Bluetooth. Praktyczne programowanie

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf".
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Człowiek-najlepsza inwestycja

UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ
WYDZIAŁ MATEMATYKI, FIZYKI I INFORMATYKI
INSTYTUT INFORMATYKI

Bluetooth. Praktyczne programowanie

Andrzej Daniluk

LUBLIN 2012

Instytut Informatyki UMCS
Lublin 2012

Andrzej Daniluk
BLUETOOTH. PRAKTYCZNE PROGRAMOWANIE

Recenzent: Kazimierz Skrobas
Opracowanie techniczne: Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach
Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl

Wydawca
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Instytut Informatyki
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

Druk
FIGARO Group Sp. z o.o. z siedziba w Rykach
ul. Warszawska 10
08-500 Ryki
www: www.figaro.pl

ISBN: 978-83-62773-35-0

SPIS TREŚCI

PRZEDMOWA

Dynamiczny rozwój systemów informatycznych oraz coraz większa pojawiająca się na rynku liczba urządzeń zdolnych do wzajemnej wymiany informacji w czasie rzeczywistym skłoniła producentów sprzętu i oprogramowania do opracowania technologii bezprzewodowego przesyłu danych, która zapewniłby prostotę wykrywania urządzeń przez systemy, a zarazem zadawalającą uniwersalność i funkcjonalność obsługi. W założeniu, nowa technologia miała łączyć istniejące standardy komunikacji bezprzewodowej i doprowadzić do tego aby wszystkie mobilne i stacjonarne urządzenia elektroniczne mogły współpracować ze sobą bezprzewodowo.

Początki historii standardu Bluetooth sięgają drugiej połowy lat 90. ubiegłego wieku, kiedy to konsorcjum pięciu firm komputerowych i komunikacyjnych (Ericsson, Intel, IBM, Nokia i Toshiba) ogłosiło rozpoczęcie prac nad nową technologią bezprzewodowego przesyłu danych. Firmy te w 1998 roku zawiązały organizację pod nazwą Special Interest Group (SIG) w celu opracowania podstaw nowego standardu transmisji bezprzewodowej krótkiego zasięgu. W 1999 roku ogłoszono specyfikację kompletnego standardu Bluetooth wersji 1.0. Niewątpliwe zalety nowej technologii skłoniły grupę standaryzacyjną Instytutu Inżynierów Elektryków i Elektroników (IEEE) zajmującą się bezprzewodowymi sieciami osobistymi 802.15 do rozpoczęcia prac nad nowym standardem warstwy fizycznej i łącza danych sieci osobistych PAN (Personal Area Network). Podstawą oficjalnie zatwierdzonego przez IEEE w 2002 roku standardu 802.15.1 są dokumenty organizacji SIG. Obecnie SIG skupia ponad 2000 firm z całego świata.

Celem niniejszej publikacji jest zaprezentowanie opisu nowoczesnego standardu Bluetooth ze szczególnym uwzględnieniem praktycznych sposobów realizowania bezprzewodowej transmisji danych w formalizmie języka programowania C++ z wykorzystaniem przeznaczonych do tego celu zasobów systemów operacyjnych Windows. Książka nie jest jedynie prezentacją typów danych, funkcji czy struktur oferowanych przez systemy operacyjne Windows, ale przede wszystkim zawiera dużo wskazówek w postaci szczegółowo przedstawionych przykładowych aplikacji.

W trakcie przygotowywania niniejszego opracowania autor korzystał z kompilatora Compiler 5.5 języka C++ zgodnego ze standardem ANSI/ISO i generującego kod wykonywalny dla systemów Microsoft Windows. Pakiet zawiera bibliotekę STL. C++ Compiler 5.5 dostępny jest nieodpłatnie na stronie firmy Embarcadero (http://edn.embarcadero.com/article/20633). Wszystkie programy były testowane w systemach operacyjnych Windows XP SP3, Vista oraz 7. Autor korzystał też z zasobów Microsoft Windows Software Development Kit

(SDK) oferujących dokumentację oraz zestawy sterowników Bluetooth. SDK można bezpłatnie pobrać ze stron MSDN.

Wszytkie programy przedstawone w podręczniku można również testować korzystając z innych kompilatorów C++, np. VC++ oferujących pełną zgodność z biblioteką SDK. Biblioteka Windows Software Development Kit jest w pełni kompatybilna jedynie z kompilatorami VC++. W definicjach struktur i funkcji w sposób niejednolity SDK używa dla typów zmiennych rozszerzeń `IN` lub `__in` w celu oznaczenia parametrów wejściowych, `OUT` lub `__out` dla oznaczenia parametrów wyjściowych lub `__opt` dla oznaczenia parametrów opcjonalnych. Możliwe jest również występowanie oznaczeń będących kombinacją tych parametrów, np. `__inout` lub `__in__opt`. Niektóre kompilatory C++ mogą zgłaszać błędy w trakcie kompilacji modułów zawierających tego rodzaju oznaczenia w deklaracjach zmiennych. W przypadku napotkania przez kompilator problemów z używanymi przez SDK rozszerzeniami należy podjąć próbę zmiany ustawień opcji kompilatora lub bez jakiejkolwiek szkody dla oprogramowania nierozpoznawalne przez kompilator opisane wyżej elementy można samodzielnie usunąć z odpowiednich plików nagłówkowych.

Niektóre z dostępnych kompilatorów języka C++ mogą też niewłaściwie obliczać rozmiar struktur (za pomocą operatora `sizeof()`). Błędne obliczenie rozmiaru którejkolwiek z używanych struktur niezmiennie skutkować będzie błędami w trakcie uruchamiania programu. W takich sytuacjach należy zadbać o właściwe ustalenie opcji kompilatora na podstawie jego dokumentacji.
Stosowana przez autora konstrukcja:

```
#pragma option push -a1
  //...
#pragma option pop
```

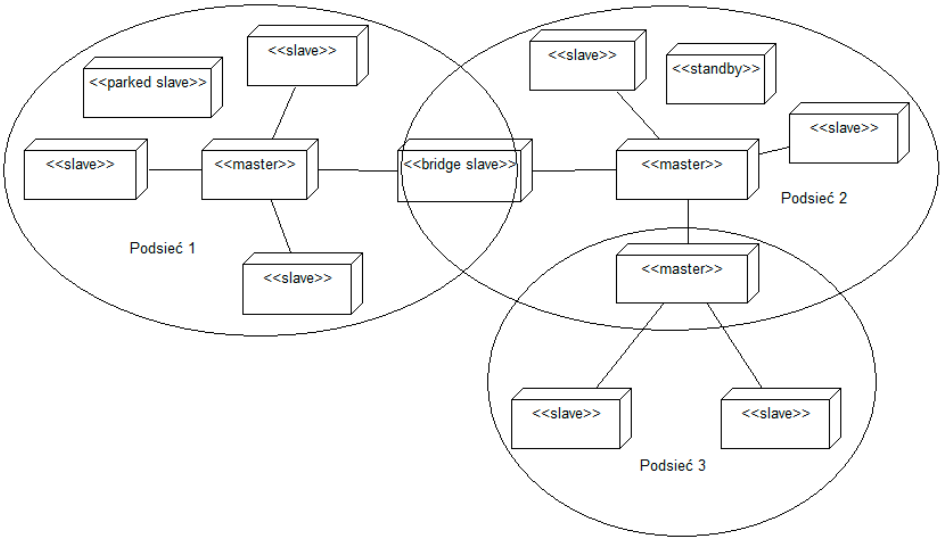
odpowiada opisanej sytuacji. Inne przykłady rozwiązania tego typu pojawiających się problemów można znaleźć w artykule dostępnym pod adresem: http://support.codegear.com/article/35751.

ROZDZIAŁ 1

PODSTAWY TECHNOLOGII BLUETOOTH

1.1. Topologia sieci

Elementy systemu Bluetooth mogą być zorganizowane w formie dwu lub wielopunktowych łącz zwanych podsieciami, pikosieciami lub pikonetami (ang. *piconet*) składających się z jednego aktywnego urządzenia nadrzędnego (ang. *master*) oraz jednego lub kilku aktywnych (maksymalnie 7) jednostek podrzędnych (ang. *slave*). Dwie lub więcej komunikujących się podsieci działających na wspólnym obszarze tworzy tzw. sieć rozproszoną (ang. *scattered*). W sieci rozproszonej urządzenia mogą należeć do więcej niż jednej podsieci, przy czym jednostka nadrzędna w jednej podsieci może pozostać jednostką nadrzędną w innej, zaś urządzenia podrzędne posiadają taki sam status w każdej z nich. Jedno z urządzeń podrzędnych (ang. *bridge slave*) może spełniać rolę mostu integrującego kilka podsieci w sieć rozproszoną. Na rysunku 1.1 w sposób schematyczny za pomocą diagramu wdrożenia zaprezentowano omawianą konfigurację.



Rysunek 1.1. Przykładowa topologia podsieci w systemie Bluetooth

Podsieci Bluetooth mogą być konfigurowane statycznie oraz dynamicznie pozwalając na wykrywanie urządzeń będących aktualnie w zasięgu nadrzędnego odbiornika radiowego [1]. Jeżeli adres któregoś z punktów końcowych jest nieznany, jedno z urządzeń nadrzędnych w celu nawiązania połączenia może używać odpowiednio skonfigurowanych zapytań. Po uzyskaniu poprawnej odpowiedzi oba urządzenia pozostają w stanie połączenia. W tym stanie urządzenie podrzędne będzie synchronizowane z zegarem urządzenia nadrzędnego i z ustalonym algorytmem zmienności przedziałów częstotliwości transmisji danych. Po wymianie danych ustalających połączenie, urządzenie

nadrzędne jest w stanie okresowo inicjować transmisję poprzez synchronizowanie aktywnych podsieci, tak jak pokazano to na rysunku 1.2.

Rysunek 1.2. Sieć rozproszona w praktyce

1.1.1. Zasięg i przepustowość łącza radiowego

Zasięg modułu radiowego urządzenia Bluetooth określany jest poprzez podanie klasy jego mocy. W tabeli 1.1 zaprezentowano opis odpowiednich klas urządzenia.

Tabela 1.1. Zasięg sygnału radiowego Bluetooth

Klasa urządzenia	Moc sygnału	Zasięg
1	100 mW	100 m
2	2,5 mW	10 m
3	1 mW	1 m

Tabela 1.2. Przepustowość łącza radiowego Bluetooth

Nr wersji Bluetooth	Przepustowość
1.0	21 kb/s
1.1	124 kb/s
1.2	328 kb/s
2.0	2,1 Mb/s
2.0 + EDR (Enhanced Data Rate)	3,1 Mb/s
3.0 + HS (High Speed)	24 Mb/s
3.1 + HS (High Speed)	40 Mb/s
4.0 BLE	200 kB/s

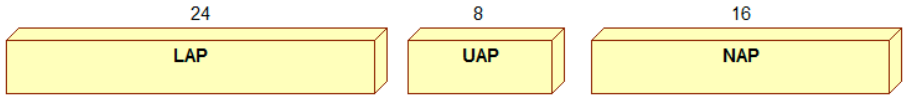
Przepustowość łącza radiowego jest cechą charakterystyczną wersji standardu Bluetooth. Tabela 1.2 zawiera porównanie poszczególnych wersji standardu ze względu na przepustowość łącza radiowego. Bluetooth w wersji 4.0 (BLE – *Bluetooth Low Energy*) różni się przede wszystkim od poprzednika v3.0 tym, iż znacząco ograniczono pobór energii, jednak kosztem obniżonego transferu danych. Ma to umożliwić stosowanie tego interfejsu w coraz bardziej zminiaturyzowanych urządzeniach zasilanych przez baterie lub akumulatory niewielkich rozmiarów, np. przenośne medyczne urządzenia diagnostyczne, urządzenia sportowe, zegarki, itp.

1.1.2. Adresowanie urządzeń Bluetooth

Istnieją cztery główne typy adresów wykorzystywanych w urządzeniach Bluetooth: BD_ADDR, AM_ADDR, PM_ADDR oraz AR_ADDR [1-5]. Każdy nadajnik Bluetooth scharakteryzowany jest poprzez unikalny 48-bitowy adres BD_ADDR (ang. *Bluetooth Device Address*) zgodny ze standardem IEEE 802. Adres ten składa się z:

- 24-bitowej niższej części LAP (ang. *Lower Address Part*) wykorzystywanej do generowania przeskoków częstotliwościowych oraz generowania procedur synchronizujących łącze bezprzewodowe;
- 8-bitowej wyższej części UAP (ang. *Upper Address Part*) wykorzystywanej do inicjowania obliczeń sum kontrolnych CRC oraz korekty błędów HEC (ang. *Header Error Check*) potencjalnie występujących w nagłówku ramki;
- 16-bitowej części nie znaczącej NAP (ang. *Non-significant Address Part*) wykorzystywanej do inicjowania procedur szyfrowania.

Na rysunku 1.3 schematycznie pokazano budowę adresu BD_ADDR.



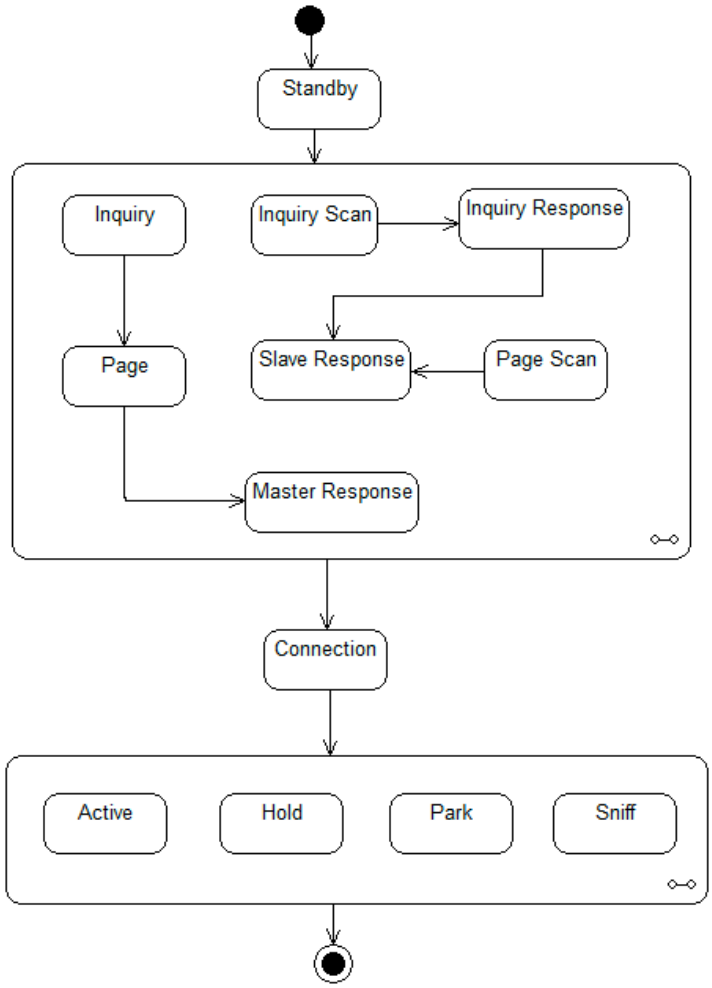
Rysunek 1.3. Elementy składowe adresu BD_ADDR

Adres urządzenia aktywnego AM_ADDR (ang. *Active Member Address*) jest 3-bitowym unikalnym adresem generowanym przez urządzenie nadrzędne (*master*) i przypisanym do urządzenia podrzędnego (*slave*). Oba urządzenie muszą pozostawać w stanie połączenia.

Adres urządzenia pozostającego w stanie wyczekiwania PM_ADDR (ang. *Parked Member Address*) jest 8-bitowym adresem nieaktywnego w danym przedziale czasu (ale zsynchronizowanego z pikosiecią) urządzenia podrzędnego. Adres żądania przyłączenia AR_ADDR (ang. *Access Request Address*) jest przypisany do urządzenia podrzędnego pozostającego w stanie wyczekiwania. Adres ten wykorzystywany jest do ustalenia, w której szczelinie czasowej określone urządzenie może przesłać żądanie przejścia do stanu aktywnego.

1.1.3. Połączenie, konfiguracja i rozłączenie

Urządzenia Bluetooth mogą przebywać w dwóch podstawowych stanach: w stanie gotowości (ang. *Standby*) lub połączenia (ang. *Connection*). Przejścia pomiędzy stanami podstawowymi możliwe są do realizacji poprzez tzw. stany pośrednie, tak jak pokazano to na rysunku 1.4. W stanach pośrednich przebywają urządzenia aktualnie włączane do lub czasowo usuwane z podsieci. Przed ustanowieniem połączenia wszystkie włączone i pozostające w zasięgu głównego modułu radiowego urządzenia przebywają w stanie wykrywania dostępnych urządzeń prowadząc nasłuch łącza radiowego. Połączenie jest inicjowane przez urządzenie wykrywające, które po ustanowieniu połączenia z jednostką wywoływaną staje się urządzeniem nadrzędnym (urządzeniem *master*) w podsieci.



Rysunek 1.4. Ogólny diagram stanów urządzenia w podsieci Bluetooth

Stan wykrywania dostępnych urządzeń (ang. *Inquiry*) wykorzystywany jest podczas dołączania będących w zasięgu głównego modułu radiowego urządzeń o początkowo nierozpoznanych adresach. Przebywając w tym stanie jednostka wywołująca za pomocą protokołu wyszukiwania usług tworzy listę potencjalnie dostępnych urządzeń. Urządzenia będące w zasięgu głównego modułu radiowego przekazują jednostce skanującej pakiety FHS (ang. *Frequency Hopping Synchronization*) umożliwiając zebranie niezbędnych do nawiązania połączenia informacji takich jak wartości CLKN (ang. *Clock Native*) oraz adresy BD_ADDR [1-5].

Stan oczekiwania na wykrycie (ang. *Inquiry Scan*) przeznaczony jest dla urządzeń z włączoną opcją widoczności w podsieci, które czasowo umożliwiają dostęp do siebie innym urządzeniom będącym aktualnie w stanie wykrywania. Po odebraniu żądanej wiadomości, urządzenie będące początkowo w stanie oczekiwania na wykrycie przechodzi do stanu odpowiedzi na wykrycie (ang. *Inquiry Response*) wysyłając odpowiedni pakiet FHS. Cechą charakterystyczną urządzeń pozostających w stanie oczekiwania na wykrycie jest wykorzystywanie szybkiej dla głównego modułu radiowego, a wolnej dla modułów podrzędnych określonej sekwencji przeskoków częstotliwościowych umożliwiających sprawne dopasowanie częstotliwości pomiędzy urządzeniami. W celu ustanowienia połączenia, urządzenie nadrzędne wykonuje procedurę wywołania na rzecz określonego urządzenia podrzędnego. Potwierdzając odpowiednie komunikaty urządzenie nadrzędne przechodzi do stanu odpowiedzi urządzenia nadrzędnego.

W celu zainicjowania połączenia urządzenie nadrzędne inicjuje procedurę wywoływania przechodząc w stan wywoływania (ang. *Page*). Wykorzystując dane zebrane w trakcie wykrywania urządzeń, urządzenie nadrzędne wysyła odpowiednie komunikaty do urządzenia podrzędnego. Potwierdzając komunikaty wywoływania generowane przez jednostkę nadrzędną, urządzenie podrzędne przechodzi do stanu odpowiedzi urządzenia podrzędnego (ang. *Slave response*), zaś urządzenie nadrzędne przechodzi do stanu odpowiedzi urządzenia nadrzędnego (ang. *Master response*).

Występujący okresowo stan oczekiwania na wywołanie (ang. *Page Scan*) pozwala nawiązać połączenie z urządzeniem zgłaszającym gotowość do współpracy. Po odebraniu pakietu wywołującego, jednostka wywoływana przechodzi do stanu odpowiedzi urządzenia podrzędnego. Należy zauważyć, iż procedury składające się na stan wywoływania mogą zostać wykonane bez konieczności wykrywania wówczas, gdy adresy odpowiednich urządzeń są znane.

W stanie połączenia aktywnego (ang. *Active*) urządzenie podrzędne przełącza się na zegar CLK urządzenia nadrzędnego. Przełączenie to następuje poprzez dodanie odpowiedniego offsetu do własnego zegara CLKN, co w konsekwencji umożliwia urządzeniu podrzędnemu na używanie sekwencji przeskoków częstotliwościowych urządzenia nadrzędnego. W celu weryfikacji poprawności połączenia urządzenie nadrzędne przesyła pakiet POOL, oczekując

potwierdzenia pakietem NULL. W przypadku niepowodzenia ww. procedury, urządzenia przechodzą do stanu wywoływania.

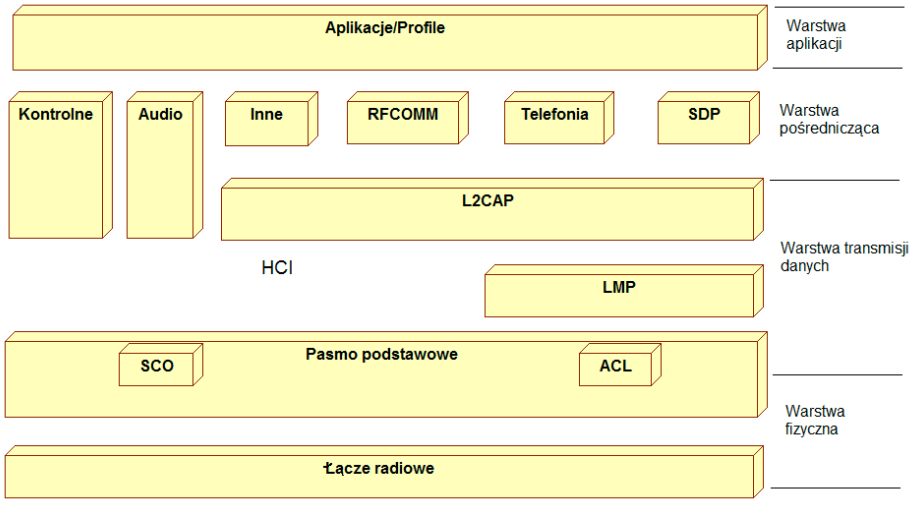
Jeżeli asynchroniczna transmisja danych między jednostkami ma być czasowo zawieszona, urządzenie nadrzędne może wysłać do urządzeń podrzędnych komunikat zalecający im przejście w stan połączenia wstrzymanego (ang. *Hold*).

Urządzenie może pozostawać w stanie okresowego nasłuchiwania lub monitorowania (ang. *Sniff*), w którym prowadzi nasłuch podsieci ze zmniejszoną aktywnością oraz w stanie uśpienia (jest to tzw. tryb wyczekiwania i niskiego poboru mocy) (ang. *Park*), w którym przekazuje swój adres AM_ADDR zarazem nie uczestnicząc aktywnie w wymianie danych, a jedynie okresowo nasłuchując przepływ danych w podsieci.

Urządzenie pozostające w stanie synchronizacji może w dowolnym momencie otrzymać sygnał aktywacyjny lub nawigacyjny od jednostki typu *master*. Stan synchronizacji wykorzystuje się głównie wtedy, gdy architektura systemu wymaga, aby jednostka nadrzędna prowadziła wymianę danych z więcej niż 7 urządzeniami podrzędnymi. Przełączenie niektórych urządzeń podrzędnych w stan synchronizacji pozwala na włączenie do podsieci dodatkowych, aktualnie wymaganych urządzeń, w chwili gdy są one potrzebne.

1.2. Architektura systemu Bluetooth

Procedury sprzętowego sterowanie łączem radiowym Bluetooth zaimplementowane są w warstwie fizycznej systemu obejmującej łącze radiowe oraz pasmo podstawowe [6]. Warstwa fizyczna zajmuje się transmisją radiową oraz przetwarzaniem sygnałów cyfrowych dla protokołów pasma podstawowego.



Rysunek 1.5. Architektura warstw systemu Bluetooth

Jej funkcje obejmują ustalenie połączeń, obsługę transmisji asynchronicznej dla danych i synchronicznej dla głosu, korygowanie błędów transmisji danych oraz uwierzytelnianie urządzeń. Z kolei oprogramowanie sprzętowe do zarządzania łączem (ang. *Link Manager*) odpowiedzialne jest za wykonanie niskopoziomowej detekcji urządzeń, procedur uwierzytelniania oraz konfiguracji łącza. Wiele procedur sprzętowego sterowania łączem może być dostępnych za pośrednictwem interfejsu HCI będącego standardowym, wewnętrznym interfejsem do oprogramowania. Na rysunku 1.5 w sposób schematyczny zaprezentowano architekturę warstw systemu Bluetooth.

