USBHcoreProcess, czyli główny punkt wejścia do automatu stanowego realizującego protokół USB.

```
void USBglobalInterruptHandler() {
    ...
    FineTimerStart(1, CoreProcessHandler, 2);
}

static void CoreProcessHandler(void) {
    USBHcoreProcess();
    if (USBHgetFrameClocksRemaining() >= DeadScheduleClocks)
        FineTimerStart(1, CoreProcessHandler, CoreScheduleTime);
    else
        FineTimerStop(1);
}
```

Żeby automat mógł zainicjować transmisję danych na żądanie aplikacji również wtedy, gdy nie pojawia się żadne przerwanie, wywołujemy funkcję USBHcoreProcess po czasie określonym przez zmienną CoreScheduleTime. Jednak nie wywołujemy jej po raz kolejny, jeśli do końca ramki lub mikroramki pozostało mniej czasu, niż określa to zmienna DeadScheduleClocks inicjowana w ten sposób, aby odpowiadało to w mikrosekundach podwojonemu czasowi określzonemu w zmiennej CoreScheduleTime, która z kolei jest inicjowana zależnie od szybkości pracy urządzenia. Odpowiednie stałe są zdefiniowane w pliku *usb_def.h*:

- HOST_LS_SCHEDULE_US – 275 µs dla HS,
- HOST_FS_SCHEDULE_US – 30 µs dla FS,
- HOST_HS_SCHEDULE_US – 15 µs dla LS.

Zaprezentowany tu sposób pobudzania automatów stanowych należy traktować jako przykład. Czasy zostały dobrane dość arbitralnie, z uwagi na czas transmisji pakietu i wydajność mikrokontrolera. W konkretnych zastosowaniach można wypróbować inne czasy albo wręcz zupełnie inną heurystykę wywoływania automatów.

usbh_core.c

Przejdźmy teraz do omówienia implementacji automatów realizujących protokołów USB po stronie kontrolera. Punktem wejścia do tych automatów jest funkcja `USBHcoreProcess`. Dane niezbędne do działania wszystkich automatów przechowuje globalna struktura `Machine`. Składowa `g_state` zawiera aktualny stan głównego automatu. Składowa `g_prev_state` zawiera jego poprzedni stan. Główny automat może przebywać w jednym z czterech stanów:

- `HOST_CONTROL_TRANSFER` – obsługa żądania, czyli transmisja danych sterujących;
- `HOST_ENUMERATION` – przyznawanie urządzeniu adresu;
- `HOST_CLASS` – obsługa protokołu specyficznego dla konkretnej klasy urządzeń;
- `HOST_IDLE` – stan spoczynkowy.

W każdym z trzech pierwszych stanów uruchamiany jest odpowiedni podautomat. Dla każdego z tych podautomatów struktura `Machine` zawiera odpowiednią podstrukturę, przechowującą jego dane. Są to odpowiednio podstruktury `control`, `enumeration` i `class`.

```
void USBHcoreProcess() {  
    switch (Machine.g_state) {  
        case HOST_CONTROL_TRANSFER:  
            if (USBHhandleControlTransfer())  
                Machine.g_state = Machine.g_prev_state;  
            return;  
        case HOST_ENUMERATION:  
            if (USBHhandleEnumeration()) {  
                Machine.g_state = HOST_IDLE;  
                if (Machine.enumeration.error != USBHLIB_SUCCESS)  
                    USBHdeviceHardReset(DEVICE_RESET_TIME_MS);  
            }  
            return;  
        case HOST_CLASS:  
            if (Machine.class.machine) {  
                if (Machine.class.machine(Machine.class.parameter)) {  
                    Machine.g_state = HOST_IDLE;  
                    Machine.class.machine = 0;  
                    Machine.class.at_sof = 0;  
                    Machine.class.at_disconnect = 0;  
                }  
            }  
    }  
}
```

```
        Machine.class.parameter = 0;
    }
}

return;
default: /* HOST_IDLE */
return;
}

}
```

W stanie `HOST_CONTROL_TRANSFER` uruchamiamy automat obsługujący transmisję danych sterujących. Jest on zaimplementowany w funkcji `USBHandleControlTransfer`. Po zakończeniu tego automatu, czyli gdy funkcja ta zwróci wartość jeden, główny automat wraca do stanu `g_prev_state`. Jest to `HOST_IDLE`, `HOST_ENUMERATION` albo `HOST_CLASS`, zależnie od tego, w którym z tych stanów zostało zainicjowane żądanie.

W stanie `HOST_ENUMERATION` uruchamiamy automat nadający urządzeniu docelowy adres. Jest on zaimplementowany w funkcji `USBHandleEnumeration`. Funkcja ta zwraca wartość jeden, gdy automat skończył swoje działanie. Składowa `errno` podstruktury `enumeration` zawiera kod zakończenia operacji według tabeli 7.1. Jeśli nie udało się nadać urządzeniu adresu, gdyż wystąpił jakiś problem w komunikacji z nim, wywołujemy funkcję `USBDeviceHardReset`, która odłącza zasilanie szyny USB na czas określony przez parametr `DEVICE_RESET_TIME_MS`. Aktualnie czas ten jest zdefiniowany w pliku `usb_def.h` na 300 ms. Celem takiego działania jest wyzerowanie urządzenia i podjęcie ponownej próby jego uruchomienia (zakładamy, że urządzenie jest zasilane z szyny USB). Z punktu widzenia kontrolera zachodzi taka sama sekwencja zdarzeń, jakby urządzenie zostało odłączone i ponownie podłączone.

W stanie `HOST_CLASS`, jeśli został zainstalowany, uruchamiamy automat specyficzny dla klasy urządzeń. Wywołujemy w tym celu funkcję określoną w składowej `machine`, do której przekazujemy parametr przechowywany w składowej `parameter`. Jeśli automat zakończył swoje działanie, czyli gdy funkcja go implementująca zwróciła jedynkę, główny automat wraca do stanu spoczynkowego, a automat zostaje odinstalowany – zerujemy składowe podstruktury `class`. Składowa `at_sof` zawiera adres funkcji, która ma być wywoływana na początku każdej ramki lub mikroramki. Składowa `at_disconnect` zawiera adres funkcji, która ma być wywołana po wykryciu odłączenia urządzenia. Ponieważ po odinstalowaniu automatu nie zostanie ona wywołana, automat przed zakończeniem swojego działania musi zwolnić wszelkie przydzielone mu zasoby.

Automat realizujący przyznawanie urządzeniu adresu zaimplementowany jest w funkcji `USBHandleEnumeration`. Jego działanie sprowadza się do inicjowania kolejnych żądań. W zmiennej `res` zapisujemy wynik realizacji żądania. Składowa `state` podstruktury `enumeration` zawiera aktualny stan tego automatu.

```
static int USBHhandleEnumeration(void) {
    int res;
    switch (Machine.enumeration.state) {
        case ENUM GET DEV DESC:
```

```
res = USBHgetDeviceDescriptor(0, (uint8_t *)  
                             &Device.dev_desc, 8);  
if (res == USBHLIB_SUCCESS) {  
    Machine.control.max_packet = Device.dev_desc.bMaxPacketSize0;  
    USBHmodifyChannel(Machine.control.hc_num_out,  
                      0, Machine.control.max_packet);  
    USBHmodifyChannel(Machine.control.hc_num_in,  
                      0, Machine.control.max_packet);  
    Machine.enumeration.state = ENUM_GET_FULL_DEV_DESC;  
}  
break;  
case ENUM_GET_FULL_DEV_DESC:  
    res = USBHgetDeviceDescriptor(0, (uint8_t *)&Device.dev_desc,  
                                 sizeof(usb_device_descriptor_t));  
    if (res == USBHLIB_SUCCESS) {  
        Machine.enumeration.state = ENUM_SET_DEV_ADDR;  
    }  
    break;  
case ENUM_SET_DEV_ADDR:  
    res = USBHsetDeviceAddress(0, DEVICE_ADDRESS);  
    if (res == USBHLIB_SUCCESS) {  
        Device.address = DEVICE_ADDRESS;  
        Device.visible_state = ADDRESS;  
        USBHmodifyChannel(Machine.control.hc_num_in,  
                           Device.address, 0);  
        USBHmodifyChannel(Machine.control.hc_num_out,  
                           Device.address, 0);  
        Machine.enumeration.state = ENUM_IDLE;  
    }  
    break;  
default: /* ENUM_IDLE */  
    res = USBHLIB_SUCCESS;  
    break;  
}  
Machine.enumeration(errno = res;  
return Machine.enumeration.state == ENUM_IDLE ||  
       (res != USBHLIB_SUCCESS && res != USBHLIB_IN_PROGRESS);  
}
```

W stanie ENUM_GET_DEV_DESC odczytujemy początkowe 8 bajtów deskryptora urządzenia, aby poznać wartość pola bMaxPacketSize0 tego deskryptora, czyli rozmiar pola danych pakietów sterujących. Wartość tę zapisujemy w składowej max_packet.

ket podstruktury `control`, gdyż będzie ona potrzebna, aby móc prawidłowo realizować transmisję danych sterujących. Ponieważ każde urządzenie musi poprawnie obsługiwać żądanie, w którym przesyłamy nie więcej niż 8 bajtów danych, taka wartość składowej `max_packet` jest ustawiona na początku i taki rozmiar pola danych pakietów mają początkowo ustawione kanały. Jeśli żądanie zakończyło się sukcesem, to za pomocą funkcji `USBHmodifyChannel` ustawiamy nowy rozmiar pola danych dla kanałów kontrolera obsługujących transmisję danych sterujących z naszym urządzeniem. Na zakończenie przechodzimy do stanu `ENUM_GET_FULL_DEV_DESC`. Otrzymany deskryptor urządzenia przechowujemy w składowej `dev_desc` globalnej struktury `Device`. Struktura ta przechowuje również adres, stan i szybkość urządzenia.

W stanie `ENUM_GET_FULL_DEV_DESC` odczytujemy cały deskryptor urządzenia. Jeśli odczyt się powiodł, to przechodzimy do stanu `ENUM_SET_DEV_ADDR`, w którym nadajemy urządzeniu jego docelowy adres. Ponieważ nasz kontroler obsługuje jednocześnie tylko jedno urządzenie, za każdym razem przyznajemy mu ten sam adres, określony przez stałą `DEVICE_ADDRESS`, która ma wartość 1. Jeśli udało się ustawić adres, to w globalnej strukturze `Device`, która przechowuje informacje o podłączonym urządzeniu, zapamiętujemy ten adres w składowej `address` i ustawiamy w składowej `visible_state`, że urządzenie jest teraz w stanie `ADDRESS`. Za pomocą funkcji `USBHmodifyChannel` modyfikujemy adres urządzenia w konfiguracji kanałów kontrolera obsługujących transmisję sterującą, aby kolejne żądania były wysyłane na nowy adres. Początkowo kanały mają ustawiony domyślny, czyli zeroowy adres urządzenia. Na zakończenie przechodzimy do stanu spoczynkowego `ENUM_IDLE`, w którym automat przebywa, gdy nie ma nic do zrobienia.

Funkcja `USBHandleEnumeration` zwraca jedynkę, gdy automat osiągnął stan `ENUM_IDLE` lub ktoś żądanie zakończyło się błędem. Oznacza to, że automat ma przestać być wywoływany. Jeśli automat nie zakończył jeszcze swojego działania i ma być ponownie wywołany, funkcja ta zwraca zero.

Automat realizujący żądania przesyłane do punktu końcowego zero, czyli do domyślnego punktu końcowego dla danych sterujących, zaimplementowany jest w funkcji `USBHandleControlTransfer`. Stan tego automatu przechowywany jest w składowej `state` podstruktury `control`. Składowa `setup` tej podstruktury zawiera strukturę wysyłaną w transakcji `SETUP`. Składowa `pid` określa PID pakietu danych w kolejnej transakcji IN lub OUT realizowanego żądania. Przyjmuje wartość `PID_DATA0` lub `PID_DATA1`. Składowa `buffer` zawiera wskaźnik na bufor, w którym znajdują się dane do wysłania lub do którego mają być skopiowane odebrane dane. Składowa `length` zawiera rozmiar danych, które mają być przesłane w ramach żądania (rozmiar bufora). Po zakończeniu realizacji żądania w składowej `transferred` znajduje się rozmiar rzeczywiście przesłanych danych. Składowa `timeout` służy do odliczania czasu, jaki pozostał do zakończenia realizacji żądania. Czas liczony jest w ramkach dla urządzeń LS i FS, a w mikroramkach dla urządzeń HS. W składowej `error_count` zlicza się liczbę nieudanych prób zrealizowania transakcji. Składowa `errno` zawiera kod błędu, czyli wartość, która informuje o poprawnym lub niepoprawnym zakończeniu realizacji żądania i która ma być zwrocona do aplikacji. Kody błędów zdefiniowane są w tabeli 7.1. Składowe `hc_num_in` i `hc_num_out` przechow-

wują numery kanałów używanych odpowiednio do odbierania danych z urządzenia i wysyłania danych do urządzenia. Wspomniana już składowa `max_packet` przechowuje maksymalny rozmiar pola danych pakietów.

```
static int USBHandleControlTransfer(void) {
    usbh_transaction_result_t res;
    uint32_t len;
    switch (Machine.control.state) {
        case CTRL_SETUP:
            if (USBHstartTransaction(Machine.control.hc_num_out,
                                      PID_SETUP,
                                      (uint8_t *)&Machine.control.setup,
                                      sizeof(usb_setup_packet_t)) == 0) {
                Machine.control.state = CTRL_SETUP_WAIT;
            }
            else {
                Machine.control(errno = USBHLIB_ERROR_IO);
                Machine.control.state = CTRL_DONE;
            }
            break;
        case CTRL_SETUP_WAIT:
            res = USBHgetTransactionResult(Machine.control.hc_num_out);
            if (res == TR_DONE) {
                if (Machine.control.length != 0) {
                    Machine.control.pid = PID_DATA1;
                    Machine.control.timeout = DATA_STAGE_TIMEOUT_MS;
                    if ((Machine.control.setup.bmRequestType &
                         REQUEST_DIRECTION) == DEVICE_TO_HOST)
                        Machine.control.state = CTRL_DATA_IN;
                    else
                        Machine.control.state = CTRL_DATA_OUT;
                }
                else {
                    Machine.control.timeout = NODATA_STAGE_TIMEOUT_MS;
                    Machine.control.state = CTRL_STATUS_IN;
                }
                if (Device.speed == HIGH_SPEED) {
                    Machine.control.timeout <= 3;
                }
                Machine.control.error_count = 0;
            }
            else if (res == TR_ERROR) {
                if (++Machine.control.error_count < TRANS_MAX REP COUNT) {
                    Machine.control.state = CTRL_SETUP;
                }
            }
            else {
```

```
        Machine.control(errno = USBHLIB_ERROR_IO;
        Machine.control.state = CTRL_DONE;
    }
}
break;
case CTRL_DATA_IN:
    len = min(Machine.control.length, Machine.control.max_packet);
    if (USBHstartTransaction(Machine.control.hc_num_in,
                                Machine.control.pid,
                                Machine.control.buffer, len) == 0) {
        Machine.control.state = CTRL_DATA_IN_WAIT;
    }
else {
    Machine.control(errno = USBHLIB_ERROR_IO;
    Machine.control.state = CTRL_DONE;
}
break;
case CTRL_DATA_IN_WAIT:
    res = USBHgetTransactionResult(Machine.control.hc_num_in);
    if (res == TR_DONE) {
        len = USBHgetTransactionSize(Machine.control.hc_num_in);
        Machine.control.buffer += len;
        Machine.control.transferred += len;
        Machine.control.length -= len;
        if (Machine.control.length == 0 ||
            len < Machine.control.max_packet) {
            Machine.control.state = CTRL_STATUS_OUT;
        }
    else {
        Machine.control.pid = USBtoggleDataPid(Machine.control.pid);
        Machine.control.state = CTRL_DATA_IN;
    }
    Machine.control.error_count = 0;
}
else if (Machine.control.timeout < 0) {
    USBHhaltChannel(Machine.control.hc_num_in);
    Machine.control(errno = USBHLIB_ERROR_TIMEOUT;
    Machine.control.state = CTRL_DONE;
}
else if (res == TR_NAK) {
    Machine.control.error_count = 0;
    Machine.control.state = CTRL_DATA_IN;
}
else if (res == TR_STALL) {
    Machine.control(errno = USBHLIB_ERROR_STALL;
```

```
        Machine.control.state = CTRL_DONE;
    }
    else if (res == TR_ERROR) {
        if (++Machine.control.error_count < TRANS_MAX REP COUNT) {
            Machine.control.state = CTRL_DATA_IN;
        }
        else {
            Machine.control(errno = USBHLIB_ERROR_IO;
            Machine.control.state = CTRL_DONE;
        }
    }
    break;
case CTRL_DATA_OUT:
    len = min(Machine.control.length, Machine.control.max_packet);
    if (USBHstartTransaction(Machine.control.hc_num_out,
                            Machine.control.pid,
                            Machine.control.buffer, len) == 0) {
        Machine.control.state = CTRL_DATA_OUT_WAIT;
    }
    else {
        Machine.control(errno = USBHLIB_ERROR_IO;
        Machine.control.state = CTRL_DONE;
    }
    break;
case CTRL_DATA_OUT_WAIT:
    res = USBHgetTransactionResult(Machine.control.hc_num_out);
    if (res == TR_DONE) {
        len = USBHgetTransactionSize(Machine.control.hc_num_out);
        Machine.control.buffer += len;
        Machine.control.transferred += len;
        Machine.control.length -= len;
        if (Machine.control.length == 0) {
            Machine.control.state = CTRL_STATUS_IN;
        }
        else {
            Machine.control.pid = USBtoggleDataPid(Machine.control.pid);
            Machine.control.state = CTRL_DATA_OUT;
        }
        Machine.control.error_count = 0;
    }
    else if (Machine.control.timeout < 0) {
        USBHhaltChannel(Machine.control.hc_num_out);
        Machine.control(errno = USBHLIB_ERROR_TIMEOUT;
        Machine.control.state = CTRL_DONE;
```

```
        }

    else if (res == TR_NAK) {
        Machine.control.error_count = 0;
        Machine.control.state = CTRL_DATA_OUT;
    }

    else if (res == TR_STALL) {
        Machine.control(errno = USBHLIB_ERROR_STALL;
        Machine.control.state = CTRL_DONE;
    }

    else if (res == TR_ERROR) {
        if (++Machine.control.error_count < TRANS_MAX REP COUNT) {
            Machine.control.state = CTRL_DATA_OUT;
        }
        else {
            Machine.control(errno = USBHLIB_ERROR_IO;
            Machine.control.state = CTRL_DONE;
        }
    }
    break;
}

case CTRL_STATUS_IN:
    if (USBHstartTransaction(Machine.control.hc_num_in, PID_DATA1,
        0, 0) == 0) {
        Machine.control.state = CTRL_STATUS_IN_WAIT;
    }
    else {
        Machine.control(errno = USBHLIB_ERROR_IO;
        Machine.control.state = CTRL_DONE;
    }
    break;
}

case CTRL_STATUS_IN_WAIT:
    res = USBHgetTransactionResult(Machine.control.hc_num_in);
    if (res == TR_DONE) {
        Machine.control(errno = USBHLIB_SUCCESS;
        Machine.control.state = CTRL_DONE;
        Machine.control.error_count = 0;
    }
    else if (Machine.control.timeout < 0) {
        USBHhaltChannel(Machine.control.hc_num_in);
        Machine.control(errno = USBHLIB_ERROR_TIMEOUT;
        Machine.control.state = CTRL_DONE;
    }
    else if (res == TR_NAK) {
        Machine.control.error_count = 0;
        Machine.control.state = CTRL_STATUS_IN;
```

```
    }
    else if (res == TR_STALL) {
        Machine.control(errno = USBHLIB_ERROR_STALL;
        Machine.control.state = CTRL_DONE;
    }
    else if (res == TR_ERROR) {
        if (++Machine.control.error_count < TRANS_MAX REP COUNT) {
            Machine.control.state = CTRL_STATUS_IN;
        }
    else {
        Machine.control(errno = USBHLIB_ERROR_IO;
        Machine.control.state = CTRL_DONE;
    }
}
break;
case CTRL_STATUS_OUT:
    if (USBHstartTransaction(Machine.control.hc_num_out,
                           PID_DATA1, 0, 0) == 0) {
        Machine.control.state = CTRL_STATUS_OUT_WAIT;
    }
else {
    Machine.control(errno = USBHLIB_ERROR_IO;
    Machine.control.state = CTRL_DONE;
}
break;
case CTRL_STATUS_OUT_WAIT:
    res = USBHgetTransactionResult(Machine.control.hc_num_out);
    if (res == TR_DONE) {
        Machine.control(errno = USBHLIB_SUCCESS;
        Machine.control.state = CTRL_DONE;
        Machine.control.error_count = 0;
    }
    else if (Machine.control.timeout < 0) {
        USBHhaltChannel(Machine.control.hc_num_out);
        Machine.control(errno = USBHLIB_ERROR_TIMEOUT;
        Machine.control.state = CTRL_DONE;
    }
    else if (res == TR_NAK) {
        Machine.control.error_count = 0;
        Machine.control.state = CTRL_STATUS_OUT;
    }
    else if (res == TR_STALL) {
        Machine.control(errno = USBHLIB_ERROR_STALL;
        Machine.control.state = CTRL_DONE;
    }
}
```

```

        else if (res == TR_ERROR) {
            if (++Machine.control.error_count < TRANS_MAX REP COUNT) {
                Machine.control.state = CTRL_STATUS_OUT;
            }
            else {
                Machine.control(errno = USBHLIB_ERROR_IO;
                Machine.control.state = CTRL_DONE;
            }
        }
        break;
    default: /* CTRL_IDLE, CTRL_DONE */
        break;
}
return Machine.control.state == CTRL_DONE;
}

```

W stanie CTRL_SETUP inicjujemy transakcję SETUP (faza ustanowienia żądania). Jeśli się powiodło, przechodzimy do stanu CTRL_SETUP_WAIT, w którym oczekujemy na jej zakończenie, a następnie przechodzimy do stanu CTRL_DATA_IN lub CTRL_DATA_OUT, zależnie od kierunku, w którym mają być przesłane dane. Ewentualnie przechodzimy od razu do stanu CTRL_STATUS_IN, jeśli żądanie nie wymaga przesyłania żadnych danych.

Znalazłszy się w stanie CTRL_DATA_IN, inicjujemy transakcję IN (faza odbierania danych), a następnie przechodzimy do stanu CTRL_DATA_IN_WAIT, w którym oczekujemy na zakończenie tej transakcji. Jeśli w wyniku transakcji urządzenie przysłało pakiet danych, to przesuwamy wskaźnik buffer, aby wskazywał na miejsce, gdzie ma się znaleźć kolejny pakiet oraz odpowiednio modyfikujemy rozmiar przesyłanych danych transferred i rozmiar danych pozostałych do przesłania length. Jeśli faza odbierania danych zakończyła się (odebraliśmy wszystkie dane lub niepełny pakiet), to przechodzimy do stanu CTRL_STATUS_OUT. W przeciwny przypadku modyfikujemy składową pid i wracamy do stanu CTRL_DATA_IN, aby zainicjować kolejną transakcję i odebrać kolejną porcję danych. W stanie CTRL_STATUS_OUT inicjujemy transakcję OUT (faza statusu żądania – wysłanie pustego pakietu bez danych), po czym przechodzimy do stanu CTRL_STATUS_OUT_WAIT, w którym czekamy na zakończenie transakcji OUT fazy statusu, czyli na zakończenie całego żądania.

Znalazłszy się w stanie CTRL_DATA_OUT, inicjujemy transakcję OUT (faza wysyłania danych), a następnie przechodzimy do stanu CTRL_DATA_OUT_WAIT, w którym oczekujemy na zakończenie tej transakcji. Po jej poprawnym zakończeniu przesuwamy wskaźnik buffer, aby wskazywał na kolejną porcję danych do wysłania i modyfikujemy w składowej transferred rozmiar wysłanych danych, a w składowej length rozmiar danych pozostałych do wysłania. Jeśli wszystkie dane zostały wysłane, to przechodzimy do stanu CTRL_STATUS_IN. W przeciwnym przypadku modyfikujemy składową pid i wracamy do stanu CTRL_DATA_OUT, aby zainicjować kolejną transakcję i wysłać kolejną porcję danych. W stanie CTRL_STATUS_IN inicjujemy transakcję IN (faza statusu żądania – oczekiwanie na pusty pakiet), po czym przechodzimy do

stanu `CTRL_STATUS_IN_WAIT`, w którym czekamy na zakończenie transakcji IN fazy statusu, czyli na zakończenie całego żądania.

Jeśli któraś transakcja została odrzucona przez urządzenie pakietem NAK (funkcja `USBHgetTransactionResult` zwróciła `TR_NAK`), powtarzamy ją. Jeśli któraś transakcja została odrzucona przez urządzenie pakietem STALL (funkcja `USBHgetTransactionResult` zwróciła `TR_STALL`), uznajemy, że urządzenie odrzuciło żądanie (np. nie obsługuje tego żądania) lub nie chce odebrać wszystkich wysyłanych do niego danych.