1.2.1. Łącze radiowe

Łącze radiowe (ang. *Physical Radio*) będąc częścią warstwy fizycznej systemu Bluetooth odpowiedzialne jest za dwukierunkowy transport danych pomiędzy urządzeniem nadrzędnym a urządzeniami podrzędnymi w ramach podsieci. Pasmo transmisyjne, w obrębie którego funkcjonuje warstwa radiowa podzielone jest na maksymalnie 79 kanałów o szerokości 1 MHz. W celu równomiernego przydziału kanałów transmisyjnych łącze radiowe wykorzystuje technologię widma rozproszonego z metodą przeskoków częstotliwości (ang. *Frequency Hopping*) pracując w paśmie ISM 2.4-2.4835 GHz. Kanał transmisyjny reprezentowany jest przez pseudolosową sekwencję przeskoków częstotliwości wykonywanych 1600 razy na sekundę. Sekwencja ta określana jest na podstawie adresu urządzenia nadrzędnego funkcjonującego w ramach podsieci. Kanał transmisyjny podzielony jest czasowo na przedziały (często nazywane szczelinami lub slotami) o szerokości 625 μs. Dane transmitowane są w postaci odpowiednio zdefiniowanych ramek. Transmisja ramki rozpoczyna się zawsze na początku zdefiniowanego przedziału czasowego i może trwać co najwyżej 5 takich jednostek. Z tego powodu przeskoki częstotliwości mogą być wstrzymywane do czasu aż cała ramka nie zostanie nadana na jednej częstotliwości. Urządzenia nadrzędne mogą nadawać jedynie w przedziałach czasowych o numerach parzystych, zaś urządzenia podrzędne - w nieparzystych. W ten sposób możliwe jest uzyskanie dwukierunkowości łącza (ang. *Time Division Duplex*) [1-4]

1.2.2. Pasmo podstawowe

Pasmo podstawowe (ang. *Baseband*) posługuje się dwoma kanałami logicznymi obsługującymi odpowiednio łącza bezpołączeniowej transmisji asynchronicznej ACL (ang. *Asynchronous Connectionless Links*), które są wykorzystywane do przesyłania standardowych danych oraz synchroniczne łącza transmisyjne SCO (ang. *Synchronous Connection Oriented*) wykorzystywane do transmisji danych audio (głosu).

1.2.3. Łącze synchroniczne

Synchroniczne łącze transmisyjne SCO jest symetrycznym łączem dwupunktowym (ang. *Point-To-Point*) tworzonym w ramach podsieci pomiędzy urządzeniem nadrzędnym (*master*) i urządzeniem podrzędnym (*slave*). W celu zagwarantowania odpowiedniego czasu transmisji, SCO wykorzystuje cykliczną rezerwację odpowiednich przedziałów czasowych, dzięki czemu jest w stanie obsługiwać transmisję danych ograniczonych czasowo (np. rozmowa telefoniczna) w czasie rzeczywistym, przy czym dane tego typu nie mogą być retransmitowane. W celu zapewnienia niezawodności przekazu wykorzystuje się odpowiednie algorytmy korekty błędów. Urządzenie podrzędne może korzystać z maksymalnie trzech kanałów typu SCO w kierunku urządzenia nadrzędnego. Każde łącze SCO może transmitować jeden kanał telefoniczny (PCM, 64 kbit/s). Za pośrednictwem SCO dopuszczalna jest także łączona transmisja danych i głosu, jednak w razie wystąpienia błędu możliwa jest jedynie retransmisja danych. Łącza SCO nie są obsługiwane przez protokół L2CAP.

1.2.4. Łącze asynchroniczne

Asynchroniczne łącze bezpołączeniowe ACL jest łączem wielopunktowym (ang. *Point-to-Multipoint*) wykorzystanym do transmisji danych dostępnych w nieregularnych odstępach czasowych. Dane mogą być transmitowane w trybie symetrycznym lub asymetrycznym między urządzeniem nadrzędnym a wszystkimi występującymi w ramach podsieci urządzeniami podrzędnymi, przy czym dla każdej pary *master-slave* można zestawić co najwyżej jedno takie łącze. ACL wykorzystując przedziały czasowe nie zarezerwowane przez SCO może obsłużyć zarówno przekaz asynchroniczny jak i izochroniczny[1]. Dane zawierające ew. błędy mogą być retransmitowane. Dane asynchroniczne przekazywane przez urządzenie nadrzędne dostarczane są pod wskazany adres, co oznacza, że odbiorcą ich powinno być konkretne urządzenie podrzędne. Jeżeli jednak nie zostało wskazane żadne urządzenie, transmitowane dane traktowane są jak wiadomość rozgłoszeniowa (ang. *broadcast*) w obrębie podsieci. Dane transmitowane za pośrednictwem ACL pochodzą od warstwy L2CAP nadajnika i są dostarczane do warstwy L2CAP funkcjonującej w ramach odbiornika.

1.3. Oprogramowanie sprzętowe do zarządzania łączem

Oprogramowanie menedżera łącza LM (ang. *Link Manager*) spełnia w systemie Bluetooth bardzo ważną rolę. Do najważniejszych realizowanych przez nie funkcji należy zaliczyć: niskopoziomową detekcję urządzeń, nadzorowanie łącza, uwierzytelnianie urządzeń, realizacja procedur związanych z

[1] Przekaz izochroniczny oznacza, że dane (ramki) transmitowane są sekwencyjnie w określonych, stałych przedziałach czasowych w tej samej kolejności, w jakich zostały nadane bez konieczności potwierdzenia odbioru.

bezpieczeństwem transmisji danych, śledzenie dostępnych usług oraz kontrola stanu pasma podstawowego. Układy zarządzające łączem w różnych urządzeniach komunikują się ze sobą za pośrednictwem specjalnie dedykowanego protokołu zarządzania łączem – LMP (ang. *Link Management Protocol*) korzystającego z łącza asynchronicznego pasma podstawowego. Pakiety LMP są odróżniane od pakietów sterowania łączem logicznym LLC (ang. *Logical Link Control*) i protokołu L2CAP bitem umieszczanym w nagłówku ACL. Pakiety LMP transmitowane są w przedziałach czasowych o jednostkowej szerokości (tzw. pakiety jednoszczelinowe) i mają wyższy priorytet niż pakiety protokołu L2CAP.

1.3.1. HCI

Sprzętowe kontrolery łącza mogą zawierać warstwę HCI (ang. *Host Controller Interface*) umiejscowioną powyżej menedżera łącza. Sterowniki host-kontrolera (ang. *host controller driver*) umożliwiają komunikację między aplikacjami Bluetooth a wybranym protokołem transportowym. Sterowniki HCI umiejscowione są w warstwie transmisji danych izolując tym samym pasmo podstawowe oraz menedżera łącza od portów komunikacyjnych (USB, RS 232C). Korzystając z HCI, aplikacje Bluetooth uzyskują bezpośredni dostęp do urządzeń bez konieczności znajomości sprzętowej implementacji warstwy transportowej.

1.3.2. L2CAP

Protokół kontroli połączenia logicznego i adaptacji L2CAP (ang. *Logical Link Control and Adaptation Protocol*) umiejscowiony jest w warstwie transmisji danych i implementowany jest programowo w ramach asynchronicznych łącz ACL. Pojedyncze łącze ACL ustanowione przez oprogramowanie menedżera jest zawsze dostępne pomiędzy urządzeniem nadrzędnym a każdym będącym w zasięgu aktywnym urządzeniem podrzędnym. Dzięki temu aplikacja użytkownika ma dostęp do wielopunktowego łącza obsługującego zarówno przesył izochroniczny jak i synchroniczny. Protokół L2CAP zapewnia usługi protokołom wyższych warstw poprzez realizację trzech głównych zadań polegających na multipleksacji danych pochodzących od warstwy wyższej, dostosowywaniu wielkości transmitowanych pakietów danych do rozmiaru ramek generowanych przez pasmo podstawowe oraz kontrolowanie parametrów jakościowych QoS (ang. *Quality of Service*) realizowanej usługi.

1.3.3. Warstwa pośrednicząca

Grupa protokołów pośredniczących stanowi interfejs, za pomocą którego warstwy aplikacji Bluetooth komunikują się ze sobą poprzez warstwę transportową. Składa się ona z następujących podstawowych elementów: RFCOMM, SDP, TCS oraz protokółu współpracy z IrDA.

1.3.4. RFCOMM

RFCOMM (ang. *Radio Frequency Communication*) stanowi zbór protokołów transportowych, umiejscowionych powyżej protokołu kontroli połączenia logicznego i adaptacji L2CAP (patrz rys. 1.5). Bardzo często protokół RFCOMM jest określany mianem emulatora standardowego portu szeregowego RS 232C [6,7]. Opisany poniżej profil wirtualnego portu szeregowego SPP bazuje na RFCOMM. Połączeniowy, strumieniowy protokół komunikacyjny RFCOMM zapewnia użytkownikowi prosty i niezawodny dostęp do strumienia danych przeznaczonych zarówno do wysłania jak i odbioru. Jest on stosowany bezpośrednio przez wiele profili związanych z telefonią, jako nośnik komend AT, a także jako warstwa transportowa dla usług wymiany danych w postaci obiektów OBEX (ang. *Object Exchange*) (patrz rys. 1.6). Wiele współczesnych aplikacji Bluetooth używa RFCOMM ze względu na jego szerokie wsparcie techniczne w postaci publicznego API dostępnego w większości systemów operacyjnych. Warto też zdawać sobie sprawę z faktu, iż aplikacje używające standardowego portu szeregowego mogą być szybko zaadoptowane na potrzeby RFCOMM. Maksymalna liczba jednocześnie dostępnych połączeń wynosi 60.

1.3.5. Telefonia

Protokół telefoniczny lub sterowania telefonem TCS - BIN (ang. *Telephony Control Specification—Binary*) jest protokołem czasu rzeczywistego, używanym w profilach zorientowanych na rozmowy. Zarządza również ustanawianiem i rozłączeniem połączenia głosowego. TCS umożliwia realizację połączeń dwupunktowych (jeżeli znany jest adres docelowy urządzenia) z wykorzystaniem ACL oraz wielopunktowych z wykorzystaniem SCO.

1.3.6. SDP

Protokół wyszukiwania usług SDP (ang. *Service Discovery Protocol*) używany do lokalizowania w ramach podsieci dostępnych usług. Implementuje on procedury, dzięki którym urządzenie podrzędne (klient SDP) może uzyskać informację na temat usług udostępnianych przez urządzenia nadrzędne (serwery SDP). Serwer SDP udostępnia informacje na temat usług w postaci rekordów opisujących parametry jednej usługi.

1.3.7. Inne protokoły

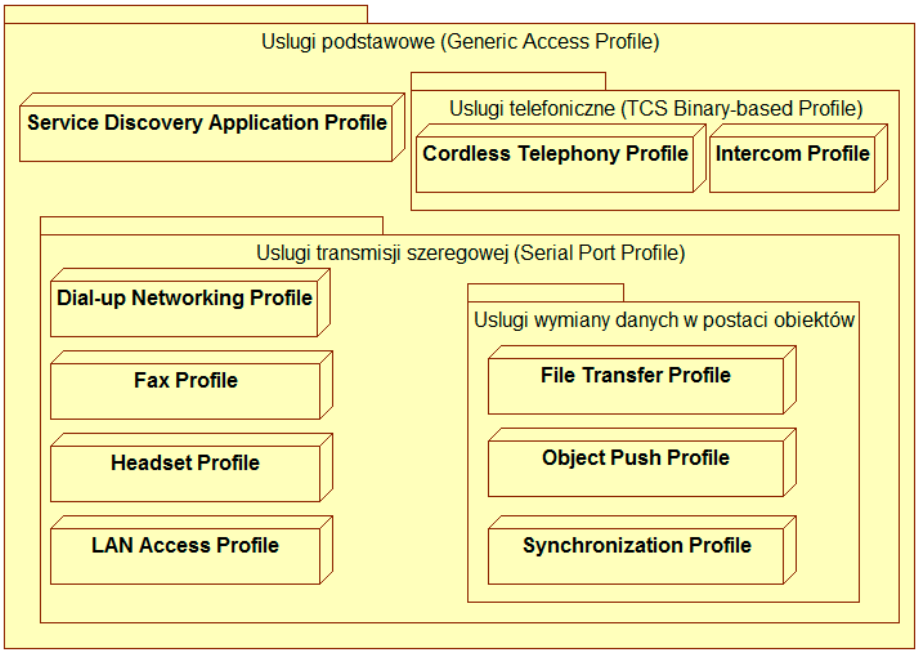
W zależności od konstrukcji systemu, warstwa pośrednicząca może być uzupełniona o inne elementy opisujące mechanizmy współpracy z takimi protokołami jak IrDA czy WAP. Protokoły te wykorzystywane są w konkretnych aplikacjach opisanych w ramach profili [6].

1.4. Profile

Dokumentacja standardu Bluetooth opracowana przez Special Interest Group [6] nie zawiera specyfikacji konkretnego API (interfejsu programistycznego)

właściwego danej platformie systemowej. SIG zakłada, iż w trakcie wdrażania technologii Bluetooth na danej platformie potrzeba opracowania właściwego API powinna spoczywać w gestii programistów. Z powyższych względów postanowiono nie opracowywać oddzielnych Linux API, Windows API, itp. Zamiast tego, za pomocą profili zdefiniowano wszystkie funkcje niezbędne programistom tworzącym oprogramowanie współpracujące z konkretnymi aplikacjami Bluetooth na danej platformy systemowej. Tym samym, mimo iż specyfikacja standardu nie zawiera właściwego API, dostarcza jednak programistom aplikacji wszystkich niezbędnych wskazówek umożliwiających w prosty sposób przekształcanie ich w API konkretnej platformy.

Zdefiniowane przez Bluetooth Special Interest Group profile określają funkcje urządzenia [6]. Obejmują one różne warstwy i protokoły służąc zapewnieniu kompatybilności między aplikacjami oraz urządzeniami Bluetooth pochodzącymi od różnych producentów. Urządzenia Bluetooth mogą współpracować ze sobą jedynie w obrębie wspólnych profili. Profile Bluetooth są uporządkowane w grupach i mogą być zależne od innych, jeżeli wykorzystują deklaracje profili nadrzędnych, tak jak pokazano to schematycznie na rysunku 1.6. Poniżej opisano kilkanaście najczęściej wykorzystywanych profili. Więcej na ten temat można znaleźć w dokumentacji SIG [6] oraz pozycjach [8, 9].



Rysunek 1.6. Podstawowe i najczęściej w praktyce wykorzystywane usługi urządzenia Bluetooth

1.4.1. GAP

Profil podstawowy GAP (ang. *Generic Access Profile*). Wszystkie urządzenia Bluetooth muszą funkcjonować w obrębie profilu ogólnego dostępu. Oznacza to, iż GAP powinien być zaimplementowany we wszystkich urządzeniach z wbudowaną funkcją Bluetooth. Definiuje on funkcjonalności, które powinny być zaimplementowane w urządzeniach, opisuje ogólne procedury niezbędne do wykrywania urządzeń, określania ich nazw, adresów, typów oraz odpowiada za ustanawianie połączenia i weryfikację kodu parowania.

1.4.2. SDAP

Profil aplikacji wykrywania usług SDAP (ang. *Service Discovery Application Profile*) definiuje operacje zaimplementowane w oprogramowaniu urządzeń Bluetooth pozwalające na wyszukiwanie usług udostępnianych przez inne urządzenia.

1.4.3. CTP

Profil telefonii bezprzewodowej CTP (ang. *Cordless Telephony Profile*) definiuje procedury, które są wymagane do osiągnięcia kompatybilności między jednostkami określanymi jako „trzy w jednym" (3-in-1) (ang. *three-in-one phone*). Umożliwia prowadzenie rozmów telefonicznych przez Bluetooth w taki sam sposób jak za pomocą standardowych bezprzewodowych telefonów stacjonarnych. W tego typu konfiguracji słuchawka może być połączona z trzema usługami. Telefon może funkcjonować jako bezprzewodowy, połączony z siecią komutowaną, połączony bezpośrednio z innymi telefonami (*walkie-talkie*), lub też jako komórkowy, połączony z infrastrukturą sieci komórkowej.

1.4.4. IntP

Profil bezprzewodowej komunikacji wewnętrznej IntP (ang. *Intercom Profile*) jest profilem analogicznym do CTP. Różnica polega na tym, iż służy on do bezpośredniego połączenia głosowego przez Bluetooth dwóch będących w zasięgu telefonów.

1.4.5. SPP

Profil wirtualnego portu szeregowego SPP (ang. *Serial Port Profile*) jest podstawowym profilem dla usługi transmisji szeregowej definiującym wszystkie niezbędne wymagania, jakie muszą spełniać urządzenia Bluetooth w celu utworzenia wirtualnego połączenia szeregowego.

1.4.6. HSP

Profil bezprzewodowego zestawu słuchawkowego HSP (ang. *Headset Profile*) specyfikuje wymagania dla urządzeń, które mogą być użyte jako zestaw słuchawkowy umożliwiający przesyłanie niskiej jakości danych audio (tzw. *Bluetooth Audio*) w obie strony (ang. *full duplex*). Profil implementuje model

Ultimate Headset określający, w jaki sposób bezprzewodowe słuchawki z zaimplementowaną funkcją Bluetooth mogą być łączone, aby działały jako interfejs audio I/O (wejścia/wyjścia) dla zdalnego urządzenia, np. laptopa.

1.4.7. DUN

Profil usług modemowych DUN lub DUNP (ang. *Dial-up Networking Profile*) udostępnia połączenia typu dial-up do sieci Internet. Profil usług modemowych bazując na wirtualnym porcie szeregowym [7] definiuje wymagania niezbędne do emulacji połączenia między modemem a terminalem (urządzeniem transmitującym dane). Działanie profilu rozpoczyna się od ustanowienia połączenia pomiędzy modemem a terminalem, tj. po podaniu w urządzeniach kodu PIN (ang. *Personal Identify Number*), wymianie kluczy szyfrujących oraz rozpoznaniu usługi. Sterowanie modemem odbywa się za pomocą komend AT przesyłanych za pośrednictwem wirtualnego portu szeregowego. Specyfikacja DUN opisuje listę komend lub zapytań wysyłanych do modemu i odpowiedzi transmitowanych przez modem.

1.4.8. LAP

Profil dostępu do sieci lokalnej LAP (ang. *LAN Access Profile*) określa mechanizm zapewniający urządzeniom Bluetooth swobodny dostęp do usług LAN (ang. *Local Area Network*) za pośrednictwem sesji protokołu PPP (ang. *Point to Point Protocol*). Urządzeniem obsługującym profil LAP może być typowy punkt dostępowy (ang. *Access Point*) podłączony do sieci lokalnej albo prosty terminal danych DT (ang. *Data Terminal*), np. laptop zgłaszający żądanie dostępu do zasobów sieci poprzez Bluetooth.

1.4.9. GOEP

Ogólny profil wymiany danych w postaci obiektów GOEP (ang. *Generic Object Exchange Profile*) precyzuje wymagania stawiane urządzeniom Bluetooth komunikującym się ze sobą w oparciu o mechanizm wymiany danych w postaci obiektów na zasadzie wyślij i pobierz (ang. *push and pull*). Komunikacja między aplikacjami po stronie klienta i serwera odbywa się według reguł opisanych protokołem OBEX. Proces wymiany danych pomiędzy aplikacją serwera a programem klienckim poprzedza wywołanie specjalnej procedury połączeniowej (ang. *bonding*). W odróżnieniu od typowej procedury tworzenia par urządzeń (ang. *pairing*), w trakcie wykonywania procedury połączeniowej oba urządzenia tworzą między sobą odpowiednio zabezpieczony kanał transmisyjny. Jego stworzenie wymaga użycia przez oba urządzenia identycznego kodu autoryzacji (klucza PIN). Proces uwierzytelniania jest inicjowany przez program klienta. Przykładem wykorzystania profilu GEOP mogą być aplikacje służące do synchronizacji danych lub przesyłania plików. Najczęściej z usług definiowanych w profilu GOEP korzystają programy

działające w urządzeniach przenośnych (laptop, notatnik elektroniczny, telefon komórkowy).

1.4.10. FAX

Profil usług telefaksowych FAX (ang. *Fax Profile*) przeznaczony jest do bezprzewodowego wysyłania i odbierania wiadomości faksowych za pośrednictwem łącza radiowego. Podobnie jak w profilu usług modemowych, transmisja danych pomiędzy terminalem a urządzeniem emulującym faks odbywa się za pośrednictwem wirtualnego portu szeregowego z wykorzystaniem komend AT.

1.4.11. OPP

Profil przesyłania obiektów OPP (ang. *Object Push Profile*) określa sposób wymiany pomiędzy urządzeniami danych w postaci tapet, dzwonków, filmów, wizytówek, pozycji z książki adresowej lub kartek z kalendarza. Informacje przesyłane są w formatach vCard/vCalendar/iCalendar, które definiują standardy formatu plików używanych do wymiany danych osobowych.

1.4.12. FTP

Profil przesyłania plików FTP (ang. *File Transfer Profile*) pozwala na przesyłanie danych zorganizowanych w pliki i foldery (katalogi). W ramach profilu użytkownik ma możliwość wyboru serwera FTP z listy serwerów pozostających w zasięgu, przeglądanie zasobów serwera, tworzenie, kopiowanie lub usuwanie pliku (lub folderu) z zasobów serwera.

1.4.13. SP

Profil synchronizacji danych SP (ang. *Synchronization Profile*) opisuje sposób porównywania i uaktualniania danych (w tym informacji osobistych użytkownika) między urządzeniami Bluetooth. Oprogramowanie SP funkcjonuje w oparciu o protokół wymiany danych między urządzeniami mobilnymi IrMC (ang. *IrDA Mobile Communications*). W ramach profilu synchronizacji zdefiniowane zostały dwa typy urządzeń wymieniających dane: IrMC serwer oraz IrMC klient. Serwerem może być telefon komórkowy lub notatnik elektroniczny PDA (ang. *Personal Data Assistant*). Rolę klienta pełni komputer z funkcją Bluetooth.

Warto zwrócić uwagę, na fakt, iż dokumentacja Bluetooth [6] opisuje szereg dodatkowych (nie pokazanych na rys. 1.5) profili krótko scharakteryzowanych poniżej.

1.4.14. ESDP

Profil rozszerzonego wykrywania usług ESDP (ang. *Extended Service Discovery Profile*) definiuje mechanizm wykorzystywania profilu SDP do

wykrywania innych urządzeń z wbudowaną obsługą usług PnP (ang. *Plug and Play*) oraz zbierania informacji o tych usługach.

1.4.15. HID

Profil urządzeń interfejsu HID (ang. *Human Interface Device Profile*) został zaadoptowany ze specyfikacji HID urządzeń USB [10]. Informacje o akcjach użytkownika na elementach sterujących urządzenia Bluetooth (np. naciśnięcie klawisza) są przesyłane do komputera w czasie rzeczywistym. Profil ten umożliwia np. zdalne sterowanie za pomocą klawiatury telefonu komórkowego aplikacjami uruchomionymi na komputerze.

1.4.16. HFP

Profil bezdotykowego sterowania HFP (ang. *Hands Free Profile*) rozszerza funkcjonalność HIDP o możliwość głosowego sterowania urządzeniem z funkcją Bluetooth.

1.4.17. HCRP

Profil wydruku bezprzewodowego HCRP lub HC2RP (ang. *Hard-Copy Cable Replacement Profile*) wykorzystywany jest przez komputer i drukarki do drukowania bezprzewodowego (zamiast kabli USB lub LPT). Telefony komórkowe nie używają tego profilu. Zamiast niego wykorzystują profil BPP (ang. *Basic Printing Profile*) umożliwiający wykonywanie prostych operacji drukowania bez konieczności instalowania dodatkowych sterowników.

1.4.18. PAN

Profil dostępu do sieci osobistej PAN lub PANP (ang. *Personal Area Networking Profile*) rozszerza funkcjonalność opisanego wcześniej profilu LAP poprzez obsługę protokołu BNEP (ang. *Bluetooth Network Encapsulation Protocol*). Profil PAN opisuje mechanizm, w jakim dwa lub więcej urządzeń Bluetooth może tworzyć podsieć doraźną (ang. *ad-hoc*) oraz jak ten sam mechanizm jest używany do uzyskania dostępu do zdalnej podsieci poprzez określony punkt dostępu. Profil definiuje sieciowy punkt dostępu, sieć rozproszoną oraz użytkownika sieci osobistej.

1.4.19. BIP

Profil podstawowego obrazowania BIP (ang. *Basic Imaging Profile*) opisuje mechanizmy zdalnego sterowania urządzeniem obrazującym (np. aparatem cyfrowym), mechanizmy drukowania obrazów na obsługującej ten profil drukarce z funkcją Bluetooth oraz mechanizmy transmisji obrazów do urządzenia magazynującego (np. komputera).

1.4.20. A2DP

Profil zaawansowanej dystrybucji audio A2DP (ang. *Advanced Audio Distribution Profile*) przeznaczony jest do transmitowania danych audio wysokiej jakości.

1.4.21. VDP

Profil dystrybucji wideo VDP (ang. *Video Distribution Profile*) przeznaczony jest do transmitowania danych video wysokiej jakości. Wykorzystanie profili A2DP oraz VDP zakłada możliwość transmisji filmów w rozdzielczości HD lub nawet FHD.

1.4.22. AVRCP

Profil zdalnego sterowania audio/wideo AVRCP (ang. *Audio/Video Remote Control Profile*) przeznaczony jest do zdalnego sterowania urządzeniami audio/wideo. Zestaw poleceń oferowanych przez ten profil umożliwia na przykład bezprzewodowe sterowanie urządzeniami audio/wideo przy pomocy jednego pilota zdalnego sterowania, lub za pomocą aplikacji uruchomionej na komputerze.

1.4.23. CIP

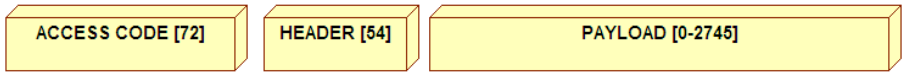
Profil wspólnego dostępu do sieci ISDN CIP (ang. *Common ISDN Access Profile*) definiuje usługi ISDN umożliwiające aplikacyjnym zachowanie wstecznej zgodności z już istniejącymi programami ISDN opartymi na interfejsie CAPI (ang. *Common-ISDN-Application Programming Interface*).

1.4.24. SAP

Profil dostępu do karty SIM (ang. *SIM Access Profile*) umożliwia urządzeniom z wbudowanym nadajnikiem-odbiornikiem GSM ustanawianie połączenia się z kartą SIM w telefonie z funkcją Bluetooth.

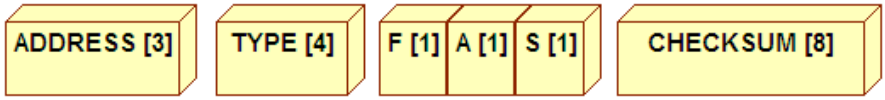
1.5. Struktura i typy ramek

Transmisja danych w systemie Bluetooth odbywa się poprzez struktury danych zwane ramkami [1-4]. Początek ramki identyfikowany jest za pomocą 72-bitowewgo kodu dostępu (ang. *Access Code*). Kod ten używany jest w celu synchronizacji oraz identyfikacji urządzenia nadrzędnego, tak aby urządzenie podrzędne będące w zasięgu dwóch (lub większej liczby) urządzeń typu master mogło zidentyfikować do którego z nich przesłać żądane informacje. Następnie występuje 54-bitowy nagłówek ramki (ang. *Header*). Ostatni element ramki stanowi pole danych (ang. *Playload*), tak jak pokazano to schematycznie na rysunku 1.7.



Rys. 1.7. Struktura ramki w systemie Bluetooth

Bardzo ważnym elementem ramki danych jest jej nagłówek (rys. 1.8). Wyszczególniony jest tutaj: 3-bitowy adres AM_ADDR identyfikujący jedno z urządzeń w podsieci, dla którego ramka jest przeznaczona, 4-bitowy typ ramki (ramki łącza synchronicznego SCO, ramki łącza asynchronicznego ACL, oraz wspólne dla obu typów – ramki sterujące z polem NULL informującym nadajnik o stanie łącza i buforów odbiornika po odebraniu przezeń informacji lub polem POLL służącym do wywoływania urządzeń *slave*, które powinny cyklicznie odpowiedzieć na zapytania kierowane przez urządzenia *master*), rodzaj używanej korekcji błędów oraz liczbę slotów czasowych w ramce. Nagłówek zawiera też 3-bitową informację na temat kontroli wypełnienia buforów transmisyjnych (bit *Flow* jest ustalany przez urządzenie podrzędne w przypadku przepełnienia swoich buforów transmisyjnych, bit *Acknowledgement* jest potwierdzeniem transmisji, zaś bit *Sequence* jest używany jako znacznik retransmisji danych). Nagłówek zakończony jest 8-bitową sumą kontrolną.



Rys. 1.8. Struktura nagłówka ramki w systemie Bluetooth

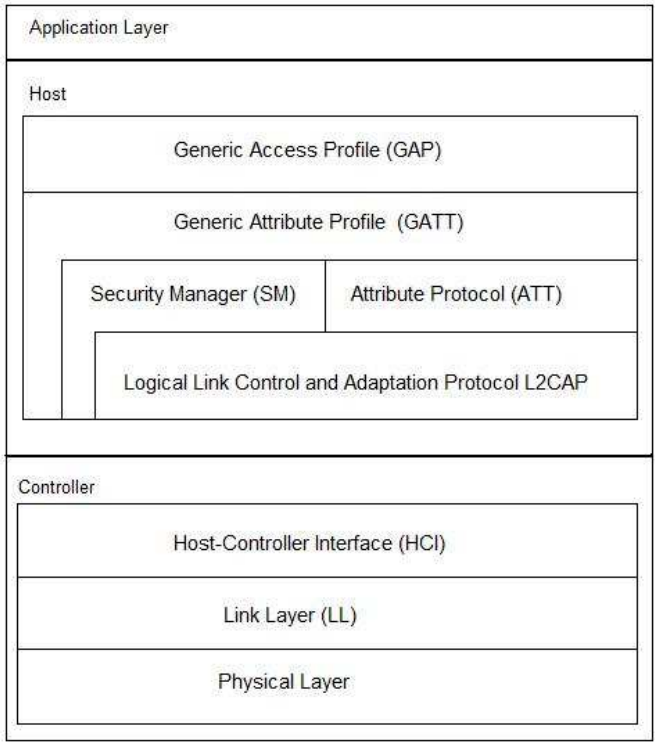
Pełny 54-bitowy nagłówek ramki danych powstaje poprzez 3-krotne powtórzenia sekwencji wyżej opisanych 18 bitów. W trakcie transmisji danych układ odbiornika sprawdza wszystkie trzy kopie każdego bitu. Jeśli wszystkie są identyczne, wówczas bit jest zaakceptowany. Jeśli nie, to w przypadku, gdy otrzymano dwa zera i jedynkę – wartość końcowa jest zerem, jeśli zaś dwie jedynki i zero – wynikiem jest jedynka.

1.6. Podstawowe protokoły Bluetooth Low Energy

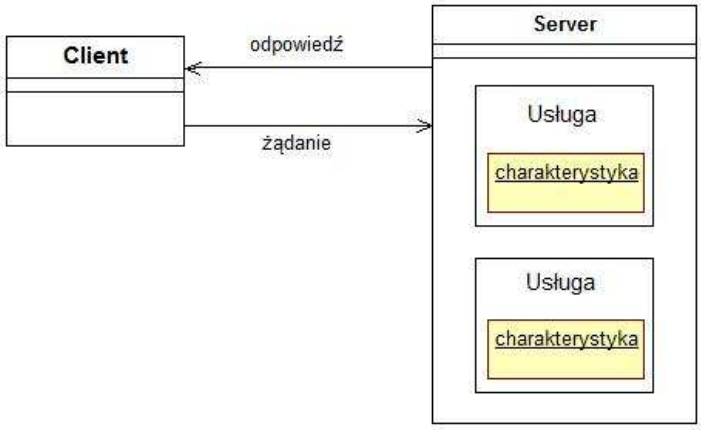
Standard Bluetooth Low Energy (BLE) definiuje dwa nowe elementy: ogólny profil atrybutów GATT (ang. *Generic Attribute Profile*) oraz protokół atrybutów ATT (ang. *Attribute Protocol*) [11]. Są one wykorzystywane przede wszystkim w urządzeniach o niskim poborze energii – każdy dodatkowy profil LE powinien z nich korzystać. Niemniej jednak mogą być również używane przez standardowe urządzenia Bluetooth BR/EDR. Rysunek 1.9 schematyczne obrazuje architekturę stosu protokołów Bluetooth LE.

Stos protokołów Bluetooth LE składa się z dwóch głównych warstw znajdujących się bezpośrednio poniżej warstwy aplikacji: hosta oraz kontrolera.

Na szczycie warstwy hosta umiejscowiony jest (opisany wcześniej) profil usług podstawowych GAP. Ogólny profil atrybutów (GATT) jest podstawowym profilem na stosie profili Bluetooth LE. Poniżej GATT umiejscowione są warstwy menedżera zabezpieczeń (SM) oraz protokołu atrybutów (ATT). Warstwa SM definiuje metody parowania i dystrybucji kluczy, udostępnia funkcje umożliwiające bezpieczne łączenie i wymianę danych z innym urządzeniem. Ogólny profil atrybutów GATT jest zbudowany na bazie protokołu atrybutów (ATT) i ustanawia wspólne ramy funkcjonowania dla danych transportowych urządzeń LE. Oznacza to, iż transmisja danych pomiędzy dwoma urządzeniami BLE obsługiwana jest przez GATT. GATT definiuje dwie role: serwera i klienta.
Serwer GATT określa funkcje urządzenia, definiuje dostępne atrybuty usług oraz ich charakterystyki, przechowuje dane transportowe, określa ich format, akceptuje żądania pochodzące od klienta i po odpowiednim skonfigurowaniu asynchronicznie wysyła informacje zwrotną do klienta, tak jak schematycznie pokazano to na rysunku 1.10 [11].

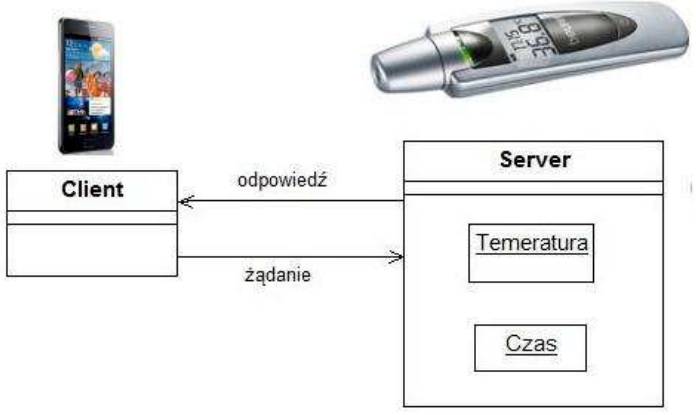


Rys. 1.9. Stos protokołów Bluetooth Low Energy (BLE)



Rys. 1.10. Role klienta i serwera w ramach GATT [http://www.bluegiga.com/home]

Protokół atrybutów ATT używany jest do lokalizowania dostępnych usług serwera. Implementuje on procedury, dzięki którym klient może uzyskać informację na temat usług udostępnianych przez serwer, tak jak schematycznie obrazuje to rysunek 1.11. Zadaniem ATT jest odpowiednie formatowanie danych w postaci usług i ich charakterystyk (właściwości). Usługi mogą zawierać zbiór właściwości. Charakterystyka może zawierać pojedynczą wartość i dowolną liczbę deskryptorów usług. W standardzie BLE, wszystkie zbiory danych, które są używane przez profil lub usługę nazywane są charakterystykami lub właściwościami.



Rys. 1.11. Role klienta i serwera w ramach ATT na przykładzie zdalnego pomiaru temperatury [http://www.bluegiga.com/home]

1.7. Podsumowanie

W rozdziale tym zostały zaprezentowane podstawowe wiadomości dotyczące standardu Bluetooth. Omawiane zagadnienia zostały potraktowane w sposób zwięzły, ale zupełnie wystarczający do zrozumienia problemów związanych z programową kontrolą transmisji bezprzewodowej realizowanej w standardzie Bluetooth. W dalszej części książki, wraz z wprowadzaniem konkretnych algorytmów mogących obsługiwać komunikację Bluetooth, omówione zagadnienia będą stopniowo uzupełniane. Bardziej szczegółowe informacje dotyczące teoretycznych podstaw standardu Bluetooth Czytelnik może znaleźć w bogatej literaturze przedmiotu [1-6, 8, 11-13].

ROZDZIAŁ 2

DETEKCJA I IDENTYFIKACJA URZĄDZEŃ BLUETOOTH. CZĘŚĆ I

2.1. Wiadomości podstawowe

Proces detekcji i identyfikacji urządzeń współpracujących ze sobą w obrębie doraźnie (ang. *ad och*) tworzonych sieci bezprzewodowych jest zasadniczo odmienny od procesu detekcji i identyfikacji występującego w ramach połączeń przewodowych [7,10]. W przypadku urządzeń komunikujących się za pośrednictwem łącza radiowego użytkownik początkowo nie ma wiedzy na temat potencjalnie dostępnych jednostek oraz oferowanych przez nie usług. W systemie Bluetooth problem ten rozwiązywany jest poprzez specjalnie dedykowane mechanizmy identyfikacji urządzeń będących aktualnie w zasięgu modułu radiowego urządzenia wykrywającego oraz przez odpowiednie wykorzystanie protokołu SDP oferowanego przez jednostki wykrywane. Aby jednostki Bluetooth mogły współpracować ze sobą muszą wyrazić zgodę na połączenie[1]. Pierwszym etapem tworzenia połączenia jest określenie jakie urządzenia Bluetooth są w zasięgu lokalnego modułu radiowego urządzenia wykrywającego. W tym celu oprogramowanie urządzenia wykrywającego wysyła serię specjalnych pakietów identyfikujących ID opatrzonych wspólnym dla wszystkich urządzeń kodem GIAC (ang. *General Inquiry Access Code*) i oczekuje odpowiedzi od urządzenia wykrywanego w postaci pakietu FHS (ang. *Frequency Hop Synchronization*), dzięki któremu oba urządzenia mogą zostać zsynchronizowane. Urządzenie wykrywające generuje pakiety ID za pośrednictwem tzw. wykrywającej sekwencji przeskoków częstotliwości. Oznacza to, iż w jednostkowym przedziale czasowym (ang. *slot time*) wysyła dwa pakiety ID, które powinny trafić do urządzenia wykrywanego, którego łącze radiowe generuje przeskoki częstotliwości co 2048 jednostkowych przedziałów czasowych (odpowiada to 1,28 sek).

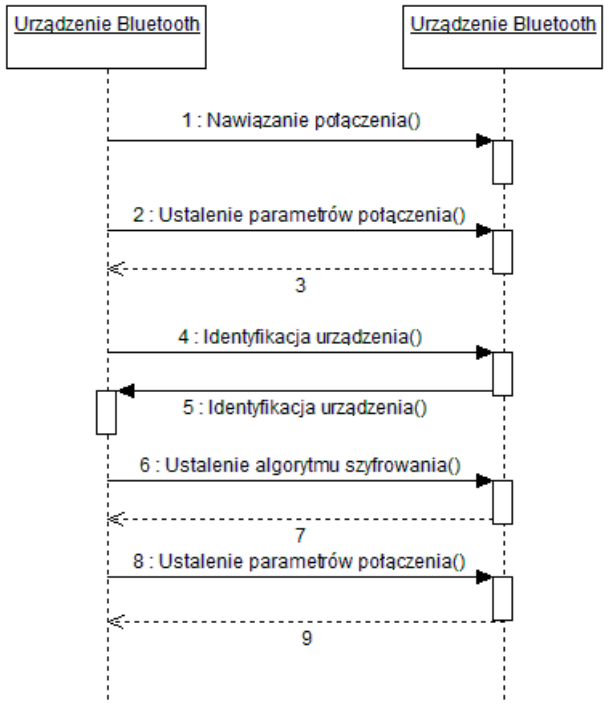
Aby uniknąć sytuacji, w której kilka urządzeń zdecyduje się jednocześnie odpowiedzieć na wezwanie jednostki wykrywającej, każde urządzenie po odebraniu pakietu wstrzymuje nadawanie, a następnie czeka przez pewną losową liczbę przedziałów czasowych i ponownie rozpoczyna nasłuchiwanie odpowiadając po ponownym odebraniu pakietu zawierającego ten sam kod GIAC. Na tej podstawie aplikacja sterująca urządzeniem wykrywającym tworzy listę jednostek wykrytych. Wyboru urządzenia, z którym ma być nawiązane połączenie dokonuje użytkownik lub zaimplementowana procedura (w zależności od wykorzystywanego oprogramowania).

W celu nawiązania połączenia oprogramowanie urządzenia inicjującego pobiera odpowiedni rekord z własnej bazy danych dostępnych urządzeń i kieruje bezpośrednie żądanie do jednostki, z którą chce nawiązać połączenie. Wywoływane urządzenie powinno pozostawać w stanie oczekiwania na wywołanie, w którym nasłuchuje pakiety zawierające własny adres. Gdy

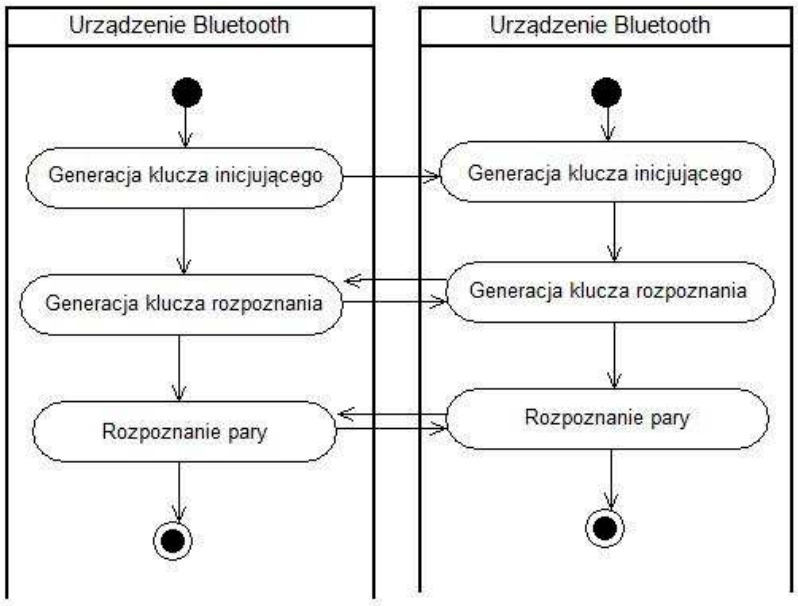
[1] Urządzenia z funkcją Bluetooth mogą być skonfigurowane w sposób uniemożliwiający skanowanie sygnałów identyfikujących – wówczas będą nierozpoznawalne.

urządzenie wywoływane odbierze taki pakiet, potwierdza możliwość nawiązania połączenia. Potwierdzenie transmitowane jest w postaci ramki danych zawierającej ten sam adres. Po odebraniu potwierdzenia jednostka wywołująca wysyła pakiet FHS. W oparciu o zawarte w nim informacje urządzenie wywołane wyznacza sekwencję skoków częstotliwości urządzenia wywołującego i zaczyna ją stosować. Po tej sekwencji wymiany danych oba urządzenia przechodzą w stan nawiązania połączenia. Urządzenie wywołujące staje się urządzeniem nadrzędnym, a urządzenie wywoływane urządzeniem podrzędnym. Przełączając się na ustaloną wcześniej sekwencję skoków częstotliwości urządzenie nadrzędne wysyła pakiet kontrolny w celu weryfikacji poprawności nawiązanego połączenia. Po weryfikacji nawiązania połączenia wykryte urządzenie przesyła do urządzenia wykrywającego informacje o udostępnianych usługach. W wyniku takiego procesu urządzenie nadrzędne gromadzi w pamięci informacje na temat usług udostępnianych przez jednostki podrzędne.

W trakcie wywoływania procedur detekcji i identyfikacji uzgadniane są warunki ustanawiania połączenia oraz dobierania w pary urządzeń [14,15,16]. Na rysunkach 2.1 oraz 2.2 pokazano ogólne schematy sekwencji oraz czynności odpowiednio dla ustanawiania połączenia oraz dobierania w pary urządzeń w systemie Bluetooth.



Rysunek 2.1. Ogólny diagram sekwencji dla procesu ustanawiania połączenia urządzeń w systemie Bluetooth



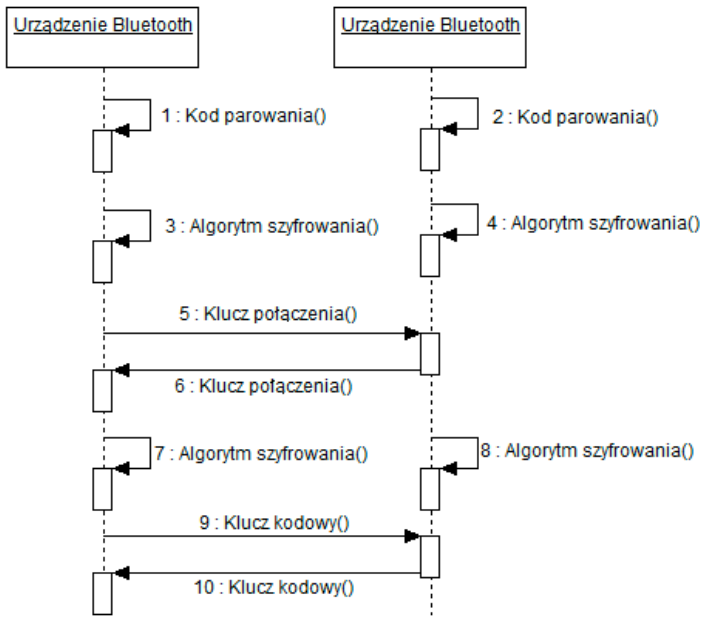
Rysunek 2.2. Ogólny diagram czynności dla procesu rozpoznawania urządzeń w systemie Bluetooth

Proces wzajemnego rozpoznawania urządzeń polega na znajomości ich adresów oraz na wiedzy o wspólnym kodzie dobierania w pary będącym ciągiem znaków (zawierającym zwykle od 4 do 16 cyfr), który wiąże jednostkę nadrzędną z urządzeniem podrzędnym. W zależności od kontekstu używania kod dobierania w pary nazywany jest hasłem, kodem, kluczem dostępu lub numerem PIN (ang. *Personal Identification Number*). W profilu GAP dwa urządzenia korzystające z tego samego klucza nazywane są połączonymi lub powiązanymi (ang. *bonded*), zestawionymi w pary, dobranymi w pary lub sparowanymi. W trakcie tworzenia powiązania menedżer łącza sprawdza czy urządzenia posługują się wspólnym kluczem. Proces ten określany jest mianem uwierzytelniania (ang. *authentication*).

Rozpoczęcie procesu uwierzytelniania poprzedzone jest ustaleniem metod szyfrowania z wykorzystaniem takiej samej liczby pseudolosowej w obu urządzeniach. W tym celu urządzenie nadrzędne wysyła wygenerowaną przez specjalny algorytm liczbę pseudolosową do urządzenia podrzędnego. Obie jednostki inicjują mechanizmy szyfrowania na podstawie tej liczby. W następnym etapie urządzenie nadrzędne transmituje do jednostki podrzędnej kolejną liczbę pseudolosową, która za pomocą klucza jest szyfrowana przez algorytm zaimplementowany w oprogramowaniu tego urządzenia. Klucz ten oparty jest na wspólnym dla obu urządzeń kodzie dobierania w pary. Zaszyfrowana w ten sposób liczba retransmitowana jest do urządzenia nadrzędnego, które porównuje dane otrzymane od jednostki podrzędnej z

wynikiem analogicznej operacji wykonanej przez macierzyste oprogramowanie. Jeżeli wyniki są zgodne, oznacza to iż oba urządzenia posługują się takim samym kluczem.