Jeśli wystąpił błąd transmisji lub został przekroczony czas oczekiwania na pakiet, to funkcja `USBHgetTransactionResult` zwraca wartość `TR_ERROR`. W tej sytuacji kontroler ponawia próbę zrealizowania transakcji. Maksymalną liczbę prób określa stała `TRANS_MAX REP COUNT` zdefiniowana w pliku `usb_def.h`. Liczba dotyczących niepowodzeń zliczana jest w składowej `error_count` podstruktury `control`. Składowa ta jest zerowana po każdym poprawnym zakończeniu transakcji, a także wtedy, gdy transakcja zostanie odrzucona pakietem NAK.

Po zakończeniu swojego działania automat przechodzi do stanu `CTRL_DONE`. Wtedy funkcja `USBHhandleControlTransfer` sygnalizuje to, zwracając jedynkę. W każdym innym stanie funkcja ta zwraca zero. Jeśli automat zakończył swoje działanie i składowa `errno` podstruktury `control` zawiera wartość `USBHLIB_SUCCESS`, to żądanie zostało zrealizowane pomyślnie. W przeciwnym przypadku składowa `errno` zawiera kod błędu.

Automat ma dwa stany spoczynkowe: `CTRL_DONE` i `CTRL_IDLE`. Automat znajduje się w jednym z tych stanów, gdy nie realizuje żadnego żądania. Automat nie może sam opuścić stanu `CTRL_DONE`. Czeka w nim na odczytanie wyniku realizacji żądania zapisanego w składowej `errno`, po czym jest wprowadzany w stan `CTRL_IDLE`, w którym jest gotowy do przyjęcia nowego żądania.

W oczekiwaniu na zakończenie poszczególnych transakcji sprawdzamy też wartość składowej `timeout` podstruktury `control`, czy nie został przekroczony czas przeznaczony na realizację żądania. Istotne jest, że sprawdzenie to jest wykonywane przed sprawdzeniem innych błędów, aby automat nie wpadł w nieskończoną pętlę, gdy urządzenie permanentnie odrzuca pakietem NAK wszystkie transakcje. Warto też zwrócić uwagę na inicjowanie składowej `timeout`. Jeśli żądanie wymaga przesłania danych, to nadajemy jej początkową wartość `DATA_STAGE_TIMEOUT_MS`, która jest zdefiniowana w pliku `usb_def.h` na 5000 ms. Jeśli żądanie nie wymaga przesłania danych, to nadajemy jej początkową wartość `NODATA_STAGE_TIMEOUT_MS`, która jest zdefiniowana w pliku `usb_def.h` na 50 ms. Wartości tych czasów wynikają z zapisów standardu. Dodatkowo, jeśli jest podłączone urządzenie HS, to mnożymy początkową wartość `timeout` przez 8, aby zliczać mikroramki. Czas, jaki pozostały do końca realizacji żądania, odmierzany jest w funkcji `USBHsof`, wywoływanej na początku każdej ramki lub mikroramki.

```
void USBHsof(uint16_t frnum) {
    if (Machine.control.timeout >= 0)
        --Machine.control.timeout;
}
```

W fazie danych, zamiast inicjować kolejne transakcje za pomocą funkcji `USBH-startTransaction`, można byłoby użyć pojedynczego wywołania funkcji `USBH-startTransfer`. Jednak w praktyce zysk byłby niewielki, gdyż rozmiar przesyłanych danych nie jest duży. W przypadku transakcji OUT praktycznie zawsze jest to jeden pakiet, a dla transakcji IN najwyżej kilka pakietów (wynika to z rozmiaru największego odczytywanego przez kontroler deskryptora). Natomiast niewątpliwymi zaletami zaprezentowanego tu rozwiązania są wyraźnie wydzielenie warstwy transakcji oraz implementacja całego protokołu transmisji danych sterujących w jednym miejscu. Dzięki temu łatwo jest przeanalizować działanie tego protokołu, a jego implementacja może być bez przeszkód przeniesiona na inny sprzęt.

Żeby zainicjować żądanie z poziomu aplikacji, wywołujemy funkcję `USBHcontrolRequest`. Jak już zostało powiedziane, jej zachowanie istotnie zależy od wartości parametru `synch`. Jeżeli ma on wartość zero, to wywołanie jest nieblokujące. W tym przypadku, jeśli automat jest w stanie `CTRL_IDLE`, to wywołujemy funkcję `USBH-submitControlRequest`, aby zainicjować nowe żądanie. Jeśli natomiast automat jest w stanie `CTRL_DONE`, to zmieniamy jego stan na `CTRL_IDLE` oraz ewentualnie za pomocą wskaźnika `length` zwracamy rozmiar przesłanych danych. Za każdym razem zwracamy kod zakończenia zapisany w składowej `errno`.

```
int USBHcontrolRequest(int synch, usb_setup_packet_t const *setup,
                       uint8_t *buffer, uint32_t *length) {
    if (synch == 0) {
        if (Machine.control.state == CTRL_IDLE) {
            USBHsubmitControlRequest(setup, buffer);
        }
        else if (Machine.control.state == CTRL_DONE) {
            Machine.control.state = CTRL_IDLE;
            if (length)
                *length = Machine.control.transferred;
        }
    }
    else {
        uint32_t x;
        x = USBHprotectInterrupt();
        if (Machine.control.state != CTRL_IDLE) {
            USBHunprotectInterrupt(x);
            return USBHLIB_ERROR_BUSY;
        }
        else {
            USBHsubmitControlRequest(setup, buffer);
            while (Machine.control.state != CTRL_DONE) {
                USBHunprotectInterrupt(x);
                Delay(200); /* How long should we wait? */
                x = USBHprotectInterrupt();
            }
            Machine.control.state = CTRL_IDLE;
        }
    }
}
```

```

        if (length)
            *length = Machine.control.transferred;
    }
    USBHunprotectInterrupt(x);
}
return Machine.control(errno);
}

```

Jeżeli parametr `synch` ma wartość niezerową, to wywołanie funkcji `USBHcontrolRequest` jest blokujące. Ponieważ wywołanie blokujące ma miejsce poza kontekstem przerwania USB, wszelkie dostępy do zmiennych globalnych odbywają się przy zablokowanym przerwaniu USB, aby wykluczyć niechciane przeploty modyfikacji takich zmiennych. Nie blokujemy odczytu zmiennych całkowitych, jak np. składowej `errno`, gdyż ich odczyt jest atomowy. Jeśli automat nie jest w stanie `CTRL_IDLE`, to opuszczamy funkcję, informując, że automat jest zajęty – zwracamy wartość `USBHLIB_ERROR_BUSY`. Podobnie jak przy wywołaniu nieblokującym, jeśli automat jest w stanie `CTRL_IDLE`, to inicjujemy nowe żądanie za pomocą funkcji `USBHsubmitControlRequest`. Po zainicjowaniu żądania czekamy w pętli `while` na jego zakończenie. Wewnątrz pętli odblokowujemy na chwilę przerwania, aby dać szansę zadziałać automatowi. Czas ten został wybrany dość arbitralnie. Nic nie stoi na przeszkodzie, aby po eksperymentować z innymi wartościami czasu lub użyć dość dokładnego licznika z modułu timer zamiast funkcji `Delay`. Po opuszczeniu pętli `while` przywracamy automat do stanu `CTRL_IDLE` oraz ewentualnie za pomocą wskaźnika `length` zwracamy rozmiar przesyłanych danych. Następnie odblokowujemy przerwania i opuszczamy funkcję, zwracając kod zakończenia ze składowej `errno`. Funkcja `USBHsubmitControlRequest` inicjuje automat, aby mógł rozpoczęć realizację nowego żądania. Aktualny główny stan automatu zapamiętujemy w składowej `g_prev_state`, aby po zakończeniu realizacji żądania móc do niego powrócić. Jako nowy główny stan automatu ustawiamy stan `HOST_CONTROL_TRANSFER`, co spowoduje uruchomienie podautomatu realizującego transmisję danych sterujących. Żeby rozpocząć działanie tego automatu, inicjujemy jego stan na `CTRL_SETUP`. Inicjujemy też odpowiednio pozostałe pola podstruktury `control`, jak to widać na poniższym wydruku.

```

static void USBHsubmitControlRequest(usb_setup_packet_t const *setup,
                                      uint8_t *buffer) {
    Machine.g_prev_state = Machine.g_state;
    Machine.g_state = HOST_CONTROL_TRANSFER;
    Machine.control.state = CTRL_SETUP;
    Machine.control errno = USBHLIB_IN_PROGRESS;
    Machine.control errno candidate = USBHLIB_SUCCESS;
    Machine.control setup.bmRequestType = setup->bmRequestType;
    Machine.control setup.bRequest = setup->bRequest;
    Machine.control setup.wValue = HTOUSBS(setup->wValue);
    Machine.control setup.wIndex = HTOUSBS(setup->wIndex);
    Machine.control setup.wLength = HTOUSBS(setup->wLength);
    Machine.control timeout = -1;
}

```

```

    Machine.control.buffer = buffer;
    Machine.control.length = setup->wLength;
    Machine.control.transferred = 0;
    Machine.control.error_count = 0;
}

```

Zwróćmy uwagę, że składowa `length` podstruktury `control` dubluje informację przechowywaną w polu `wLength` struktury `setup`. Jest to konieczne, gdyż pole `wLength` może mieć inną kolejność bajtów niż obowiązująca w architekturze, na której działa program.

Na koniec warto jeszcze zobaczyć przykład implementacji jakiejś prostej funkcji inicjującej żądanie, na przykład żądanie `SET_ADDRESS`. Tekst źródłowy wszystkich tych funkcji jest bardzo do siebie podobny. Inicjowanie żądania polega przede wszystkim na właściwym wypełnieniu pól struktury typu `usb_setup_packet_t`, reprezentującej dane wysypane w transakcji `SETUP`, a następnie wywołaniu omówionej powyżej funkcji `USBHcontrolRequest`.

```

int USBHsetDeviceAddress(int synch, uint8_t addr) {
    usb_setup_packet_t setup;
    setup.bmRequestType = HOST_TO_DEVICE | STANDARD_REQUEST |
        DEVICE_RECIPIENT;
    setup.bRequest = SET_ADDRESS;
    setup.wValue = addr;
    setup.wIndex = 0;
    setup.wLength = 0;
    return USBHcontrolRequest(synch, &setup, 0);
}

```

7.2. Obsługa myszy i klawiatury

Obsługę urządzenia klasy HID podzieliłem na dwie części. Pierwsza część implementuje protokół fazy rozruchu. Starałem się, aby była ona dość uniwersalna i ogólna. Dzięki temu może być wykorzystana w innych projektach i de facto jest fragmentem przedstawionej w poprzednim podrozdziale biblioteki kontrolera. Zadaniem drugiej części jest tylko zademonstrowanie działania urządzeń HID. Należy ją traktować raczej jako przykład pokazujący zalecaną kolejność wywołań poszczególnych funkcji i program pozwalający na przetestowanie działania myszy i klawiatury USB.

7.2.1. Protokół fazy rozruchu

`usbh_hid_core.h`
`usbh_hid_core.c`

Protokół fazy rozruchu dla myszy i klawiatury jest zaimplementowany w module `usbh_hid_core`. Plik `usbh_hid_core.h` zawiera niezbędne deklaracje, a w pliku `usbh_hid_core.c` znajduje się właściwa implementacja. Odczyt stanu myszy i klawiatury odbywa się głównie poprzez zmienne globalne. Nie jest to najszczególniejsze

rozwiązań. Udostępnianie przez moduł jakichkolwiek zmiennych niesie ryzyko ich nieuprawnionej modyfikacji. Dostęp do zmiennych modułu powinien odbywać się wyłącznie poprzez wywołania funkcji, które mają wtedy szansę sprawdzać poprawność tych odwołań. Ponadto poważną wadą zastosowanego rozwiązania jest to, że pamiętany jest tylko ostatni stan myszy i klawiatury, więc jeśli nie zostanie on na czas odczytany, to zostanie zgubiony. Jednak w prezentowanym programie nie jest to wielkim problemem, gdyż stan myszy i klawiatury jest czytany i wyświetlany w pętli dostatecznie często. Docelowo należałoby w jakiś sposób kolejkować zdarzenia pochodzące od myszy i klawiatury oraz zaprojektować interfejs odczytu tych zdarzeń. W tym przypadku o zastosowaniu zmiennych globalnych zadecydowała prostota tego rozwiązania i to, że zupełnie wystarcza ono do zademonstrowania działania myszy i klawiatury. Zadanie zaprojektowania i zaimplementowania porządnego interfejsu modułu *usbhid_core* pozostawiam Czytelnikowi jako ćwiczenie. Tymczasem przejdźmy do opisu aktualnej implementacji. W pliku nagłówkowym *usbhid_core.h* zadeklarowane są zmienne przedstawione na poniższym wydruku.

```
extern int new_mouse_data;
extern unsigned mouse_buttons;
extern int mouse_x, mouse_y;
extern int new_keyboard_data;
extern unsigned keyboard_modifiers;
extern uint8_t keyboard_scan_code[KEYBOARD_MAX_PRESSED_KEYS];
```

Zmienne *mouse_x* i *mouse_y* zawierają aktualne położenie kurSORA myszy w pikselach. Trzy najmniej znaczące bity zmiennej *mouse_buttons* zawierają stan aktualnie wciśniętych przycisków myszy. Stałe reprezentujące bity odpowiadające poszczególnym przyciskom są zdefiniowane w pliku *usb_def.h*.

```
#define MOUSE_LEFT_BUTTON      0x01
#define MOUSE_RIGHT_BUTTON     0x02
#define MOUSE_MIDDLE_BUTTON    0x04
```

Tablica *keyboard_scan_code* zawiera kody aktualnie wciśniętych klawiszy. Typowa klawiatura od komputera PC zwraca kody o wartościach od 4 (klawisz A) do 101 (klawisz *Menu*). Osiem najmniej znaczących bitów zmiennej *keyboard_modifiers* zawiera aktualny stan klawiszy modyfikujących. Stałe reprezentujące bity odpowiadające poszczególnym klawiszom modyfikującym są zdefiniowane w pliku *usb_def.h*.

```
#define KEYBOARD_LEFT_CTRL    0x01
#define KEYBOARD_LEFT_SHIFT    0x02
#define KEYBOARD_LEFT_ALT     0x04
#define KEYBOARD_LEFT_GUI      0x08
#define KEYBOARD_RIGHT_CTRL   0x10
#define KEYBOARD_RIGHT_SHIFT  0x20
#define KEYBOARD_RIGHT_ALT    0x40
#define KEYBOARD_RIGHT_GUI    0x80
```

Zmienna *new_mouse_data* informuje, czy stan myszy uległ zmianie. Jest ustawiana na jedynkę po każdej zmianie położenia myszy lub któregoś z jej przycisków.

Aplikacja po odczytaniu nowego stanu myszy powinna wyzerować tę zmienną. Zmienna new_keyboard_data informuje, czy stan klawiatury uległ zmianie. Jest ustawiana na jedynkę po każdym wciśnięciu lub puszczeniu jakiegoś klawisza. Aplikacja po odczytaniu nowego stanu klawiatury powinna wyzerować tę zmienną. Każda zmiana stanu myszy lub klawiatury nadpisuje poprzedni stan, więc jeśli aplikacja nie odczyta tego stanu na czas, to zostanie on zgubiony.

```
int HIDsetMachine(usb_speed_t speed, uint8_t dev_addr,
                   usb_interface_descriptor_t const *if_desc,
                   usb_hid_main_descriptor_t const *hid_desc,
                   usb_endpoint_descriptor_t const *ep_desc,
                   unsigned ep_count);
```

Funkcja HIDsetMachine instaluje automat stanowy obsługujący protokół fazy rozruchu. Korzysta w tym celu z funkcji USBHsetClassMachine. Parametry speed i dev_addr zawierają odpowiednio szybkość i adres urządzenia. Do prawidłowego działania protokołu potrzebne są informacje zawarte w deskryptorach. Parametry if_desc i hid_desc muszą wskazywać odpowiednio na deskryptor interfejsu i deskryptor HID. Parametr ep_desc musi wskazywać na tablicę z deskryptorami punktów końcowych. Parametr ep_count zawiera liczbę deskryptorów punktów końcowych. Funkcja ta zwraca USBHLIB_SUCCESS, gdy zakończyła się sukcesem, albo odpowiedni kod błędu. Wywołanie tej funkcji jest nieblokujące i może być ona wywoływana poza kontekstem obsługi przerwania.

```
int HIDisDeviceReady(void);
```

Bezparametrowa funkcja HIDisDeviceReady zwraca jedynkę, gdy urządzenie klasy HID jest podłączone i aktywne, czyli jest gotowe do komunikacji. W przeciwnym przypadku funkcja ta zwraca zero. Wywołanie tej funkcji jest nieblokujące i może być ona wywoływana poza kontekstem obsługi przerwania.

<i>usbh_hid_req.h</i>
<i>usbh_hid_req.c</i>

W pliku *usbh_hid_req.h* zadeklarowane są funkcje realizujące najczęściej używane żądania HID. Ich implementacja znajduje się w pliku *usbh_hid_req.c*. Wszystkie funkcje obudowują wywołanie funkcji USBHcontrolRequest. Pierwszy ich argument synch ma takie samo znaczenie i jest do niej przekazywany. Wszystkie też zwracają wartość przekazaną przez funkcję USBHcontrolRequest.

```
int HIDgetReportDescriptor(int synch, uint8_t *desc, uint16_t length);
```

Funkcja HIDgetReportDescriptor odczytuje deskryptor raportu. Parametr desc jest wskaźnikiem na bufor, do którego ma być skopiowany ten deskryptor. Parametr length zawiera maksymalną liczbę bajtów, które mają być odebrane. Nie może to być więcej niż rozmiar bufora. Po zakończeniu realizacji żądania funkcja ta sprawdza, czy rozmiar odebranych danych zgadza się z wartością parametru length.

```
int HIDsetIdle(int synch, uint8_t iface, uint8_t report_id,
               uint8_t interval);
```

Funkcja HIDsetIdle wysyła żądanie SET_IDLE. Parametr iface zawiera numer interfejsu, do którego kierowane jest żądanie. Parametr report_id jest numerem raportu, którego dotyczy żądanie – zwykle jest to raport numer 0. Parametr interval określa maksymalny czas między kolejnymi rapportami generowanymi przez urządzenie. Urządzenie generuje nowy raport, gdy użytkownik wcisnął klawisz lub poruszył myszką albo gdy upłynął czas zadany tym parametrem. Jednostką są 4 ms, a wartość zero oznacza czas nieskończony. Przy czym kolejne raporty nie są generowane częściej, niż wynika to z wartości parametru bInterval w deskryptorze punktu końcowego.

```
int HIDsetBootProtocol(int synch, uint8_t iface,
                       usb_hid_protocol_t protocol);
```

Funkcja HIDsetBootProtocol wysyła żądanie SET_PROTOCOL. Parametr iface zawiera numer interfejsu, do którego kierowane jest żądanie. Parametr protocol określa, jaki protokół ma być ustawiony. Wartość HID_BOOT_PROTOCOL (0) oznacza protokół fazy rozruchu. Wartość HID_REPORT_PROTOCOL (1) oznacza protokół opisany w deskryptorze raportu.

```
int HIDsetReport(int synch, uint8_t iface, uint8_t report_id,
                  uint8_t *report, uint16_t length);
```

Funkcja HIDsetReport wysyła do urządzenia raport wyjściowy (za pomocą żądania SET_REPORT). Parametr iface zawiera numer interfejsu, do którego kierowane jest żądanie. Parametr report_id zawiera numer raportu. Parametr report jest wskaźnikiem na bufor, w którym znajduje się raport do wysłania. Parametr length określa rozmiar raportu do wysłania. Po zakończeniu realizacji żądania funkcja ta sprawdza, czy rozmiar wysłanych danych zgadza się z wartością tego parametru.

7.2.2. Program demonstrujący

ex_hid_host.c

Program demonstrujący działanie kontrolera znajduje się w pliku *ex_hid_host.c*. Zaczyna się on od funkcji main, w której najpierw za pomocą funkcji GetBootParams odczytujemy, z jaką częstotliwością ma być taktowany rdzeń (zmienna clk) oraz który nadajnik-odbiornik USB ma być użyty (zmienna phy). Wybór nadajnika-odbiornika USB jednoznacznie determinuje, który układ peryferyjny USB ma być użyty. Następnie konfigurujemy kolejne układy i moduły: diody świecące, przerwania, pętle fazowe, wyświetlacz ciekłokrystaliczny i na końcu interfejs USB. Błąd podczas konfigurowania sprzętu spowoduje, że mikrokontroler zostanie wyzerowany przez funkcję ErrorResetable.

```

int main(void) {
    int clk;
    usb_phy_t phy;
    GetBootParams(&clk, 0, &phy);
    AllPinsDisable();
    LEDconfigure();
    RedLEDon();
    IRQprotectionConfigure();
    ErrorResetable(ClockConfigure(clk), 2);
    ErrorResetable(LCDconfigure(), 3);
    ErrorResetable(USBHconfigure(phy, MIDDLE_IRQ_PRIO), 5);
    for (;;)
        HIDexample();
}

```

Po poprawnym skonfigurowaniu wszystkich układów i modułów wywołujemy w nieskończonej pętli funkcję `HIDexample`, która realizuje pełny cykl obsługi urządzenia od wykrycia jego podłączenia do szyny aż do jego odłączenia lub wystąpienia błędu. Jej tekst źródłowy jest bardzo rozwlekły. Dlatego w dalszym ciągu prezentuję tylko kluczowe fragmenty jej implementacji. W większości przypadków pomijam opis obsługi błędów. Dla każdej funkcji, która może zawieść, sprawdzamy, czy jej wywołanie zakończyło się sukcesem. Jeśli wystąpił błąd, to wypisujemy informację o nim i opuszczamy funkcję `HIDexample`. Pomijam też fragmenty tekstu źródłowego służące do formatowania tekstu i wypisywania go na ekranie wyświetlacza ciekłokrystalicznego. Nie omawiam również szczegółowo pomocniczych funkcji wywoływanych przez funkcję `HIDexample`. Całość można zobaczyć w archiwum z przykładami.

Funkcja `HIDexample` deklaruje następujące zmienne lokalne:

```

usb_speed_t speed;
usb_device_descriptor_t dev_desc;
usb_configuration_descriptor_t cfg_desc;
usb_interface_descriptor_t if_desc;
usb_hid_main_descriptor_t hid_desc;
usb_endpoint_descriptor_t ep_desc[MAX_ENDPOINT_COUNT];
unsigned ep_count;
uint16_t len16;
int res;
uint8_t dev_addr;
char temp[BUFF_LEN];

```

Zaczynamy od aktywnego oczekiwania na podłączenie urządzenia. Oczekивание to realizuje funkcja `USBHopenDevice`.

```
res = USBHopenDevice(&speed, &dev_addr, &dev_desc, SHORT_TIMEOUT);
```

Jesli wywołanie funkcji `USBHopenDevice` zakończy się błędem, to wyświetlamy na środku ekranu napis NO DEVICE. Jeśli wywołanie zakończyło się sukcesem, to w zmiennej `speed` mamy szybkość wykrytego urządzenia, w zmiennej `dev_addr`

jego adres, a struktura `dev_desc` zawiera deskryptor urządzenia i uzyskane informacje możemy wypisać na wyświetlaczu.

Zakładamy, że urządzenie ma tylko jeden deskryptor konfiguracji o numerze 0. Odczytujemy go w sposób blokujący za pomocą funkcji `USBGetConfDescriptor`.

```
res = USBHgetConfDescriptor(1, 0, (uint8_t *)&cfg_desc,  
                           sizeof(cfg_desc));
```

Głównym celem powyższego wywołania jest poznanie całkowitego rozmiaru konfiguracji, znajdującego się w polu deskryptora konfiguracji `wTotalLength`. Znając ten rozmiar, podejmujemy próbę odczytania całej konfiguracji.

```
len16 = min(cfg_desc.wTotalLength, BUFF_LEN);  
res = USBHgetConfDescriptor(1, 0, (uint8_t *)temp, len16);
```

Po odczytaniu konfiguracji wyciągamy z niej potrzebne deskryptory za pomocą pomocniczej funkcji `ParseConfiguration`. Pomijam omawianie szczegółów jej działania, gdyż polega to na zmuśnym i niezbyt ciekawym analizowaniu kolejnych pól konfiguracji.

```
ep_count = MAX_ENDPOINT_COUNT;
res = ParseConfiguration(&cfg_desc, &if_desc, &hid_desc, ep_desc,
                        &ep_count, (uint8_t *)temp, len16);
```

Jeśli powyższe wywołanie zakończyło się sukcesem, to w strukturze `cfg_desc` mamy deskryptor konfiguracji, a w strukturze `if_desc` znajduje się deskryptor interfejsu. Jeśli urządzenie ma więcej niż jeden interfejs, to otrzymujemy deskryptor interfejsu numer 0. Jeśli podłączone urządzenie jest myszą lub klawiaturą, to w strukturze `hid_desc` dostajemy jego deskryptor HID, a w tablicy `ep_desc` deskryptory punktów końcowych. Przed wywołaniem funkcji `ParseConfiguration` zmienna `ep_count` zawiera liczbę elementów w tablicy `ep_desc`, a po powrocie jest w niej liczba rzeczywiście skopiowanych do tej tablicy deskryptorów. Mając dodatkowe informacje o urządzeniu, możemy je wyświetlić.