Poufne hasła, klucze dostępu lub numery PIN nie są transmitowane do urządzeń, przez co program zarządzający może tworzyć tablice wcześniej zidentyfikowanych urządzeń, dzięki czemu powtórny proces łączenia może przebiegać dużo sprawniej. Klucz połączeniowy jest tworzony w trakcie ustanawiania połączenia. Generowany jest na podstawie kodów dobierania w pary, jakie użytkownicy wpisują do urządzeń, tak jak pokazano to na rysunku 2.3. Kody wpisane w obydwu urządzeniach muszą być identyczne. Rozmiar typowego kodu dobierania w pary nie przekracza 16 bajtów.



Rysunek 2.3. Ogólny diagram sekwencji dla generowania klucza połączeniowego na podstawie kodu dobierania w pary

Urządzenia zgodne ze specyfikacją Bluetooth SIG w celu realizacji mechanizmów uwierzytelniania wykorzystują specjalistyczne algorytmy (w większości przypadków) bazujące na algorytmie o nazwie SAFER+. Generuje on 128 bitowe klucze szyfrujące w oparciu o liczby pseudolosowe, kody dobierania w pary oraz klucze połączeniowe[2].

Ze względów bezpieczeństwa większość urządzeń z obsługą funkcji Bluetooth wymaga używania kodu parowania. W zależności od konstrukcji i

[2] w 2005 roku opracowano metodę łamania kodów PIN z wykorzystaniem właściwości algorytmu SAFER+ (4 cyfrowy PIN rozszyfrowano w ok. 0,1 sekundy, zaś 7 cyfrowy w ok. 1,5 minuty).

wykorzystywanego oprogramowania, urządzenia Bluetooth można dobierać w pary wykorzystując następujące metody:

- każdorazowo uzgadniane pomiędzy urządzeniami kody,
- porównanie (przynajmniej) sześciocyfrowych liczb generowanych przez specjalnie dedykowany algorytm (ang. *Numeric Comparison*),
- dane OOB (ang. *Out-of-Band*). OOB to specjalnie zestawiany kanał komunikacyjny pozwalający m.in. na przekazywanie kluczowych danych z wykorzystaniem określonego protokołu pakietowego. Pakiety OOB zazwyczaj umieszczane są w oddzielnym buforze i wysyłane zawsze w pierwszej kolejności. Często do tego buforu nie ma dostępu ani system operacyjny, ani żaden działający pod jego nadzorem program, z wyjątkiem uprzywilejowanej aplikacji zestawiającej połączenie OOB,
- w przypadku urządzeń nie zaopatrzonych w wyświetlacz danych (np. klawiatura, mysz) można posłużyć się zdefiniowanym dla urządzenia unikalnym kluczem (ang. *passkey*). Jest to przykład tzw. „jednokierunkowego dobierania w pary".

2.2. Podstawowe funkcje

Obecny podrozdział zawiera opis podstawowych funkcji SDK API Windows służących do implementacji procedur wyszukiwania urządzeń pozostających w zasięgu lokalnego odbiornika radiowego Bluetooth. Zostaną w nim przedstawione zagadnienia związane z implementacją funkcji API Windows wykorzystywanych do programowej kontroli urządzeń Bluetooth. Definicje omawianych funkcji znajdują się w modułach *Ws2bth.h*, *Bthsdpdef.h* oraz *BluetoothAPIs.h*. Moduł *Ws2bth.h* powinien być włączany do kodu po obowiązkowo używanym module *Winsock2.h*. Dodatkowo program główny powinien być statycznie łączony z biblioteką *Bthprops.lib*. Biblioteka ta jest standardowo dostępna w zasobach systemów Windows XP z Service Pack 2 i 3, Vista oraz 7 i 8.

2.2.1. Funkcja BluetoothAuthenticateDevice()

Funkcja `BluetoothAuthenticateDevice()` wysyła żądanie identyfikacji i uwierzytelnienia do urządzenia Bluetooth.

```
DWORD BluetoothAuthenticateDevice(
    HWND hwndParent,
    HANDLE hRadio,
    BLUETOOTH_DEVICE_INFO *pbtdi,
    PWCHAR pszPasskey,
    ULONG ulPasskeyLength
);
```

Parametr hwndParent określa okno dialogowe kreatora identyfikacji urządzenia. Jeżeli parametrowi zostanie przypisana wartość NULL, okno dialogowe będzie wyświetlane na pulpicie. Parametr hRadio identyfikuje moduł radiowy Bluetooth. Moduł radiowy może być modułem wbudowanym (np. w laptopie) lub adapterem Bluetooth USB. Jeżeli hRadio zawiera wartość NULL, urządzenie identyfikowane jest na podstawie skanowania wszystkich zainstalowanych i dostępnych modułów radiowych. Parametr pbtdi wskazuje na strukturę:

```
typedef struct _BLUETOOTH_DEVICE_INFO {
  DWORD             dwSize;
  BLUETOOTH_ADDRESS Address;
  ULONG             ulClassofDevice;
  BOOL              fConnected;
  BOOL              fRemembered;
  BOOL              fAuthenticated;
  SYSTEMTIME        stLastSeen;
  SYSTEMTIME        stLastUsed;
  WCHAR             szName[BLUETOOTH_MAX_NAME_SIZE];
} BLUETOOTH_DEVICE_INFO;
```

Parametr pszPasskey jest kodem używanym do parowania urządzeń. Kod dobierania w pary powinien być przechowywany w formie łańcucha znaków zakończonego zerowym ogranicznikiem (ang. *null-terminated string*). ulPasskeyLength jest długością kodu parowania, którego rozmiar nie powinien przekraczać wartości BLUETOOTH_MAX_PASSKEY_SIZE. Prawidłowo wykonana funkcja zwraca wartość ERROR_SUCCESS.
Najczęściej występujące kody błędów to:
ERROR_CANCELLED – użytkownik przerwał operację identyfikacji urządzenia,
ERROR_INVALID_PARAMETER – struktura przechowująca informacje na temat urządzenia jest niewłaściwie użyta,
ERROR_NO_MORE_ITEMS – wskazane urządzenie zostało już wcześniej zidentyfikowane.
Programując w systemach Windows Vista z SP2 lub Windows 7 zamiast BluetoothAuthenticateDevice() należy używać funkcji BluetoothAuthenticateDeviceEx().

W tabeli 2.1 zaprezentowano zasobu struktury BLUETOOTH_DEVICE_INFO.

Tabela 2.1.Specyfikacja struktury BLUETOOTH_DEVICE_INFO

Element struktury	Znaczenie
dwSize	Rozmiar struktury w bajtach , który każdorazowo należy prawidłowo określić za pomocą operatora sizeof(). W przeciwnym

	wypadku funkcja nie będzie działać poprawnie.
`Address`	Adres urządzenia
`ulClassofDevice`	Klasa urządzenia
`fConnected`	Określa czy urządzenie jest podłączone i aktualnie używane
`fRemembered`	Określa czy urządzenie jest zapamiętane przez system. Nie wszystkie urządzenia zapamiętane przez system były identyfikowane
`fAuthenticated`	Określa czy urządzenie jest zidentyfikowane, podłączone lub sparowane. Wszystkie zidentyfikowane urządzenia są pamiętane w systemie (jeżeli nie zostaną wcześniej jawnie usunięte).

`stLastSeen`	Data ostatniego wykrycia urządzenia zapisana jest w polach struktury
	`typedef struct _SYSTEMTIME {`
	`WORD wYear;`
	`WORD wMonth;`
	`WORD wDayOfWeek;`
	`WORD wDay;`
	`WORD wHour;`
	`WORD wMinute;`
	`WORD wSecond;`
	`WORD wMilliseconds;`
	`} SYSTEMTIME, *PSYSTEMTIME;`

`stLastUsed`	Data ostatniego połączenia z urządzeniem zapisana jest w polach struktury `SYSTIME`
`szName`	Nazwa urządzenia

2.2.2. Funkcja BluetoothAuthenticateDeviceEx()

Stosowany przez Windows API protokół dobierania urządzeń w pary SSP (ang. *Security Simple Paring*) implementuje bezpieczne i nieskomplikowane z technicznego punku widzenia mechanizmy uwierzytelniania i dobierania w pary urządzeń. Mechanizmy te chronią system przed biernym podsłuchiwaniem a także przed atakami typu MITM (ang. *Man in the middle*). W atakach typu MITM, wykorzystuje się urządzenie pośredniczące, które przekazuje informacje między dwoma urządzeniami stwarzając wrażenie że te dwa urządzenia komunikują się bezpośrednio między sobą.

Funkcja BluetoothAuthenticateDeviceEx() wysyła rozkaz uwierzytelniania do urządzenia z włączoną obsługą Bluetooth. W odróżnieniu od poprzedniej, umożliwia przekazanie szczegółowych żądań za pośrednictwem danych OOB.

```
HRESULT WINAPI BluetoothAuthenticateDeviceEx(
            HWND hwndParentIn,
  __in_opt  HANDLE hRadioIn,
  __inout   BLUETOOTH_DEVICE_INFO *pbtdiInout,
  __in_opt  PBTH_OOB_DATA pbtOobData,
            BLUETOOTH_AUTHENTICATION_REQUIREMENTS
            authenticationRequirement
);
```

Parametr hwndParentIn identyfikuje okno dialogowe kreatora uwierzytelniania. Jeżeli parametrowi zostanie przypisana wartość NULL, okno kreatora będzie wyświetlane na pulpicie. Opcjonalnie występujące parametry hRadioIn oraz pbtdiInout spełniają taką samą rolę jak hRadio i pbtdi w funkcji BluetoothAuthenticateDevice().

Tabela 2.2. Specyfikacja typu wyliczeniowego BLUETOOTH_AUTHENTICATION_REQUIREMENTS

Element typu wyliczeniowego	Wartość	Znaczenie
MITMProtectionNotRequired	0x00	Zabezpieczenia na wypadek ataku MITM nie są wymagane w trakcie uwierzytelniania.
MITMProtectionRequired	0x01	Zabezpieczenia na wypadek ataku MITM są bezwzględnie wymagane w trakcie uwierzytelniania.
MITMProtectionNotRequiredBonding	0x02	Zabezpieczenia na wypadek ataku MITM nie są wymagane w trakcie inicjowania połączenia.
MITMProtectionRequiredBonding	0x03	Zabezpieczenia na wypadek ataku MITM są bezwzględnie wymagane w trakcie inicjowania połączenia.
MITMProtectionNotRequiredGeneralBonding	0x04	Zabezpieczenia na wypadek ataku MITM nie są wymagane w trakcie realizowania połączenia.
MITMProtectionRequiredGeneralBonding	0x05	Zabezpieczenia na wypadek ataku MITM nie są wymagane w trakcie realizowania połączenia.
MITMProtectionNotDefined	0xff	Nie zdefiniowano zabezpieczeń na wypadek ataku MITM.

Opcjonalny parametr pbtOobData określa możliwość przetwarzania danych OOB. Parametr authenticationRequirement jest elementem typu wyliczeniowego BLUETOOTH_AUTHENTICATION_REQUIREMENTS (patrz tabela 2.2) przechowującego dane wymagane dla ochrony przed atakami typu MITM.

2.2.3. Funkcja BluetoothDisplayDeviceProperties()

Funkcja BluetoothDisplayDeviceProperties() uruchamia panel kontrolny wyświetlany w postaci okna dialogowego zawierającego właściwości aktualnie sparowanego urządzenia.

```
BOOL BluetoothDisplayDeviceProperties(
    HWND hwndParent,
    BLUETOOTH_DEVICE_INFO *pbtdi
);
```

Parametr hwndParent identyfikuje właściciela okna dialogowego, w którym wyświetlane są właściwości urządzenia. Wskaźnik pbtdi wskazuje na strukturę BLUETOOTH_DEVICE_INFO. Prawidłowo wykonana funkcja zwraca wartość TRUE, w przeciwnym wypadku FALSE.

2.2.4. Funkcja BluetoothEnableDiscovery()

Funkcja BluetoothEnableDiscovery() modyfikuje znacznik usług aktualnie udostępnianych przez moduł radiowy.

```
BOOL BluetoothEnableDiscovery(
    HANDLE hRadio,
    BOOL fEnabled
);
```

Parametr hRadio identyfikuje moduł radiowy Bluetooth. Znacznik fEnabled określa czy usługa powinna być dostępna (TRUE), czy nie (FALSE).

2.2.5. Funkcja BluetoothEnableIncomingConnections()

Funkcja BluetoothEnableIncomingConnections() określa akceptowalność lokalnych połączeń Bluetooth.

```
BOOL BluetoothEnableIncomingConnections(
    HANDLE hRadio,
    BOOL fEnabled
);
```

Parametr hRadio identyfikuje lokalny moduł radiowy. Jeżeli wskaźnik jest zerowy (NULL) oznacza to, iż skanowane będą wszystkie istniejące moduły.

Znacznik fEnabled określa akceptowalność połączenia. Wartość TRUE oznacza, iż połączenie jest akceptowane.

2.2.6. Funkcja BluetoothEnumerateInstalledServices()

Funkcja BluetoothEnumerateInstalledServices() wylicza identyfikatory GUID identyfikujące usługi dostępne w urządzeniach z włączoną opcją Bluetooth.

```
DWORD BluetoothEnumerateInstalledServices(
    HANDLE hRadio,
    BLUETOOTH_DEVICE_INFO *pbtdi,
    DWORD *pcServices,
    GUID *pGuidServices
);
```

Znaczenie parametru hRadio jest identyczne jak poprzednio. Wskaźnik pbtdi wskazuje na strukturę BLUETOOTH_DEVICE_INFO. Traktowany jako parametr wejściowy pcServices jest liczbą rekordów (usług) wskazywanych przez pGuidServices. Wykorzystywany jako parametr wyjściowy pcServices podaje liczbę rekordów opisujących usługi wskazywane przez pGuidServices. Wskaźnik pGuidServices wskazuje na bufor pamięci przechowujący identyfikatory GUID zainstalowanych usług urządzenia. Rozmiar bufora nie powinien być mniejszy niż sizeof(pGuidServices) bajtów. Jeżeli wskaźnikowi pGuidServices przypisana jest wartość NULL, parametr wyjściowy pcServices wskazuje na liczbę dostępnych usług. Prawidłowo wykonana funkcja zwraca wartość ERROR_SUCCESS.

2.3. Funkcje rodziny BluetoothXxxDeviceXxx()

Funkcje rodziny BluetoothXxxDeviceXxx() umożliwiają programom współpracę z będącymi w zasięgu urządzeniami Bluetooth.

2.3.1. Funkcja BluetoothFindFirstDevice()

Funkcja BluetoothFindFirstDevice() rozpoczyna proces wyszukiwania będących w zasięgu lokalnego modułu radiowego zewnętrznych urządzeń z włączoną opcją Bluetooth.

```
HBLUETOOTH_DEVICE_FIND BluetoothFindFirstDevice(
    BLUETOOTH_DEVICE_SEARCH_PARAMS *pbtsp,
    BLUETOOTH_DEVICE_INFO *pbtdi
);
```

Wskaźnik pbtsp wskazuje na strukturę BLUETOOTH_DEVICE_SEARCH_-PARAMS:

```
typedef struct {
  DWORD  dwSize;
  BOOL   fReturnAuthenticated;
  BOOL   fReturnRemembered;
  BOOL   fReturnUnknown;
  BOOL   fReturnConnected;
  BOOL   fIssueInquiry;
  UCHAR  cTimeoutMultiplier;
  HANDLE hRadio;
} BLUETOOTH_DEVICE_SEARCH_PARAMS;
```

której zasoby zaprezentowano w tabeli 2.3. Wskaźnik `pbtdi` wskazuje na strukturę `BLUETOOTH_DEVICE_INFO`.

Tabela 2.3. Specyfikacja struktury BLUETOOTH_DEVICE_SEARCH_PARAMS

Element struktury	Znaczenie
`dwSize`	Rozmiar struktury w bajtach, który każdorazowo należy prawidłowo określić za pomocą operatora `sizeof()`. W przeciwnym wypadku funkcja nie będzie działać poprawnie.
`fReturnAuthenticated`	Znacznik określający czy funkcja, wyszuka urządzenia uprzednio uwierzytelnione.
`fReturnRemembered`	Znacznik określający czy funkcja, wyszuka urządzenia uprzednio zapamiętane w systemie.
`fReturnUnknown`	Znacznik określający czy funkcja, wyszuka niezidentyfikowane urządzenie.
`fReturnConnected`	Znacznik określający czy funkcja, wyszuka przyłączone i aktualnie używane urządzenie.
`fIssueInquiry`	Znacznik określający, czy w trakcie wyszukiwania będzie wysłane do urządzenia zapytanie o jego identyfikację.
`cTimeoutMultiplier`	Wartość z zakresu <1;48> określająca czas przeterminowania w oczekiwaniu na wykrycie urządzenia. Parametr ten wyrażony jest w jednostkach równych 1.28 sekund. Np., `cTimeoutMultiplier=10` odpowiada 12.8 sek.
`hRadio`	Identyfikator odbiornika radiowego. Wartość NULL oznacza rozpoczęcie procesu wyszukiwania urządzeń dla wszystkich dostępnych w systemie odbiorników radiowych Bluetooth.

Prawidłowo wykonana funkcja BluetoothFindFirstDevice() zwraca identyfikator typu HBLUETOOTH_DEVICE_FIND, lub w przypadku niepowodzenia wartość NULL. Kody ewentualnych błędów można odczytać za pomocą funkcji GetLastError() i mogą one przyjmować następujące wartości: ERROR_INVALID_PARAMETER – jeden ze wskaźników pbtsp lub pbtdi wskazuje wartość pustą NULL,
ERROR_REVISION_MISMATCH – rozmiary struktur wskazywanych przez pbtsp lub pbtdi nie zostały prawidłowo określone i wpisane do ich atrybutów dwSize.

2.3.2. Funkcja BluetoothFindNextDevice()

Funkcja BluetoothFindNextDevice() kontynuuje proces wyszukiwania urządzeń z włączoną opcją Bluetooth na podstawie identyfikatora hFind zwracanego przez funkcję BluetoothFindFirstDevice().

```
BOOL BluetoothFindNextDevice(
    HBLUETOOTH_DEVICE_FIND hFind,
    BLUETOOTH_DEVICE_INFO *pbtdi
);
```

Wskaźnik pbtdi wskazuje na strukturę BLUETOOTH_DEVICE_INFO, w której polach przechowywane są informacje na temat kolejnego zidentyfikowanego urządzenia. Prawidłowo wykonana funkcja zwraca wartość TRUE, zaś w przypadku niepowodzenia – FALSE. Kod ewentualnego błędu powinien być określony za pomocą funkcji GetLastError() i może być jednym z:
ERROR_INVALID_HANDLE – identyfikator urządzenia wskazuje na wartość NULL,
ERROR_NO_MORE_ITEMS – nie znaleziono więcej urządzeń,
ERROR_OUTOFMEMORY – nie ma wystarczającej ilości pamięci, aby kontynuować proces wyszukiwania.

2.3.3. Funkcja BluetoothFindDeviceClose()

Funkcja BluetoothFindDeviceClose() zwalnia identyfikator hFind uprzednio przydzielony za pomocą funkcji BluetoothFindFirstDevice() i kończy proces wyszukiwania będących w zasięgu urządzeń w włączoną opcją Bluetooth.

```
BOOL BluetoothFindDeviceClose(
    HBLUETOOTH_DEVICE_FIND hFind
);
```

Prawidłowo wykonana funkcja zwraca wartość TRUE. Ewentualne błędy powstałe w trakcie jej wywołania powinny być diagnozowane za pomocą funkcji GetLastError().

2.3.4. Funkcja BluetoothGetDeviceInfo()

Funkcja BluetoothGetDeviceInfo() aktualizuje bufor danych przechowujący pola struktury wskazywanej przez pbtdi. W polach struktury BLUETOOTH_DEVICE_INFO zapisane są informacje o zidentyfikowanym zewnętrznym urządzeniu Bluetooth.

```
DWORD BluetoothGetDeviceInfo(
    HANDLE hRadio,
    BLUETOOTH_DEVICE_INFO *pbtdi
);
```

Identyfikator hRadio zwracany jest poprzez funkcję BluetoothFindFirstRadio() lub SetupDiEnumerateDeviceInterfaces() [10].
Prawidłowo wykonana funkcja zwraca wartość ERROR_SUCCESS. W przypadku niepowodzenie wartość zwracana prze funkcję może być jedną z:
ERROR_REVISION_MISMATCH – rozmiar struktury BLUETOOTH_DEVICE_INFO został niewłaściwie określony;
ERROR_NOT_FOUND – w systemie nie zidentyfikowano żądanego modułu radiowego Bluetooth lub adres urządzenia zapisany w polu Address struktury BLUETOOTH_DEVICE_INFO jest zerowy;
ERROR_INVALID_PARAMETER – wskaźnik pbtdi wskazuje wartość pustą.

2.3.5. BluetoothRemoveDevice()

Funkcja BluetoothRemoveDevice() usuwa urządzenie o adresie wskazywanym przez pAddress z listy zidentyfikowanych urządzeń zewnętrznych.

```
DWORD BluetoothRemoveDevice(
    BLUETOOTH_ADDRESS *pAddress
);
```

Prawidłowo wykonana funkcja zwraca wartość ERROR_SUCCESS, w przeciwnym wypadku – ERROR_NOT_FOUND.

2.3.6. Funkcja BluetoothSelectDevices()

Funkcja BluetoothSelectDevices() uaktywnia okno dialogowe kreatora, za pomocą którego można określić i sparować wykryte przez moduł radiowy zewnętrzne urządzenie z włączoną opcją Bluetooth.

```
BOOL BluetoothSelectDevices(
```

```
        BLUETOOTH_SELECT_DEVICE_PARAMS *pbtsdp
    );
```

Lista wyświetlanych urządzeń określona jest przez odpowiednie kombinacje znaczników będących polami struktury BLUETOOTH_SELECT_DEVICE_PARAMS wskazywanej przez parametr pbtsdp. W tabeli 2.4 omówiono zasoby tej struktury.

```
typedef struct _BLUETOOTH_SELECT_DEVICE_PARAMS {
  DWORD                  dwSize;
  ULONG                  cNumOfClasses;
  BLUETOOTH_COD_PAIRS    *prgClassOfDevices;
  LPWSTR                 pszInfo;
  HWND                   hwndParent;
  BOOL                   fForceAuthentication;
  BOOL                   fShowAuthenticated;
  BOOL                   fShowRemembered;
  BOOL                   fShowUnknown;
  BOOL                   fAddNewDeviceWizard;
  BOOL                   fSkipServicesPage;
  PFN_DEVICE_CALLBACK    pfnDeviceCallback;
  LPVOID                 pvParam;
  DWORD                  cNumDevices;
  PBLUETOOTH_DEVICE_INFO pDevices;
} BLUETOOTH_SELECT_DEVICE_PARAMS;
```

Tabela 2.4. Specyfikacja struktury BLUETOOTH_SELECT_DEVICE_PARAMS

Element struktury	Znaczenie
dwSize	Rozmiar struktury w bajtach, który każdorazowo należy prawidłowo określić za pomocą operatora sizeof(). W przeciwnym wypadku funkcja nie będzie działać poprawnie.
cNumOfClasses	Numer klasy instalacji urządzenia w tablicy klas prgClassOfDevices. Wartość zero oznacza, iż będą wyświetlane urządzenia na podstawie wszystkich klas instalacji.
prgClassOfDevices	Tablica klas instalacji urządzeń.
pszInfo	Informacje podawane są w formie tekstowej.
hwndParent	Identyfikator nadrzędnego okna dialogowego. Wartość NULL oznacza, że okno dialogowe kreatora będzie wyświetlane bezpośrednio na pulpicie.
fForceAuthentication	Znacznik określający konieczność (TRUE) potwierdzenia uwierzytelnienia urządzenia.

`fShowAuthenticated`	Znacznik określający, czy uprzednio uwierzytelnione urządzenia będą pokazywane przez program, czy też nie (`FALSE`).
`fShowRemembered`	Znacznik określający, czy będą pokazywane (`TRUE`) zapamiętane uprzednio w systemie urządzenia Bluetooth.
`fShowUnknown`	Znacznik określający, czy będą pokazywane (`TRUE`) urządzenia, których autentyczność nie została wcześniej potwierdzona lub urządzenia wcześniej nie zapamiętane w systemie.
`fAddNewDeviceWizard`	Znacznik określający, czy należy wyświetlić (`TRUE`) nowe okno dialogowe „Sparuj z urządzeniem bezprzewodowym”, czy jedynie okno wyboru nowego urządzenia (`FALSE`).
`fSkipServicesPage`	Znacznik określający, czy pominąć (`TRUE`) kartę „Usługi” w widoku właściwości urządzenia.