Jeśli urządzenie oferuje deskryptory tekstowe opisujące producenta, to próbujemy go odczytać, a następnie, jeśli się powiodło, wypisujemy go, co już nie jest pokazane na poniższym wydruku.

Podobnie postępujemy z deskryptorem tekstowym opisującym produkt.

Jeśli podłączone urządzenie jest myszą lub klawiaturą, to przystępujemy do jego skonfigurowania. Aktywujemy jego jedyną konfigurację. Następnie odczytujemy deskryptor raportu, mimo że z niego nie korzystamy. Okazuje się jednak, że niektóre tanie myszy nie działają, jeśli tego nie zrobimy. Potem ustawiamy protokół fazy rozruchu i nieskończony maksymalny czas między kolejnymi rapportami. Ustawienie tego czasu jest opcjonalne dla myszy, więc jeśli w tym przypadku wywołanie HIDsetIdle zwróci wartość USBHLIB_ERROR_STALL, co oznacza, że urządzenie odrzuciło żądanie, to powinniśmy to zignorować. Na koniec instalujemy automat obsługujący protokół fazy rozruchu. Oczywiście za każdym razem sprawdzamy wartość zwróconą w zmiennej res i odpowiednio reagujemy, jeśli któreś wywołanie się nie powiodło, co nie jest uwidocznione na poniższym wydruku, aby nie zmniejszać jego czytelności.

```

if (dev_desc.bDeviceClass == 0 &&
    dev_desc.bDeviceSubClass == 0 &&
    dev_desc.bDeviceProtocol == 0 &&
    if_desc.bInterfaceClass == HUMAN_INTERFACE_DEVICE_CLASS &&
    if_desc.bInterfaceSubClass == BOOT_INTERFACE_SUBCLASS) {
    res = USBHsetConfiguration(1, cfg_desc.bConfigurationValue);
    len16 = min(hid_desc.wDescriptorLength1, BUFF_LEN);
    res = HIDgetReportDescriptor(1, (uint8_t *)temp, len16);
    res = HIDsetBootProtocol(1, if_desc.bInterfaceNumber,
                             HID_BOOT_PROTOCOL);
    res = HIDsetIdle(1, if_desc.bInterfaceNumber, 0, 0);
    res = HIDsetMachine(speed, dev_addr, &if_desc, &hid_desc,
                         ep_desc, ep_count);
    while (HIDisDeviceReady()) {
        LCDrefresh();
        Delay(SHORT_TIMEOUT);
    }
}
else {
    while (USBHisDeviceReady())
        Delay(SHORT_TIMEOUT);
}

```

Po tych wszystkich czynnościami kręcimy się w pętli while, dopóki urządzenie pozostaje podłączone. Pętla zakończy się, gdy funkcja HIDisDeviceReady zwróci zero, co oznacza, że urządzenie zostało odłączone. Wewnątrz pętli wywołujemy funkcję LCDrefresh, która sprawdza, czy zmienił się stan myszy lub klawiatury i ewentualnie wyświetla ten nowy stan. Jeśli podłączone urządzenie nie jest ani myszą, ani klawiaturą, to wywołujemy w pętli funkcję USBHisDeviceReady, która zwróci zero, gdy urządzenie zostało odłączone. Na koniec po wyjściu z pętli while, a przed opuszczeniem funkcji HIDexample należy jeszcze wyczyścić zawartość ekranu za pomocą funkcji LCDclear.

Automat realizujący protokół fazy rozruchu dla HID nie jest specjalnie ciekawy, dlatego pomijam tu omawianie jego implementacji – zawsze można zatrzymać do pliku źródłowego w archiwum. Bardziej pouczającą implementację znacznie ciekawszego automatu zamieszczam w ostatnim rozdziale.

7.2.3. Kompilowanie i testowanie

Archiwum z przykładami zawiera dwie wersje sprzętowe projektu tego kontrolera. Był on testowany na zestawie ZL29ARM z wyświetlaczem graficznym WG12864A oraz na zestawie STM3220G-EVAL, który ma wbudowany wyświetlacz graficzny. Program kontrolera wykorzystuje te wyświetlacze jedynie do wypisywania tekstu. W katalogu `./make` znajdują się podkatalogi, których nazwa rozpoczyna się przedrostkiem `usb7_hid_host` i w których umieszczone są pliki *makefile* umożliwiające skompilowanie projektu. Można go oczywiście dostosować do innego sprzętu, posługując się wskazówkami z rozdziału 2. Plik *makefile* zawiera listę plików źródłowych niezbędnych do skompilowania programu. Jest zatem przydatny również dla tych, którzy nie chcą korzystać bezpośrednio z programu `make`. Wtedy należy pamiętać, aby dołączyć też plik `startup_stm32.c` i właściwą wersję biblioteki STM32.

W przypadku płytka ZL29ARM należy pamiętać o przestawieniu zwór w pozycję kontrolera (patrz też opis tego zestawu w rozdziale 2). Płyty STM3220G-EVAL ma dwa interfejsy USB: CN8 oznaczony też jako USB OTG FS oraz CN9 oznaczony jako USB OTG HS. Wyboru interfejsu dokonuje się za pomocą przycisku *Wakeups*. Jeśli kontroler ma być uruchomiony na interfejsie CN8, to podczas startu programu przycisk *Wakeups* musi być puszczyony. Jeśli kontroler ma być uruchomiony na interfejsie CN9, to należy wcisnąć przycisk *Wakeups* podczas startu programu (np. przytrzymać ten przycisk podczas wciskania przycisku *Reset*). Jeśli wybieramy interfejs CN8, to za pomocą przełącznika BOOT2 wybiera się częstotliwość taktowania rdzenia: 48 lub 120 MHz (patrz też tabela 2.5). Ponieważ interfejs CN9 może pracować z wysoką szybkością, w tym przypadku częstotliwość taktowania jest ustalona na 120 MHz.

Stan kontrolera i informacje o podłączonym urządzeniu prezentowane są za pomocą diod świecących i wyświetlacza. Jeśli urządzenie nie jest podłączone, to świeci czerwona dioda, a na środku wyświetlacza pojawia się napis NO DEVICE. Po wykryciu, że urządzenie zostało podłączone, czerwona dioda gaśnie, a kontroler podejmuje próbę odczytania deskryptorów. Na wyświetlaczu powinniśmy zobaczyć informacje podobne do poniższych:

```
LOW SPEED      15d9:2003  
00 00 00      70 mA  
03 01 01      rem. wakeup  
Unknown USB   Keyboard
```

W pierwszym wierszu po lewej wyświetla się szybkość, z jaką pracuje urządzenie, a po prawej jego VID i PID. W drugim po lewej wyświetlone są szesnastkowo klasa, podklasa i protokół, umieszczone w deskryptorze urządzenia. W trzecim wierszu po lewej są te same pola, ale z deskryptora interfejsu. W drugim i trzecim wierszu po prawej znajdują się atrybuty związane z zasilaniem, odczytane z deskryptora

konfiguracji. Poniżej, w czwartym i ewentualnie piątym wierszu wyświetlane są nazwa producenta i nazwa urządzenia z odpowiednich deskryptów tekstowych, jeśli takie deskrepty są dostępne.

Jeśli kontroler nie może uruchomić urządzenia, to cyklicznie odłącza na chwilę zasilanie portu USB i ponawia próby, aż do skutku. Każde odłączenie zasilania sygnalizuje krótkim (ok. 300 ms) mignięciem czerwonej diody. Pewne błędy wykryte podczas uruchamiania lub działania kontrolera sygnalizowane są trzema sekwencjami mignięć czerwonej diody, po których następuje wyzerowanie mikrokontrolera. Błędy te zebrane są w **tabeli 7.2**. Mają one charakter diagnostyczny. Wystąpienie któregoś z nich najprawdopodobniej świadczy o błędzie w implementacji – okazały się przydatne podczas uruchamiania projektu, a w poprawnie działającym programie nie powinny się już pojawić. Liczba mignień w sekwencji oznacza numer błędu. Jeśli urządzenie jest podłączone i udało się odczytać jego konfigurację, to czerwona dioda gaśnie, a zaczyna migać zielona dioda, z częstotliwością 1 Hz dla urządzenia LS lub FS, a 8 Hz dla urządzenia HS.

Tab. 7.2. Błędy sygnalizowane przez kontroler

Numer błędu	Opis błędu
2	Funkcja <code>ClockConfigure</code> , niepoprawna lub niemożliwa do ustawienia częstotliwość takowania
3	Funkcja <code>LCDconfigure</code> , błąd podczas uruchamiania wyświetlacza, brak wyświetlacza
5	Funkcja <code>USBHconfigure</code> , niepoprawna wartość parametru, błąd podczas konfigurowania układu peryferyjnego USB w trybie kontrolera
8	Układ peryferyjny OTG-FS lub OTG-HS w trybie urządzenia, plik <code>usbh_interrupt.c</code>
9	Błąd kolejności pakietów DATA0 i DATA1 (ang. <i>data toggle error</i>), plik <code>usbh_interrupt.c</code>

Dodatkowe informacje pojawiają się na wyświetlaczu, gdy zostanie podłączone urządzenie HID. Jeśli jest to klawiatura, to w wierszach szóstym i siódmym wyświetlany jest stan klawiszy modyfikujących (*Alt*, *Ctrl*, *GUI*, *Shift*) – mniej więcej zgodnie z ich rozmieszczeniem na klawiaturze. Wciswanie klawiszy *Caps Lock* i *Num Lock* zmienia na klawiaturze stan diod świecących związanych z tymi klawiszami. W wierszu ósmym pojawiają się nazwy aktualnie wcisniętych zwykłych klawiszy, w kolejności, w jakiej zostały odebrane w raporcie przysłanym przez klawiaturę. Wyświetlane są nazwy, a nie kody klawiszy, dla zwiększenia czytelności. Program kontrolera nie dokonuje konwersji kodu klawisza na znak, który powinien zostać rozpoznany po jego wcisnięciu. W większości przypadków jako nazwę klawisza wybrałem symbol (lub jeden z symboli) znajdujący się na tym klawiszku w klawiaturze o układzie amerykańskim. Wszystko to ma na celu ułatwienie testowania. Raport klawiatury umożliwia sygnalizowanie jednoczesnego wcisnięcia do sześciu klawiszy. Okazuje się, że klawiatury zachowują się dość różnie. Na przykład w posiadanym przeze mnie modelu wcisnięcie klawiszy QWERTY powoduje zasygnalizowanie tylko czterech pierwszych, czyli QWER. Natomiast kombinacja QWERUI sygnalizowana jest w całości.

Jeśli podłączone urządzenie jest myszą, to w wierszu siódmym wyświetlany jest stan jej przycisków, a w wierszu ósmym położenie myszy w pikselach względem położenia początkowego, jakie było podczas uruchamiania urządzenia.

Mikrokontrolery mają zwykle mało pamięci i jeśli muszą gromadzić duże ilości danych, to trzeba użyć pamięci zewnętrznej, np. karty SD. Mając do dyspozycji interfejs kontrolera USB, dość atrakcyjnym rozwiązaniem wydaje się podłączenie w tym celu pamięci USB Flash (ang. *pendrive*). Pojemności takich pamięci sięgają już obecnie wielu gigabajtów. Ponadto użycie pamięci USB Flash bardzo upraszcza przenoszenie danych między mikrokontrolerem a komputerem osobistym. Dlatego w ostatnim rozdziale przedstawiam projekt kontrolera pamięci masowej USB.

8.1. System plików

Pamięci USB Flash na ogół korzystają z systemu plików FAT. Jest to bardzo stary system plików, opracowany na początku lat 80. ubiegłego wieku na potrzeby systemu operacyjnego DOS. Jego zaletami są: prostota, jawność specyfikacji, dostępność darmowych implementacji, obsługa przez wszystkie popularne systemy operacyjne. Poważną wadą, zwłaszcza w przypadku systemów wbudowanych, jest brak odporności na niespodziewane wyłączenie zasilania, co się przydarza w systemach wbudowanych i może doprowadzić do niespójności informacji zapisanej na dysku, a w konsekwencji do poważnego uszkodzenia systemu plików i utraty danych. Mimo tej wady przedstawiona w tym rozdziale aplikacja korzysta z systemu plików FAT. O takim wyborze zadecydowała przede wszystkim prostota takiego rozwiązania. Ponieważ aplikacja ma strukturę modułową i warstwową, nie jest trudno podmienić fragment obsługujący system plików na inny, jeśli zajdzie taka potrzeba.

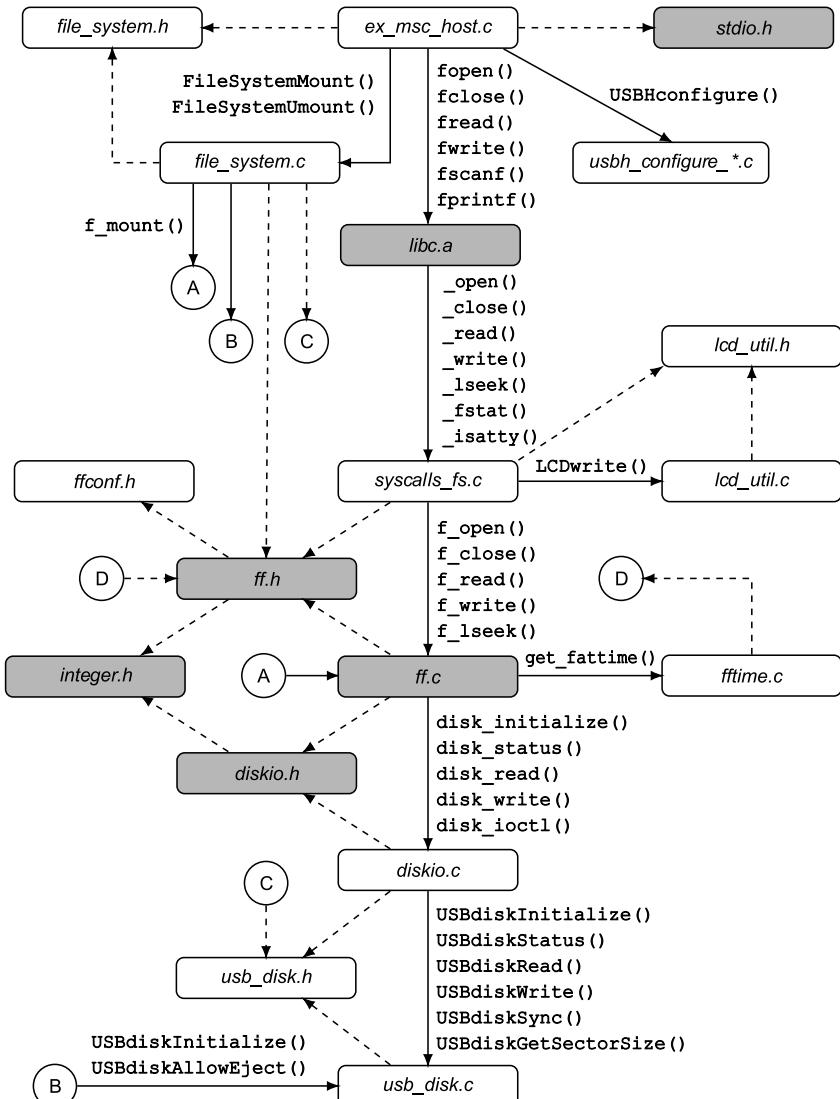
8.1.1. Struktura aplikacji

Żeby tworzenie aplikacji korzystającej z zewnętrznej pamięci było proste, a jej tekst źródłowy łatwo dawał się przenosić na inne platformy sprzętowe, należy zastosować uniwersalny wysokopoziomowy interfejs do obsługi plików. Gdy piszemy aplikację w języku C, to naturalnym wyborem staje się interfejs dostarczany przez standardową bibliotekę tego języka zadeklarowany w pliku nagłówkowym *stdio.h*. Interfejs ten obejmuje funkcje *fopen*, *fclose*, *fread*, *fwrite*, *fscanf*, *fprintf*, a także *printf*, *putchar* itp. Jest on niezależny od zastosowanego systemu plików i od fizycznego medium.

Schemat implementacji systemu plików w przykładowym kontrolerze pamięci USB Flash pokazany jest na **rysunku 8.1**. Pliki biblioteczne, które wykorzystuje się bez żadnej modyfikacji, umieszczone są na szarym tle. Pozostałe pliki, które trzeba zaimplementować albo dostosować do konkretnej aplikacji, znajdują się na białym tle. Jak widać, implementacja podzielona jest na kilka warstw. Zastosowana standardowa biblioteka języka C, czyli biblioteka Newlib, nie może oczywiście bezpośrednio implementować funkcji z pliku nagłówkowego *stdio.h*, gdyż nie ma dostępu do warstwy sprzętu. W zamian za to tłumaczy wywołania wysokopoziomowych funkcji plikowych na wywołania prostszych funkcji: *_open*, *_close*, *_read*, *_write*, *_lseek*, *_fstat* i *_isatty*. Semantyka tych drugich funkcji jest taka sama jak odpowiednich funkcji w systemach uniksowych, czyli funkcji o analogicznych nazwach, ale bez podkreślenia na początku.

W pliku nagłówkowym *stdio.h* zdefiniowano trzy strumienie (pseudopliki), które służą zwykle do komunikacji z konsolą. Są to *stdin* – standardowe wejście, *stdout*

– standardowe wyjście i `stderr` – standardowe wyjście błędów. Funkcje do obsługi plików działają również na tych strumieniach, z tą różnicą, że przed użyciem nie trzeba ich otwierać, a przed zakończeniem programu nie trzeba ich zamknić. W pliku `syscalls_fs.c` następuje rozdzielenie wywołań dotyczących tych trzech standar-dowych strumieni i wywołań rzeczywistych plików. Standardowe wejście nie jest obsługiwane przez przykładową aplikację. Dane kierowane na standardowe wyjście i standardowe wyjście błędów są wysyłane na wyświetlacz ciekłokrystaliczny za pomocą ogólnego interfejsu tekstowego, który jest zadeklarowany w pliku `lcd_util.h`, a zaimplementowany w pliku `lcd_util.c`. Następnie wywoływane są niskopoziom-



Rys. 8.1. Implementacja systemu plików

mowe funkcje obsługujące wybrany typ wyświetlacza zaimplementowane w module *lcd* – patrz rozdział 2.4.3.

Natomiast wywołania dotyczące rzeczywistych plików zostają w pliku *syscalls_fs.c* przetłumaczone na wywołania biblioteki obsługującej wybrany system plików. W naszym przypadku jest to biblioteka FatFs, która implementuje system plików FAT i która udostępnia w tym celu funkcje *f_open*, *f_close*, *f_read*, *f_write* i *f_lseek*. Są to funkcje o podobnej semantyce jak odpowiednie uniksowe funkcje bez przedrostka *f* i podkreślenia, lecz nieco prostsze, np. funkcja *f_lseek* nie ma parametru pozwalającego wybrać, od jakiej pozycji liczone jest przesunięcie – zawsze liczone jest od początku pliku.

Główna część implementacji biblioteki FatFs znajduje się w pliku *ff.c*, który włącza bezpośrednio i pośrednio kilka plików nagłówkowych. Plik nagłówkowy *ff.h* zawiera deklarację interfejsu biblioteki. W pliku nagłówkowym *integer.h* znajdują się definicje typów całkowitoliczbowych, z których korzysta biblioteka. W pliku nagłówkowym *ffconf.h* definiuje się parametry, z którymi biblioteka zostanie skompilowana. Plik ten jest dostarczany wraz z biblioteką, ale jego zawartość trzeba dopasować do konkretnego jej użycia. Plik nagłówkowy *diskio.h* deklaruje interfejs łączący bibliotekę z warstwą obsługującą bezpośrednio sprzęt.

Biblioteka FatFs, podobnie jak Newlib, również nie odwołuje się bezpośrednio do sprzętu. Tłumaczy ona jedynie odwołania do plików na odwołania do fizycznych sektorów dysku. Fizyczne odwołania do dysku są realizowane za pomocą funkcji *disk_initialize*, *disk_status*, *disk_read*, *disk_write* i *disk_ioctl*. Funkcje te należy zaimplementować w pliku *diskio.c*, który rozdziela wywołania do poszczególnych dysków fizycznych. Biblioteka FatFs może obsługiwać do 10 dysków fizycznych, które mogą się różnić sposobem dostępu czy interfejsem (np. z kartą pamięci SD komunikujemy się na ogół za pomocą interfejsu SPI). Jeśli potrzebujemy obsługiwać tylko jeden dysk, to plik ten jest w zasadzie zbędny, ale został zachowany, aby było jasne, jak rozbudować przykład kontrolera o obsługę kolejnych dysków. W pliku *diskio.c* następuje tłumaczenie fizycznych odwołań do dysku na odwołania do konkretnego typu dysku. W naszym przypadku implementacja obsługi pamięci USB Flash znajduje się w pliku *usb_disk.c*, który udostępnia funkcje *USBdiskInitialize*, *USBdiskStatus*, *USBdiskRead*, *USBdiskWrite*, *USBdiskSync* i *USBdiskGetSectorSize*. Ich nagłówki są zadeklarowane w pliku nagłówkowym *usb_disk.h*.

Gdy biblioteka FatFs zapisuje lub modyfikuje plik, musi uaktualnić na dysku wpis w odpowiednim katalogu, a w szczególności musi zapisać również czas jego ostatniej modyfikacji. W tym celu aktualny czas zwracany jest przez funkcję *get_fattime* zaimplementowaną w pliku *fftime.c*.

Żeby rozpocząć korzystanie z pamięci USB Flash, trzeba najpierw zamontować system plików, czyli zainicjować bibliotekę FatFs za pomocą funkcji *FileSystemMount*. Po zakończeniu korzystania z pamięci trzeba odmontować system plików, czyli zwolnić wszelkie zasoby za pomocą funkcji *FileSystemUnmount*. Obie te funkcje zadeklarowane są w pliku nagłówkowym *file_system.h*, a zaimplementowane w pliku *file_system.c*. Korzystają one z funkcji *f_mount* biblioteki FatFs oraz funkcji *USBdiskInitialize* i *USBdiskAllowEject* z modułu *usb_disk.c*.

Przykładowa aplikacja znajduje się w pliku *ex_msc_host.c*. Przed zamontowaniem systemu plików musimy skonfigurować interfejs USB w trybie kontrolera za pomocą funkcji *USBHConfigure*. Implementacja tej funkcji zależy od typu mikrokontrolera i znajduje się w odpowiednim pliku *usbh_configure_*.c*. Gwiazdka w nazwie oznacza tu, że w archiwum jest kilka plików, których nazwa pasuje do tego wzorca i które przeznaczone są dla poszczególnych modeli mikrokontrolerów.

Jak wynika z powyższego opisu, każde odwołanie przechodzi przez kilka etapów tłumaczenia. Wydaje się to mało efektywne, ale pozwala na bardzo elastyczne podmienianie systemu plików czy fizycznego medium bez konieczności modyfikowania tekstu źródłowego aplikacji i większości modułów. Zmiana systemu plików wymaga zastosowania innej biblioteki oraz zmiany plików *syscalls_fs.c* oraz *diskio.c* sklejających ją z sąsiednimi modułami. Zmiana lub dodanie nowego medium wymaga dodania implementacji tego medium oraz modyfikacji pliku *diskio.c*.

Tłumaczenie wywołań większości funkcji jest bardzo proste. Jedyny problem sprawiają funkcje służące do otwierania pliku. Poniżej przedstawiam sygnatury trzech funkcji, z których korzystamy w coraz niższych warstwach implementacji systemu plików. Mimo że wydają się one podobne, to subtelne różnice w znaczeniu poszczególnych parametrów sprawiają sporo kłopotów.

```
FILE *fopen(const char *path, const char *mode);
```

Funkcja *fopen* z biblioteki standardowej języka C otwiera plik, którego nazwa wskazana jest za pomocą parametru *path*. Parametr *mode* określa sposób dostępu do pliku. Funkcja ta zwraca wskaźnik do strumienia lub wskaźnik zerowy (*NULL*), jeśli próba otwarcia pliku się nie powiodła.

```
int _open(const char *path, unsigned flags, ...);
```

Funkcja *_open* otwiera plik, którego nazwa wskazana jest za pomocą parametru *path*. Parametr *flags* określa sposób dostępu do pliku. Funkcja ta ma semantykę zgodną z semantyką funkcji *open* w systemach uniksowych. Może ona być wywołana jeszcze z trzecim parametrem określającym prawa dostępu i atrybuty pliku. Nie korzystamy z tego, gdyż nasza aplikacja i system plików nie obsługują ich. Funkcja ta zwraca deskryptor pliku (uchwyt do pliku), będący małą liczbą całkowitą, albo wartość ujemną, jeśli próba otwarcia pliku się nie powiodła.

```
FRESULT f_open(FIL *fp, const TCHAR *path, BYTE mode);
```

Funkcja *f_open* z biblioteki FatFs otwiera plik, którego nazwa wskazana jest za pomocą parametru *path*. Parametr *mode* określa sposób dostępu do pliku. Za pomocą parametru *fp* przekazuje się wskaźnik do struktury typu *FIL* opisującej plik. Struktura ta odgrywa analogiczną rolę do struktury *FILE* w standardowej bibliotece języka C. Jeśli otwarcie powiodło się, to funkcja zwraca wartość *FR_OK*. W przeciwnym przypadku zwraca właściwy kod błędu.

Parametr *mode* funkcji *fopen* jest napisem, natomiast parametr *flags* funkcji *_open* jest alternatywą znaczników. **Tabela 8.1** opisuje, jak biblioteka Newlib tłumaczy

wartości tych parametrów. Opisy pochodzą z [1]. Natomiast naszym zadaniem jest przetłumaczenie wartości parametru `flags` funkcji `_open` na odpowiadającą mu wartość parametru `mode` funkcji `f_open`. Odpowiednie konwersje zamieszczone są w **tabeli 8.2**. Aby łatwiej było zrozumieć, co dzieje się w tekście źródłowym, zamieszczone są również szesnastkowe wartości parametrów.

Tab. 8.1. Konwersja parametrów funkcji otwierających plik

Funkcja <code>fopen</code>	Funkcja <code>_open</code>	Opis
r	O_RDONLY	Otwórz istniejący plik do czytania
w	O_WRONLY O_CREAT O_TRUNC	Otwórz plik do pisania. Skasuj poprzednią zawartość, jeśli plik już istnieje
a	O_WRONLY O_CREAT O_APPEND	Otwórz istniejący lub utwórz nowy plik do dopisywania na jego końcu ¹⁾
r+	O_RDWR	Otwórz istniejący plik do czytania i pisania
w+	O_RDWR O_CREAT O_TRUNC	Utwórz plik do czytania i pisania. Skasuj poprzednią zawartość, jeśli plik już istnieje
a+	O_RDWR O_CREAT O_APPEND	Otwórz istniejący lub utwórz nowy plik do czytania i dopisywania na jego końcu ¹⁾
b	O_BINARY	Otwórz plik jako binarny ²⁾

¹⁾ Biblioteka *Newlib* w celu przesunięcia bieżącej pozycji w pliku, oprócz wywołania funkcji `_open`, wywołuje dodatkowo funkcję `_lseek`.

²⁾ Może wystąpić w kombinacji łącznie z poprzednimi parametrami, np. napis `rb` odpowiada wartości `O_RDONLY | O_BINARY`.

Tab. 8.2. Konwersja parametrów funkcji otwierających plik

Funkcja <code>_open</code>	Wartość	Funkcja <code>f_open</code>	Wartość	Opis
O_RDONLY	0x0000	FA_READ	0x01	Otwórz plik tylko do czytania
O_WRONLY	0x0001	FA_WRITE	0x02	Otwórz plik tylko do pisania
O_RDWR	0x0002	FA_READ FA_WRITE	0x03	Otwórz plik do czytania i pisania
	0x0000	FA_OPEN_EXISTING	0x00	Otwierany plik musi istnieć
O_CREAT	0x0200	FA_OPEN_ALWAYS	0x10	Jeśli otwierany plik nie istnieje, zostanie utworzony nowy plik
O_CREAT O_TRUNC	0x0600	FA_CREATE_ALWAYS	0x08	Jeśli otwierany plik istnieje, jego zawartość zostanie skasowana
O_CREAT O_EXCL	0x0C00	FA_CREATE_NEW	0x04	Jeśli otwierany plik istnieje, jego otwarcie się nie powiedzie
O_APPEND	0x0008			Nie ma odpowiednika ³⁾
O_BINARY	0x10000			Nie ma odpowiednika ⁴⁾

³⁾ Po wywołaniu funkcji `f_open` należy przesunąć wskaźnik zapisu na koniec pliku za pomocą funkcji `f_lseek`.

⁴⁾ Wszystkie pliki są otwierane jako binarne.

8.1.2. Implementacja

Przyjrzyjmy się teraz dokładniej implementacji poszczególnych modułów z rysunku 8.1.

`syscalls_fs.c`

W pliku `syscalls_fs.c` należy zaimplementować funkcje systemowe wymagane przez bibliotekę *Newlib*. Należ do nich, opisana w rozdziale 2.4.10, funkcja `_sbrk`

potrzebna do zarządzania pamięcią na stercie. Poniżej opisuję pozostałe zaimplementowane w tym pliku funkcje, które są przeznaczone do obsługi systemu plików. Korzystamy z biblioteki FatFs, która używa arytmetyki bez znaku, co na maszynach 32-bitowych (a takie są mikrokontrolery STM32) ogranicza maksymalny rozmiar pliku. Plik musi być krótszy niż 4 GiB. Ponieważ w niektórych miejscach, np. jako wartość zwracaną przez funkcję, zmuszeni jesteśmy stosować typ całkowity ze znakiem (`long`), maksymalny rozmiar pliku, który możemy bezpiecznie obsługiwać bez dodatkowego narzutu obliczeniowego, wynosi 2 GiB minus jeden bajt. Nie wydaje się to jednak istotnym ograniczeniem w systemach wbudowanych, przynajmniej na razie.

W celu identyfikacji pliku funkcje wywoływanne przez bibliotekę Newlib posługują się deskryptorami, które są małymi nieujemnymi liczbami całkowitymi, a funkcje biblioteki FatFs korzystają ze struktur typu `FIL`. Musimy zapewnić odwzorowanie pomiędzy tymi dwoma sposobami identyfikowania plików. Tradycyjnie deskryptory 0, 1 i 2 oznaczają odpowiednio standardowe wejście, standardowe wyjście i standardowe wyjście błędów. Deskryptory te nie będą odwzorowywane w struktury typu `FIL`. Odwzorowanie realizujemy za pomocą globalnej tablicy `fd_table` struktur typu `FIL`. Indeks w tej tablicy jest deskryptorem pliku pomniejszonym o wartość 3. Dodatkowo deklarujemy globalną tablicę `fd_used`, w której zaznaczamy, czy odpowiedni deskryptor i struktura zostały przydzielone. Korzystamy z tego, że początkowo (zapewnia nam to procedura startowa w pliku `startup_stm32.c` – patrz rozdział 2.4.6) tablica ta jest wyzerowana. W aktualnej implementacji stała `MAX_OPEN_FILES` ma wartość 4, aby nie zajmować niepotrzebnie zbyt wiele pamięci, ale nic nie stoi na przeszkodzie, aby zwiększyć tę wartość, gdy aplikacja będzie potrzebowała otworzyć jednocześnie więcej plików.

```
static FIL fd_table[MAX_OPEN_FILES];
static char fd_used[MAX_OPEN_FILES];
```

W funkcji `_open` przed próbą otwarcia pliku najpierw musimy znaleźć wolny deskryptor. Jeśli wszystkie deskryptory plików są zajęte, ustawiamy globalną zmienną `errno` i kończymy funkcję, zwracając wartość ujemną oznaczającą błąd. Znalazłyśmy wolny deskryptor, musimy skonwertować wartość parametru `flags` na wartość parametru `mode_flags`, który zostanie przekazany do funkcji `f_open`. Najpierw korzystamy z małego triku polegającego na tym, że w obu tych parametrach dwa najmłodsze bity decydują, czy plik ma być otwarty do odczytu lub zapisu, czy jednocześnie do odczytu i zapisu. Przy czym stałe `O_RDONLY`, `O_WRONLY` i `O_RDWR` mają odpowiednio kolejne wartości 0, 1 i 2. Natomiast stałe `FA_READ` i `FA_WRITE` mają wartości 1 i 2. Zatem, aby otrzymać właściwą wartość parametru `mode_flags`, wystarczy zwiększyć wartość parametru `flags` o jeden i ograniczyć do dwóch najmłodszych bitów. Następnie, zgodnie z tabelą 8.2, konwertujemy parametry zawiązane ze sposobem otwierania istniejącego pliku i tworzenia nowego pliku, czyli konwertujemy parametry `O_CREAT`, `O_TRUNC` i `O_EXCL` na parametry `FA_CREATE_ALWAYS`, `FA_CREATE_NEW`, `FA_OPEN_ALWAYS` i `FA_OPEN_EXISTING`. Parametr `O_APPEND` ignorujemy, gdyż w tym przypadku biblioteka Newlib po otwarciu pliku i tak wywoła funkcję `_lseek` w celu ustalenia wskaźnika odczytu-zapisu na końcu pliku. Ignorujemy

również parametr `O_BINARY`, gdyż biblioteka FatFs nie rozróżnia plików tekstowych i binarnych – wszystkie pliki są otwierane jako binarne.

```
int _open(const char *path, unsigned flags, ...) {
    int fd;
    BYTE mode_flags;
    for (fd = 0; fd < MAX_OPEN_FILES; ++fd)
        if (fd_used[fd] == 0)
            break;
    if (fd >= MAX_OPEN_FILES) {
        errno = ENFILE;
        return -1;
    }
    else {
        mode_flags = (flags + 1) & 3;
        if (flags & O_CREAT) {
            if (flags & O_TRUNC) {
                mode_flags |= FA_CREATE_ALWAYS;
            }
            else if (flags & O_EXCL) {
                mode_flags |= FA_CREATE_NEW;
            }
            else {
                mode_flags |= FA_OPEN_ALWAYS;
            }
        }
        else {
            mode_flags |= FA_OPEN_EXISTING;
        }
        memset(&fd_table[fd], 0, sizeof(FIL));
        if (FR_OK == f_open(&fd_table[fd], path, mode_flags)) {
            fd_used[fd] = 1;
            errno = 0;
            return fd + 3;
        }
        else {
            errno = EIO;
            return -1;
        }
    }
}
```

Gdy mamy już ustalony sposób otwierania pliku, zerujemy znalezioną wolną strukturę typu `FIL` i próbujemy otworzyć plik za pomocą funkcji `f_open`. Jeśli otwarcie się powiodło, to zapisujemy jedynkę w odpowiednie miejsce tabeli `fd_used`, aby zaznaczyć, że deskryptor jest zajęty. Następnie zerujemy globalną zmienną `errno`, gdyż wywołanie zakończyło się bez błędu. Zwracaną wartość zwiększamy o trzy, aby uwzględnić deskryptory standardowego wejścia, standardowego wyjścia i standardowego wyjścia błędów. Jeśli otwarcie się nie powiodło, to ustawiamy kod błędu w zmiennej `errno` i zwracamy wartość ujemną, co sygnalizuje, że wywołanie zakończyło się błędem. W celu uproszczenia implementacji niezależnie od przyczyny błędu zawsze ustawiamy kod błędu `EIO`, oznaczający ogólny błąd operacji wejścia-wyjścia. Podobnie postępujemy w pozostałych funkcjach operujących na plikach. Być może docelowo należałyby uzależnić ustawiany kod błędu od wartości zwróconej przez funkcję biblioteki `FatFs`.

W funkcji `_close` najpierw sprawdzamy poprawność przekazanego deskryptora pliku `fd`. Jeśli jest poprawny, to próbujemy zamknąć plik za pomocą funkcji `f_close`. Niezależnie od tego, czy jej wywołanie zakończyło się powodzeniem, zwalniamy deskryptor. W przypadku niepowodzenia zamknięcia pliku na ogół nic lepszego zrobić się nie da. Funkcja ta zwraca zero, gdy wszystko przebiegło poprawnie, a wartość ujemną, gdy wystąpił błąd. W obu przypadkach odpowiednio ustawiamy też globalną zmienną `errno`.

```
int _close(int fd) {
    fd -= 3;
    if (fd < 0 || fd >= MAX_OPEN_FILES || fd_used[fd] == 0) {
        errno = EBADF;
        return -1;
    }
    else {
        if (FR_OK == f_close(&fd_table[fd])) {
            fd_used[fd] = 0;
            errno = 0;
            return 0;
        }
        else {
            fd_used[fd] = 0;
            errno = EIO;
            return -1;
        }
    }
}
```

Zadaniem funkcji `_read` są sprawdzenie poprawności przekazanego jej deskryptora pliku `fd` i przekierowanie wywołania do funkcji `f_read`, a następnie sprawdzenie poprawności jej zakończenia. Parametr `ptr` jest wskaźnikiem na bufor, do którego mają zostać skopiowane odczytane dane. Parametr `len` zawiera rozmiar tego bufora.

Jeśli udało się odczytać jakieś dane, to zwracamy ich rozmiar. W przypadku błędu ustawiamy odpowiednio zmienną globalną `errno` i zwracamy wartość ujemną.

```
long _read(int fd, char *ptr, long len) {
    if (ptr == 0 || len < 0) {
        errno = EINVAL;
        return -1;
    }
    fd -= 3;
    if (fd < 0 || fd >= MAX_OPEN_FILES || fd_used[fd] == 0) {
        errno = EBADF;
        return -1;
    }
    else {
        UINT br;
        if (FR_OK == f_read(&fd_table[fd], ptr, len, &br)) {
            errno = 0;
            return br;
        }
        else {
            errno = EIO;
            return -1;
        }
    }
}
```

Funkcja `_write` sprawdza poprawność przekazanego jej deskryptora pliku `fd`. Jeśli jest to deskryptor standardowego wyjścia lub standardowego wyjścia błędów, to przekazuje sterowanie do funkcji `LCDwriteLenWrap`, która wypisuje tekst na ekranie wyświetlacza ciekłokrystalicznego. Jeśli jest to deskryptor otwartego pliku, to przekazuje sterowanie do funkcji `f_write`. W pozostałych przypadkach sygnalizujemy błąd. Parametr `ptr` jest wskaźnikiem na dane. Parametr `len` zawiera rozmiar danych. Jeśli wszystko przebiegło poprawnie, zwracamy liczbę bajtów wypisanych na ekran lub zapisanych do pliku.

```
long _write(int fd, char const *ptr, long len) {
    if (ptr == 0 || len < 0) {
        errno = EINVAL;
        return -1;
    }
    else if (fd == 1 || fd == 2) {
        LCDwriteLenWrap(ptr, len);
        return len;
    }
}
```

```

fd -= 3;
if (fd < 0 || fd >= MAX_OPEN_FILES || fd_used[fd] == 0) {
    errno = EBADF;
    return -1;
}
else {
    UINT bw;
    if (FR_OK == f_write(&fd_table[fd], ptr, len, &bw)) {
        errno = 0;
        return bw;
    }
    else {
        errno = EIO;
        return -1;
    }
}
}

```

Z każdym plikiem związany jest wskaźnik odczytu-zapisu. Wyznacza on miejsce, od którego rozpocznie się następna operacja odczytu lub zapisu na tym pliku. Po każdej takiej operacji wskaźnik ten jest przesuwany o liczbę odczytanych lub zapisanych bajtów. Funkcja _lseek pozwala przesunąć ten wskaźnik bez odczytywania lub zapisywania czegokolwiek. Parametr `fd` jest deskryptorem pliku, którego dotyczy wywołanie. Parametr `offset` określa, o ile bajtów przesunąć wskaźnik. Przesunięcie może mieć wartość zarówno dodatnią, jak i ujemną. Parametr `whence` określa pozycję, od której ma być liczone przesunięcie. Przyjmuje on trzy wartości:

- `SEEK_CUR` oznacza, że przesunięcie jest liczone od bieżącej pozycji wskaźnika w pliku;
- `SEEK_END` oznacza, że przesunięcie jest liczone od końca pliku;
- `SEEK_SET` oznacza, że przesunięcie jest liczone od początku pliku.

```

long _lseek(int fd, long offset, int whence) {
    fd -= 3;
    if (fd < 0 || fd >= MAX_OPEN_FILES || fd_used[fd] == 0) {
        errno = EBADF;
        return -1;
    }
    else if (whence == SEEK_CUR) {
        offset += (long)f_tell(&fd_table[fd]);
    }
    else if (whence == SEEK_END) {
        offset += (long)f_size(&fd_table[fd]);
    }
}

```

```

        else if (whence != SEEK_SET) {
            errno = EINVAL;
            return -1;
        }
        if (offset < 0) {
            errno = EINVAL;
            return -1;
        }
        else if (FR_OK == f_lseek(&fd_table[fd], offset)) {
            errno = 0;
            return f_tell(&fd_table[fd]);
        }
        else {
            errno = EIO;
            return -1;
        }
    }
}

```

W celu wykonania właściwego przesunięcia wskaźnika odczytu-zapisu wywołujemy funkcję `f_lseek`, która wymaga jednak, aby podane jej przesunięcie było wyliczone względem początku pliku. Dlatego należy je przeliczyć, do czego przydaje się funkcja `f_tell` podająca bieżącą pozycję wskaźnika odczytu-zapisu i funkcja `f_size` zwracająca rozmiar pliku. Przy czym w aktualnej implementacji biblioteki FatFs są to makra operujące na strukturze typu `FIL`. Funkcja `_lseek` sprawdza poprawność przekazanego jej deskryptora pliku. Sprawdza też, czy nie próbujemy przesunąć wskaźnika odczytu-zapisu przed początek pliku. Przy poprawnym zaakończeniu funkcja `_lseek` zwraca uaktualnioną pozycję wskaźnika. Jeśli wystąpił błąd, zwraca wartość ujemną. W obu przypadkach odpowiednio ustawia zmienną globalną `errno`.

Funkcja `_fstat` dostarcza informacji o pliku identyfikowanym przez deskryptor podany jako jej pierwszy parametr `fd`. Parametr `st` jest wskaźnikiem na strukturę typu `stat`, która opisuje własności pliku i którą należy wypełnić. Funkcja powinna zwrócić zero, gdy zakończyła się powodzeniem, a wartość ujemną w przypadku błędu, czyli gdy parametr `st` ma wartość zero lub podano niepoprawny deskryptor pliku `fd`. Odpowiednio powinna też zostać ustawiona zmienna globalna `errno`. Jeśli zapytanie dotyczy standardowego wejścia, standardowego wyjścia lub standardowego wyjścia błędów, zerujemy całą strukturę wskazywaną przez parametr `st` i ustawiamy w jej składowej `st_mode` wartość `S_IFCHR`, co oznacza, że jest to deskryptor urządzenia znakowego. Ponadto ustawiamy uniksowe prawa dostępu do pliku `rw-rw-rw-`, czyli wartość `0666`. Biblioteka Newlib nie robi jednak żadnego użytku z tych praw. Ponieważ nie ma łatwego sposobu wyciągnięcia z biblioteki FatFs wszystkich potrzebnych informacji do wypełnienia struktury typu `stat`, w przypadku, gdy zapytanie dotyczy otwartego pliku, sygnalizujemy błąd.

```
int _fstat(int fd, struct stat *st) {
    if (st == 0) {
        errno = EINVAL;
        return -1;
    }
    else if (fd >= 0 && fd <= 2) {
        memset(st, 0, sizeof(struct stat));
        st->st_mode = S_IFCHR | 0666;
        errno = 0;
        return 0;
    }
    fd -= 3;
    if (fd < 0 || fd >= MAX_OPEN_FILES || fd_used[fd] == 0) {
        errno = EBADF;
        return -1;
    }
    else {
        errno = EIO;
        return -1;
    }
}
```

Funkcja `_isatty` jest wywoływana przez bibliotekę Newlib w celu sprawdzenia, czy podany deskryptor pliku `fd` odpowiada terminalowi. Jeśli jest to deskryptor standardowego wejścia, standardowego wyjścia lub standardowego wyjścia błędów, to zwracamy jedynkę, a w przeciwnym przypadku zero.

```
int _isatty(int fd) {
    return fd >= 0 && fd <= 2;
}
```

diskio.h
diskio.c

Zadaniem modułu `diskio` jest rozdzielenie niskopoziomowych operacji dyskowych, czyli operacji realizowanych na poziomie sektorów, do właściwych modułów realizujących te operacje dla poszczególnych rodzajów dysków. Plik nagłówkowy `diskio.h` jest dostarczany wraz z biblioteką FatFs i zawiera deklaracje funkcji, które wywołuje ta biblioteka, oraz definicje typów, z których te funkcje korzystają. Wraz z biblioteką FatFs dostarczany jest też plik `diskio.c` ze szkieletem implementacji, którą musimy uzupełnić. Poniżej prezentuję przykład, co powinno się znaleźć w pliku `diskio.c`. Przykład ten obsługuje tylko jeden dysk, mianowicie dysk USB, ale łatwo jest go rozbudować o obsługę kolejnych dysków. Pierwszy parametr każdej funkcji, czyli `drv`, określa numer dysku, do którego chce się odwołać biblioteka FatFs. Numery dysków należą do przedziału 0...9. Na podstawie numeru dysku

każda funkcja przekierowuje wywołanie do właściwej funkcji, która ma zrealizować żądaną operację. Następnie sprawdza wynik wykonania tej operacji i zwraca stosowną wartość.

Zadaniem funkcji `disk_initialize` jest zainicjowanie dysku, aby był gotowy do wykonywania kolejnych operacji. Jeśli wywołanie dotyczy dysku USB, przekierujemy je do funkcji `USBdiskInitialize`. Sukces sygnalizujemy, zwracając zero, a błąd przez kombinację wartości oznaczających, że nie ma dysku (`STA_NODISK`) i nie został on zainicjowany (`STA_NOINIT`).

```
DSTATUS disk_initialize(BYTE drv) {
    if (drv == USB_DRIVE_NUMEBR && USBdiskInitialize() == 0)
        return 0;
    else
        return STA_NOINIT | STA_NODISK;
}
```

Funkcja `disk_status` sprawdza, czy dysk jest aktywny. Jej działanie jest analogiczne jak funkcji `disk_initialize`, z tą różnicą, że wywołanie przekierowujemy do funkcji `USBdiskStatus`.

```
DSTATUS disk_status(BYTE drv) {
    if (drv == USB_DRIVE_NUMEBR && USBdiskStatus() == 0)
        return 0;
    else
        return STA_NOINIT | STA_NODISK;
}
```

Funkcja `disk_read` jest wywoływana przez bibliotekę FatFs w celu odczytania danych z dysku. Parametr `buff` jest wskaźnikiem na bufor, do którego mają być skopiowane odczytane dane. Parametr `sector` określa numer pierwszego sektora, który ma być odczytany. Parametr `count` określa liczbę sektorów do odczytania (jest to liczba z przedziału 1...128). W przypadku dysku USB wywołanie przekierujemy do funkcji `USBdiskRead`. Zależnie od jej wyniku zwracamy `RES_OK` lub `RES_ERROR`. Jeśli wywołanie dotyczy nieobsługiwanej dysku, zwracamy `RES_PARERR`.

```
DRESULT disk_read(BYTE drv, BYTE *buff, DWORD sector, BYTE count) {
    if (drv == USB_DRIVE_NUMEBR) {
        if (USBdiskRead(buff, sector, count) == 0)
            return RES_OK;
        else
            return RES_ERROR;
    }
    return RES_PARERR;
}
```

Funkcja `disk_write` jest wywoływana przez bibliotekę FatFs w celu zapisania danych na dysku. Parametr `buff` jest wskaźnikiem na bufor z danymi. Parametr

sector określa numer pierwszego sektora, który ma być zapisany. Parametr count określa liczbę sektorów do zapisania (jest to liczba z przedziału 1...128). W przypadku dysku USB wywołanie przekierowujemy do funkcji USBdiskWrite. Zależnie od jej wyniku zwracamy RES_OK lub RES_ERROR. Jeśli wywołanie dotyczy nieobsługiwanego dysku, zwracamy RES_PARERR.

```
DRESULT disk_write(BYTE drv, BYTE const *buff,
                    DWORD sector, BYTE count) {
    if (drv == USB_DRIVE_NUMEBR) {
        if (USBdiskWrite(buff, sector, count) == 0)
            return RES_OK;
        else
            return RES_ERROR;
    }
    return RES_PARERR;
}
```

Funkcja disk_ioctl jest wywoływana przez bibliotekę FatFs w celu wykonania na dysku innych operacji niż odczyt lub zapis. Parametr ctrl określa operację do wykonania. Parametr buff służy do przekazania dodatkowych danych. Wywołanie z parametrem ctrl o wartości CTRL_SYNC oznacza, że dysk powinien zakończyć wszystkie operacje zapisu, czyli zsynchronizować stan dysku ze stanem wynikającym z dotychczasowej sekwencji operacji zapisu. W tym celu wywołujemy funkcję USBdiskSync. Wartość parametru buff jest wtedy nieistotna (biblioteka FatFs ustawia go na zero). Wywołanie z parametrem ctrl o wartości GET_SECTOR_SIZE oznacza, że biblioteka FatFs chce odczytać rozmiar sektora, jaki jest używany na dysku. Takie wywołanie przekierowujemy do funkcji USBdiskGetSectorSize. W tym przypadku parametr buff wskazuje na 16-bitową zmienną, w której ma zostać zapisany żądany rozmiar sektora.

```
DRESULT disk_ioctl(BYTE drv, BYTE ctrl, void *buff) {
    if (drv == USB_DRIVE_NUMEBR) {
        if (ctrl == CTRL_SYNC) {
            if (USBdiskSync() == 0)
                return RES_OK;
            else
                return RES_ERROR;
        }
        else if (ctrl == GET_SECTOR_SIZE) {
            if (USBdiskGetSectorSize(buff) == 0)
                return RES_OK;
            else
                return RES_ERROR;
        }
    }
    return RES_PARERR;
}
```

fftime.c

W pliku *fftime.c* znajduje się funkcja `get_fattime`, która zwraca 32-bitową liczbę zawierającą bieżący czas w formacie stosowanym w systemie plików FAT. Format czasu opisany jest w **tablicy 8.3**. Starsze 16 bitów zawiera datę. Możliwe do zapisania daty zaczynają się 1 stycznia 1980 r. (system operacyjny DOS, w którym stosowano FAT, powstał na początku lat 80. ubiegłego wieku), a kończą się 31 grudnia 2107 r. Młodsze 16 bitów zawiera czas dzienny z dokładnością do dwóch sekund. Biblioteka FatFs wywołuje funkcję `get_fattime`, gdy tworzy nowy albo modyfikuje istniejący plik, aby zmodyfikować wpis w katalogu dotyczący tego pliku. Aktualna implementacja funkcji `get_fattime` zwraca zawsze tę samą datę 13 listopada 2012 r. i godzinę 22.18.04. Docelowo należałoby skorzystać z zegara czasu rzeczywistego RTC (ang. *Real Time Clock*), który jest dostępny w mikrokontrolerach STM32 jako układ peryferyjny. Rozszerzenie przykładu o obsługę RTC pozostawiam Czytelnikowi jako ćwiczenie.

```
DWORD get_fattime() {
    return (2012 - 1980) << 25 | 11 << 21 | 13 << 16 |
        22 << 11 | 18 << 5 | 4 >> 1;
}
```