`pfnDeviceCallback`	Fakt zakończenia wykonywania np. operacji asynchronicznej może być sygnalizowany poprzez wywołanie określonej funkcji powrotnej. `pfnDeviceCallback` wskazuje na funkcję o prototypie:
	`BOOL PFN_DEVICE_CALLBACK(`
	`  LPVOID pvParam,`
	`  PBLUETOOTH_DEVICE_INFO pDevice`
	`);`

`pvParam`	Parametr `pvParam` w funkcji wskazywanej przez `pfnDeviceCallback`
`cNumDevices`	Określa liczbę identyfikowanych urządzeń, wartość 0 oznacza, że nie ustalono górnego limitu.
`pDevices`	Wskaźnik do tablicy struktur `BLUETOOTH_DEVICE_INFO`.

Prawidłowo wykonana funkcja `BluetoothSelectDevices()` zwraca wartość `TRUE`, zaś w przypadku niepowodzenia – `FALSE`. Kod ewentualnego błędu powinien być określony za pomocą funkcji `GetLastError()` i może być jednym z:
`ERROR_CANCELLED` – użytkownik anulował żądanie,
`ERROR_INVALID_PARAMETER` – wskaźnik wskazuje wartość `NULL`,
`ERROR_REVISION_MISMATCH` – rozmiar struktury wskazywanej przez `pbtsdp` nie został określony.

2.3.7. Funkcja BluetoothSelectDevicesFree()

Funkcja `BluetoothSelectDevicesFree()` zwalnia zasoby struktury wskazywanej przez `pbtsdp` uprzednio przydzielone za pomocą funkcji `BluetoothSelectDevices()`.

```
BOOL BluetoothSelectDevicesFree(
    BLUETOOTH_SELECT_DEVICE_PARAMS *pbtsdp
);
```

Prawidłowo wykonana funkcja zwraca wartość `TRUE`.

2.3.8. Funkcja BluetoothUpdateDeviceRecord()

Funkcja `BluetoothUpdateDeviceRecord()` uaktualnia informacje na temat urządzenia uprzednio zapisane w polach struktury wskazywanej przez `pbtdi`.

```
DWORD BluetoothUpdateDeviceRecord(
    BLUETOOTH_DEVICE_INFO *pbtdi
);
```

Prawidłowo wykonana funkcja zwraca wartość `ERROR_SUCCESS`. Kody ewentualnych błędów, to:
`ERROR_INVALID_PARAMETER` – wskaźnik `pbtdi` wskazuje wartość `NULL`,
`ERROR_REVISION_MISMATCH` – rozmiar struktury wskazywanej przez `pbtdi` nie został określony prawidłowo lub/i nie został wpisany do pola `dwSize`.

2.3.9. Przykłady

Na listingu 2.1 pokazano jeden ze sposobów praktycznego wykorzystania w programie funkcji `BluetoothSelectDevices()` oraz `BluetoothSelectDevicesFree()`. Program używa wygodnych procedur za pomocą których można szybko zidentyfikować urządzenie z włączoną opcją Bluetooth oraz sparować je z komputerem za pomocą wybranej opcji oraz kodu parowania. Wynik działania programu obrazuje sekwencja rysunków 2.4-2.7.

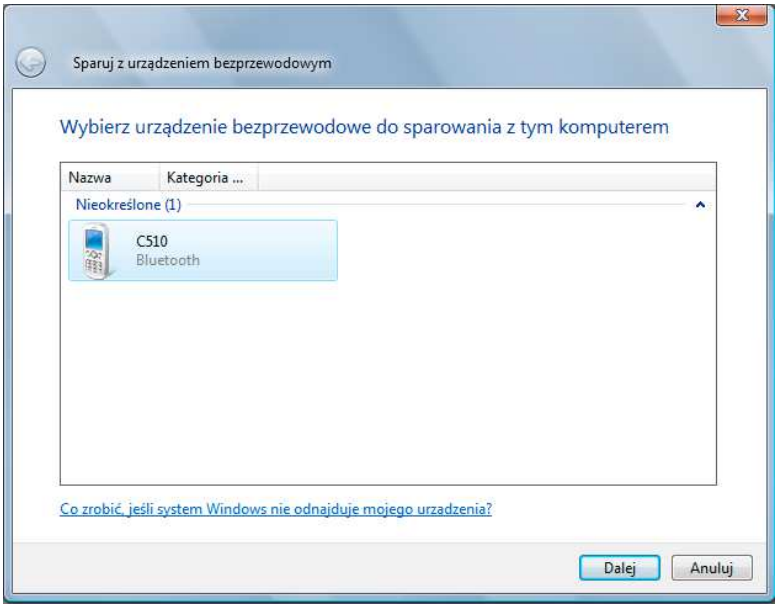
Listing 2.1. Programowa obsługa wykrywania i parowania urządzeń bezprzewodowych Bluetooth

```
#include <iostream>
#pragma option push -a1
  #include <winsock2>
  #include "Ws2bth.h"
  #include "BluetoothAPIs.h"
#pragma option pop

#pragma comment(lib, "Bthprops.lib")
```

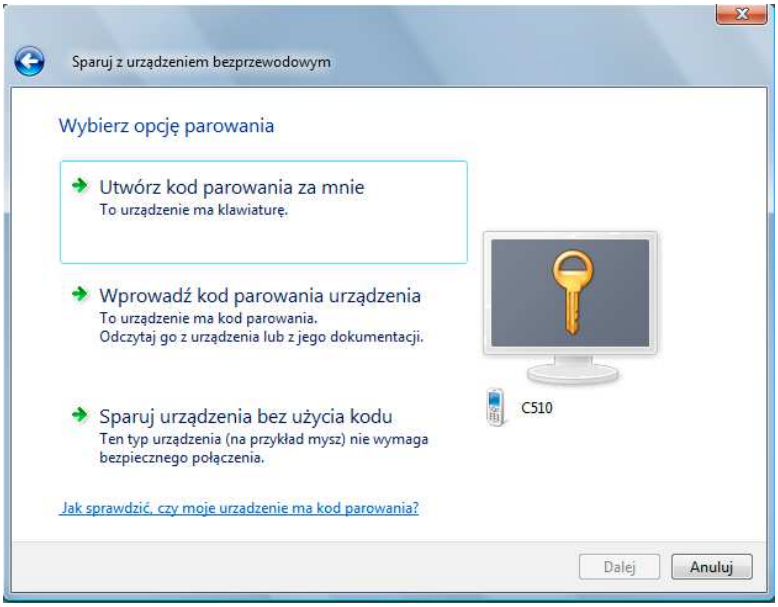
```
using namespace std;

BOOL __stdcall devInfoCallback(LPVOID pvParam,
                          PBLUETOOTH_DEVICE_INFO pDevice)
{
  //...
  return TRUE;
}
//-----------------------------------------------------
int main()
{
   BLUETOOTH_SELECT_DEVICE_PARAMS bthsdp={sizeof(bthsdp)};

   bthsdp.hwndParent = NULL;
   bthsdp.fShowUnknown = TRUE;
   bthsdp.fAddNewDeviceWizard = TRUE;
   bthsdp.fSkipServicesPage = FALSE;
   bthsdp.fShowRemembered = TRUE;
   bthsdp.fShowAuthenticated = TRUE;
   bthsdp.pfnDeviceCallback = devInfoCallback;
   bthsdp.cNumDevices = 0;
   //...
   BOOL bthSelectDevices =
        BluetoothSelectDevices(&bthsdp);
   if (bthSelectDevices) {
      BLUETOOTH_DEVICE_INFO * pbtdi = bthsdp.pDevices;
      pbtdi->dwSize = sizeof(pbtdi);
      for (ULONG cDevice = 0; cDevice <
           bthsdp.cNumDevices; cDevice ++ ) {
         if (pbtdi->fAuthenticated ||
             pbtdi->fRemembered ) {
             wprintf(L"Device: %s\n",pbtdi->szName);
             wprintf(L"Class of Device: %d\n",
                     pbtdi->ulClassofDevice);
             //...
         }
         pbtdi = (BLUETOOTH_DEVICE_INFO *)
                 ((LPBYTE)pbtdi + pbtdi->dwSize);
      }
      BluetoothSelectDevicesFree(&bthsdp);
   }//koniec if
   system("PAUSE");
   return 0;
}
//-----------------------------------------------------
```

Ze względów bezpieczeństwa większość urządzeń z obsługą funkcji Bluetooth wymaga dokonania wyboru odpowiedniej opcji dobierania w pary oraz użycia specjalnego kodu parowania. Kody na urządzeniu i na komputerze, z którym jest ono zestawiane muszą być zgodne. Przed zakończeniem procesu dobierania w pary urządzeń jest wyświetlany monit o sprawdzenie tego kodu, tak jak pokazano to na rysunku 2.6.

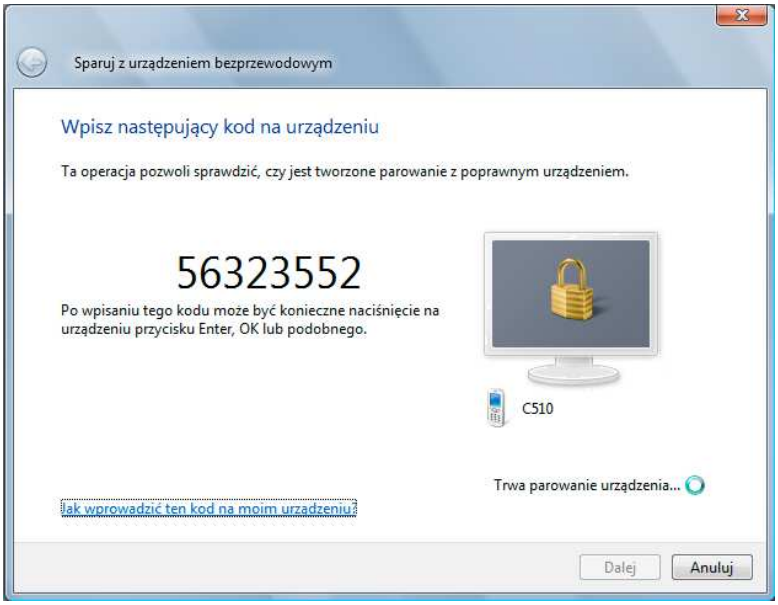
Podczas inicjowania procedur uwierzytelniania kod wybranego urządzenia jest wykorzystywany do utworzenia 128 bitowego klucza zależnego od adresu urządzenia nadrzędnego oraz wygenerowanej liczby pseudolosowej wspólnej dla obu zestawianych w parę urządzeń. Procedura ta zapewnia, że oba urządzenia posługują się wspólnym kluczem szyfrowania oraz że ten sam kod został wprowadzony w obu urządzeniach. Z kolei klucz ten wykorzystywany jest do utworzenia nowego 128 bitowego klucza zwanego kluczem łącza. Klucz łącza jest tworzony na bazie klucza bieżącego, adresu wykrytego urządzenia oraz nowej liczby pseudolosowej. Dla klucza łącza i określonego adresu generowany jest z kolei klucz szyfrowania zwany też kluczem kodowym. Klucz kodowy wraz z aktualną wartością wskazań zegara urządzenia oraz jego adresem używany jest do inicjowania mechanizmów szyfrowania wykorzystywanych w trakcie szyfrowania i deszyfrowania transmitowanych danych.



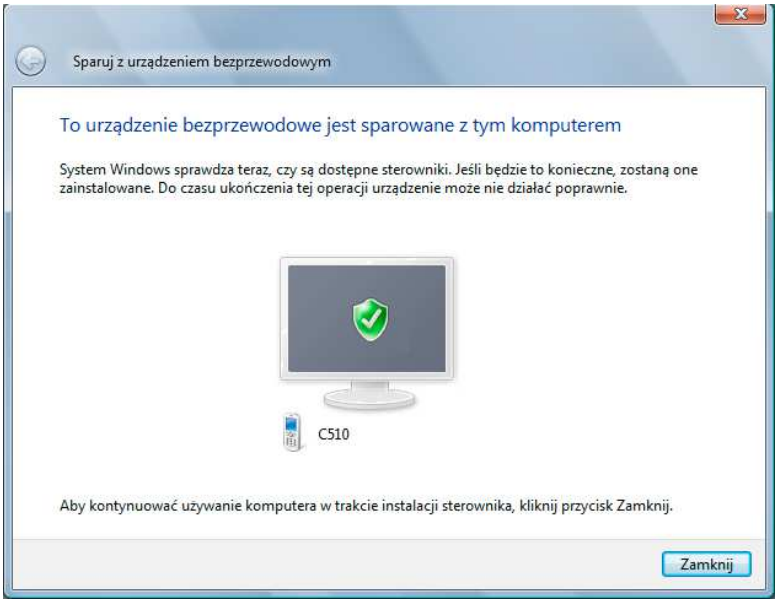
Rysunek 2.4. Okno dialogowe wyboru będących w zasięgu urządzeń z włączoną funkcją Bluetooth



Rysunek 2.5. Wybór opcji dobierania urządzeń w pary



Rysunek 2.6. Sprawdzanie kodu dobierania w pary



Rysunek 2.7. Komunikat o zakończeniu procesu dobierania urządzeń w pary

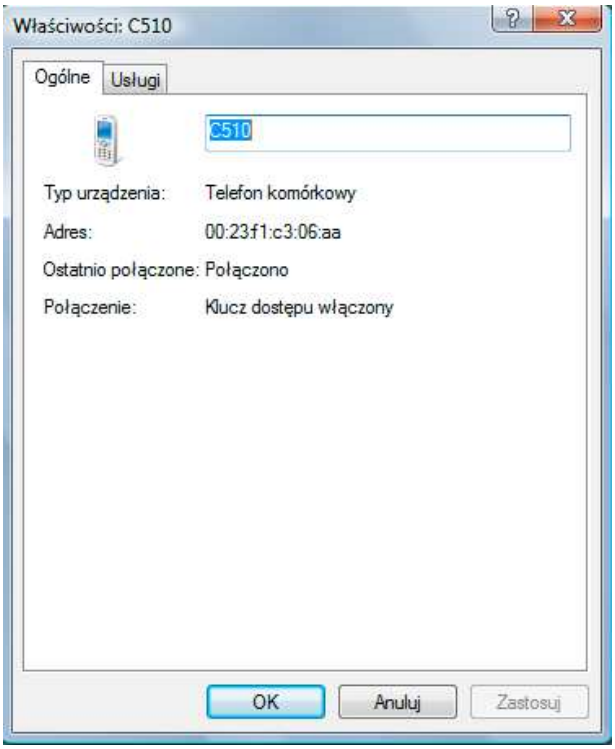
Na listingu 2.2 zilustrowano ogólne zasady wykorzystania w programie wybranych funkcji rodziny `BluetoothXxxDeviceXxx()`, za pomocą których można w wygodny sposób odczytać informacje na temat sparowanego urządzenia oraz udostępnianych przez nie usług. Wynik działania programu pokazano na rysunkach 2.8 oraz 2.9.

Listing. 2.2. Odczyt podstawowych właściwości oraz usług sparowanego z głównym modułem radiowym urządzenia Bluetooth

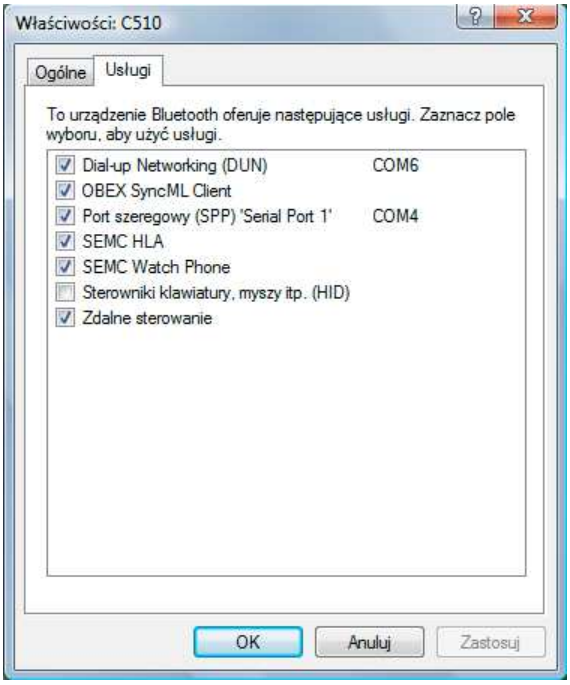
```
#include <iostream>
#pragma option push -a1
  #include <winsock2>
  #include "Ws2bth.h"
  #include "BluetoothAPIs.h"
#pragma option pop

#pragma comment(lib, "Bthprops.lib")

#define MAX_DEVICES 20

using namespace std;

int main()
{
   HANDLE phRadio = NULL;
   BLUETOOTH_FIND_RADIO_PARAMS pbtfrp;
```

```
pbtfrp.dwSize = sizeof(pbtfrp);

HBLUETOOTH_RADIO_FIND bthRadioFind =
            BluetoothFindFirstRadio(&pbtfrp, &phRadio);
if (bthRadioFind != NULL) {
  do {
    BLUETOOTH_RADIO_INFO pRadioInfo;
    pRadioInfo.dwSize = sizeof(pRadioInfo);
    if (ERROR_SUCCESS ==
        BluetoothGetRadioInfo(phRadio, &pRadioInfo)) {
        wprintf(L"Master radio: %s\n",
                pRadioInfo.szName);
      //...
    }
    BLUETOOTH_DEVICE_INFO pbtdi;
    pbtdi.dwSize = sizeof(pbtdi);

    BLUETOOTH_DEVICE_SEARCH_PARAMS pbtsp;
    //memset(&pbtsp, 0, sizeof(pbtsp));
    pbtsp.dwSize = sizeof(pbtsp);
    pbtsp.fReturnAuthenticated = true;
    pbtsp.fReturnRemembered = true;
    pbtsp.fReturnUnknown = true;
    pbtsp.fReturnConnected = true;
    pbtsp.hRadio = phRadio;

    HANDLE hbthDeviceFind =
           BluetoothFindFirstDevice(&pbtsp, &pbtdi);
    if (hbthDeviceFind != NULL)
       do {
           //...
           BluetoothDisplayDeviceProperties(NULL,
           &pbtdi);
       } while(BluetoothFindNextDevice(hbthDeviceFind,
                &pbtdi));
    BluetoothFindDeviceClose(hbthDeviceFind);
    if(ERROR_SUCCESS==BluetoothGetDeviceInfo(phRadio,
                &pbtdi)) {
       wprintf(L"Conected device: %s\n",
                pbtdi.szName);
        //...
    }

    GUID pGuidServices[MAX_DEVICES];
    DWORD pcServices = sizeof(pGuidServices);
```

```
        BluetoothEnumerateInstalledServices(phRadio,
             &pbtdi, &pcServices, pGuidServices);
        wprintf(L"Services: %d\n",pcServices);
        CloseHandle(phRadio);
        BLUETOOTH_SELECT_DEVICE_PARAMS btsdp;
        btsdp.dwSize = sizeof(btsdp);

      } while(BluetoothFindNextRadio(bthRadioFind,
             &phRadio));

      BluetoothFindRadioClose(bthRadioFind);
    }//koniec if
    system("PAUSE");
    return 0;
}
//---------------------------------------------------
```



Rysunek 2.8. Ogólne właściwości urządzenia Bluetooth



Rysunek 2.9. Usługi udostępniane przez sparowane urządzenie

2.4. Funkcje rodziny BluetoothXxxRadioXxx()

Funkcje rodziny `BluetoothXxxRadioXxx()` umożliwiają programom sterującym urządzeniami Bluetooth współpracę z istniejącymi w systemie odbiornikami radiowymi.

2.4.1. Funkcja BluetoothFindFirstRadio()

Funkcja `BluetoothFindFirstRadio()` rozpoczyna proces detekcji dostępnych modułów radiowych Bluetooth.

```
HBLUETOOTH_RADIO_FIND BluetoothFindFirstRadio(
    BLUETOOTH_FIND_RADIO_PARAMS *pbtfrp,
    __out  HANDLE *phRadio
);
```

Wskaźnik `pbtfrp` wskazuje na strukturę `BLUETOOTH_FIND_RADIO_PARAMS`:

```
typedef struct {
  DWORD dwSize;
} BLUETOOTH_FIND_RADIO_PARAMS;
```

Wskaźnik `phRadio` wskazuje na wykryty identyfikator modułu radiowego. Moduł radiowy może być urządzeniem wbudowanym (np. w laptopie) lub

adapterem Bluetooth USB. Po wykryciu ostatniego modułu identyfikator ten powinien zostać zwolniony za pomocą funkcji `CloseHandle()`. Prawidłowo wykonana funkcja lokalizuje identyfikator modułu radiowego. Zwrócona przez funkcję wartość `NULL` oznacza niewłaściwe jej wykonanie. Kod ewentualnego błędu powinien być określony za pomocą funkcji `GetLastError()` i może być jednym z:
`ERROR_INVALID_PARAMETER` – wskaźnik `pbtfrp` wskazuje na wartość `NULL`,
`ERROR_REVISION_MISMATCH` – rozmiar struktury wskazywanej przez `pbtfrp` został niewłaściwie określony,
`ERROR_OUTOFMEMORY` – nie ma wystarczającej ilości pamięci do zlokalizowania zainstalowanych w systemie modułów radiowych Bluetooth.

2.4.2. Funkcja BluetoothFindNextRadio()

Funkcja `BluetoothFindNextRadio()` kontynuuje proces wyszukiwania dostępnych modułów radiowych Bluetooth.

```
BOOL BluetoothFindNextRadio(
  __in   HBLUETOOTH_RADIO_FIND hFind,
  __out  HANDLE *phRadio
);
```

Parametr wejściowy `hFind` zwracany jest przez uprzednio wywołaną funkcję `BluetoothFindFirstRadio()`. Wskaźnik `phRadio` wskazuje na identyfikator kolejnego wykrytego modułu radiowego. W momencie zaprzestania jego wykorzystywania identyfikator ten powinien być zwalniany poprzez wywołanie funkcji `CloseHandle()`. Prawidłowo wykonana funkcja zwraca wartość `TRUE`. Zwrócona przez funkcję wartość `FALSE` oznacza, iż nie znaleziono kolejnych modułów radiowych. Kod ewentualnego błędu w trakcie wykonywania funkcji powinien być określany za pomocą funkcji `GetLastError()` i jest jednym z:
`ERROR_INVALID_HANDLE` – identyfikator odbiornika wskazuje na wartość pustą `NULL`,
`ERROR_NO_MORE_ITEMS` – nie znaleziono więcej modułów radiowych,
`ERROR_OUTOFMEMORY` – zbyt mało pamięci, aby kontynuować proces wyszukiwania.

2.4.3. Funkcja BluetoothFindRadioClose()

Funkcja `BluetoothFindRadioClose()` kończy proces wyszukiwania modułów radiowych i zwalnia ich identyfikatory.

```
BOOL BluetoothFindRadioClose(
    HBLUETOOTH_RADIO_FIND hFind
);
```

Parametr hFind jest identyfikatorem modułu radiowego wykrytego uprzednio dzięki wywołaniu funkcji BluetoothFindFirstRadio(). Prawidłowo wykonana funkcja zwraca wartość TRUE.

2.4.4. Funkcja BluetoothGetRadioInfo()

Funkcja BluetoothGetRadioInfo() wypełnia pola struktury wskazywanej przez pRadioInfo informacjami na temat modułu radiowego Bluetooth.

```
DWORD BluetoothGetRadioInfo(
    HANDLE hRadio,
    PBLUETOOTH_RADIO_INFO pRadioInfo
);
```

Identyfikator hRadio zwracany jest przez funkcję BluetoothFindFirstRadio() lub SetupDiEnumerateDeviceInterfances() [10]. Prawidłowo wykonana funkcja zwraca wartość ERROR_SUCCESS.
Kody ewentualnych błędów to:
ERROR_INVALID_PARAMETER – wskaźniki hRadio lub/i pRadioInfo wskazują wartość NULL,
ERROR_REVISION_MISMATCH – rozmiar struktury BLUETOOTH_RADIO_INFO został niewłaściwie określony i nie został wpisany do pola dwSize.
W polach struktury BLUETOOTH_RADIO_INFO:

```
typedef struct {
  DWORD             dwSize;
  BLUETOOTH_ADDRESS address;
  WCHAR             szName[BLUETOOTH_MAX_NAME_SIZE];
  ULONG             ulClassofDevice;
  USHORT            lmpSubversion;
  USHORT            manufacturer;
} BLUETOOTH_RADIO_INFO;
```

przechowywane są wszystkie istotne informacje na temat modułu radiowego Bluetooth. W tabeli 2.5 podano jej specyfikację.