Tab. 8.3. Format czasu w systemie plików FAT

Bity	Opis
31...25	Rok pomniejszony o 1980 (0...127)
24...21	Miesiąc (1...12)
20...16	Dzień miesiąca (1...31)
15...11	Godzina (0...23)
10...5	Minuta (0...59)
4...0	Sekunda podzielona przez 2 (0...29)

file_system.h
file_system.c

Moduł *file_system* dostarcza dwóch funkcji. Funkcja `FileSystemMount` inicjuje (montuje) system plików. Najpierw inicjuje bibliotekę FatFs, wywołując funkcję `f_mount` z pierwszym parametrem o wartości `USB_DRIVE_NUMEBR`. Jest to numer monitorowanego dysku. Drugi parametr jest wskaźnikiem na strukturę typu `FATFS`, która przechowuje potrzebne bibliotece FatFs informacje związane z systemem plików FAT. Jeśli wywołanie funkcji `f_mount` zakończy się pomyślnie (zwróci ona wartość `FR_OK`), to za pomocą funkcji `USBdiskInitialize` inicjujemy pamięć dyskową USB Flash. Wywołanie to nie jest konieczne, gdyż i tak biblioteka FatFs spróbuje zainicjować dysk przy pierwszym dostępie do pliku, czyli przy pierwszej próbie otwarcia jakiegoś pliku na tej pamięci. Jednak mimo to wywołujemy tę funkcję tutaj, aby w przypadku, gdy pamięć USB nie działa lub po prostu nie jest podłączona, zgłosić błąd możliwie wcześnie, zanim spróbujemy otworzyć jakiś plik. Funkcja

FileSystemMount zwraca zero, gdy zakończyła się powodzeniem, a wartość ujemną w przypadku błędu.

```
int FileSystemMount() {
    static FATFS fatfs;
    if (f_mount(USB_DRIVE_NUMEBR, &fatfs) != FR_OK)
        return -1;
    return USBdiskInitialize();
}
```

Funkcja FileSystemUnmount odmontowuje system plików. Najpierw w celu zwolnienia zasobów biblioteki FatFs wywołuje funkcję f_mount. Pierwszy parametr jest numerem odmontowywanego dysku, a drugi parametr musi mieć wartość zero. Następnie wywołuje funkcję USBdiskAllowEject, aby pamięć USB Flash zakończyła wszystkie operacje, w szczególności wszystkie zapisy, co umożliwi jej bezpieczne wyjście. Ze względów wydajnościowych pamięć może buforować i opóźniać zapisy. Odłączenie pamięci, gdy zapis się nie zakończył, może uszkodzić system plików. Teoretycznie obie wywoływane funkcje mogą zakończyć się niepowodzeniem. Bezpieczne jest wywołanie obu tych funkcji, nawet gdy jedna z nich zwróciła błąd. Funkcja FileSystemUnmount zwraca zero, gdy obie funkcje zakończyły się sukcesem, a wartość ujemną w przeciwnym przypadku.

```
int FileSystemUnmount() {
    FRESULT fr;
    int re;
    fr = f_mount(USB_DRIVE_NUMEBR, 0);
    re = USBdiskAllowEject();
    return fr == FR_OK && re == 0 ? 0 : -1;
}
```

ffconf.h

Plik nagłówkowy *ffconf.h* jest dostarczany wraz z biblioteką FatFs. Zdefiniowane są w nim stałe konfigurujące tę bibliotekę. Należy go zmodyfikować w celu dostrojenia biblioteki do potrzeb konkretnego projektu. Wszystko jest szczegółowo wyjaśnione w dokumentacji biblioteki i w komentarzach zamieszczonych w tym pliku. Poniżej przedstawiam tylko te stałe, które zmodyfikowałem na potrzeby niniejszego projektu.

```
#define _FS_TINY 1
```

Stała *_FS_TINY* determinuje ilość pamięci RAM zużywanej przez bibliotekę FatFs. Wartość 1 zmniejsza to zapotrzebowanie przez alokowanie wspólnego bufora dla całego systemu plików zamiast osobno dla każdego pliku. Oryginalna wartość to 0.

```
#define _FS_MINIMIZE 2
```

Stała *_FS_MINIMIZE* określa, które funkcje biblioteki FatFs zostaną skompilowane. Wartość 0 oznacza wszystkie funkcje. Wartość 1 oznacza, że nie będą kompilowa-

ne różne zaawansowane funkcje: `f_stat`, `f_getfree`, `f_unlink`, `f_mkdir`, `f_chmod`, `f_truncate` i `f_rename`. Wartość 2 oznacza, że dodatkowo będą też wyrzucone funkcje do obsługi katalogów: `f_opendir` i `f_readdir`. Wartość 3 oznacza, że nie będzie również dostępna funkcja `f_lseek`. Oryginalna wartość to 0.

```
#define _CODE_PAGE 1
```

Stała `_CODE_PAGE` określa numer strony kodowej używanej do zapisywania nazw plików. Wartość 1 oznacza, że dopuszczalne są tylko znaki ze standardowego zestawu ASCII, nie można stosować żadnych diakrytycznych znaków narodowych, a nazwa pliku musi być w formacie 8.3, czyli maksymalnie 8 znaków na nazwę, po czym ewentualnie kropka i maksymalnie trzy znaki rozszerzenia. W nazwach plików nie rozróżnia się wielkich i małych liter. Zatem jest to konfiguracja zgodna z tym, co było w pierwszych wersjach systemu operacyjnego DOS. Biblioteka FatFs pozwala korzystać z później wprowadzonych rozszerzeń, czyli długich nazw plików kodowanych w różnych alfabetach za pomocą różnych stron kodowych lub unikodu. Wymaga to właściwego ustawienia wartości stałych `_CODE_PAGE`, `_USE_LFN`, `_MAX_LFN`, `_LFN_UNICODE` oraz skompilowania odpowiednich tablic konwersji, które znajdują się w plikach źródłowych w katalogu `/libraries/fatfs/src/option`. Rozszerzenie projektu o obsługę długich nazw w języku polskim pozostawiam Czytelnikowi jako ćwiczenie.

```
#define _MAX_SS 4096
```

Stała `_MAX_SS` określa maksymalny rozmiar sektora. Może przyjmować wartości 512, 1024, 2048 lub 4096. Oryginalna wartość to 512.

8.2. Obsługa pamięci USB Flash

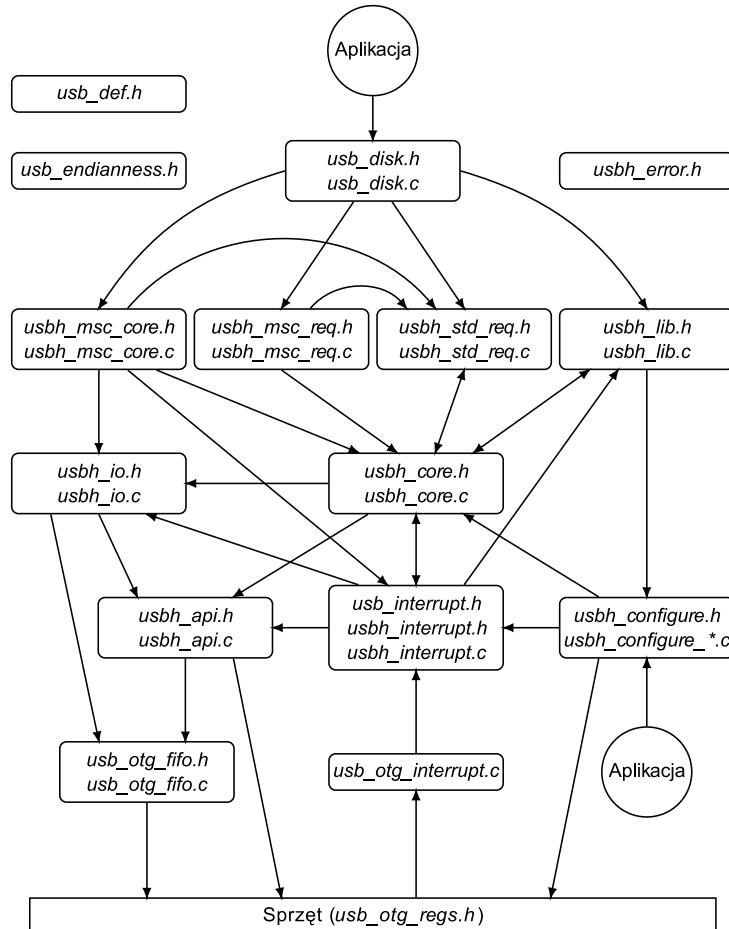
Na **rysunku 8.2** przedstawione są zależności między poszczególnymi modułami biblioteki wykorzystanymi w projekcie kontrolera pamięci masowej. Rysunku tego dotyczą wszelkie uwagi zamieszczone w poprzednim rozdziale przy opisie rysunku 7.1.

8.2.1. Protokół BOT

Implementacja protokołu BOT podzielona jest na dwa moduły. Omawiam je kolejno w tym podrozdziale. Pierwszy z tych modułów implementuje komunikację z użyciem danych sterujących, a drugi moduł obsługuje komunikację korzystającą z danych masowych.

<code>usbh_msc_req.h</code>
<code>usbh_msc_req.c</code>

Moduł `usbh_msc_req` rozszerza implementację kontrolera USB o obsługę żądań specyficznych dla protokołu BOT. Funkcje udostępniane przez ten moduł zadeklarowane są w pliku nagłówkowym `usbh_msc_req.h`, a zaimplementowane w pliku `usbh_msc_req.c`. Wszystkie te funkcje obudowują bezpośrednio lub pośrednio wywołanie funkcji `USBHcontrolRequest`. Pierwszy ich argument `synch` ma takie samo znaczenie jak w tej funkcji i jest do niej przekazywany. Wszystkie funkcje



Rys. 8.2. Zależności między modułami biblioteki dla projektu kontrolera pamięci masowej

udostępniane przez ten moduł zwracają wartość otrzymaną z wywołania funkcji `USBHcontrolRequest`.

```
int MSCgetMaxLun(int synch, uint8_t iface, uint8_t *max_lun);
```

Funkcja `MSCgetMaxLun` realizuje żądanie `MSC_GET_MAX_LUN` (patrz tabela 6.2). Parametr `iface` zawiera numer interfejsu, do którego skierowane jest to żądanie. Parametr `max_lun` jest wskaźnikiem na zmienną, w której po poprawnym zrealizowaniu żądania ma być zapisana otrzymana wartość, czyli maksymalny numer jednostki logicznej obsługiwanej przez urządzenie.

```
int MSCresetBOT(int synch, uint8_t iface);
```

Funkcja `MSCresetBOT` realizuje żądanie `MSC_BULK_ONLY_RESET` (patrz tabela 6.2). Parametr `iface` zawiera numer interfejsu, do którego skierowane jest to żądanie.

```
int MSCresetRecovery(int synch, uint8_t interface_number,
                      uint8_t in_ep_addr, uint8_t out_ep_addr);
```

Funkcja `MSCresetRecovery` realizuje procedurę przywracania stanu początkowego (ang. *reset recovery*). Przy czym jest ona zaimplementowana tylko w wersji synchronicznej (dla niezerowej wartości parametru `synch`). Zaimplementowanie wersji asynchronicznej pozostawiam Czytelnikowi jako ćwiczenie. Procedura ta składa się z trzech kolejnych kroków:

- przywrócenie początkowego stanu komunikacji za pomocą żądania `MSC_BULK_ONLY_RESET`;
- uaktywnienie wejściowego punktu końcowego dla danych masowych za pomocą skierowanego do tego punktu końcowego żądania `CLEAR_FEATURE` z parametrem `wValue` równym `ENDPOINT_HALT` (zero, patrz też tabele 1.15 i 1.16);
- uaktywnienie wyjściowego punktu końcowego dla danych masowych za pomocą skierowanego do tego punktu końcowego żądania `CLEAR_FEATURE` z parametrem `wValue` równym `ENDPOINT_HALT` (zero, patrz też tabele 1.15 i 1.16).

<code>usbh_msc_core.h</code>
<code>usbh_msc_core.c</code>

Moduł `usbh_msc_core` rozszerza implementację rdzenia protokołu kontrolera USB o obsługę tej części komunikacji z urządzeniem klasy MSC, która korzysta z przesyłania danych masowych.

```
typedef struct {
    usb_speed_t          speed;
    uint16_t              out_ep_max_packet;
    uint16_t              in_ep_max_packet;
    uint16_t              sector_size;
    uint8_t                dev_addr;
    uint8_t              configuration_value;
    uint8_t              interface_number;
    uint8_t              out_ep_addr;
    uint8_t              in_ep_addr;
    uint8_t              interval;
    uint8_t              max_lun;
} msc_configuration_t;
```

W pliku nagłówkowym `usbh_msc_core.h` zdefiniowana jest struktura typu `msc_configuration_t`, opisująca konfigurację urządzenia klasy MSC. Składowa `speed` tej struktury zawiera szybkość, z jaką odbywa się komunikacja z urządzeniem. Składowe `out_ep_max_packet` i `in_ep_max_packet` określają maksymalne rozmiary pola danych odpowiednio wyjściowego i wejściowego punktu końcowego dla danych masowych. Składowa `sector_size` zawiera rozmiar sektora na dysku

USB Flash. Składowa `dev_addr` przechowuje adres urządzenia. Składowa `configuration_value` zawiera numer konfiguracji, czyli wartość z pola `bConfigurationValue` deskryptora konfiguracji. Składowa `interface_number` zawiera numer interfejsu, czyli wartość z pola `bInterfaceNumber` deskryptora interfejsu. Składowe `out_ep_addr` i `in_ep_addr` przechowują adresy odpowiednio wyjściowego i wejściowego punktu końcowego dla danych masowych. Składowa `interval` zawiera wartość z pola `bInterval` deskryptora wychodzącego punktu końcowego dla danych masowych, jeśli jest to urządzenie HS. W przeciwnym przypadku składowa ta ma wartość zero. Składowa `max_lun` określa maksymalny numer jednostki logicznej na dysku USB Flash. Przedstawiona kolejność składowych w strukturze może się wydać nielogiczna. Pomieszczone są informacje dotyczące interfejsu USB i organizacji dysku. Celem takiego układu składowych jest zminimalizowanie zajmowanej przez strukturę pamięci. Kompilator, optymalizując rozmieszczenie składowych w pamięci, umieszcza je tak, aby zawsze znajdowały się pod adresami, które są wielokrotnościami ich rozmiarów. Umieszczanie składowych w porządku nierosnących rozmiarów sprawia, że kompilator nie musi pozostawiać między nimi odstępów, aby spełnić to wymaganie.

Ponadto w pliku nagłówkowym `usbh_msc_core.h` zadeklarowane są trzy niżej przedstawione funkcje. Ich implementacja znajduje się w pliku `usbh_msc_core.c`.

```
int MSCsetMachine(msc_configuration_t const *cfg);
```

Funkcja `MSCsetMachine` instaluje automat stanowy obsługujący protokół BOT urządzenia klasy MSC. Korzysta w tym celu z funkcji `USBHsetClassMachine`. Parametr `cfg` wskazuje strukturę opisującą konfigurację urządzenia. Funkcja ta zwraca `USBHLIB_SUCCESS`, gdy zakończyła się sukcesem, albo odpowiedni kod błędu.

```
int MSCisDeviceReady(void);
```

Bezparametrowa funkcja `MSCisDeviceReady` zwraca jedynkę, gdy urządzenie klasy MSC jest podłączone i aktywne, czyli gdy działa automat obsługujący komunikację i urządzenie jest gotowe do komunikacji. W przeciwnym przypadku funkcja ta zwraca zero.

```
uint8_t MSCBOT(int synch, uint8_t lun,
               uint8_t const *scsi_cmd, uint32_t scsi_len,
               uint8_t const *out_buff, uint8_t *in_buff,
               uint32_t *len);
```

Funkcja `MSCBOT` realizuje pojedyncze przesłanie protokołu BOT, który opisałem w rozdziale 6.2.1. Pod względem działania i struktury implementacji funkcja ta jest podobna do funkcji `USBHcontrolRequest`. Parametr `synch` określa, czy wywołanie ma być nieblokujące (asynchroniczne, wartość zero), czy blokujące (synchroniczne, wartość niezerowa). Parametr `lun` zawiera numer jednostki logicznej. Parametr `scsi_cmd` wskazuje polecenie protokołu SCSI, które ma być wysłane w pakiecie CBW. Parametr `scsi_len` zawiera rozmiar w bajtach tego polecenia. Parametr `out_`

buff jest wskaźnikiem, który jeśli ma niezerową wartość, to wskazuje na bufor z danymi do wysłania. Parametr `in_buff` jest wskaźnikiem, który jeśli ma niezerową wartość, to wskazuje na bufor, do którego mają zostać skopiowane odebrane dane. Parametr `len` jest wskaźnikiem na zmienną, która przy wywołaniu funkcji zawiera rozmiar bufora, a po jej zakończeniu przypisywany jest do niej rozmiar rzeczywiście przesłanych danych. Kierunek transmisji danych określa się na podstawie wartości wskaźników `out_buff` i `in_buff`. Tylko jeden z nich może mieć niezerową wartość. Jeśli oba mają wartość zero, to żadne dane nie będą przesyłane – po wysłaniu do urządzenia pakietu CBW następuje od razu przejście do oczekiwania na przysłanie przez urządzenie pakietu CSW. Wtedy zmienna wskazywana parametrem `len` powinna mieć również wartość zero. Funkcja ta zwraca wartość z pola `bCSWStatus` pakietu CSW, czyli wynik wykonania polecenia, albo wartość `MSC_BOT_UNDEF_ERROR`, jeśli wystąpił błąd transmisji i nie udało się odebrać po-prawnego pakietu CSW.

Przyjrzymy się teraz implementacji automatu stanowego realizującego protokół BOT (patrz też rozdział 6.2.1). Dane automatu przechowujemy w strukturze typu `usbh_msc_data_t`. Składowa `state` tej struktury zawiera bieżący stan automatu. Składowe `out_ch_num` i `in_ch_num` przechowują numery kanałów kontrolera używanych do komunikacji z punktami końcowymi urządzenia. Są to dwa punkty końcowe odpowiednio do wysyłania i odbierania danych masowych. Składowa `buffer` jest wskaźnikiem na bufor przechowujący dane do wysłania lub odebrane dane. Składowa `length` zawiera rozmiar danych do przesłania. Natomiast składowa `transferred` zawiera rozmiar rzeczywiście przesłanych danych. Składowe `nak_delay`, `nak_timeout` i `csw_attempts` są potrzebne do tego, aby automat mógł powtarzać pewne akcje i się przy tym nie zapętlil. Ich znaczenie stanie się jasne po omówieniu szczegółów implementacji automatu. Składowa `status` przechowuje wartość, która ma być zwrócona przez funkcję `MSCBOT`. Składowa `cfg` zawiera konfigurację podłączonego urządzenia. Składowa `cbw` przechowuje pakiet CBW, który ma być wysłany. Do składowej `csw` zapisuje się odebrany pakiet CSW.

```
typedef struct {
    usbh_msc_state_t           state;
    int                         out_ch_num;
    int                         in_ch_num;
    uint8_t                     *buffer;
    uint32_t                    length;
    uint32_t                    transferred;
    unsigned                    nak_delay;
    unsigned                    nak_timeout;
    unsigned                    csw_attempts;
    uint8_t                     status;
    msc_configuration_t         cfg;
    msc_bot_cbw_t               cbw;
    msc_bot_csw_t               csw;
} usbh_msc_data_t;
```

Automat korzysta z kilku pomocniczych funkcji. Pierwszy, a czasem jedyny, parametr tych funkcji, czyli `md`, jest wskaźnikiem na strukturę przechowującą dane automatu. Żeby móc przesyłać dane masowe między kontrolerem a urządzeniem, musimy przydzielić kanały. Zajmuje się tym funkcja `InitState`. Jeżeli wszystko się powiodło, to automat przechodzi do stanu `MSC_IDLE`, w którym oczekuje na zainicjowanie sesji protokołu BOT. W przypadku niepowodzenia przydzielenia kanałów automat przechodzi do stanu `MSC_EXIT`, w którym kończy swoje działanie. Funkcja `InitState` nie zwraca żadnej wartości.

```
static void InitState(usbh_msc_data_t *md) {
    md->out_ch_num = USBHallocChannel();
    md->in_ch_num = USBHallocChannel();
    if (md->out_ch_num >= 0 && md->in_ch_num >= 0) {
        USBHopenChannel(md->out_ch_num, md->cfg.dev_addr,
                         md->cfg.out_ep_addr, md->cfg.speed,
                         BULK_TRANSFER, md->cfg.out_ep_max_packet);
        USBHopenChannel(md->in_ch_num, md->cfg.dev_addr,
                         md->cfg.in_ep_addr, md->cfg.speed,
                         BULK_TRANSFER, md->cfg.in_ep_max_packet);
        md->state = MSC_IDLE;
    }
    else {
        md->state = MSC_EXIT;
    }
}
```

Jeśli kanały zostały przydzielone, to przed zakończeniem działania automatu trzeba je zwolnić. Zajmuje się tym funkcja `FreeChannels`. Zakładamy, że w momencie wywoływania tej funkcji nie odbywa się żadna transmisja, a kanały zostały wstrzymane za pomocą funkcji `USBHhaltChannel`. Kanał jest wstrzymywany w procedurze obsługi przerwania (plik `usbh_interrupt.c`) zarówno po poprawnym zakończeniu transmisji, jak i w przypadku wystąpienia błędu. Funkcja `FreeChannels` nie zwraca żadnej wartości.

```
static void FreeChannels(usbh_msc_data_t *md) {
    if (md->out_ch_num >= 0) {
        USBHfreeChannel(md->out_ch_num);
        md->out_ch_num = -1;
    }
    if (md->in_ch_num >= 0) {
        USBHfreeChannel(md->in_ch_num);
        md->in_ch_num = -1;
    }
}
```

Funkcja StartState jest wywoływana w sytuacji, gdy chcemy zainicjować transmisję danych. Parametr `ch_num` zawiera numer kanału, którym mają być przesyłane dane. Kierunek transmisji jest zdeterminowany konfiguracją tego kanału. Parametr `buffer` jest wskaźnikiem na bufor z danymi do wysłania lub bufor, do którego mają zostać skopiowane odebrane dane. Parametr `length` zawiera rozmiar danych do przesłania. Parametr `next_state` określa stan automatu, w którym będziemy oczekiwali na zakończenie transmisji. Przesyłanie jest inicjowane za pomocą funkcji `USBHstartTransfer`. Jeśli transmisja została poprawnie zainicjowana, to ustawiamy składową `nak_timeout`, która określa, jak długo mamy powtarzać próby transmisji, gdy urządzenie permanentnie odrzuca pakietem NAK kolejne próby zrealizowania transakcji. Zabezpiecza to kontroler przed zapętlaniem się. Wartość tej składowej jest wyrażana w liczbie ramek lub mikroramek, dlatego w przypadku komunikacji HS, chcąc odliczyć zadaną liczbę milisekund, musimy ją pomnożyć przez 8. W aktualnej implementacji wartość stałej `NAK_TIMEOUT_MS` została dobrana eksperymentalnie i wynosi 10 000. W niektórych przypadkach może być konieczne dopasowanie tej wartości. Jeśli transmisja została poprawnie zainicjowana, to przechodzimy do stanu określonego parametrem `next_state`, a w przeciwnym przypadku kończymy sesję protokołu BOT, przechodząc do stanu `MSC_DONE`. W tym drugim przypadku, ponieważ nie odebraliśmy jeszcze pakietu CSW, ustawiamy w składowej `status` wartość `MSC_BOT_UNDEF_ERROR`, którą ma zwrócić funkcja `MSCBOT`. Funkcja `StartState` nie zwraca żadnej wartości.

```
static void StartState(usbh_msc_data_t *md, int ch_num,
                      uint8_t *buffer, uint32_t length,
                      usbh_msc_state_t next_state) {
    if (USBHstartTransfer(ch_num, buffer, length) == 0) {
        md->nak_timeout = md->cfg.speed == HIGH_SPEED ?
            NAK_TIMEOUT_MS << 3 : NAK_TIMEOUT_MS;
        md->state = next_state;
    } else {
        md->state = MSC_DONE;
    }
    md->status = MSC_BOT_UNDEF_ERROR;
}
```