Tabela 2.5. Specyfikacja struktury BLUETOOTH_RADIO_INFO

Element struktury	Znaczenie
dwSize	Rozmiar struktury w bajtach, który każdorazowo należy prawidłowo określić za pomocą operatora sizeof(). W przeciwnym wypadku funkcja nie będzie działać poprawnie.
address	Adres lokalnego modułu radiowego.

`szName`	Nazwa lokalnego modułu radiowego.
`ulClassofDevice`	Klasa instalacji lokalnego modułu radiowego.
`lmpSubversion`	Dane specyfikujące wersje lokalnego modułu radiowego.
`manufacturer`	Wartość zapisana w jednostkach `BTH_MFG_Xxx` identyfikująca producenta danego modułu radiowego Bluetooth. Szczegółowe informacje na ten temat producentów modułów radiowych Bluetooth dostępne są na stronie *www.bluetooth.com*.

2.4.5. Przykłady

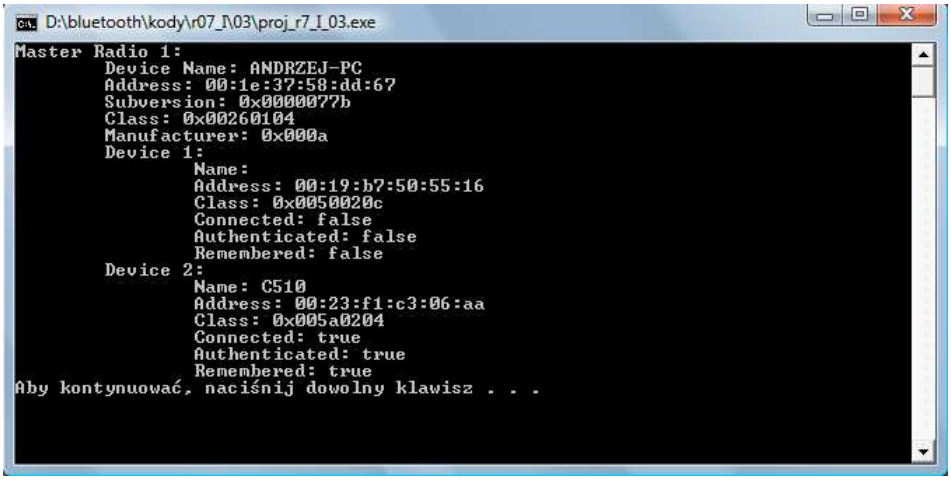
Na listingu 2.3 zaprezentowano szkielet kodu będącego ilustracją ogólnych zasad posługiwania się w programie omówionymi wcześniej funkcjami przeznaczonymi do współpracy z modułami radiowymi oraz będącymi w zasięgu włączonymi urządzeniami Bluetooth. Na rysunku 2.10 pokazano wynik działania programu.

Listing 2.3. Skanowanie istniejących modułów radiowych oraz włączonych, będących w zasięgu zewnętrznych urządzeń Bluetooth

```
#include <iostream>
#pragma option push -a1
  #include <winsock2>
  #include "Ws2bth.h"
  #include "BluetoothAPIs.h"
#pragma option pop

#pragma comment(lib, "Bthprops.lib")

using namespace std;

BLUETOOTH_FIND_RADIO_PARAMS pbtfrp = {
  sizeof(BLUETOOTH_FIND_RADIO_PARAMS)
};

BLUETOOTH_RADIO_INFO pRadioInfo = {
  sizeof(BLUETOOTH_RADIO_INFO),  0,
};

BLUETOOTH_DEVICE_SEARCH_PARAMS pbtsp = {
  sizeof(BLUETOOTH_DEVICE_SEARCH_PARAMS),
  TRUE, TRUE, TRUE, TRUE, TRUE, 10 /*12.28 sek*/, NULL
};
```

```
BLUETOOTH_DEVICE_INFO pbtdi = {
  sizeof(BLUETOOTH_DEVICE_INFO), 0,
};

HANDLE phRadio = NULL;
HBLUETOOTH_RADIO_FIND bthRadioFind = NULL;
HBLUETOOTH_DEVICE_FIND hbthDeviceFind = NULL;

int main() {

    bthRadioFind = BluetoothFindFirstRadio(&pbtfrp,
                   &phRadio);

    int radioNumber = 0;
    do {
      radioNumber++;

      BluetoothGetRadioInfo(phRadio, &pRadioInfo);

      wprintf(L"Master Radio %d:\n", radioNumber);
      wprintf(L"\tDevice Name: %s\n",
              pRadioInfo.szName);
      wprintf(L"\tAddress:
              %02x:%02x:%02x:%02x:%02x:%02x\n",
              pRadioInfo.address.rgBytes[5],
              pRadioInfo.address.rgBytes[4],
              pRadioInfo.address.rgBytes[3],
              pRadioInfo.address.rgBytes[2],
              pRadioInfo.address.rgBytes[1],
              pRadioInfo.address.rgBytes[0]);
      wprintf(L"\tSubversion: 0x%08x\n",
              pRadioInfo.lmpSubversion);
      wprintf(L"\tClass: 0x%08x\n",
              pRadioInfo.ulClassofDevice);
      wprintf(L"\tManufacturer: 0x%04x\n",
              pRadioInfo.manufacturer);

      pbtsp.hRadio = phRadio;

      memset(&pbtdi, 0, sizeof(BLUETOOTH_DEVICE_INFO));
      pbtdi.dwSize = sizeof(BLUETOOTH_DEVICE_INFO);

      hbthDeviceFind = BluetoothFindFirstDevice(&pbtsp,
                       &pbtdi);
```

```
      int deviceNumber = 0;
      do {
        deviceNumber++;

        wprintf(L"\tDevice %d:\n", deviceNumber);
        wprintf(L"\t\tName: %s\n", pbtdi.szName);
        wprintf(L"\t\tAddress:
                %02x:%02x:%02x:%02x:%02x:%02x\n",
                pbtdi.Address.rgBytes[5],
                pbtdi.Address.rgBytes[4],
                pbtdi.Address.rgBytes[3],
                pbtdi.Address.rgBytes[2],
                pbtdi.Address.rgBytes[1],
                pbtdi.Address.rgBytes[0]);
        wprintf(L"\t\tClass: 0x%08x\n",
                pbtdi.ulClassofDevice);
        wprintf(L"\t\tConnected: %s\n",
                pbtdi.fConnected ? L"true" : L"false");
        wprintf(L"\t\tAuthenticated: %s\n",
                pbtdi.fAuthenticated ? L"true" :
                L"false");
        wprintf(L"\t\tRemembered: %s\n",
                pbtdi.fRemembered ? L"true" :
                L"false");

      } while(BluetoothFindNextDevice(hbthDeviceFind,
              &pbtdi));

      BluetoothFindDeviceClose(hbthDeviceFind);

    } while(BluetoothFindNextRadio(&pbtfrp, &phRadio));

    BluetoothFindRadioClose(bthRadioFind);

system("PAUSE");
return 0;
}
//---------------------------------------------------
```



Rysunek 2.10. Program skanujący moduły radiowe oraz będące w zasięgu urządzenia z włączoną funkcją Bluetooth

2.5. Funkcje rodziny BluetoothSdpXxx()

Protokół wyszukiwania usług SDP zapewnia środki niezbędne do określenia, jakie usługi Bluetooth są dostępne w danym urządzeniu. Urządzenie Bluetooth może funkcjonować jako klient SDP pytający o usługi, serwer SDP dostarczający usług lub jako klient i serwer. Jedno urządzenie Bluetooth nie może funkcjonować jako więcej niż jeden serwer SDP, ale może być klientem dla więcej niż jednego serwera. SDP oferuje dostęp jedynie do informacji o usługach, zaś wykorzystanie tych usług musi być zapewnione przez inny protokół. Serwery SDP obsługują rekordy wykorzystywane do katalogowania wszystkich dostępnych usług dostarczanych przez urządzenie. Każda usługa jest reprezentowana przez jeden rekord. Elementy struktury `SDP_ELEMENT_DATA` opisują i definiują obsługiwane rekordy usług, wliczając w to: atrybuty usług, identyfikatory atrybutów, wartości atrybutów, klasy usług, uniwersalne unikatowe identyfikatory (UUID) klas usług. W tabelach 2.6 oraz 2.7 podano odpowiednio wartości identyfikatorów UUID dla wybranych protokołów oraz klas usług i profili Bluetooth [6].

Tabela 2.6. Identyfikatory UUID protokołów

Nazwa protokołu	UUID
`SDP`	0x0001
`UDP`	0x0002
`RFCOMM`	0x0003
`TCP`	0x0004
`TCS-BIN`	0x0005

TCS-ATT	0x0006
ATT	0x0007
OBEX	0x0008
IP	0x0009
FTP	0x000A
HTTP	0x000C
WSP	0x000E
BNEP	0x000F
ESDP	0x0010
HIDP	0x0011
Hardcopy Control Channel	0x0012
Hardcopy Data Channel	0x0014
Hardcopy Notification	0x0016
AVCTP	0x0017
AVDTP	0x0019
CIP	0x001B
MCAP Control Channel	0x001E
MCAP Data Channel	0x001F
L2CAP	0x0100

Tabela 2.7. Identyfikatory UUID wybranych klas usług/profili

Nazwa klasy usług/profilu	UUID
SerialPort	0x1101
LAN Access Using PPP	0x1102
Dialup Networking	0x1103
IrMCSync	0x1104
OBEX Object Push	0x1105
OBEX File Transfer	0x1106
Cordless Telephony	0x1109
AudioSource	0x110A
Fax	0x1111
Headset - Audio Gateway (AG)	0x1112
WAP	0x1113

NAP	0x1116
Reference Printing	0x1119
Basic Printing Profile	0x111A
Human Interface Device Service	0x1224

2.5.1. Funkcja BluetoothSdpGetElementData()

Funkcja `BluetoothSdpGetElementData()` odczytuje i przeprowadza analizę składniową pojedynczego rekordu z protokołu wyszukiwania usług SDP.

```
DWORD BluetoothSdpGetElementData(
  __in   LPBYTE pSdpStream,
  __in   ULONG cbSpdStreamLength,
  __out  PSDP_ELEMENT_DATA pData
);
```

Parametr wejściowy `pSdpStream` wskazuje wykorzystywaną strukturę: `string`, `url`, `sequence` lub `alternative`, której pola opisują atrybuty usługi. Atrybut usługi składa się z dwóch części: identyfikatora oraz wartości. Parametr wejściowy `cbSpdStreamLength` jest rozmiarem wskazywanego rekordu usług. Wskaźnik `pData` wskazuje na strukturę:

```
typedef struct _SPD_ELEMENT_DATA {
  SDP_TYPE         type;
  SDP_SPECIFICTYPE specificType;
  union {
    SDP_LARGE_INTEGER_16  int128;
    LONGLONG              int64;
    LONG                  int32;
    SHORT                 int16;
    CHAR                  int8;
    SDP_ULARGE_INTEGER_16 uint128;
    ULONGLONG             uint64;
    ULONG                 uint32;
    USHORT                uint16;
    UCHAR                 uint8;
    UCHAR                 booleanVal;
    GUID                  uuid128;
    ULONG                 uuid32;
    USHORT                uuid16;
    struct {
      LPBYTE value;
      ULONG  length;
    } string;
```

```
        struct {
          LPBYTE value;
          ULONG  length;
        } url;
        struct {
          LPBYTE value;
          ULONG  length;
        } sequence;
        struct {
          LPBYTE value;
          ULONG  length;
        } alternative;
      } data;
    } SDP_ELEMENT_DATA, *PSDP_ELEMENT_DATA;
```

Prawidłowo wykonana funkcja zwraca wartość ERROR_SUCCESS, w przeciwnym razie - ERROR_INVALID_PARAMETER.

2.5.2. Funkcja BluetoothSdpEnumAttributes()

Funkcja BluetoothSdpEnumAttributes() analizuje strumień rekordów SDP i wywołuje funkcję powrotną callback() dla każdego atrybutu rekordu usług. Każdy atrybut występujący w rekordzie usług opisuje pojedynczą usługę udostępnianą przez urządzenie Bluetooth.

```
BOOL BluetoothSdpEnumAttributes(
    LPBYTE pSDPStream,
    ULONG cbStreamSize,
    PFN_BLUETOOTH_ENUM_ATTRIBUTES_CALLBACK pfnCallback,
    LPVOID pvParam
);
```

Wskaźnik pSDPStream wskazuje na pojedynczy rekord SDP. Parametr cbStreamSize jest rozmiarem strumienia rekordów. Wskaźnik pfnCallback wskazuje na funkcję callback(), której przykładowa konstrukcja została zamieszczona poniżej:

```
BOOL __stdcall callback(ULONG uAttribId, LPBYTE
                        pValueStream,
                        ULONG cbStreamSize,
                        LPVOID pvParam)
{
   SDP_ELEMENT_DATA sdpElementData;
   wprintf(L"callback() uAttribId: %ul\n", uAttribId);
   wprintf(L"callback() pValueStream: %d\n ",
```

```
            pValueStream);
    wprintf(L"callback() cbStreamSize: %ul\n ",
            cbStreamSize);
    if(BluetoothSdpGetElementData(pValueStream,
                                  cbStreamSize,
                                  &sdpElementData)
       != ERROR_SUCCESS) {
         //...
         return FALSE;
    }
    else {
         //...
         return TRUE;
    }
}
//-------------------------------------------------
```

`pvParam` jest parametrem opcjonalnym. Prawidłowo wykonana funkcja zwraca wartość `TRUE`. Zwrócona przez funkcję wartość `FALSE` oznacza, iż nie można zanalizować strumienia rekordów SDP. Kod ewentualnego błędu w trakcie wykonywania funkcji powinien być określony za pomocą `GetLastError()` i może być jednym z:
`ERROR_INVALID_PARAMETER` – wskaźniki `pSDPStream` lub/i `pfnCallback` wskazują na wartość pustą `NULL`,
`ERROR_INVALID_DATA` – zawartość strumienia rekordów SDP jest niewłaściwa.

2.5.3. Funkcja BluetoothSdpGetAttributeValue()

Funkcja `BluetoothSdpGetAttributeValue()` pobiera wartość atrybutu właściwą jego identyfikatorowi. Identyfikator atrybutu jest 16-bitową daną typu `unsigned int` (`UINT16`).

```
DWORD BluetoothSdpGetAttributeValue(
  __in   LPBYTE pRecordStream,
  __in   ULONG cbRecordLength,
  __in   USHORT usAttributeId,
  __out  PSDP_ELEMENT_DATA pAttributeData
);
```

Wskaźnik `pRecordStream` wskazuje na rekord usług SDP. Parametr `cbRecordLength` określa jego rozmiar w bajtach. Parametr `usAttributeId` jest identyfikatorem atrybutu w rekordzie usług. Wartości wszystkich identyfikatorów atrybutów występujących w rekordach usług zapisane są w formacie `SDP_ATTRIB_Xxx` w module *bthdef.h*. Wskaźnik `pAttributeData` wskazuje

na strukturę SDP_ELEMENT_DATA. Prawidłowo wykonana funkcja zwraca wartość ERROR_SUCCESS. Wartości ewentualnych błędów mogą być następujące:
ERROR_INVALID_PARAMETER – jeden ze wskaźników wskazuje na wartość pustą NULL,
ERROR_FILE_NOT_FOUND – w rekordzie usług nie występuje żądany identyfikator usAttributeId.

2.5.4. Funkcja BluetoothSdpGetString()

Funkcja BluetoothSdpGetString() konwertuje łańcuch znaków opisujących usługę z rekordu usług na łańcuch typu Unicode.

```
DWORD BluetoothSdpGetString(
  __in     LPBYTE pRecordStream,
  __in     ULONG cbRecordLength,
  __in     PSDP_STRING_DATA_TYPE pStringData,
  __in     USHORT usStringOffset,
  __out    PWCHAR pszString,
  __inout  PULONG pcchStringLength
);
```

Wskaźnik pRecordStream wskazuje na rekord usług. cbRecordLength jest rozmiarem wskazywanego rekordu. Parametr pStringData wskazuje na strukturę:

```
typedef struct _SDP_STRING_TYPE_DATA {
  USHORT encoding;
  USHORT mibeNum;
  USHORT attributeID;
} SDP_STRING_TYPE_DATA, *PSDP_STRING_TYPE_DATA;
```

Parametr usStringOffset jest offsetem dodawanym do identyfikatora usługi i może być jednym z: STRING_NAME_OFFSET, STRING_DESCRIPTION_OFFSET, oraz STRING_PROVIDER_NAME_OFFSET. Wskaźnik pszString != NULL wskazuje na przekonwertowany łańcuch znaków. Parametr pcchStringLength określa długość łańcucha znaków wskazywanego przez pszString. Prawidłowo wykonana funkcja zwraca wartość ERROR_SUCCESS.

2.6. Funkcje rodziny BluetoothXxxAuthenticationXxx()

Funkcje rodziny BluetoothXxxAuthenticationXxx() służą do programowej kontroli procesu rejestrowania i uwierzytelnia urządzeń Bluetooth.

2.6.1. Funkcja BluetoothRegisterForAuthentication()

Funkcja `BluetoothRegisterForAuthentication()` rejestruje w systemie uwierzytelnione urządzenie z włączoną obsługą Bluetooth. Funkcja weryfikuje zgodność wprowadzonych kodów przez oba dobierane w pary urządzenia.

```
DWORD BluetoothRegisterForAuthentication(
    BLUETOOTH_DEVICE_INFO *pbtdi,
    HBLUETOOTH_AUTHENTICATION_REGISTRATION
    *phRegHandle,
    PFN_AUTHENTICATION_CALLBACK pfnCallback,
    PVOID pvParam
);
```

Wskaźnik `pbtdi` wskazuje na strukturę `BLUETOOTH_DEVICE_INFO`. Wskaźnik `phRegHandle` wskazuje na identyfikator `HBLUETOOTH_AUTHENTICATION_REGISTRATION` określający rezultat przeprowadzanej operacji parowania urządzeń przez aktualnie używany program (aplikację). Parametr `pfnCallback` wskazuje na funkcję przechwytującą odpowiedź zidentyfikowanego zewnętrznego urządzenia Bluetooth pod kątem posługiwania się przez nie poprawnym kodem zestawiania w pary. Jej prototyp charakteryzuje się następującą konstrukcją:

```
BOOL PFN_AUTHENTICATION_CALLBACK(
    LPVOID pvParam,
    PBLUETOOTH_DEVICE_INFO pDevice
);
```

Z reguły na rzecz funkcji wskazywanej przez `pfnCallback` wywoływana jest funkcja `BluetoothSendAuthenticationResponse()`. Opcjonalny parametr `pvParam` może być jednym z argumentów funkcji wskazywanej przez `pfnCallback`. Prawidłowo wykonana funkcja zwraca wartość `ERROR_SUCCESS`. W przypadku niepowodzenia zwracana jest wartość `ERROR_OUTOFMEMORY` – niewystarczająca ilość pamięci do zarejestrowania operacji dobierania urządzeń w pary.

2.6.2. Funkcja BluetoothRegisterForAuthenticationEx()

Funkcja `BluetoothRegisterForAuthenticationEx()` rozszerza funkcjonalność poprzedniej i jest rekomendowana do wykorzystania w programach działających w systemach Windows Vista z SP2,3 oraz Windows 7.

```
HRESULT WINAPI
        BluetoothRegisterForAuthenticationEx(
        const BLUETOOTH_DEVICE_INFO *pbtdiln,
```

```
        __out     HBLUETOOTH_AUTHENTICATION_REGISTRATION
                  *phRegHandleOut,
        __in_opt  PFN_AUTHENTICATION_CALLBACK_EX
                  pfnCallbackIn,
        __in_opt  PVOID pvParam
);
```

Znaczenie parametrów oraz wartości zwracanych przez funkcję jest analogiczne jak w przypadku funkcji `BluetoothRegisterForAuthentication()`. Wyjątkiem jest wskaźnik `pfnCallbackIn`, który wskazuje na funkcję powrotną o prototypie:

```
BOOL CALLBACK PFN_AUTHENTICATION_CALLBACK_EX(
  __in_opt  LPVOID pvParam,
  __in      PBLUETOOTH_AUTHENTICATION_CALLBACK_PARAMS
            pAuthCallbackParams
);
```

gdzie wskaźnik `pAuthCallbackParams` wskazuje na strukturę:

```
typedef struct
_BLUETOOTH_AUTHENTICATION_CALLBACK_PARAMS {
  BLUETOOTH_DEVICE_INFO
  deviceInfo;
  BLUETOOTH_AUTHENTICATION_METHOD
  authenticationMethod;
  BLUETOOTH_IO_CAPABILITY
  ioCapability;
  BLUETOOTH_AUTHENTICATION_REQUIREMENTS
  authenticationRequirements;
  union {
    ULONG Numeric_Value;
    ULONG Passkey;
  } ;
} BLUETOOTH_AUTHENTICATION_CALLBACK_PARAMS,
*PBLUETOOTH_AUTHENTICATION_CALLBACK_PARAMS;
```

przechowującą informacje na temat rejestrowanego w systemie urządzenia Bluetooth. W tabeli 2.6 zaprezentowano jej zasoby.

Tabela 2.6. Specyfikacja struktury BLUETOOTH__AUTHENTICATION_CALLBACK_PARAMS

Element struktury	Znaczenie
`deviceInfo`	Wskaźnik do struktury `BLUETOOTH_DEVICE_INFO`.

`authentication Method`	Elementy typu wyliczeniowego
	`typedef enum {`
	`BLUETOOTH_AUTHENTICATION_METHOD_LEGACY = 0x1,`
	`BLUETOOTH_AUTHENTICATION_METHOD_OOB = ,`
	`BLUETOOTH_AUTHENTICATION_METHOD_NUMERIC_COMPARISON=,`
	`BLUETOOTH_AUTHENTICATION_METHOD_PASSKEY_NOTIFICATION=,`
	`BLUETOOTH_AUTHENTICATION_METHOD_PASSKEY =`
	`} BLUETOOTH_AUTHENTICATION_METHOD;`
	definiującego metody używane w trakcie uwierzytelniania urządzeń:
	`BLUETOOTH_AUTHENTICATION_METHOD_LEGACY` – urządzenie Bluetooth jest uwierzytelnianie (potwierdza swoją autentyczność) za pomocą kodu PIN.
	`BLUETOOTH_AUTHENTICATION_METHOD_OOB` – urządzenie Bluetooth jest uwierzytelnianie za pomocą danych OOB.
	`BLUETOOTH_AUTHENTICATION_METHOD_NUMERIC_COMPARISON` – urządzenie Bluetooth jest uwierzytelnianie za pomocą kodu numerycznego.
	`BLUETOOTH_AUTHENTICATION_METHOD_PASSKEY_NOTIFICATION` – urządzenie potwierdza swoją autentyczność za pomocą powiadomienia o konieczności użycia i potwierdzenia unikalnego klucza.
	`BLUETOOTH_AUTHENTICATION_METHOD_PASSKEY` – urządzenie jest uwierzytelniane za pomocą unikalnego klucza bez konieczności jego potwierdzania.

`ioCapability`	Elementy typu wyliczeniowego określające sposób przeprowadzania uwierzytelniania urządzeń: `BLUETOOTH_IO_CAPABILITY_DISPLAYONLY` – urządzenie jest tylko wyświetlane w systemie. `BLUETOOTH_IO_CAPABILITY_DISPLAYYESNO` – urządzenie reprezentowane jest w formie okna dialogowego, w którym przewidziano opcje potwierdzenia/anulowania wykonywanych operacji. `BLUETOOTH_IO_CAPABILITY_KEYBOARDONLY` - urządzenie rejestruje dane wprowadzane jedynie z klawiatury. `BLUETOOTH_IO_CAPABILITY_NOINPUTNOOUTPUT` – urządzenie nie jest interaktywne. `BLUETOOTH_IO_CAPABILITY_UNDEFINED` – nie zdefiniowano.
`authentication Requirements`	Elementy typu wyliczeniowego `BLUETOOTH_AUTHENTICATION_REQUIREMENTS` (patrz tabela 2.2).
`Numeric`	Wartość numeryczna (przynajmniej sześciocyfrowa liczba) używana w trakcie dobierania urządzeń w pary.