Funkcja WaitState jest wywoływana, gdy automat czeka na zakończenie transmisji. Parametr `ch_num` zawiera numer kanału, którym są przesyłane dane. Parametr `next_state` zawiera stan, do którego automat ma przejść po poprawnym zakończeniu transmisji. Parametr `interval` określa, jak długo po odebraniu od urządzenia pakietu NAK należy poczekać na ponowne zainicjowanie transmisji. W tym przypadku wartość tego parametru służy do zainicjowania składowej `nak_delay`, a automat przechodzi do stanu określonego parametrem `nak_state`, chyba że upłynął czas zapisany w składowej `nak_timeout` – wtedy automat kończy sesję protokołu BOT i przechodzi do stanu `MSC_DONE`, a w składowej `status` ustawiamy wartość `MSC_BOT_UNDEF_ERROR`. Parametr `stall_state` zawiera stan, do którego automat ma

przejść po odrzuceniu transmisji pakietem STALL. Protokół BOT przewiduje dwie możliwości. Jeśli odrzucona została transmisja pakietu CBW, to najprawdopodobniej nastąpiła utrata synchronizacji protokołu BOT i kontroler powinien rozpoczęć procedurę przywracania stanu początkowego (patrz opis funkcji MSCresetRecovery). Aby zasygnalizować tę sytuację, ustawiamy w składowej status wartość MSC_BOT_CSW_PHASE_ERROR. Jeśli natomiast urządzenie odrzuciło transmisję danych, to oznacza, że nie chce wysłać ani odebrać więcej danych. Wtedy kontroler powinien spróbować odebrać pakiet CSW. W tej sytuacji wartość ustawiona w składowej status nie ma znaczenia – zostanie zmieniona w stanie określonym parametrem stall_state. Gdy wystąpił błąd transmisji, czyli gdy funkcja USBHgetTransferResult zwróciła wartość TR_ERROR, automat kończy sesję protokołu BOT. Zwrócenie przez tę funkcję wartości TR_UNDEF oznacza, że transmisja wciąż trwa. Sytuację tę wyłączamy w sekcji default instrukcji switch. Funkcja WaitState zwraca jedynkę, gdy transmisja zakończyła się poprawnie, a zero w przeciwnym przypadku.

```
static int WaitState(usbh_msc_data_t *md, int ch_num,
                     unsigned interval, usbh_msc_state_t next_state,
                     usbh_msc_state_t nak_state,
                     usbh_msc_state_t stall_state) {
    switch (USBHgetTransferResult(ch_num)) {
        case TR_DONE:
            md->state = next_state;
            return 1;
        case TR_NAK:
            if (md->nak_timeout > 0) {
                md->nak_delay = interval;
                md->state = nak_state;
            }
            else {
                md->status = MSC_BOT_UNDEF_ERROR;
                md->state = MSC_DONE;
            }
            return 0;
        case TR_STALL:
            md->status = MSC_BOT_CSW_PHASE_ERROR;
            md->state = stall_state;
            return 0;
        case TR_ERROR:
            md->status = MSC_BOT_UNDEF_ERROR;
            md->state = MSC_DONE;
            return 0;
        default:
            return 0;
    }
}
```

Funkcja NakedState jest wywoływaną, gdy automat czeka na wznowienie transmisji po odebraniu od urządzenia pakietu NAK. Parametr ch_num zawiera numer kanału, którym mają być przesyłane dane. Parametr prev_state określa stan, do którego automat ma wrócić po wznowieniu transmisji i w którym automat ma oczekiwania na jej zakończenie. Transmisję wznowiamy za pomocą funkcji USBHrestartTransfer. Podobnie jak poprzednio, gdy wystąpił błąd, automat kończy sesję protokołu BOT, przechodząc do stanu MSC_DONE i odpowiednio ustawiając składową status. Funkcja NakedState nie zwraca żadnej wartości.

```
static void NakedState(usbh_msc_data_t *md, int ch_num,
                      usbh_msc_state_t prev_state) {
    if (md->nak_delay == 0) {
        if (USBHrestartTransfer(ch_num) == 0) {
            md->state = prev_state;
        }
        else {
            md->status = MSC_BOT_UNDEF_ERROR;
            md->state = MSC_DONE;
        }
    }
}
```

W funkcji StalledState próbujemy za pomocą funkcji USBHclearEndpointHalt ponownie uaktywnić punkt końcowy, który odrzucił transmisję pakietem STALL. Parametr ep_addr zawiera adres tego punktu końcowego. Jeśli się udało, to automat przechodzi do stanu MSC_CSW_START, w którym ma zostać zainicjowane odbieranie pakietu CSW. Jeśli się nie udało uaktywnić punktu końcowego, automat kończy sesję protokołu BOT. Funkcja StalledState nie zwraca żadnej wartości.

```
static void StalledState(usbh_msc_data_t *md, uint8_t ep_addr) {
    int res;
    res = USBHclearEndpointHalt(0, ep_addr);
    if (res == USBHLIB_SUCCESS) {
        md->state = MSC_CSW_START;
    }
    else if (res != USBHLIB_IN_PROGRESS) {
        md->status = MSC_BOT_UNDEF_ERROR;
        md->state = MSC_DONE;
    }
}
```

Funkcja IsCSWvalidAndMeaningful sprawdza poprawność odebranego pakietu CSW. Zwraca jedynkę, gdy jest on poprawny, a zero w przeciwnym przypadku. Pakiet CSW jest poprawny, gdy rozmiar odebranych danych jest równy rozmiarowi struktury oraz gdy poprawne są wartości jej pól csw.dCSWSignature i csw.dCSW-Tag. Sprawdzamy też poprawność wartości w polu csw.bCSWStatus. W przypadku,

gdy pole to ma wartość MSC_BOT_CSW_COMMAND_PASSED lub MSC_BOT_CSW_COMMAND_FAILED, rozmiar przesyłanych danych (składowa transferred) musi być różnicą rozmiaru danych, które miały być przesłane (składowa length), i rozmiaru danych, które nie zostały przetworzone przez urządzenie (pole dCSWDataResidue).

```
static int IsCSWisValidAndMeaningful(usbh_msc_data_t const *md) {
    return USBHgetTransferSize(md->in_ch_num) == sizeof(msc_bot_csw_t) &&
        md->csw.dCSWSignature == MSC_BOT_CSW_SIGNATURE &&
        md->csw.dCSWTag == md->cbw.dCBWTag &&
        (md->csw.bCSWStatus == MSC_BOT_CSW_PHASE_ERROR ||
        (md->transferred == md->length - md->csw.dCSWDataResidue &&
        (md->csw.bCSWStatus == MSC_BOT_CSW_COMMAND_PASSED ||
        md->csw.bCSWStatus == MSC_BOT_CSW_COMMAND_FAILED)));
}
```

Zasadniczy automat jest zaimplementowany w funkcji MSCstateMachine. Jej jedyny parametr p jest wskaźnikiem na strukturę przechowującą dane automatu. Funkcja ta zwraca wartość niezerową, gdy ma przestać być wywoływana, czyli gdy automat zakończył swoje działanie. Funkcja zwraca zero, gdy ma być ponownie wywołana. Automat zaimplementowany jest tradycyjnie za pomocą instrukcji switch, która rozdziela sterowanie zależnie od bieżącego stanu. Stan MSC_INIT służy do zainicjowania automatu. Ze stanu tego automat może przejść do stanu MSC_IDLE lub MSC_EXIT (patrz opis funkcji InitState). Ze stanu MSC_IDLE możemy przeprowadzić automat do stanu MSC_CBW_START, w którym inicjuje on wysyłanie pakietu CBW. Ze stanu MSC_CBW_START automat może przejść do stanu MSC_CBW_WAIT lub MSC_DONE (patrz opis funkcji StartState). W stanie MSC_CBW_WAIT automat czeka na zakończenie wysyłania pakietu CBW. Zależnie od tego, czy są jakieś dane do przesłania i w jakim kierunku mają być transmitowane, ze stanu MSC_CBW_WAIT automat przechodzi do stanu MSC_CSW_START, MSC_DATA_IN_START lub MSC_DATA_OUT_START. Ponadto, w przypadku wystąpienia problemów, może też przejść do stanu MSC_CBW_NAKED lub MSC_DONE (patrz opis funkcji WaitState). W stanie MSC_CBW_NAKED automat próbuje ponowić transmisję pakietu CBW. Ze stanu tego wraca do stanu MSC_CBW_WAIT lub przechodzi do stanu MSC_DONE (patrz opis funkcji NakedState).

```
static int MSCstateMachine(void *p) {
    usbh_msc_data_t *md = p;
    switch (md->state) {
        case MSC_INIT:
            InitState(md);
            break;
        case MSC_CBW_START:
            StartState(md, md->out_ch_num, (uint8_t *)&md->cbw,
                sizeof(msc_bot_cbw_t), MSC_CBW_WAIT);
            break;
        case MSC_CBW_WAIT:
```

```

        if (WaitState(md, md->out_ch_num, md->cfg.interval,
                      MSC_CSW_START, MSC_CBW_NAKED, MSC_DONE)) {
            if (md->cbw.bmCBWFlags == MSC_BOT_CBW_DATA_IN) {
                md->state = MSC_DATA_IN_START;
            }
            else if (md->length > 0) {
                md->state = MSC_DATA_OUT_START;
            }
        }
        break;
    case MSC_CBW_NAKED:
        NakedState(md, md->out_ch_num, MSC_CBW_WAIT);
        break;
    case MSC_DATA_IN_START:
        StartState(md, md->in_ch_num, md->buffer, md->length,
                   MSC_DATA_IN_WAIT);
        break;
    case MSC_DATA_IN_WAIT:
        WaitState(md, md->in_ch_num, 0, MSC_CSW_START,
                  MSC_DATA_IN_NAKED, MSC_DATA_IN_STALLED);
        md->transferred = USBHgetTransferSize(md->in_ch_num);
        break;
    case MSC_DATA_IN_NAKED:
        NakedState(md, md->in_ch_num, MSC_DATA_IN_WAIT);
        break;
    case MSC_DATA_IN_STALLED:
        StalledState(md, md->cfg.in_ep_addr);
        break;
    case MSC_DATA_OUT_START:
        StartState(md, md->out_ch_num, md->buffer, md->length,
                   MSC_DATA_OUT_WAIT);
        break;
    case MSC_DATA_OUT_WAIT:
        WaitState(md, md->out_ch_num, md->cfg.interval, MSC_CSW_START,
                  MSC_DATA_OUT_NAKED, MSC_DATA_OUT_STALLED);
        md->transferred = USBHgetTransferSize(md->out_ch_num);
        break;
    case MSC_DATA_OUT_NAKED:
        NakedState(md, md->out_ch_num, MSC_DATA_OUT_WAIT);
        break;
    case MSC_DATA_OUT_STALLED:

```

```

        StalledState(md, md->cfg.out_ep_addr);
        break;
    case MSC_CSW_START:
        StartState(md, md->in_ch_num, (uint8_t *)&md->csw,
                    sizeof(msc_bot_csw_t), MSC_CSW_WAIT);
        break;
    case MSC_CSW_WAIT:
        if (WaitState(md, md->in_ch_num, 0,
                      MSC_DONE, MSC_CSW_NAKED, MSC_CSW_STALLED)) {
            md->csw.dCSWSignature = USBTOHL(md->csw.dCSWSignature);
            md->csw.dCSWDataResidue = USBTOHL(md->csw.dCSWDataResidue);
            if (IsCSWValidAndMeaningful(md))
                md->status = md->csw.bCSWStatus;
            else
                md->status = MSC_BOT_CSW_PHASE_ERROR;
        }
        break;
    case MSC_CSW_NAKED:
        NakedState(md, md->in_ch_num, MSC_CSW_WAIT);
        break;
    case MSC_CSW_STALLED:
        if (++md->csw_attempts < 2) {
            StalledState(md, md->cfg.in_ep_addr);
        }
        else {
            md->status = MSC_BOT_UNDEF_ERROR;
            md->state = MSC_DONE;
        }
        break;
    default: /* MSC_EXIT, MSC_IDLE, MSC_DONE */
        break;
    }
    if (md->state == MSC_EXIT)
        FreeChannels(md);
    return md->state == MSC_EXIT;
}

```

W stanie MSC_DATA_IN_START automat inicjuje odbieranie danych. Ze stanu tego może przejść do stanu MSC_DATA_IN_WAIT lub MSC_DONE (patrz opis funkcji StartState). W stanie MSC_DATA_IN_WAIT automat czeka na zakończenie odbierania danych. Z tego stanu, zależnie od wyniku transmisji, automat może przejść do jednego ze stanów: MSC_CSW_START, MSC_DATA_IN_NAKED, MSC_DATA_IN_STALLED, MSC_DONE (patrz opis funkcji WaitState). W stanie MSC_DATA_IN_NAKED automat

próbuje wznowić transmisję danych. Ze stanu tego wraca do stanu `MSC_DATA_IN_WAIT` lub przechodzi do stanu `MSC_DONE` (patrz opis funkcji `NackedState`). W stanie `MSC_DATA_IN_STALLED` automat uaktywnia wejściowy punkt końcowy dla danych masowych. Jeśli się powiedzie, przechodzi do stanu `MSC_CSW_START`, a w przeciwnym przypadku do stanu `MSC_DONE` (patrz opis funkcji `StalledState`). Wysyłanie danych odbywa się w stanach `MSC_DATA_OUT_START`, `MSC_DATA_OUT_WAIT`, `MSC_DATA_OUT_NAKED`, `MSC_DATA_OUT_STALLED`. Zachowanie automatu w tych stanach jest analogiczne jak przy odbieraniu danych.

W stanie `MSC_CSW_START` automat inicjuje odbieranie pakietu CSW. Ze stanu tego może przejść do stanu `MSC_CSW_WAIT` lub `MSC_DONE` (patrz opis funkcji `StartState`). W stanie `MSC_CSW_WAIT` automat czeka na zakończenie transmisji pakietu CSW. Jeśli transmisja zakończy się powodzeniem, to automat przechodzi do stanu `MSC_DONE`, ale przedtem sprawdzamy poprawność zawartości pakietu CSW za pomocą funkcji `IsCSWvalidAndMeaningful`. Zależnie od wyniku tego sprawdzenia ustawiamy wartość składowej `status`, czyli wartość, którą ma zwrócić funkcja `MSCBOT`. Jeśli wystąpił problem podczas transmisji, automat może przejść ze stanu `MSC_CSW_WAIT` do stanu `MSC_CSW_NAKED`, `MSC_CSW_STALLED` lub `MSC_DONE` (patrz opis funkcji `WaitState`). W stanie `MSC_CSW_NAKED` automat próbuje ponówić odbieranie pakietu CSW. Ze stanu tego wraca do stanu `MSC_CSW_WAIT` lub przechodzi do stanu `MSC_DONE` (patrz opis funkcji `NackedState`). W stanie `MSC_CSW_STALLED` automat uaktywnia wejściowy punkt końcowy dla danych masowych. Jeśli się to powiedzie, przechodzi do stanu `MSC_CSW_START`, aby ponownie spróbować odebrać pakiet CSW, a w przeciwnym przypadku przechodzi do stanu `MSC_DONE` (patrz opis funkcji `StalledState`). Przy czym automat wykonuje co najwyżej dwie próby odebrania pakietu CSW. Liczbę prób zlicza składowa `csw_attempts`.

Sekcja `default` obejmuje pozostałe stany automatu, czyli `MSC_EXIT`, `MSC_IDLE` i `MSC_DONE`. Przejście automatu do stanu `MSC_EXIT` powoduje zwolnienie kanałów za pomocą funkcji `FreeChannels` i zakończenie pracy automatu. W stanie `MSC_IDLE` automat czeka na zainicjowanie sesji protokołu BOT. W stanie `MSC_DONE` automat czeka po zakończeniu sesji protokołu BOT na odczytanie jej wyniku.

Funkcja `MSCatSof` jest wywoływana na początku każdej ramki lub mikroramki. Jej zadaniem jest zmniejszenie wartości liczników `nak_delay` i `nak_timeout` odmierzających czas. Parametr `p` jest wskaźnikiem na strukturę przechowującą dane automatu. Parametr `frnum` zawiera numer ramki lub mikroramki – nie jest on wykorzystywany. Funkcja ta nie zwraca żadnej wartości.

```
static void MSCatSof(void *p, uint16_t frnum) {
    usbh_msc_data_t *md = p;
    if (md->nak_delay > 0)
        --md->nak_delay;
    if (md->nak_timeout > 0)
        --md->nak_timeout;
}
```

Funkcja `MSCatDisconnect` jest wywoływana, gdy urządzenie zostanie odłączone – wyjąte z gniazda USB. Parametr `p` jest wskaźnikiem na strukturę przechowującą

dane automatu. Zadaniem tej funkcji jest zakończenie działania automatu i zwolnienie kanałów. Nie zwraca ona żadnej wartości.

```
static void MSCatDisconnect(void *p) {
    usbh_msc_data_t *md = p;
    FreeChannels(md);
    md->status = MSC_BOT_UNDEF_ERROR;
    md->state = MSC_EXIT;
}
```

Globalna struktura MSCdata typu `usbh_msc_data_t` przechowuje dane automatu. Adres tej struktury jest przekazywany jako pierwszy parametr wyżej opisanych funkcji. Początkowym stanem automatu jest `MSC_EXIT`, co oznacza, że automat nie działa. Numery kanałów są inicjowane wartościami ujemnymi, co oznacza, że kanały są nieprzydzielone.

```
static usbh_msc_data_t MSCdata = {MSC_EXIT, -1, -1};
```

Poniżej widzimy implementację funkcji `MSCsetMachine`, która instaluje omówiony automat. W tym celu najpierw inicjuje główne składowe struktury `MSCdata` (ustawia stan automatu `MSC_INIT` i niezdefiniowany wynik komunikacji `MSC_BOT_UNDEF_ERROR`), a potem wywołuje funkcję `USBHsetClassMachine`. Jeśli zainstalowanie automatu się nie powiodło, zwracamy kod błędu otrzymany z wywołania tej funkcji. Jeśli automat został zainstalowany, to czekamy, dopóki nie opuści on stanu `MSC_INIT`. Jeśli automat przeszedł do stanu `MSC_IDLE`, to inicjowanie automatu powiodło się, kanały do komunikacji z urządzeniem zostały przydzielone i automat jest gotowy do komunikacji z urządzeniem. W tym przypadku funkcja `MSCsetMachine` zwraca wartość `USBHLIB_SUCCESS`. Jeśli automat przeszedł do innego stanu (a będzie to stan `MSC_EXIT`), to znaczy, że nie udało się przydzielić kanałów i automat nie został uruchomiony. W tym przypadku funkcja `MSCsetMachine` zwraca wartość `USBHLIB_ERROR_BUSY`. Wartość opóźnienia w funkcji `Delay` jest wybrana zupełnie arbitralnie. Można poeksperymentować z innymi wartościami.

```
int MSCsetMachine(msc_configuration_t const *cfg) {
    int res;
    if ((cfg->out_ep_addr & ENDP_DIRECTION_MASK) != ENDP_OUT ||
        (cfg->in_ep_addr & ENDP_DIRECTION_MASK) != ENDP_IN)
        return USBHLIB_ERROR_INVALID_PARAM;
    MSCdata.state = MSC_INIT;
    MSCdata.status = MSC_BOT_UNDEF_ERROR;
    memcpy(&MSCdata.cfg, cfg, sizeof(msc_configuration_t));
    res = USBHsetClassMachine(MSCstateMachine, MSCatSof,
                               MSCatDisconnect, &MSCdata);
    if (res == USBHLIB_SUCCESS) {
        while (MSCdata.state == MSC_INIT)
            Delay(200);
        if (MSCdata.state != MSC_IDLE)
```

```

        res = USBHLIB_ERROR_BUSY;
    }
    return res;
}

```

Poniżej przedstawiona jest implementacja funkcji MSCisDeviceReady. Zwraca ona zero, gdy automat jest w stanie MSC_EXIT, a jedynkę w przeciwnym przypadku.

```

int MSCisDeviceReady() {
    return MSCdata.state != MSC_EXIT;
}

```

Do zainicjowania sesji protokołu BOT korzystamy z pomocniczej funkcji MSCsubmit. Jej parametry mają identyczne znaczenie jak parametry funkcji MSCBOT o takich samych nazwach. Nie zwraca ona żadnej wartości. Jej zadaniem jest zainicjowanie danych automatu, czyli struktury MSCdata. Ustawia ona m.in. stan automatu MSC_CBW_START. Przy każdym wywołaniu funkcji zwiększamy statyczną zmienną tag, aby kolejne sesje protokołu BOT miały różne wartości w polu dCBWTag. Strukturę przechowującą pakiet CSW wypełniamy w taki sposób, aby w przypadku odebrania tylko jego fragmentu ostatnie pole bCSWStatus nie miało poprawnej wartości, co spowoduje, że cały pakiet zostanie uznany za niepoprawny. Przy przypisywaniu wartości wskaźnika out_buff do składowej buffer następuje rzutowanie typu uint8_t const * na typ uint8_t *, co jest poprawne przy założeniu, że wysyłane dane nie będą modyfikowane. Składowa length dubluje informację przechowywaną w polu dCBWDataTransferLength pakietu CBW, ale jest to konieczne, gdyż potencjalnie mogą one mieć różny porządek bajtów.

```

static void MSCsubmit(uint8_t lun, uint8_t const *scsi_cmd,
                      uint32_t scsi_len, uint8_t const *out_buff,
                      uint8_t *in_buff, uint32_t len) {
    static uint32_t tag;
    memset(&MSCdata.csw, 0xFF, sizeof(msc_bot_csw_t));
    memset(&MSCdata.cbw, 0, sizeof(msc_bot_cbw_t));
    MSCdata.cbw.dCBWSignature = HTOUSBL(MSC_BOT_CBW_SIGNATURE);
    MSCdata.cbw.dCBWTag = HTOUSBL(++tag);
    MSCdata.cbw.dCBWDataTransferLength = HTOUSBL(len);
    MSCdata.cbw.bCBWIJUN = lun;
    MSCdata.cbw.bCBWCBLength = scsi_len;
    memcpy(MSCdata.cbw.CBWCB, scsi_cmd, scsi_len);
    if (in_buff) {
        MSCdata.cbw.bmCBWFlags = MSC_BOT_CBW_DATA_IN;
        MSCdata.buffer = in_buff;
    }
    else {
        MSCdata.cbw.bmCBWFlags = 0;
    }
}

```

```
    MSCdata.buffer = (uint8_t *)out_buff;
}
MSCdata.length = len;
MSCdata.transferred = 0;
MSCdata.csw_attempts = 0;
MSCdata.status = MSC_BOT_UNDEF_ERROR;
MSCdata.state = MSC_CBW_START;
}
```

Poniżej przedstawiam implementację funkcji `MSCBOT`. Jest ona analogiczna jak przedstawiona w rozdziale 7.1.7 implementacja funkcji `USBHcontrolRequest`. Jedyną istotną różnicą jest dodanie na początku sprawdzenia poprawności argumentów.

```
uint8_t MSCBOT(int synch, uint8_t lun,
                uint8_t const *scsi_cmd, uint32_t scsi_len,
                uint8_t const *out_buff, uint8_t *in_buff,
                uint32_t *len) {
    if (lun > 15 || scsi_cmd == 0 || scsi_len > MSC_CBWCB_LENGTH ||
        len == 0 || (out_buff == 0 && in_buff == 0 && *len > 0))
        return MSC_BOT_UNDEF_ERROR;
    if (synch == 0) {
        if (MSCdata.state == MSC_IDLE) {
            MSCsubmit(lun, scsi_cmd, scsi_len, out_buff, in_buff, *len);
        }
        else if (MSCdata.state == MSC_DONE) {
            MSCdata.state = MSC_IDLE;
            *len = MSCdata.transferred;
        }
    }
    else {
        uint32_t x;
        x = USBProtectInterrupt();
        if (MSCdata.state != MSC_IDLE) {
            USBUnprotectInterrupt(x);
            return USBHLIB_ERROR_BUSY;
        }
        else {
            MSCsubmit(lun, scsi_cmd, scsi_len, out_buff, in_buff, *len);
            while (MSCdata.state != MSC_DONE) {
                USBUnprotectInterrupt(x);
                Delay(200); /* How long should we wait? */
                x = USBProtectInterrupt();
            }
        }
    }
}
```

```

        MSCdata.state = MSC_IDLE;
        *len = MSCdata.transferred;
    }
    USBUnprotectInterrupt(x);
}
return MSCdata.status;
}

```

8.2.2. Protokół SCSI

usb_disk.h
usb_disk.c

Moduł *usb_disk* zawiera uproszczoną implementację protokołu SCSI. Obejmuje ona tylko absolutne minimum niezbędne do działania pamięci USB Flash. Plik nagłówkowy *usb_disk.h*, oprócz deklaracji funkcji udostępnianych przez ten moduł, definiuje numer dysku USB.

```
#define USB_DRIVE_NUMEBR 0
```

Przyjrzyjmy się teraz implementacji zawartej w pliku *usb_disk.c*. Zaczynamy od funkcji, które generują polecenia SCSI. Nazwy tych funkcji odpowiadają nazwom poszczególnych poleceń. Liczba kończąca nazwę oznacza rozmiar polecenia w bajtach (patrz tabela 6.6 w rozdziale 6.2.2). Każda z tych funkcji zwraca wskaźnik do statycznej tablicy bajtów przechowującej polecenie.

```

static uint8_t const * SCSI_TEST_UNIT_READY6(void) {
    static const uint8_t cmd[6] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00
    };
    return cmd;
}

```

Polecenie TEST UNIT READY służy do sprawdzenia, czy jednostka logiczna jest gotowa akceptować polecenia wymagające dostępu do nośnika, na którym przechowywane są dane.

```

static uint8_t const * SCSI_REQUEST_SENSE6(uint8_t len) {
    static uint8_t cmd[6] = {
        0x03, 0x00, 0x00, 0x00, 0x00, 0x00
    };
    cmd[4] = len;
    return cmd;
}

```

Jeśli jakieś polecenie zakończyło się błędem, to polecenie REQUEST SENSE pozwala odczytać szczegółowy opis tego błędu. Parametr *len* określa rozmiar opisu, który ma być zwrócony przez urządzenie.

```

static uint8_t const * SCSI_START_STOP_UNIT6(uint8_t action) {
    static uint8_t cmd[6] = {
        0x1B, 0x00, 0x00, 0x00, 0x00, 0x00
    };
    cmd[4] = action;
    return cmd;
}

```

Za pomocą polecenia START STOP UNIT wysyła się do urządzenia żądanie zmiany stanu zasilania lub żądanie załadowania (ang. *load*) bądź wysunięcia (ang. *eject*) nośnika. Jeśli kontroler chce wydać urządzeniu polecenie „wysuń”, parametr *action* powinien mieć wartość 2. Polecenie to musi być wysłane do pamięci USB Flash, aby użytkownik mógł bezpiecznie wyjąć ją z gniazda, bez ryzyka utraty danych lub powstania niespójności w systemie plików.

```

static uint8_t const * SCSI_PREVENT_ALLOW_MEDIUM_REMOVAL6(int prevent)
{
    static uint8_t cmd[6] = {
        0x1E, 0x00, 0x00, 0x00, 0x00, 0x00
    };
    cmd[4] = prevent != 0;
    return cmd;
}

```

Za pomocą polecenia PREVENT ALLOW MEDIUM REMOVAL wysyła się do urządzenia żądanie zablokowania bądź odblokowania wysuwanego nośnika. Trzeba je wysłać przed poleceniem START STOP UNIT, podając zero jako wartość parametru *prevent*.

```

static uint8_t const * SCSI_READ_CAPACITY10(void) {
    static const uint8_t cmd[10] = {
        0x25, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
    };
    return cmd;
}

```

Za pomocą polecenia READ CAPACITY odczytuje się pojemność dysku, czyli rozmiar bloku logicznego (sektora) w bajtach oraz liczbę bloków logicznych.

```

static uint8_t const * SCSI_READ10(uint32_t lba, uint16_t lbc) {
    static uint8_t cmd[10] = {
        0x28, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
    };
    cmd[2] = lba >> 24;
    cmd[3] = lba >> 16;
    cmd[4] = lba >> 8;
    cmd[5] = lba;
}

```

```

cmd[7] = lbc >> 8;
cmd[8] = lbc;
return cmd;
}

```

Polecenie READ pozwala odczytać ciąg kolejnych bloków logicznych. Parametr lba zawiera numer pierwszego bloku logicznego do odczytania. Parametr lbc określa liczbę bloków logicznych do odczytania. Wewnątrz polecenia obie wartości zapisuje się w porządku grubokońcowkowym.

```

static uint8_t const * SCSI_WRITE10(uint32_t lba, uint16_t lbc) {
    static uint8_t cmd[10] = {
        0x2A, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
    };
    cmd[2] = lba >> 24;
    cmd[3] = lba >> 16;
    cmd[4] = lba >> 8;
    cmd[5] = lba;
    cmd[7] = lbc >> 8;
    cmd[8] = lbc;
    return cmd;
}

```

Polecenie WRITE pozwala zapisać ciąg kolejnych bloków logicznych. Parametr lba zawiera numer pierwszego bloku logicznego do zapisania. Parametr lbc określa liczbę bloków logicznych do zapisania. Wewnątrz polecenia obie wartości zapisuje się w porządku grubokońcowkowym.

Globalna zmienna `msc_cfg` typu `msc_configuration_t` przechowuje konfigurację urządzenia.

```
static msc_configuration_t msc_cfg;
```

Przejdźmy teraz do omówienia funkcji, które są udostępniane przez moduł `usb_disk` i wywoływane z innych modułów, czyli `diskio` i `file_system`. Bezparametrowa funkcja `USBdiskInitialize` inicjuje i konfiguruje dysk USB, aby był gotowy do operacji odczytu i zapisu. Zwraca zero, gdy zakończyła się sukcesem, a wartość ujemną, gdy wystąpił błąd. W funkcji tej sprawdzamy najpierw za pomocą funkcji `MSCIsDeviceReady`, czy działa automat obsługujący komunikację z urządzeniem. Jeśli automat działa, to wywołujemy funkcję `USBdiskStatus`, aby sprawdzić, czy dysk USB odpowiada na polecenia SCSI. Jeśli automat nie działa, to próbujemy go uruchomić. Najpierw za pomocą funkcji `USBHopenDevice` sprawdzamy, czy do gniazda USB jest podłączone jakieś urządzenie. Funkcja `CheckDeviceDescriptor` sprawdza poprawność odczytanego deskryptora urządzenia. Pomijam opis implementacji tej funkcji. Jeśli do gniazda USB podłączone jest urządzenie, to za pomocą funkcji `USBHgetConfDescriptor` odczytujemy jego deskryptor konfiguracji, a następnie całą konfigurację. Funkcja `ParseConfiguration` sprawdza poprawność odczytanej konfiguracji. Sprawdza też, czy jest to konfiguracja pamięci masowej.

Jeśli sprawdzenia wypadły pozytywnie, to wypełnia strukturę `msc_cfg` opisującą konfigurację urządzenia. Pomijam opis implementacji tej funkcji. Mając odczytaną konfigurację urządzenia, ustawiamy aktywną konfigurację, wywołując funkcję `USBHsetConfiguration`. Za pomocą funkcji `MSCgetMaxLun` odczytujemy maksymalny numer jednostki logicznej. Wywołanie to nie jest konieczne, gdyż w dalszym ciągu zakładamy, że dysk ma tylko jedną jednostkę logiczną o numerze zero. Jeśli wszystkie dotychczasowe wywołania zakończyły się powodzeniem, to komunikacja USB z urządzeniem działa poprawnie i możemy za pomocą funkcji `MSCsetMachine` zainstalować automat obsługujący protokół BOT. Po poprawnym zainstalowaniu automatu sprawdzamy za pomocą funkcji `USBdiskStatus`, czy dysk odpowiada na polecenia SCSI. Jeśli dysk odpowiada na te polecenia, to na koniec odczytujemy rozmiar sektora, wysyłając polecenie SCSI READ CAPACITY za pomocą funkcji `MSCBOT`.

```
int USBdiskInitialize() {
    usb_device_descriptor_t *dev_desc;
    usb_configuration_descriptor_t *cfg_desc;
    scsi_capacity_data_t *disk_capacity;
    uint32_t length;
    uint16_t len16;
    uint8_t status, temp[BUFF_LEN];
    if (MSCisDeviceReady())
        return USBdiskStatus();
    memset(&temp, 0, BUFF_LEN);
    dev_desc = (usb_device_descriptor_t *)temp;
    if (USBHopenDevice(&msc_cfg.speed, &msc_cfg.dev_addr, dev_desc,
                       TIMEOUT) < 0)
        return -1;
    if (CheckDeviceDescriptor(dev_desc) < 0)
        return -1;
    memset(&temp, 0, BUFF_LEN);
    len16 = sizeof(usb_configuration_descriptor_t);
    if (USBHgetConfDescriptor(1, 0, temp, len16) < 0)
        return -1;
    cfg_desc = (usb_configuration_descriptor_t *)temp;
    len16 = USBTOHS(cfg_desc->wTotalLength);
    if (len16 > BUFF_LEN)
        return -1;
    memset(&temp, 0, BUFF_LEN);
    if (USBHgetConfDescriptor(1, 0, temp, len16) < 0)
        return -1;
    if (ParseConfiguration(&msc_cfg, temp, len16) < 0)
        return -1;
    if (USBHsetConfiguration(1, msc_cfg.configuration_value) < 0)
```

```

        return -1;
    if (MSCgetMaxLun(1, msc_cfg.interface_number, &msc_cfg.max_lun) < 0)
        return -1;
    if (MSCsetMachine(&msc_cfg) < 0)
        return -1;
    if (USBdiskStatus() < 0)
        return -1;
    length = sizeof(scsi_capacity_data_t);
    status = MSCBOT(1, 0, SCSI_READ_CAPACITY10(), 10, 0,
                    temp, &length);
    if (status != MSC_BOT_CSW_COMMAND_PASSED)
        return -1;
    if (length != sizeof(scsi_capacity_data_t))
        return -1;
    disk_capacity = (scsi_capacity_data_t *)temp;
    length = SCSITOHL(disk_capacity->block_length);
    if (length > 0x1000)
        return -1;
    else
        msc_cfg.sector_size = length;
    return 0;
}

```

Bezparametrowa funkcja `USBdiskStatus` sprawdza, czy dysk USB odpowiada na polecenia SCSI i jest gotowy do operacji odczytu lub zapisu. Funkcja ta zwraca zero, gdy zakończyła się sukcesem, a wartość ujemną, gdy wystąpił jakiś błąd. Sprawdzenie polega na wysłaniu polecenia SCSI TEST UNIT READY. Jeśli polecenie to zostało wykonane poprawnie, czyli urządzenie odpowiedziało statusem `MSC_BOT_CSW_COMMAND_PASSED`, to dysk jest gotowy. Jeśli polecenie zostało odrzucone, czyli urządzenie odpowiedziało statusem `MSC_BOT_CSW_COMMAND_FAILED`, to wysyłamy polecenie SCSI REQUEST SENSE, aby odczytać przyczynę odrzucenia poprzedniego polecenia. Przy czym ignorujemy odczytaną przyczynę błędu. Stała `SCSI_SENSE_DATA_LEN` ma wartość 18. Powyższą sekwencję powtarzamy co najwyżej dwukrotnie. Jeśli po pierwszej próbie któreś z polecień zakończyło się statusem `MSC_BOT_CSW_PHASE_ERROR`, to najprawdopodobniej nastąpiło rozsynchronizowanie się protokołu BOT i próbujemy go ponownie zsynchronizować, wywołując funkcję `MSCresetRecovery`.

```

int USBdiskStatus() {
    int i;
    uint32_t length;
    uint8_t status, temp[SCSI_SENSE_DATA_LEN];
    for (i = 0; i < 2; ++i) {
        length = 0;
        status = MSCBOT(1, 0, SCSI_TEST_UNIT_READY6(),

```

```

        6, 0, 0, &length);
if (status == MSC_BOT_CSW_COMMAND_PASSED) {
    return 0;
}
else if (status == MSC_BOT_CSW_COMMAND_FAILED) {
    length = SCSI_SENSE_DATA_LEN;
    status = MSCBOT(1, 0, SCSI_REQUEST_SENSE6(SCSI_SENSE_DATA_LEN),
                    6, 0, temp, &length);
}
if (i == 0 && status == MSC_BOT_CSW_PHASE_ERROR) {
    if (MSCresetRecovery(1, msc_cfg.interface_number,
                         msc_cfg.in_ep_addr,
                         msc_cfg.out_ep_addr) < 0) {
        return -1;
    }
}
else if (status != MSC_BOT_CSW_COMMAND_PASSED &&
         status != MSC_BOT_CSW_COMMAND_FAILED) {
    return -1;
}
return -1;
}

```

Funkcja `USBdiskRead` wykonuje operację odczytu danych z dysku za pomocą polecenia SCSI DISK READ. Parametr `buff` jest wskaźnikiem na bufor, do którego mają zostać skopiowane odczytane dane. Parametr `sector` określa numer pierwszego odczytywanego sektora. Parametr `count` zawiera liczbę sektorów do odczytania. Funkcja ta zwraca zero, gdy wszystko przebiegło poprawnie, a wartość ujemną, gdy wystąpił błąd. Ponieważ użyty wariant polecenia DISK READ zapisuje liczbę odczytywanych sektorów jako wartość 16-bitową, próba odczytania większej liczby sektorów niż 65 535 (szesnastkowo 0xFFFF) musi zakończyć się błędem.

```

int USBdiskRead(uint8_t *buff, uint32_t sector, uint32_t count) {
    uint32_t length1, length2;
    uint8_t status;
    if (count > 0xFFFF)
        return -1;
    length1 = length2 = count * msc_cfg.sector_size;
    status = MSCBOT(1, 0, SCSI_READ10(sector, count), 10,
                   0, buff, &length1);
    return status != MSC_BOT_CSW_COMMAND_PASSED ||
           length1 != length2 ? -1 : 0;
}

```

Funkcja `USBdiskWrite` wykonuje operację zapisu danych na dysku za pomocą polecenia SCSI DISK WRITE. Parametr `buff` jest wskaźnikiem na bufor z danymi do zapisania. Parametr `sector` określa numer pierwszego zapisywanej sektora. Parametr `count` zawiera liczbę sektorów do zapisania. Funkcja ta zwraca zero, gdy wszystko przebiegło poprawnie, a wartość ujemną, gdy wystąpił błąd. Ponieważ użyty wariant polecenia DISK WRITE zapisuje liczbę sektorów jako wartość 16-bitową, to próba zapisania większej liczby sektorów niż 65 535 (szesnastkowo 0xFFFF) musi zakończyć się błędem.

```
int USBdiskWrite(uint8_t const *buff, uint32_t sector, uint32_t count)
{
    uint32_t length1, length2;
    uint8_t status;
    if (count > 0xFFFF)
        return -1;
    length1 = length2 = count * msc_cfg.sector_size;
    status = MSCBOT(1, 0, SCSI_WRITE10(sector, count), 10,
                    buff, 0, &length1);
    return status != MSC_BOT_CSW_COMMAND_PASSED ||
           length1 != length2 ? -1 : 0;
}
```

Zadaniem bezparametrowej funkcji `USBdiskSync` jest spowodowanie, aby dysk zakończył wszystkie operacje zapisu, czyli zsynchronizował swój stan ze stanem wynikającym z dotychczasowej sekwencji operacji zapisu. Podejrzenie, co w takiej sytuacji robią popularne systemy operacyjne, pokazało, że należy wywołać polecenie SCSI TEST UNIT READY za pomocą funkcji `USBdiskStatus`. Obie funkcje zwracają zero, gdy wszystko przebiegło poprawnie, a wartość ujemną, gdy wystąpił błąd.

```
int USBdiskSync() {
    return USBdiskStatus();
}
```

Funkcja `USBdiskGetSectorSize` zapisuje w zmiennej wskazanej parametrem `sector_size` rozmiar sektora odczytany podczas inicjowania komunikacji z dyskiem. Funkcja ta zwraca zero, gdy rozmiar dysku jest znany, a wartość ujemną, gdy automat nie działa, czyli gdy rozmiar sektora przechowywany w strukturze opisującej parametry urządzenie nie jest wiarygodny.

```
int USBdiskGetSectorSize(uint16_t *sector_size) {
    if (!MSCisDeviceReady())
        return -1;
    *sector_size = msc_cfg.sector_size;
    return 0;
}
```

Bezparametrowa funkcja `USBdiskAllowEject` jest wywoływana, gdy po zakończeniu wszystkich operacji na pamięci USB Flash chcemy pozwolić użytkownikowi wyjąć ją z gniazda USB. Zwraca ona zero, gdy wszystko przebiegło poprawnie, a wartość ujemną, gdy wystąpił błąd. Stała `SCSI_EJECT` ma wartość 2.

```
int USBdiskAllowEject() {
    uint32_t length;
    uint8_t status1, status2;
    length = 0;
    status1 = MSCBOT(1, 0, SCSI_PREVENT_ALLOW_MEDIUM_REMOVAL6(0), 6,
                     0, 0, &length);
    length = 0;
    status2 = MSCBOT(1, 0, SCSI_START_STOP_UNIT6(SCSI_EJECT), 6,
                     0, 0, &length);
    return status1 != MSC_BOT_CSW_COMMAND_PASSED ||
           status2 != MSC_BOT_CSW_COMMAND_PASSED ? -1 : 0;
}
```

8.3. Przykład użycia

W ostatnim podrozdziale przedstawiam przykładowy program demonstrujący użycie dysku USB, czyli pamięci USB Flash.

8.3.1. Program demonstrujący

ex_msc_host.c

W pliku `ex_msc_host.c` znajduje się funkcja `main` przykładowego programu. Zaczynamy jak zwykle od odczytania parametrów, z jakimi ma być uruchomiona aplikacja, a potem skonfigurowania portów wejścia-wyjścia, diod świecących i systemu przerwań. Potem konfigurujemy po kolej układy taktujące mikrokontrolera, wyświetlacz ciekłokrystaliczny i na koniec układ peryferyjny USB w trybie kontrolera.

```
int main(void) {
    int clk;
    usb_phy_t phy;
    GetBootParams(&clk, 0, &phy);
    AllPinsDisable();
    LEDconfigure();
    RedLEDOn();
    IRQprotectionConfigure();
    ErrorResetable(ClockConfigure(clk), 2);
```

```
    ErrorResetable(LCDconfigure(), 3);
    ErrorResetable(USBHconfigure(phy), 5);
    ...
}
```

W drugiej części funkcji main wykonujemy w nieskończonej pętli właściwe testy. Na początku używamy funkcji USBHisDeviceReady wywoływanej w pętli while, aby poczekać na podłączenie urządzenia do gniazda USB. Jeśli kontroler wykryje podłączone urządzenie, sprawdzamy, czy jest to pamięć USB Flash, próbując zamontować system plików za pomocą funkcji FileSystemMount. Jeśli jej wywołanie zakończyło się powodzeniem, to wykonujemy właściwy test, a potem odmontujemy system plików za pomocą funkcji FileSystemUnmount. Na koniec ponownie wywołujemy funkcję USBHisDeviceReady w pętli while, tym razem, aby poczekać na wyjęcie urządzenia (pamięci Flash) z gniazda USB. Stała TIMEOUT ma wartość 1000, ale nie jest to wartość krytyczna.

```
int main(void) {
    ...
    for (;;) {
        printf("Waiting for device...\n");
        while (!USBHisDeviceReady())
            Delay(TIMEOUT);
        LCDclear();
        if (FileSystemMount() < 0) {
            fprintf(stderr, "mount failed\n");
        }
        else {
            int n;
            n = ReadTestNumber("testnum.txt");
            printf("Test %d\n", n);
            BiggerTest(n, "testbig", "w");
            WriteTestNumber(n, "testlog.txt", "a");
            BiggerTest(n, "testbig", "r");
            WriteTestNumber(n, "testnum.txt", "w");
            if (FileSystemUnmount() < 0)
                fprintf(stderr, "umount failed\n");
            else
                printf("Device can be removed.\n");
        }
        while (USBHisDeviceReady())
            Delay(TIMEOUT);
        LCDclear();
    }
}
```

Właściwy test składa się z kilku operacji plikowych. Za pomocą funkcji `ReadTestNumber` czytamy numer testu, czyli liczbę zapisaną w pliku `testnum.txt`. Funkcja ta zwraca tę liczbę zwiększoną o jeden lub jedynkę, gdy plik nie istnieje. Za pomocą funkcji `BiggerTest` zapisujemy dużą porcję danych do pliku `testbig`. Jeśli plik ten istniał już przed rozpoczęciem testu, to jest zastępowany nowym. Zapisywany ciąg danych zależy od wartości pierwszego argumentu przekazanego tej funkcji. Za pomocą funkcji `WriteTestNumber` dopisujemy numer wykonanego testu na końcu pliku `testlog.txt`. Ponownie wywołujemy funkcję `BiggerTest`, tym razem w celu odczytania pliku `testbig`, aby sprawdzić, czy jego zawartość zgadza się z zapisanymi poprzednio danymi. Nowy numer przeprowadzonego testu zapisujemy do pliku `testnum.txt` za pomocą funkcji `WriteTestNumber`. Dla porządku poniżej zamieszczam wydruki wspomnianych funkcji.

```
static int ReadTestNumber(char const *filename) {
    FILE *file;
    int testnum;
    file = fopen(filename, "r");
    if (file) {
        if (fscanf(file, "%d", &testnum) == 1) {
            ++testnum;
        }
        else {
            testnum = 1;
            fprintf(stderr, "fscanf %.12s failed\n", filename);
        }
        if (fclose(file) != 0) {
            fprintf(stderr, "fclose %.12s failed\n", filename);
        }
    }
    else {
        testnum = 1;
    }
    return testnum;
}

static void WriteTestNumber(int testnum, char const *filename,
                           char const *mode) {
    FILE *file;
    file = fopen(filename, mode);
    if (file == 0) {
        fprintf(stderr, "fopen %.12s failed\n", filename);
    }
}
```

```
else {
    if (fprintf(file, "%d\n", testnum) < 0) {
        fprintf(stderr, "fprintf %.12s failed\n", filename);
    }
    if (fclose(file) != 0) {
        fprintf(stderr, "fclose %.12s failed\n", filename);
    }
}

static void BiggerTest(int testnum, char const *filename,
                      char const *mode) {
    static const size_t size[4] = {1024, 1024 + 76, 1024 - 41,
                                 1024 - 35};

    static char buff[1024 + 76];
    FILE *file;
    size_t i, j, s, r;
    char c;
    file = fopen(filename, mode);
    if (file == 0) {
        fprintf(stderr, "fopen %.12s failed\n", filename);
    }
    else {
        for (i = 0; i < 17; ++i) {
            s = size[(i + testnum) & 3];
            c = 5 * i + testnum;
            if (mode[0] == 'w') {
                memset(buff, c, s);
                r = fwrite(buff, 1, s, file);
                if (r != s) {
                    fprintf(stderr, "fwrite %.12s failed\n", filename);
                    break;
                }
            }
            else if (mode[0] == 'r') {
                r = fread(buff, 1, s, file);
                if (r != s) {
                    fprintf(stderr, "fread %.12s failed\n", filename);
                    break;
                }
            }
        }
    }
}
```

```
        for (j = 0; j < s; ++j) {
            if (buff[j] != c) {
                fprintf(stderr, "compare %.12s failed\n", filename);
                break;
            }
        }
        putchar('.');
        fflush(stdout);
    }
    putchar('\n');
    if (fclose(file) != 0) {
        fprintf(stderr, "fclose %.12s failed\n", filename);
    }
}
}
```

8.3.2. Kompilowanie i testowanie

Archiwum z przykładami zawiera dwie wersje sprzętowe projektu kontrolera pamięci masowej. Podobnie jak projekt z rozdziału 7 był on testowany na zestawie prototypowym ZL29ARM z wyświetlaczem graficznym WG12864A oraz na zestawie prototypowym STM3220G-EVAL, który ma wbudowany wyświetlacz graficzny. W katalogu *.make* znajdują się podkatalogi, których nazwa rozpoczyna się przedrostkiem *usb8_msc_host* i w których umieszczone są pliki *makefile* umożliwiające skompilowanie projektu. Do tego projektu odnoszą się również zamieszczone w rozdziale 7.2.3 uwagi dotyczące dostosowania go do innego sprzętu, konfigurowania zestawów prototypowych (ustawienia zwór, wybór interfejsu USB i częstotliwości taktowania mikrokontrolera) oraz sygnalizowania stanu kontrolera i błędów przez diody świecące.

Podeczas testu program tworzy lub modyfikuje w głównym katalogu dysku USB Flash trzy pliki: *testnum.txt*, *testlog.txt* i *testbig*. Zawartość tych plików można sobie oczywiście obejrzeć, podłączając pamięć do komputera osobistego. Jeśli żadne urządzenie nie jest podłączone do kontrolera, na wyświetlaczu ciekłokrystalicznym zestawu prototypowego pojawia się napis:

```
Waiting for device...
```

Po podłączeniu urządzenia napis ten znika, a po poprawnym zakończeniu testu na ekranie wyświetlacza ciekłokrystalicznego powinniśmy zobaczyć jego numer oraz informację, że urządzenie może być wyjęte z gniazda USB:

Test 20

.....
.....

Device can be removed.

Jeśli wystąpił błąd, to wyświetlany jest odpowiedni komunikat. Formaty tych komunikatów są zaszyte w tekście źródłowym funkcji przedstawionych w poprzednim podrozdziale.

Celem tego dodatku jest pokazanie, jak pod systemem Linux zainstalować minimalny zestaw programów narzędziowych i bibliotek, aby kompilować programy dla procesorów ARM z rdzeniami Cortex-M0, Cortex-M3 i Cortex-M4. Nie ma oczywiście problemu ze ściągnięciem z Internetu gotowych pakietów, które instaluje się kilkoma kliknięciami myszy. Często są to jednak wersje limitowane (ograniczony rozmiar generowanego pliku wykonywalnego, ograniczony czas używania lub tym podobne), a za ich pełną wersję trzeba zapłacić. Ponadto dobrze jest wiedzieć, jak to jest zrobione, a samodzielna instalacja wcale nie jest bardzo trudna. Pozwala też na indywidualne dostrojenie wielu parametrów, czego efektem możliwe być istotne zmniejszenie rozmiaru generowanego kodu wykonywalnego, co bywa bardzo pożądane w aplikacjach wbudowanych. Eksperymenty z programami zamieszczonymi w tej książce pokazały, że kompilując je za pomocą narzędzi skonfigurowanych według niniejszej instrukcji, uzyskuje się zmniejszenie rozmiaru kodu wynikowego od 10 do 25 procent w stosunku do jednego z popularnych komercyjnych zestawów narzędzi (przy ustawieniu tych samych opcji optymalizacji). Jest to dość istotne zmniejszenie, które pośrednio poprzez zastosowanie mikrokontrolera o mniejszej pamięci może przełożyć się na zmniejszenie końcowego kosztu wytwarzania projektowanego urządzenia.