_Value	
Passkey	Klucz identyfikujący urządzenie.

2.6.3. Funkcja BluetoothSendAuthenticationResponse()

Funkcja BluetoothSendAuthenticationResponse() wykorzystywana jest, gdy program żąda potwierdzenia prawidłowości wprowadzonego kodu zestawiania w pary.

```
DWORD BluetoothSendAuthenticationResponse(
    HANDLE hRadio,
    BLUETOOTH_DEVICE_INFO *pbtdi,
    LPWSTR pszPasskey
);
```

Identyfikator hRadio identyfikuje lokalny moduł radiowy Bluetooth. Wskaźnik pbtdi wskazuje na strukturę BLUETOOTH_DEVICE_INFO, której pola przechowują informacje na temat dobieranego w parę zewnętrznego urządzenia Bluetooth. Wskaźnik pszPasskey wskazuje na zakończony zerowym ogranicznikiem łańcuch znaków UNICODE reprezentujący kod zestawiania w pary. Długość klucza (kodu) nie może być większa niż BLUETOOTH_MAX_PASSKEY_SIZE.

Prawidłowo wykonana funkcja zwraca wartość ERROR_SUCCESS. W przypadku niepowodzenia wartościami zwracanymi mogą być:

ERROR_CANCELLED – kod dobierania w pary nie został wprowadzony w urządzeniu Bluetooth przed upływem czasu przeterminowania,

E_FAIL – został wprowadzony nieodpowiedni kod i urządzenia nie mogą być prawidłowo zestawione.

2.6.4. Funkcja BluetoothSendAuthenticationResponseEx()

Funkcja BluetoothSendAuthenticationResponseEx() rozszerza funkcjonalność poprzedniej i jest rekomendowana do wykorzystania w programach działających w systemach Windows Vista z SP2 oraz Windows 7.

```
HRESULT WINAPI BluetoothSendAuthenticationResponseEx(
  __in_opt  HANDLE hRadioIn,
  __in      PBLUETOOTH_AUTHENTICATE_RESPONSE
            pauthResponse
);
```

Identyfikator hRadioIn wskazuje na lokalny moduł radiowy Bluetooth. Wskaźnik pauthResponse wskazuje na strukturę BLUETOOTH_AUTHENTICATE_RESPONSE, której specyfikacja została przedstawiona w tabeli 2.7.

```
typedef struct _BLUETOOTH_AUTHENTICATE_RESPONSE {
  BLUETOOTH_ADDRESS bthAddressRemote;
  BLUETOOTH_AUTHENTICATION_METHOD authMethod;
  union {
    BLUETOOTH_PIN_INFO                pinInfo;
    BLUETOOTH_OOB_DATA_INFO           oobInfo;
    BLUETOOTH_NUMERIC_COMPARISON_INFO numericCompInfo;
    BLUETOOTH_PASSKEY_INFO            passkeyInfo;
  } ;
  UCHAR negativeResponse;
} BLUETOOTH_AUTHENTICATE_RESPONSE,
*PBLUETOOTH_AUTHENTICATE_RESPONSE;
```

Tabela 2.7. Specyfikacja struktury BLUETOOTH__AUTHENTICATE_RESPONSE

Element struktury	Znaczenie

`bthAddressRemote`	Wskaźnik do struktury
	`typedef struct _BLUETOOTH_ADDRESS {`
	`  union {`
	`  BTH_ADDR ullLong;`
	`  BYTE rgBytes[6];`
	`  } ;`
	`} BLUETOOTH_ADDRESS;`
	Pola struktury przechowują adres urządzenia Bluetooth.

`authMethod`	Element typu wyliczeniowego `BLUETOOTH_AUTHENTICATION_METHOD` (patrz tabela 2.6).

`pinInfo`	Wskaźnik do struktury
	`Typedef struct _BLUETOOTH_PIN_INFO {`
	`  UCHAR pin[BTH_MAX_PIN_SIZE];`
	`  UCHAR pinLength;`
	`} BLUETOOTH_PIN_INFO,`
	`*PBLUETOOTH_PIN_INFO;`
	Pola struktury przechowują wartość oraz długość kodu PIN.

`oobInfo`	Wskaźnik do struktury
	`typedef struct`
	`_BLUETOOTH_OOB_DATA_INFO {`
	`  UCHAR C[16];`
	`  UCHAR R[16];`

	} BLUETOOTH_OOB_DATA_INFO, *PBLUETOOTH_OOB_DATA_INFO; Pola struktury przechowują odpowiednio: 128-bitowy klucz kryptograficzny wykorzystywany w trakcie dwukierunkowego zestawiania urządzeń (C[16]), oraz liczbę pseudolosową wykorzystywaną w trakcie jednokierunkowego dobierania w pary urządzeń (R[16]).
numericCompInfo	Wskaźnik do struktury typedef struct _BLUETOOTH_NUMERIC_COMPARISON_INFO { ULONG NumericValue; } BLUETOOTH_NUMERIC_COMPARISON_INFO, *PBLUETOOTH_NUMERIC_COMPARISON_INFO; Pole struktury przechowuje wartość liczbową wykorzystywaną w trakcie zestawiania urządzeń w pary.
passkeyInfo	Wskaźnik do struktury typedef struct _BLUETOOTH_PASSKEY_INFO { ULONG passkey; } BLUETOOTH_PASSKEY_INFO, *PBLUETOOTH_PASSKEY_INFO; Pole struktury przechowuje klucz wykorzystywany w trakcie dobierania w pary urządzeń.
negativeResponse	Wartość TRUE, jeżeli uwierzytelnianie urządzeń nie powiodło się (np., ze względu na niezgodność wprowadzonych kodów zestawiania w pary).

Wartości zwracane przez funkcję BluetoothSendAuthentication-ResponseEx() są analogiczne jak w przypadku funkcji BluetoothSend-AuthenticationResponse().

2.6.5. Funkcja BluetoothUnregisterAuthentication()

Funkcja BluetoothUnregisterAuthentication() zwalnia identyfikator hRegHandle przydzielony uprzednio za pomocą BluetoothRegisterFor-Authentication().

```
BOOL BluetoothUnregisterAuthentication()
    HBLUETOOTH_AUTHENTICATION_REGISTRATION hRegHandle
);
```

Prawidłowo wykonana funkcja zwraca wartość TRUE.

2.6.6. Przykłady

W trakcie procesu dobierania urządzeń w pary użytkownik może zostać poproszony o wprowadzenie lub/i potwierdzenie odpowiedniego kodu. Kod ten bywa często wyświetlany na urządzeniu lub/i na komputerze, w zależności od typu urządzenia. Stanowi on gwarancję, że są zestawiane w pary odpowiednie urządzenia Bluetooth. Na listingu 2.4 zaprezentowano szkielet kodu obrazującego praktyczne aspekty wykorzystywania w aplikacji zarządzającej urządzeniami Bluetooth niektórych funkcji rodziny `BluetoothXxx-AutenticationXxx()` umożliwiających zestawianie w pary urządzeń na podstawie wspólnego kodu. Na rysunku 2.11 zaprezentowano wynik działania programu.

Listing 2.4. Dobieranie w pary urządzeń Bluetooth z wykorzystaniem wspólnego kodu

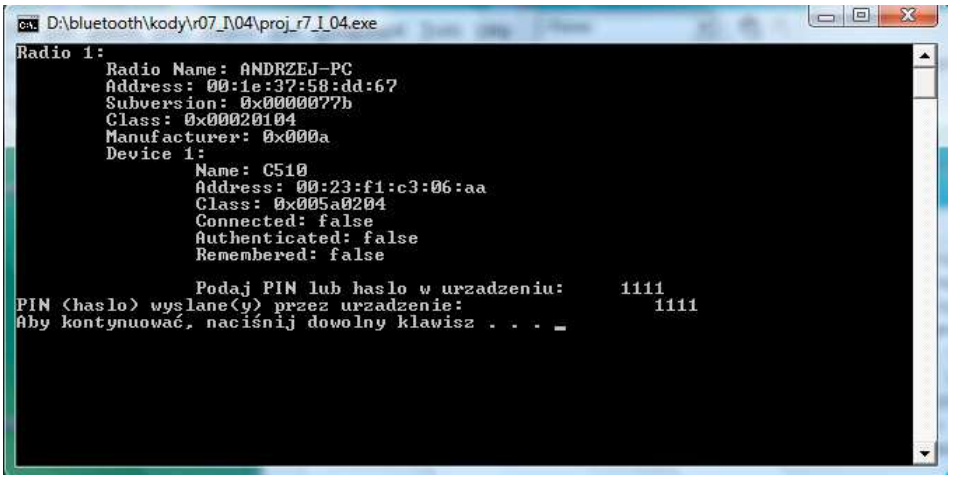
```
#include <iostream>
#pragma option push -a1
  #include <winsock2>
  #include "Ws2bth.h"
  #include "BluetoothAPIs.h"
#pragma option pop

#pragma comment(lib, "Bthprops.lib")

using namespace std;

BLUETOOTH_FIND_RADIO_PARAMS pbtfrp = {
  sizeof(BLUETOOTH_FIND_RADIO_PARAMS)
};
BLUETOOTH_RADIO_INFO pRadioInfo = {
  sizeof(BLUETOOTH_RADIO_INFO),  0,
};
BLUETOOTH_DEVICE_SEARCH_PARAMS pbtsp = {
  sizeof(BLUETOOTH_DEVICE_SEARCH_PARAMS),
  TRUE, TRUE, TRUE, TRUE, TRUE, 5 /*6.14 sek*/, NULL
};
BLUETOOTH_DEVICE_INFO pbtdi = {
  sizeof(BLUETOOTH_DEVICE_INFO), 0,
};

HANDLE phRadio = NULL;
HBLUETOOTH_RADIO_FIND bthRadioFind = NULL;
HBLUETOOTH_DEVICE_FIND hbthDeviceFind = NULL;
//---------------------------------------------------
void showError()
```

```
{
  LPVOID lpMsgBuf;
  FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
                 FORMAT_MESSAGE_FROM_SYSTEM |
                 FORMAT_MESSAGE_IGNORE_INSERTS,
                 NULL, GetLastError(), 0, (LPTSTR)
                 &lpMsgBuf, 0, NULL );
                 fprintf(stderr, "\n%s\n", lpMsgBuf);
  free(lpMsgBuf);
}
//--------------------------------------------------
BOOL __cdecl pfnCallback(LPVOID pvParam,
                         PBLUETOOTH_DEVICE_INFO pDevice)
{
  HANDLE hRadio = (HANDLE)pvParam;
  WCHAR temp[15] = {0};
  LPWSTR pszPasskey = temp;

  for (int j = 0; j < 8; j+=2) {
    ((PUCHAR)pszPasskey)[j] = '1';
  }

  if (ERROR_SUCCESS ==
      BluetoothSendAuthenticationResponse(hRadio,
      pDevice, pszPasskey)){
      wprintf(L"\nPIN (hasło) wysłane(y) przez "\
              " urządzenie: %20s\n", pszPasskey);
     system("PAUSE");
     return TRUE;
  }
   else {
       showError();
       return FALSE;
   }
}
//--------------------------------------------------
int main() {

    bthRadioFind = BluetoothFindFirstRadio(&pbtfrp,
                   &phRadio);

    int radioNumber = 0;
    do {
      radioNumber++;
```

```
BluetoothGetRadioInfo(phRadio, &pRadioInfo);

wprintf(L"Radio %d:\n", radioNumber);
wprintf(L"\tRadio Name: %s\n",
        pRadioInfo.szName);
        wprintf(L"\tAddress:
        %02x:%02x:%02x:%02x:%02x:%02x\n",
        pRadioInfo.address.rgBytes[5],
        pRadioInfo.address.rgBytes[4],
        pRadioInfo.address.rgBytes[3],
        pRadioInfo.address.rgBytes[2],
        pRadioInfo.address.rgBytes[1],
        pRadioInfo.address.rgBytes[0]);
wprintf(L"\tSubversion: 0x%08x\n",
        pRadioInfo.lmpSubversion);
wprintf(L"\tClass: 0x%08x\n",
        pRadioInfo.ulClassofDevice);
wprintf(L"\tManufacturer: 0x%04x\n",
        pRadioInfo.manufacturer);

pbtsp.hRadio = phRadio;

memset(&pbtdi, 0, sizeof(BLUETOOTH_DEVICE_INFO));
pbtdi.dwSize = sizeof(BLUETOOTH_DEVICE_INFO);

hbthDeviceFind = BluetoothFindFirstDevice(&pbtsp,
                 &pbtdi);

int deviceNumber = 0;
do {
  deviceNumber++;

  wprintf(L"\t\Device %d:\n", deviceNumber);
  wprintf(L"\t\tName: %s\n", pbtdi.szName);
  wprintf(L"\t\tAddress:
          %02x:%02x:%02x:%02x:%02x:%02x\n",
          pbtdi.Address.rgBytes[5],
          pbtdi.Address.rgBytes[4],
          pbtdi.Address.rgBytes[3],
          pbtdi.Address.rgBytes[2],
          pbtdi.Address.rgBytes[1],
          pbtdi.Address.rgBytes[0]);
  wprintf(L"\t\tClass: 0x%08x\n",
          pbtdi.ulClassofDevice);
  wprintf(L"\t\tConnected: %s\n",
```

```
            pbtdi.fConnected ? L"true" \
                 : L"false");
      wprintf(L"\t\tAuthenticated: %s\n",
            pbtdi.fAuthenticated ? \
            L"true" : L"false");
      wprintf(L"\t\tRemembered: %s\n",
            pbtdi.fRemembered ? L"true"\
            : L"false");

      HBLUETOOTH_AUTHENTICATION_REGISTRATION
      phRegHandle;
      if (ERROR_SUCCESS !=
          BluetoothRegisterForAuthentication(&pbtdi,
          &phRegHandle, pfnCallback, phRadio)) {
         //showError();
      }

      LPWSTR pszPasskey;

      for (int j = 0; j < 8; j+=2) {
        ((PUCHAR)pszPasskey)[j] = '1';
      }

      ULONG ulPasskeyLength = 4;
      wprintf(L"\n\t\t\Podaj PIN lub hasło w
            urządzeniu: %10s\n", pszPasskey);

      DWORD result =
            BluetoothAuthenticateDevice(NULL,
            phRadio, &pbtdi, pszPasskey,
            ulPasskeyLength);
      if (result == ERROR_NO_MORE_ITEMS) {
        printf("\n\t\t%s\n", "Urządzenie zostało
              wcześniej zidentyfikowane\n");
      }
      else if(result != ERROR_SUCCESS) {
        showError();
      }

    } while(BluetoothFindNextDevice(hbthDeviceFind,
            &pbtdi));

    BluetoothFindDeviceClose(hbthDeviceFind);

  } while(BluetoothFindNextRadio(&pbtfrp,
```

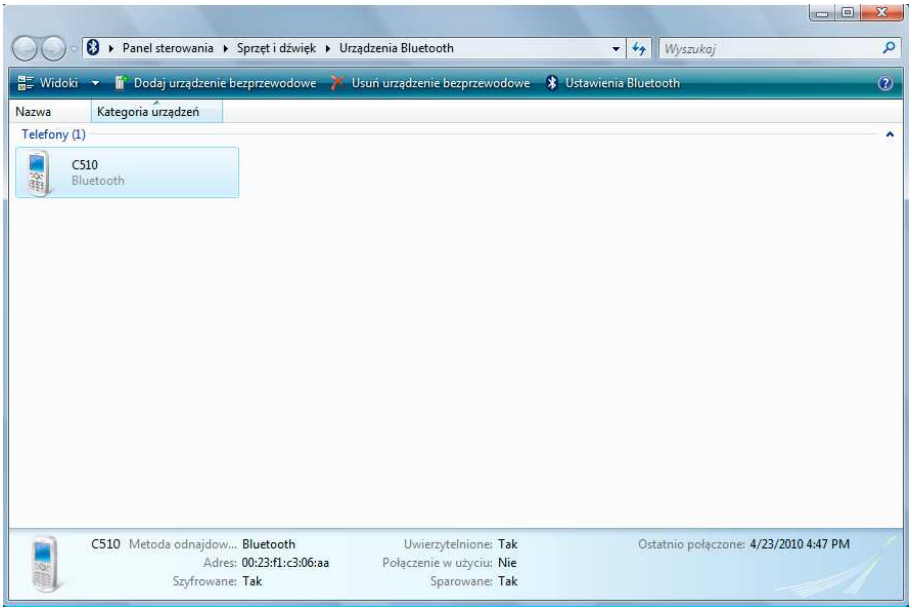
```
            &phRadio));

    BluetoothFindRadioClose(bthRadioFind);

    cin.get();
    return 0;
}
//---------------------------------------------------
```



Rysunek 2.11. Wynik działania programu dobierającego w pary urządzenia na podstawie wspólnego kodu

Testując powyższy program łatwo możemy zauważyć, iż kod zestawiania w pary urządzeń został zaprogramowany w postaci czterech jednakowych cyfr: `1111`. Po uruchomieniu na komputerze, program skanuje dostępne w systemie moduły radiowe oraz urządzenia z włączoną funkcją Bluetooth. W następnej kolejności prosi pierwsze wykryte urządzenie o wprowadzenie kodu `1111`. Jeżeli kod zostanie prawidłowo wprowadzony – urządzenie zostaje zarejestrowane w systemie otrzymując status sprzętu uwierzytelnionego i sparowanego. Panel sterowania (rys. 2.12) dostarcza podstawowych informacji na temat tak dobranego urządzenia.

Warto pamiętać, iż nie tylko komputer może aranżować operację zestawiania w pary. Ignorując monit programu o wprowadzenie odpowiedniego kodu w urządzeniu, można przejść do trybu kiedy to samo urządzenie zaaranżuje tryb dobierania w pary. W takiej sytuacji należy w urządzeniu wybrać opcję „*Nowe urządzenia*”, następnie określić nazwę żądanego modułu (odbiornika) radiowego i wprowadzić odpowiedni kod.



Rysunek 2.12. Panel sterowania z dostępnym urządzeniem Bluetooth

2.7. Funkcje rodziny BluetoothXxxServiceXxx()

Funkcje rodziny `BluetoothXxxServiceXxx()` udostępniają programom informacje na temat usług oferowanych przez urządzenia Bluetooth będące aktualnie w zasięgu lokalnego odbiornika radiowego.

2.7.1. Funkcja BluetoothSetLocalServiceInfo()

Funkcja `BluetoothSetLocalServiceInfo()` ustala informacje na temat lokalnej usługi udostępnianej przez urządzenie Bluetooth.

```
DWORD WINAPI BluetoothSetLocalServiceInfo(
  __in_opt  HANDLE hRadioIn,
  __in      const GUID *pClassGuid,
            ULONG ulInstance,
            const BLUETOOTH_LOCAL_SERVICE_INFO
            *pServiceInfoIn
);
```

Wskaźnik `hRadioIn` wskazuje na identyfikator lokalnego modułu radiowego. Wskaźnik `pClassGuid` wskazuje na identyfikator `GUID` usługi. Parametr `ulInstance` jest identyfikatorem ID urządzenia używanego przez menedżera PnP. Wskaźnik `pServiceInfoIn` wskazuje na strukturę `BLUETOOTH_LOCAL_SERVICE_INFO`. Specyfikacja tej struktury została zamieszczona w tabeli 2.8.

```
typedef struct _BLUETOOTH_LOCAL_SERVICE_INFO {
  BOOL Enabled;
  BLUETOOTH_ADDRESS btAddr;
  WCHAR szName[BLUETOOTH_MAX_SERVICE_NAME_SIZE];
  WCHAR szDeviceString[BLUETOOTH_DEVICE_NAME_SIZE];
} BLUETOOTH_LOCAL_SERVICE_INFO;
```

Tabela 2.8. Specyfikacja struktury BLUETOOTH__LOCAL_SERVICE_INFO

Element struktury	Znaczenie
`Enabled`	Znacznik określający dostępność usługi (`TRUE` lub `FALSE`).
`btAddr`	Wskaźnik do struktury `BLUETOOTH_ADDRESS` przechowującej adres urządzenia Bluetooth.
`szName`	Wskaźnik do łańcucha znaków (zakończonego zerowym ogranicznikiem) opisującego nazwę usługi. Całkowita długość łańcucha nie może przekraczać 256 znaków.
`szDeviceString`	Wskaźnik do łańcucha znaków (zakończonego zerowym ogranicznikiem) opisującego nazwę lokalnego urządzenia kojarzonego w daną usługą. Lokalnym urządzeniem może być np. wirtualny port szeregowy. Całkowita długość łańcucha nie może przekraczać 256 znaków.

Prawidłowo wykonana funkcja `BluetoothSetLocalServiceInfo()` zwraca wartość `ERROR_SUCCESS`. Kody ewentualnych błędów to:
`ERROR_NOT_FOUND` – nie znaleziono wskazywanego modułu radiowego,
`ERROR_BAD_UNIT` – w systemie nie wykryto żadnego modułu radiowego,
`STATUS_INSUFFICIENT_RESOURCES` – zbyt mało pamięci aby wykonać operację,
`STATUS_PRIVILEGE_NOT_HELD` – użytkownik nie posiada uprawnień aby uzyskać dostęp do określonych zasobów lub urządzeń.

2.7.2. Funkcja BluetoothSetServiceState()

Funkcja `BluetoothSetServiceState()` umożliwia określenie usługi udostępnianej przez urządzenie Bluetooth.

```
DWORD BluetoothSetServiceState(
    HANDLE hRadio,
    BLUETOOTH_DEVICE_INFO *pbtdi,
    GUID *pGuidService,
    DWORD dwServiceFlags
);
```

Wskaźnik hRadio wskazuje na moduł radiowy Bluetooth. Wskaźnik pbtdi wskazuje na strukturę BLUETOOTH_DEVICE_INFO przechowującej informacje na temat urządzenia zewnętrznego. Wskaźnik pGuidService wskazuje na identyfikator GUID usługi. Parametr dwServiceFlags określa czy wskazywana usługa ma być dostępna – BLUETOOTH_SERVICE_ENABLE, czy też nie – BLUETOOTH_SERVICE_DISABLE.
Prawidłowo wykonana funkcja zwraca wartość ERROR_SUCCESS. Kody ewentualnych błędów to:
ERROR_INVALID_PARAMETER – błędnie określono parametr dwServiceFlags,
ERROR_SERVICE_DOES_NOT_EXIST – usługa określona identyfikatorem GUID nie jest dostępna dla urządzenia,
E_INVALIDARG – wyspecyfikowana za pomocą dwServiceFlags usługa jest już dostępna.

2.8. Podsumowanie

W niniejszym rozdziale zostały omówione zasoby API Windows, za pomocą których programiści uzyskują bezpośredni dostęp do urządzeń z funkcją Bluetooth. Zaprezentowane zostały przykłady praktycznego wykorzystania omawianych funkcji i struktur. Konstrukcje przykładowych programów starano się przedstawić w sposób na tyle przejrzysty, by Czytelnik nie miał żadnych problemów z samodzielną ich modyfikacją i dostosowaniem do swoich wymagań sprzętowych i programowych. Starano się również zadbać o czytelność kodu, stosując oryginalne nazewnictwo dla zmiennych i funkcji wiernie odzwierciedlające ich rolę w programie. Więcej przykładów praktycznego wykorzystania obecnie omawianych zasobów systemowych zostanie zaprezentowanych w następnym rozdziale opisującym zagadnienia związane z uzyskiwaniem dostępu do urządzeń Bluetooth za pośrednictwem interfejsu programisty biblioteki gniazd WinSock.