We wszystkich poniższych poleceniach konfiguratora (polecenie `configure`) opcja `--prefix` określa miejsce, gdzie zostaną zainstalowane programy i biblioteki. Można je zainstalować w katalogu domowym użytkownika (poniższe przykłady dotyczą użytkownika o nazwie `user`), podając na przykład

```
--prefix=/home/user/arm
```

lub w jakimś katalogu systemowym, podając na przykład

```
--prefix=/usr/local/arm
```

Jeśli instalujemy programy w katalogu systemowym, do którego modyfikowania potrzebne są prawa administratora (ang. `root`), to wszystkie poniższe polecenia `make install` powinny wykonywać się z uprawnieniami administratora, dlatego należy poprzedzić je polecienniem `sudo` (i podać hasło), czyli

```
sudo make install
```

Zakładamy, że w bieżącym katalogu mamy następujące paczki (można i zwykle należy oczywiście użyć najnowszych wersji tych pakietów):

- `binutils-2.22.tar.gz`,
- `gcc-4.6.3.tar.gz`,
- `newlib-1.20.0.tar.gz`,
- `libftdi-0.20.tar.gz`,
- `openocd-0.6.1.tar.gz`.

Binutils

Instalowanie środowiska programistycznego należy rozpocząć od pakietu Binutils. Jego najnowszą wersję można znaleźć na stronie <http://ftp.gnu.org/gnu/binutils>.

Rozpakowujemyściągniętą paczkę. W katalogu, w którym się rozpakowała, tworzymy podkatalog *build*. W tym podkatalogu będą tworzone wszystkie pliki tymczasowe i docelowe. Dzięki temu, gdy coś pójdzie źle, można ten podkatalog po prostu skasować i rozpocząć instalowanie od nowa, ewentualnie ze zmienionymi parametrami. Cały proces instalowania przedstawia poniższy wydruk.

```
tar -zxf binutils-2.22.tar.gz
cd binutils-2.22
mkdir build
cd build
../configure --target=arm-elf --prefix=/home/user/arm
make
make install
cd ../../
```

Instalujemy kompilator skroşny (ang. *cross compiler*) – kompilowanie naszych programów będzie wykonywane na komputerze o innej architekturze niż architektura procesora, na której ma być wykonywany skompilowany program. Dlatego za pomocą parametru *--target* określa się docelową architekturę, na którą będą generowane pliki wynikowe. Ponadto, aby umożliwić jednokrotnie korzystanie z wielu zestawów narzędzi, nazwy programów wykonywalnych będą miały przedrostek określony za pomocą tego parametru. Żeby móc wywoływać zainstalowane programy, trzeba dodać do ścieżek poszukiwania plików katalog określony parametrem *--prefix*. Najprościej można to zrobić, dodając odpowiedni wpis na końcu pliku *.bash_profile* (w niektórych dystrybucjach *.profile* – uwaga na kropkę rozpoczynającą nazwę pliku), który znajduje się w katalogu domowym użytkownika. Przykładowy wpis wygląda następująco:

```
PATH=$PATH:/home/user/bin:/home/user/arm/bin:/home/user/msp430/bin
export PATH
```

Zmiana będzie widoczna dopiero po ponownym wykonaniu skryptu *.bash_profile* (na przykład po wylogowaniu i ponownym zalogowaniu użytkownika). Na czas instalowania kolejnych składników środowiska programistycznego wystarczy wykonać następujące polecenie:

```
export PATH=$PATH:/home/user/arm/bin
```

Należy koniecznie pamiętać o dodaniu we wszystkich ścieżkach podkatalogu *bin*.

GCC

Kolejnym pakietem, który trzeba zainstalować, jest GCC (ang. *GNU Compiler Collection*). Adresy serwerów oferujących GCC dostępne są na stronie <http://gcc.gnu.org/mirrors.html>. Ze względu na występujący w rdzeniach Cortex-M3 drobny błąd w niektórych wariantach instrukcji *ldr*, należy użyć wersji co najmniej 4.4.0. Od tej wersji dla Cortex-M3 domyślnie ustawiona jest opcja *-mfix-cortex-m3-lldr*, która zapobiega używaniu przez kompilator wadliwie działających instruk-

cji. Cortex-M0 jest wspierany od wersji 4.5.0, a Cortex-M4 – od wersji 4.6.0. Pościągnięciu pakietu należy go rozpakować:

```
tar -zxf gcc-4.6.3.tar.gz
cd gcc-4.6.3
```

Procesory ARM używają trzech zestawów instrukcji: ARM, Thumb i Thumb-2. Dla każdego z tych zestawów instrukcji trzeba mieć oddzielnny zestaw bibliotek. Umożliwia to opcja `--enable-multilib` konfiguratora (program `configure`). Nie trzeba jej jednak podawać, gdyż jest to opcja domyślna. Natomiast w pliku `gcc-4.6.3/gcc/config/arm/t-arm-elf` musimy zdefiniować, jakie wersje bibliotek mają zostać skompilowane. Dla wygody zmodyfikowany plik `t-arm-elf` umieszczony jest w katalogu `.make/linux` w archiwum z przykładami. W stosunku do oryginału zmodyfikowane są w nim wiersze z parametrami `MULTILIB`:

```
MULTILIB_OPTIONS      = mthumb
MULTILIB_DIRNAMES    = thumb
MULTILIB_OPTIONS      += mcpu=cortex-m0/mcpu=cortex-m3/mcpu=cortex-m4
MULTILIB_DIRNAMES    += cortex-m0 cortex-m3 cortex-m4
MULTILIB_OPTIONS      += mfloat-abi=softfp
MULTILIB_DIRNAMES    += soft-fp-abi
MULTILIB_OPTIONS      += mfpu=fpv4-sp-d16
MULTILIB_DIRNAMES    += sp-fpu
MULTILIB_EXCEPTIONS   = mthumb
MULTILIB_EXCEPTIONS   += mcpu=cortex-m*
MULTILIB_EXCEPTIONS   += mfloat-abi=softfp*
MULTILIB_EXCEPTIONS   += mfpu=fpv4-sp-d16*
MULTILIB_EXCEPTIONS   += mthumb/mfloat-abi=softfp*
MULTILIB_EXCEPTIONS   += mthumb/mfpu=fpv4-sp-d16*
MULTILIB_EXCEPTIONS   += mthumb/mcpu=cortex-m0/mfloat-abi=softfp*
MULTILIB_EXCEPTIONS   += mthumb/mcpu=cortex-m3/mfloat-abi=softfp*
MULTILIB_EXCEPTIONS   += mthumb/mcpu=cortex-m0/mfpu=fpv4-sp-d16*
MULTILIB_EXCEPTIONS   += mthumb/mcpu=cortex-m3/mfpu=fpv4-sp-d16*
MULTILIB_EXCEPTIONS   += mthumb/mcpu=cortex-m4/mfloat-abi=softfp
MULTILIB_EXCEPTIONS   += mthumb/mcpu=cortex-m4/mfpu=fpv4-sp-d16
MULTILIB_MATCHES      =
MULTILIB_OSDIRNAMES   = mthumb/mcpu.cortex-m0!=cortex-m0
MULTILIB_OSDIRNAMES   += mthumb/mcpu.cortex-m3!=cortex-m3
MULTILIB_OSDIRNAMES   += mthumb/mcpu.cortex-m4!=cortex-m4
MULTILIB_OSDIRNAMES   += mthumb/mcpu.cortex-m4/mfloat-abi.softfp/ \
                         mfpu.fpv4-sp-d16!=cortex-m4/fpu
```

Ponieważ w przykładowych projektach korzystamy z własnego kodu startowego (plik `startup_stm32.c`), a program uruchomiony na mikrokontrolerze działa aż do wyłączenia zasilania, nie potrzebujemy dostarczanego wraz z pakietem GCC kodu,

który wykonuje się po uruchomieniu (ang. *runtime begin*) i po zakończeniu programu (ang. *runtime end*). Dlatego w pliku *t-arm-elf* należy zakomentować poniższe wiersze:

```
# EXTRA_MULTILIB_PARTS = crtbegin.o crtend.o crtio.o crtn.o
# EXTRA_PARTS = crtbegin.o crtend.o crtio.o crtn.o
```

Ustawione jak wyżej parametry *MULTILIB* spowodują, że w domyślnym dla bibliotek katalogu znajdą się ich wersje używające zestawu instrukcji ARM, w jego podkatalogach *cortex-m0*, *cortex-m3*, *cortex-m4* wersje używające zestawu Thumb-2 odpowiednio dla rdzeni Cortex-M0, Cortex-M3, Cortex-M4. W podkatalogu *cortex-m4/fpu* zostaną umieszczone biblioteki używające jednostki zmienoprzecinkowej (ang. *floating point unit*) rdzenia Cortex-M4, a w podkatalogu *cortex-m4* będą biblioteki z programową emulacją operacji zmienoprzecinkowych.

Konsolidator (ang. *linker*) będzie dołączał właściwe biblioteki na podstawie argumentów podanych w linii poleceń (konsolidatora lub kompilatora). Domyślnie będą dołączane biblioteki używające zestawu instrukcji ARM. Zastosowanie łączenie opcji *-mthumb* oraz *-mcpu=cortex-m** sprawi, że dołączone zostaną biblioteki używające zestawu instrukcji Thumb-2 dla odpowiedniego rdzenia. Nazwy opcji kompilatora (z pominieciem początkowego myślnika) specyfikuje się za pomocą parametru *MULTILIB_OPTIONS*.

Nazwy podkatalogów, w których mają być umieszczane odpowiadające im wersje bibliotek, wymienia się za pomocą parametru *MULTILIB_DIRNAMES*. Prowadzi to jednak do utworzenia długiego ciągu podkatalogów. Parametr *MULTILIB_OSDIRNAMES* pozwala spłaszczyć strukturę podkatalogów. Parametr *MULTILIB_EXCEPTIONS* opisuje zabronione kombinacje parametrów kompilatora. Na przykład GCC nie dopuszcza użycia samej opcji *-mcpu=cortex-m**, bez opcji *-mthumb*.

Po skonfigurowaniu wersji bibliotek można przystąpić do instalowania kompilatora skróstnego:

```
mkdir build
cd build
../configure --target=arm-elf --prefix=/home/user/arm
              --enable-languages=c,c++ --with-newlib
              --without-headers --disable-shared
```

Parametr *--enable-languages* określa języki programowania, dla których mają zostać zbudowane kompilatory. Ustawienie łącznie opcji *--with-newlib* i *--without-headers* sprawia, że biblioteka LIBGCC zostanie zbudowana bez wsparcia jakichkolwiek plików nagłówkowych, czyli nie będzie ona korzystać z żadnej innej biblioteki, natomiast pozostałe biblioteki będą mogły korzystać z biblioteki Newlib i jej plików nagłówkowych. Takie ustawienia są typowe, gdy buduje się kompilator dla systemów wbudowanych. Opcja *--disable-shared* wyłącza używanie bibliotek dzielonych, ładowanych dynamicznie, co w przypadku mikrokontrolerów nie miałoby po prostu sensu.

Aby skompilować GCC, trzeba mieć zainstalowane biblioteki GMP, MPFR i MPC w wersji rozwojowej (ang. *devel*), gdyż potrzebne są pliki nagłówkowe: *gmp.h*,

mpfr.h i *mpc.h*. W razie potrzeby należy je najpierw doinstalować – większość dystrybucji Linuksa dostarcza wygodnego programu graficznego umożliwiającego łatwe dodawanie pakietów. Za pomocą opcji *--with-gmp*, *--with-mpfr*, *--with-mpc* można wskazać miejsce zainstalowania odpowiedniej biblioteki, jeśli nie jest to miejsce standardowe.

Najpierw komplikujemy i instalujemy sam kompilator:

```
make all-gcc
make install-gcc
cd ../../
```

Poprawność dotychczasowych etapów instalacji można sprawdzić za pomocą dwóch poniższych poleceń. Polecenie *arm-elf-gcc -v* powinno wypisać coś następującego:

```
Using built-in specs.
COLLECT_GCC=arm-elf-gcc
COLLECT_LTO_WRAPPER=/home/user/arm/libexec/gcc/arm-elf/4.6.3/lto-wrapper
Target: arm-elf
Configured with: ../configure --target=arm-elf
--prefix=/home/user/arm --enable-languages=c,c++
--with-newlib --without-headers --disable-shared
Thread model: single
gcc version 4.6.3 (GCC)
```

Natomiast polecenie *arm-elf-gcc -print-multi-lib* powinno wypisać coś takiego:

```
.;
cortex-m0;@mthumb@mcpu=cortex-m0
cortex-m3;@mthumb@mcpu=cortex-m3
cortex-m4;@mthumb@mcpu=cortex-m4
cortex-m4/fpu;@mthumb@mcpu=cortex-m4@mfloat-abi=softfp@mcpu=fpv4-sp-d16
```

Biblioteka standardowa języka C

W kolejnym kroku instalujemy biblioteki. Potrzebujemy standardowej biblioteki języka C. Jedną z najpowszechniej stosowanych dla systemów wbudowanych jest biblioteka Newlib. Można jąściągnąć z serwera [ftp://sources.redhat.com/pub/newlib/index.html](http://sources.redhat.com/pub/newlib/index.html). Rozpakowanie, skonfigurowanie, skompilowanie i zainstalowanie tej biblioteki przebiega jak na poniższym wydruku.

```
tar -zxf newlib-1.20.0.tar.gz
cd newlib-1.20.0
mkdir build
cd build
```

```
../configure --target=arm-elf --prefix=/home/user/arm  
          --disable-shared --disable-newlib-supplied-syscalls  
make CFLAGS_FOR_TARGET="-O2 -ffunction-sections -fdata-sections  
          -D__BUFSIZ__=256"  
make install  
cd ../../
```

Polecenie `make` zakończy się niepowodzeniem, gdy zechcemy skompilować *runtime* dostarczony z GCC. Można je wtedy zastąpić następującym poleceniem:

```
make CFLAGS_FOR_TARGET="-O2 -fdata-sections -D__BUFSIZ__=256"
```

Parametr `--disable-newlib-supplied-syscalls` konfiguratora mówi, że musimy sami dostarczyć implementację niektórych wywołań systemowych (patrz plik `syscalls_dummy.c` w archiwum z przykładami). Parametr `CFLAGS_FOR_TARGET` polecenia `make` określa opcje, które mają być użyte podczas kompilowania biblioteki. Parametr `-D__BUFSIZ__` jest specyficzny dla biblioteki Newlib i określa domyślny rozmiar bufora przydzielanego dla plików otwieranych za pomocą funkcji `fopen`. Poziom optymalizacji kodu wynikowego nie jest ustawiony domyślnie, dlatego należy go podać. W powyższym przykładzie użyto parametrów `-O2`, `-ffunction-sections` i `-fdata-sections`. Można też wypróbować inne opcje optymalizacji. Eksperymenty pokazały, że zastosowanie parametru `-fomit-frame-pointer` nie wpływa na rozmiar pliku wynikowego.

Pozostało jeszcze dokończenie instalowania GCC. W tym kroku instalujemy pomocnicze biblioteki dostarczane wraz z GCC. Najważniejszą z nich jest biblioteka LIBGCC. Przy jej kompilowaniu również dobrze jest ustawić pożądany poziom optymalizacji. Tym razem parametr `-O2` jest ustawiany domyślnie, ale nic nie przeszkadza, by go tu podać. Odpowiednie wywołania wyglądają następująco:

```
cd gcc-4.6.3/build  
make all CFLAGS_FOR_TARGET="-O2 -ffunction-sections -fdata-sections"  
make install  
cd ../../
```

OpenOCD

Do programowania pamięci Flash można użyć programu OpenOCD (ang. *Open On-Chip Debugger*). Współpracuje on z wieloma adapterami JTAG i programatorami, ale przed jego skompilowaniem trzeba skonfigurować, które z nich mają być obsługiwane. Jako przykład wybierzmy adaptery JTAG podłączane do USB i zbudowane na układzie FT2232 lub FT2232H oraz programatory ST-LINK/V1 i ST-LINK/V2 też podłączane do USB. W tym przypadku do zainstalowania OpenOCD potrzebna jest biblioteka libusb w wersji rozwojowej (ang. *devel*) – jest ona standardowo dostępna w większości dystrybucji Linuksa. Adaptery z układami FT2232 lub FT2232H wymagają ponadto biblioteki libftdi w wersji co najmniej 0.16. Wiele dystrybucji Linuksa pozwala na automatyczne doinstalowanie tej biblioteki. Jeśli jej nie ma, to można ją ściągnąć ze strony <http://www.intra2net.com/en/developer/libftdi/download.php>, a jej instalowanie przebiega następująco:

```
tar -xzf libftdi-0.20.tar.gz
cd libftdi-0.20
mkdir build
cd build
../configure --prefix=/home/user/arm
make
make install
cd ../../
```

Wersję źródłową OpenOCD można ściągnąć ze strony <http://sourceforge.net/projects/openocd/files/openocd>. Instalowanie tego pakietu wygląda następująco:

```
tar -xzf openocd-0.6.1.tar.gz
cd openocd-0.6.1
mkdir build
cd build
../configure --prefix=/home/user/arm --enable-ft2232_libftdi
               --enable-stlink
make
make install
cd ../../
```

Parametry --enable określają, jakie adaptery i programatory mają być obsługiwane. Pełna lista dostępnych opcji znajduje się w pliku *openocd-0.6.1/README*. Alternatywnie, zamiast libftdi można użyć biblioteki ftd2xx, która jest dostępna pod adresem <http://www.ftdichip.com/Drivers/D2XX.htm>, gdzie jest też plik *ReadMe* opisujący, jak ją zainstalować. Należy wtedy użyć parametru `--enable-ft2232_ftd2xx`. Zeby umożliwić korzystanie bez uprawnień administratora z programu OpenOCD i adaptera JTAG lub programatora podłączanego do USB, należy skorzystać z udev, patrz rozdział 2.7.1.

Literatura

- [1] Kernighan B.W., Ritchie D.M., *Język ANSI C*, WNT, Warszawa 2002.
- [2] Peczarski M., *Mikrokontrolery STM32 w sieci Ethernet w przykładach*, Wydawnictwo BTC, Legionowo 2011.
- [3] CS43L22, Low Power Stereo DAC with Headphone and Speaker Amplifier, nota katalogowa, Cirrus Logic, 2010.
- [4] ISP1705, ULPI Hi-Speed USB transceiver, nota katalogowa, <http://www.stericsson.com>.
- [5] Keyboard Scan Code Specification, Appendix C: USB Keyboard/Keypad Page (0x07), Microsoft Corporation, wersja 1.3a, 2000.
- [6] PM0056 Programming manual, STM32F10xxx/20xxx/21xxx/L1xxxx Cortex-M3 programming manual, <http://www.st.com/stm32>.
- [7] PM0214 Programming manual, STM32F4xxx Cortex-M4 programming manual, <http://www.st.com/stm32>.
- [8] PM0215 Programming manual, STM32F0xxx Cortex-M0 programming manual, <http://www.st.com/stm32>.
- [9] RM0008 Reference manual, STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM-based 32-bit MCUs, <http://www.st.com/stm32>.
- [10] RM0033 Reference manual, STM32F205xx, STM32F207xx, STM32F215xx and STM32F217xx advanced ARM-based 32-bit MCUs, <http://www.st.com/stm32>.
- [11] RM0038 Reference manual, STM32L151xx, STM32L152xx and STM32L162xx advanced ARM-based 32-bit MCUs, <http://www.st.com/stm32>.
- [12] RM0041 Reference manual, STM32F100xx advanced ARM-based 32-bit MCUs, <http://www.st.com/stm32>.
- [13] RM0090 Reference manual, STM32F405xx, STM32F407xx, STM32F415xx and STM32F417xx advanced ARM-based 32-bit MCUs, <http://www.st.com/stm32>.
- [14] RM0091 Reference manual, STM32F05xxx advanced ARM-based 32-bit MCUs, <http://www.st.com/stm32>.
- [15] RM0313 Reference manual, STM32F37xxx and STM32F38xxx advanced ARM-based 32-bit MCUs, <http://www.st.com>.
- [16] RM0316 Reference manual, STM32F302xx, STM32F303xx and STM32F313xx advanced ARM-based 32-bit MCUs, <http://www.st.com/stm32>.
- [17] STM32L151xx and STM32L152xx Errata sheet, STM32L151xx and STM32L152xx ultralow power limitations, <http://www.st.com/stm32>.
- [18] UM0424 User manual, STM32F10xxx USB-FS-Device development kit, <http://www.st.com/stm32>.

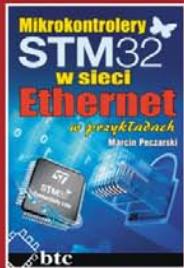
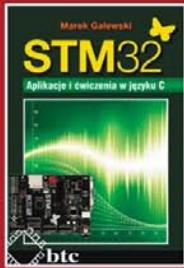
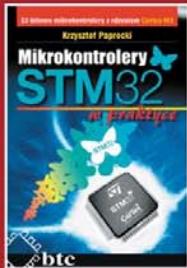
- [19] UM1057 User manual, STM3220G-EVAL evaluation board, <http://www.st.com/stm32>.
- [20] UM1079 User manual, STM32L-DISCOVERY, <http://www.st.com/stm32>.
- [21] UM1472 User manual, STM32F4DISCOVERY STM32F4 high-performance discovery board, <http://www.st.com/stm32>.
- [22] Universal Serial Bus Specification, Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips, wersja 2.0 z późniejszymi poprawkami i suplementami, 2000, <http://www.usb.org/developers/docs>.
- [23] Universal Serial Bus, Device Class Definition for Human Interface Devices (HID), wersja 1.11, 2001, <http://www.usb.org/developers/docs>.
- [24] Universal Serial Bus, Class Definition for Communications Devices, wersja 1.2, 2010, <http://www.usb.org/developers/docs>.
- [25] Universal Serial Bus, Communications Class Subclass Specification for PSTN Devices, wersja 1.2, 2007, <http://www.usb.org/developers/docs>.
- [26] Universal Serial Bus, Device Class Definition for Audio Devices, wersja 1.0, 1998, <http://www.usb.org/developers/docs>.
- [27] Universal Serial Bus, Audio Device Class Specification for Basic Audio Devices, wersja 1.0, 2009, <http://www.usb.org/developers/docs>.
- [28] Universal Serial Bus, Device Class Definition for Audio Data Formats, wersja 1.0, 1998, <http://www.usb.org/developers/docs>.
- [29] Universal Serial Bus, Device Class Definition for Terminal Types, wersja 1.0, 1998, <http://www.usb.org/developers/docs>.
- [30] Universal Serial Bus, Device Class Definition for Audio Devices, wersja 2.0, 2006, <http://www.usb.org/developers/docs>.
- [31] Universal Serial Bus, Mass Storage Class, Bulk-Only Transport, wersja 1.0, 1999, <http://www.usb.org/developers/docs>.
- [32] Universal Serial Bus, Mass Storage Class, UFI Command Specification, wersja 1.0, 1998, <http://www.usb.org/developers/docs>.
- [33] USB in a NutShell, Making sense of the USB standard, <http://www.beyondlogic.org/usbnutshell>.
- [34] SCSI Architecture Model – 5 (SAM-5), <http://www.t10.org>.
- [35] SCSI Block Commands – 3 (SBC-3), <http://www.t10.org>.
- [36] SCSI Primary Commands – 4 (SPC-4), <http://www.t10.org>.

USB dla niewtajemniczonych w przykładach na mikrokontrolery STM32

Jeżeli chcesz świadomie wykorzystać pełne możliwości interfejsu USB w swojej aplikacji wbudowanej – nie możesz przegapić tej książki!

Oprogramowanie i przykładowe aplikacje przedstawione w książce są dostępne bezpłatnie do pobrania.

Poznaj inne książki o mikrokontrolerach STM32:



także w wersji
e-book



Dr inż. Marcin Pęczarski jest absolwentem Wydziału Elektroniki Politechniki Warszawskiej. Ukończył też magisterskie studia uzupełniające i studia doktoranckie na kierunku informatyka na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, gdzie następnie uzyskał stopień doktora nauk matematycznych w zakresie informatyki. Przez wiele lat pracował w dziale telekomunikacji koncernu Siemens. Obecnie jest adiunktem w Instytucie Informatyki UW. Prowadzi zajęcia dydaktyczne z architektury komputerów, sieci komputerowych, bezpieczeństwa systemów komputerowych, programowania w języku C++, programowania mikrokontrolerów i programowalnych układów logicznych. Jest autorem książki *Mikrokontrolery STM32 w sieci Ethernet w przykładach*, która ukazała się w 2011 roku nakładem Wydawnictwa BTC.

Patronat medialny



MIKROKONTROLER.PL



Fanklub STM32

www.stm32.eu



księgarnia
naukowo-techniczna

www.btc.pl

ISBN 978-83-60233-93-1



9 788360 233931

Wydawnictwo

btc