ROZDZIAŁ 3

DETEKCJA I IDENTYFIKACJA URZĄDZEŃ BLUETOOTH. CZĘŚĆ II

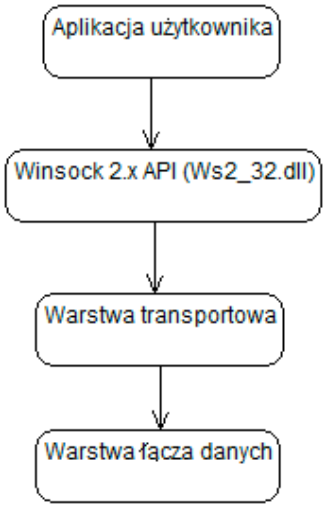
3.1. WinSock API

Systemy operacyjne Windows podtrzymują obsługę wielu protokołów komunikacyjnych dostarczanych w formie usług udostępniających ujednolicony interfejs programistyczny o nazwie Windows Sockets API (WinSock API) eksportowany z biblioteki DLL, tak jak schematycznie pokazano to na rysunku 3.1. Zadaniem interfejsu jest pośredniczenie pomiędzy programami użytkownika a wybranymi protokołami komunikacyjnymi [17]. Oznacza to, iż funkcje interfejsu maskują przed programistą warstwę transportową wybranego protokołu. Dzięki temu, z perspektywy programisty korzystanie z różnych protokołów transmisji danych wygląda niemal identycznie. Biblioteka WinSock 2.x pozwala tworzyć dwa typy usług: transportowe (ang. *transport service providers*) implementujące protokoły transportowe oraz usługi przestrzeni nazw NS_XXX (ang. *namespace resolution service providers*) związane z domenami komunikacyjnymi. W obrębie usług transportowych wyróżnione są dwie grupy dostawców: zaliczane do warstwy transportowej i sieciowej usługi podstawowe BSP (ang. *Base Service Provider*) oraz występujące w ramach sesji usługi rozszerzające LSP (ang. *Layered Service Provider*). Usługi BSP definiują szczegóły implementacji protokołu komunikacyjnego – ustanawianie połączenia, transfer danych i procedury obsługi błędów. LSP definiują procedury komunikacyjne oparte o usługi istniejące w systemie operacyjnym. Podczas wybierania odpowiedniego dostawcy, aplikacja zgłasza do interfejsu biblioteki WinSock 2.x żądanie udostępnienia gniazda[1] odpowiednio scharakteryzowanej usługi. Specjalna warstwa biblioteki WinSock 2.x zwana SPI (ang. *Service Provider Interface*) zarządzająca usługami zainstalowanymi w systemie Windows sprawdza ich dostępność przeglądając stos dostawców począwszy od jego wierzchołka. Po napotkaniu usługi zgodnej z żądanymi parametrami zwraca do aplikacji deskryptor gniazda wykorzystującego daną usługę. W wypadku istnienia więcej niż jednej usługi o tych samych parametrach, o wyborze decyduje wzajemne położenie usług na stosie, przy czym pierwszeństwo wyboru mają usługi położone bliżej wierzchołka stosu usług [17].

WinSock 2.x udostępnia dwie grupy poleceń: pierwsza z nich jest zgodna z funkcjami interfejsu gniazd (ang. *sockets interface*) stosowanymi w systemach UNIX/BSD, druga jest specyficzna dla implementacji Windows. Nazwy wszystkich funkcji interfejsu gniazd specyficznych dla systemów operacyjnych Windows rozpoczynają się od liter WSA (skrót od WinSock API). Definicje wszystkich funkcji biblioteki WinSock znajdują się w pliku nagłówkowym *winsock2.h* (dla wersji biblioteki 2.x) lub w *winsock.h* (dla wcześniejszych wersji). Gniazda w systemach Windows implementowane są w zewnętrznej bibliotece dynamicznej *ws2_32.dll*. Z tego powodu podczas kompilacji

[1] Gniazdo w telekomunikacji (ang. *socket*) jest pojęciem abstrakcyjnym reprezentującym dwukierunkowy punkt końcowy połączenia (ang. *endpoint*). Dwukierunkowość oznacza możliwość odbierania oraz wysyłania danych.

programów korzystających z WinSock 2.x należy pamiętać o statycznym łączeniu ich z biblioteką importową *ws2_32.lib*. Biblioteka ta jest standardowo dostępna w zasobach systemów Windows XP z Service Pack 2,3, Vista oraz 7.



Rysunek 3.1. Hierarchia warstw w bibliotece WinSock 2.x

3.2. Podstawowe funkcje

Poniżej omówiono podstawowe funkcje WinSock API pomocne w programowej kontroli transmisji bezprzewodowej w standardzie Bluetooth.

3.2.1. Funkcja WSAStartup()

Funkcja `WSAStartup()` odwzorowuje w przestrzeń adresową macierzystego procesu identyfikator biblioteki *ws2_32.dll*. Funkcja powinna być wywołana przed wszystkimi innymi odwołaniami do Windows Sockets API.

```
int WSAStartup(
  __in   WORD wVersionRequested,
  __out  LPWSADATA lpWSAData
);
```

Parametr `wVersionRequested` określa wymaganą przez implementację wersję biblioteki. Wersję biblioteki można wpisać do `wVersionRequested` za pomocą makrodefinicji `MAKEWORD` zdefiniowanej w module *windef.h*. Wskaźnik `lpWSAData` wskazuje na strukturę:

```
typedef struct WSAData {
  WORD           wVersion;
  WORD           wHighVersion;
```

```
    char           szDescription[WSADESCRIPTION_LEN+1];
    char           szSystemStatus[WSASYS_STATUS_LEN+1];
    unsigned short iMaxSockets;
    unsigned short iMaxUdpDg;
    char FAR       *lpVendorInfo;
} WSADATA, *LPWSADATA;
```

gdzie umieszczane są dane dotyczące dostępnej implementacji WinSock. W tabeli 3.1 zaprezentowano specyfikację tej struktury. Prawidłowo wykonana funkcja `WSAStartup()` zwraca wartość `0`.

Tabela 3.1. Specyfikacja struktury WSADATA

Element struktury	Znaczenie
`wVersion`	Numer wersji używanej biblioteki *ws2_32.dll*.
`wHighVersion`	Maksymalny numer wersji biblioteki wspierany przez daną implementację.
`szDescription`	Opis biblioteki w formie łańcucha znaków ASCII zakończonego zerowym ogranicznikiem (maksymalnie 256 znaków).
`szSystemStatus`	Wskaźnik do łańcucha znaków ASCII (zakończonego zerowym ogranicznikiem) zawierającego informacje konfiguracyjne biblioteki *ws2_32.dll*. Parametru tego nie należy traktować jako uzupełnienia danych zawartych w `szDescription`.
`iMaxSockets`	Maksymalna liczba gniazd, jaką może otworzyć pojedynczy proces (0 oznacza brak ograniczeń dla procesu).
`iMaxUdpDg`	Maksymalny rozmiar datagramu UDP (0 oznacza brak ograniczeń) .
`lpVendorInfo`	Wskaźnik do łańcucha znaków zawierającego informację na temat producenta biblioteki. W wersjach WinSock 2.x atrybut ten jest ignorowany.

3.2.2. Funkcja WSACleanup()

Bezparametrowa funkcja `WSACleanup()` usuwa z przestrzeni adresowej macierzystego procesu bibliotekę WinSock (*ws2_32.dll*) oraz zwalnia przydzielone jej zasoby.

```
int WSACleanup(void);
```

Prawidłowo wykonana funkcja zwraca `0`, zaś w przeciwnym wypadku wartość `SOCKET_ERROR`. Kody ewentualnych błędów można diagnozować za pomocą funkcji `WSAGetLastError()`.

3.2.3. Funkcja WSALookupServiceBegin()

Funkcja `WSALookupServiceBegin()` rozpoczyna proces wyszukiwania informacji o usługach udostępnianych przez bibliotekę WinSock. Podstawowymi parametrami określającymi dostępną usługę są: identyfikator ID klasy usługi, nazwa usługi, identyfikator obszaru nazw (domeny komunikacyjnej) usług oraz adresy IP z portami na których nasłuchuje serwer usług. Funkcja `WSALookupServiceBegin()` zwraca identyfikator `lphLookup` który powinien być użyty przez kolejne zapytania realizowane za pomocą funkcji `WSALookupServiceNext()`.

```
INT WSALookupServiceBegin(
  __in   LPWSAQUERYSET lpqsRestrictions,
  __in   DWORD dwControlFlags,
  __out  LPHANDLE lphLookup
);
```

Wskaźnik `lpqsRestrictions` wskazuje na strukturę:

```
 typedef struct _WSAQuerySet {
  DWORD          dwSize;
  LPTSTR         lpszServiceInstanceName;
  LPGUID         lpServiceClassId;
  LPWSAVERSION   lpVersion;
  LPTSTR         lpszComment;
  DWORD          dwNameSpace;
  LPGUID         lpNSProviderId;
  LPTSTR         lpszContext;
  DWORD          dwNumberOfProtocols;
  LPAFPROTOCOLS  lpafpProtocols;
  LPTSTR         lpszQueryString;
  DWORD          dwNumberOfCsAddrs;
  LPCSADDR_INFO  lpcsaBuffer;
  DWORD          dwOutputFlags;
  LPBLOB         lpBlob;
} WSAQUERYSET, *PWSAQUERYSET, *LPWSAQUERYSET;
```

której atrybuty przechowują informacje określające kryteria, na podstawie których będą wyszukiwane dostępne usługi. Specyfikacja struktury `WSAQUERYSET` została zaprezentowana w tabeli 3.2.

Tabela 3.2. Specyfikacja struktury WSAQUERYSET

Element struktury	Znaczenie
`dwSize`	Rozmiar struktury w bajtach , który każdorazowo należy prawidłowo określić za pomocą operatora `sizeof()`. W przeciwnym wypadku funkcja nie będzie działać poprawnie.
`lpszServiceInstanceName`	Atrybut opcjonalny. Wskaźnik do łańcucha znaków zakończonego zerowym ogranicznikiem opisującego nazwę usługi.
`lpServiceClassId`	Identyfikator GUID klasy usług.
`lpVersion`	Atrybut opcjonalny. Wskaźnik do zmiennej zawierającej wymaganą wersję przestrzeni nazw dostarczyciela usługi.
`lpszComment`	Atrybut ignorowany przy zapytaniach.
`dwNameSpace`	Identyfikator obszaru nazw do którego ma być ograniczone przeszukiwane informacji o usłudze, parametr NS_ALL oznacza przeszukiwanie wszystkich obszarów nazw, zaś NS_BTH oznacza przeszukiwanie obszarów nazw Bluetooth.
`lpNSProviderId`	Wskaźnik do identyfikatora GUID w określonym obszarze nazw identyfikującym dostawcę usługi. Odpowiednie obszary nazw można zidentyfikować za pomocą funkcji `WSAEnumNameSpaceProviders()` lub `WSAEnumNameSpaceProvidersEx()`.
`lpszContext`	Atrybut opcjonalny. Wskaźnik określa początkowy obszar nazw do którego będzie wysłane zapytanie.
`dwNumberOfProtocols`	Rozmiar tablicy (w bajtach) zawierającej listę protokołów do których ograniczone są zapytania. Wartość ta może być zerowa.

`lpafpProtocols`	Atrybut opcjonalny. Wskazuje na tablicę struktur:
	`typedef struct _AFPROTOCOLS {`
	`  INT iAddressFamily;`
	`  INT iProtocol;`
	`} AFPROTOCOLS, *PAFPROTOCOLS, *LPAFPROTOCOLS;`
	zawierającą listę protokołów, do których można ograniczyć zapytania:
	`iAddressFamily` – rodzina adresów do jakich zapytanie ma być ograniczone,
	`iProtocol` – protokół do jakiego zapytanie ma być ograniczone.

`lpszQueryString`	Atrybut opcjonalny. Służy do określania postaci łańcucha znaków poprzez który może być realizowane zapytanie.
`dwNumberOfCsAddrs`	Atrybut ignorowany w trakcie realizacji zapytań.
`lpcsaBuffer`	Jeżeli w trakcie wywoływania funkcji `WSALookupServiceNext()` znacznik `dwControlFlags` zawiera wartość `LUP_RETURN_ADDR` adres urządzenia zwracany jest poprzez wskaźnik `lpcsaBuffer>RemoteAddr.lpSockaddr`.
`dwOutputFlags`	Atrybut ignorowany w trakcie realizacji zapytań.

`lpBlob`	Atrybut opcjonalny. Wskaźnik do struktury
	`typedef struct _BLOB {`
	`ULONG cbSize;`
	`BYTE *pBlobData;`
	`} BLOB;`
	wyznaczanej z Binary Large Object zawierającej informację o bloku danych.
	`cbSize` – określa wielkość bloku danych (w bajtach) wskazywanych przez `pBlobData`.

Znacznik `dwControlFlags` określa stopień szczegółowości wyszukiwania i może być logiczną kombinacją wielu predefiniowanych stałych z których (z praktycznego punktu widzenia dla programistów urządzeń Bluetooth) najważniejsze to:
`LUP_CONTAINERS` – znacznik powinien być zawsze określony w trakcie identyfikacji urządzeń lub usług oferowanych przez urządzenia,
`LUP_FLUSHCACHE` – czyści bufor danych zawierający informacje na temat wcześniej wykrytych urządzeń,
`LUP_RETURN_TYPE` – można odzyskać informacje na temat klasy urządzeń Bluetooth. Klasy urządzeń są opisane w specyfikacji Bluetooth. Znacznik ten może być przydatny w trakcie określania typu ikony dla każdego wykrytego urządzenia (telefony komórkowe, drukarki, komputery, itp.). Moduł *bthdef.h* definiuje kilka makrodefinicji przydatnych w trakcie przeprowadzania analizy składniowej tego elementu.
`LUP_RETURN_NAME` – nazwa urządzenia wyspecyfikowana przez atrybut `lpszServiceInstanceName`.
`LUP_RETURN_ADDR` – określa adresy zidentyfikowanych urządzeń z włączoną funkcją Bluetooth.

Prawidłowo wykonana funkcja `WSALookupServiceBegin()` zwraca wartość zerową, w przeciwnym wypadku – `SOCKET_ERROR`. Kody

pojawiających się błędów wykonania można diagnozować za pomocą funkcji WSAGetLastError().

3.2.4. Funkcja WSALookupServiceNext()

Funkcja WSALookupServiceNext() jest wywoływana po uzyskaniu identyfikatora hLookup od funkcji WSALookupServiceBegin() w celu uzyskania informacji o dostępnej usłudze.

```
INT WSALookupServiceNext(
  __in     HANDLE hLookup,
  __in     DWORD dwControlFlags,
  __inout  LPDWORD lpdwBufferLength,
  __out    LPWSAQUERYSET lpqsResults
);
```

Argument dwControlFlags jest znacznikiem służącym do kontroli zapytania. Obecnie zdefiniowany jest pod postacią predefiniowanej stałej LUP_FLUSHPREVIOUS. Użyty jako parametr wejściowy wskaźnik lpdwBufferLength wskazuje na liczbę bajtów w buforze wskazywanym przez lpqsResults. Użyty jako parametr wyjściowy (gdy funkcja zwróci kod błędu WSAEFAULT), wskaźnik lpdwBufferLength wskazuje na minimalną liczbę bajtów jaką powinien zawierać bufor aby odebrać dane. Wskaźnik lpqsResults wskazuje na obszar pamięci przechowujący wartości przypisane polom struktury WSAQUERYSET. Funkcję należy wywoływać tak długo, aż nie zwróci wartości WSA_E_NO_MORE, co oznacza, że wszystkie dane zapisane w WSAQUERYSET zostały przetworzone.
Prawidłowo wykonana funkcja WSALookupServiceNext() zwraca 0, w przeciwnym wypadku – SOCKET_ERROR. Kody pojawiających się błędów wykonania można diagnozować za pomocą funkcji WSAGetLastError().

3.2.5. Funkcja WSALookupServiceEnd()

Funkcja WSALookupServiceEnd() zwalnia identyfikator hLookup uprzednio przydzielony za pomocą WSALookupServiceBegin().

```
INT WSALookupServiceEnd(
  __in  HANDLE hLookup
);
```

Prawidłowo wykonana funkcja zwraca wartość zerową, w przeciwnym wypadku – SOCKET_ERROR. Kody pojawiających się błędów wykonania można diagnozować za pomocą funkcji WSAGetLastError().

3.2.6. Funkcja WSASetService()

Aplikacje Bluetooth używają funkcji `WSASetService()` do zarejestrowania lub usunięcia rekordu SDP z odpowiedniej przestrzeni nazw rejestru systemowego.

```
INT WSASetService(
  __in  LPWSAQUERYSET lpqsRegInfo,
  __in  WSAESETSERVICEOP essOperation,
  __in  DWORD dwControlFlags
);
```

Wskaźnik `lpqsRegInfo` wskazuje na strukturę `WSAQUERYSET` zawierającej informacje na temat rejestrowanej lub usuwanej usługi. Znacznik `dwControlFlags` przyjmuje wartość `0`. Parametr `essOperation` reprezentuje jedną z operacji:
`RNRSERVICE_REGISTER` – zarejestrowanie usługi,
`RNRSERVICE_DEREGISTER` – usunięcie i wyrejestrowanie usługi,
`RNRSERVICE_DELETE` – usunięcie nazwy usługi.

Prawidłowo wykonana funkcja `WSASetService()` zwraca `0`, w przeciwnym wypadku – `SOCKET_ERROR`. Kody pojawiających się błędów wykonania można diagnozować za pomocą funkcji `WSAGetLastError()`.

Przykład praktycznego użycia funkcji `WSASetService()` w celu zarejestrowania nowego rekordu SDP można znaleźć na stronie http://msdn.microsoft.com/en-us/library/aa450140(v=MSDN.10).aspx. Pełne wykorzystanie omawianej funkcji wiąże się z koniecznością poprawnego zdefiniowania rekordu SDP. W tym celu należy zapoznać się z dokumentacją techniczną oprogramowywanego urządzenia.

3.2.7. Funkcja WSAAddressToString()

Wiele funkcji biblioteki WinSock wymaga podania struktury reprezentującej adres gniazda:

```
struct sockaddr
{
   unsigned short sa_family; //rodzina adresów, AF_XXX
   char  sa_data[14];    //co najwyżej 14 bajtów adresu
                         //właściwego protokołu
};
```

Adres będącego w zasięgu i zidentyfikowanego urządzenia Bluetooth może być konwertowany na łańcuch znaków za pomocą funkcji:

```
INT WSAAPI WSAAddressToString(
```

```
    __in      LPSOCKADDR lpsaAddress,
    __in      DWORD dwAddressLength,
    __in_opt  LPWSAPROTOCOL_INFO lpProtocolInfo,
    __inout   LPTSTR lpszAddressString,
    __inout   LPDWORD lpdwAddressStringLength
  );
```

Wskaźnik `lpsaAddress` wskazuje na strukturę typu `sockaddr`. W domenie protokołów Bluetooth zamiast struktury `sockaddr` używa się struktur `SOCKADDR_BTH` lub/i `CSADDR_INFO` wykonując przy przekazywaniu wskaźnika rzutowanie na strukturę (`struct sockaddr*`). Specyfikacja struktur `SOCKADDR_BTH` oraz `CSADDR_INFO` została zaprezentowana odpowiednio w tabelach 3.3 oraz 3.4.

```
typedef struct _SOCKADDR_BTH {
  USHORT   addressFamily;
  BTH_ADDR btAddr;
  GUID     serviceClassId;
  ULONG    port;
} SOCKADDR_BTH, *PSOCKADDR_BTH;
```

Tabela 3.3. Specyfikacja struktury SOCKADDR_BTH

Element struktury	Znaczenie
`addressFamily`	Rodzina adresów związana z danym protokołem. Stała AF_BTH definiuje rodzinę adresów Bluetooth.
`btAddr`	Adres urządzenia Bluetooth. Dla aplikacji klienckiej wartość tego pola może być równa 0.
`serviceClassId`	Identyfikator klasy usługi. Atrybut ten jest ignorowany, gdy program używa funkcji `bind()` lub gdy aplikacja używa wyspecyfikowanego portu.
`port`	RFCOMM

```
typedef struct _CSADDR_INFO {
  SOCKET_ADDRESS LocalAddr;
  SOCKET_ADDRESS RemoteAddr;
  INT            iSocketType;
  INT            iProtocol;
} CSADDR_INFO;
```

Tabela 3.4. Specyfikacja struktury CSADDR_INFO

Element struktury	Znaczenie
`LocalAddr`	Adres lokalnego odbiornika radiowego Bluetooth.
`RemoteAddr`	Adres będącego w zasięgu urządzenia Bluetooth.
`iSocketType`	Rodzaj gniazda.
`iProtocol`	Protokół komunikacyjny, z jakiego korzysta gniazdo.

Występujący jako argument wejściowy funkcji `WSAAddressToString()` parametr `dwAddressLength` reprezentuje długość adresu (w bajtach). Opcjonalnie wykorzystywany wskaźnik `lpProtocolInfo` wskazuje na strukturę `WSAPROTOCOL_INFO` zawierającej szczegółowe informacje opisujące dany protokół komunikacyjny. Wskaźnik `lpszAddressString` wskazuje na bufor danych, w którym umieszczany jest łańcuch znaków reprezentujący adres będącego w zasięgu urządzenia. Parametr `lpdwAddressStringLength` określa rozmiar bufora danych przechowującego łańcuch znaków reprezentujący adres urządzenia. Jeżeli rozmiar bufora nie został poprawnie określony funkcja zakończy swoje działanie z kodem błędu `WSAEFAULT`.
Prawidłowo wykonana funkcja zwraca wartość zerową, w przeciwnym wypadku – `SOCKET_ERROR`. Kody pojawiających się błędów wykonania można diagnozować za pomocą funkcji `WSAGetLastError()`.

3.2.8. Funkcja WSASocket()

Funkcja `WSASocket()` tworzy nowe gniazdo służące do komunikacji zwracając jego deskryptor, wykorzystywany przy każdorazowym odwoływaniu się do funkcji gniazda.

```
SOCKET WSASocket(
  __in  int af,
  __in  int type,
  __in  int protocol,
  __in  LPWSAPROTOCOL_INFO lpProtocolInfo,
  __in  GROUP g,
  __in  DWORD dwFlags
);
```

Pierwszy argument określa domenę komunikacyjną i służy do wyboru rodziny protokołów komunikacyjnych. Dla protokołów Bluetooth parametr ten powinien mieć wartość `AF_BTH`. Parametr `type` określa typ komunikacji. Dla protokołów Bluetooth parametr ten powinien mieć wartość `SOCK_STREAM` (gniazdo strumieniowe) określając tym samym możliwość dwukierunkowej komunikacji

bazującej na transmisji danych w mechanizmie OOB. Parametr `protocol` określa protokół, z jakiego gniazdo korzysta. Dla protokołów Bluetooth parametr ten powinien mieć wartość `BTHPROTO_RFCOMM`. Wskaźnik `lpProtocolInfo` wskazuje na strukturę `WSAPROTOCOL_INFO` opisującą charakterystykę tworzonego gniazda. Jeżeli `lpProtocolInfo` przypisano wartość `NULL`, biblioteka *ws2_32.dll* używać będzie trzech pierwszych argumentów funkcji (`af`, `type`, `protocol`). Parametr `g` jest zarezerwowany, zaś znacznik `dwFlags` określa dodatkowe atrybuty gniazda polegające na możliwości jego blokowania lub nieblokowania. Gniazda mogą pracować w jednym z dwóch trybów: *blokującym* lub *nieblokującym*. Standardowo ustalany jest tryb blokujący. Oznacza to, że wywołanie funkcji gniazda powoduje zatrzymanie programu dopóki funkcja nie zostanie wykonana. W celu asynchronicznego zarządzania połączeniami można przełączyć gniazda w tryb nieblokujący. Jeżeli parametrowi `dwFlags` przypiszemy wartość `WSA_FLAG_OVERLAPPED` gniazdo może być dostępne w trybie nieblokującym dla asynchronicznych operacji I/O. Możliwym jest wówczas wykorzystanie funkcji `WSASend()`, `WSASendTo()`, `WSARecv()`, `WSARecvFrom()` oraz `WSAIoctl()`.

Prawidłowo wykonana funkcja zwraca wartość określającą deskryptor gniazda, w przeciwnym wypadku – `INVALID_SOCKET`. Kody pojawiających się błędów wykonania można diagnozować za pomocą funkcji `WSAGetLastError()`.

3.2.9. Funkcja socket()

Użycie funkcji `socket()` jest równoważne z wywołaniem `WSASocket()` z odpowiednimi wartościami parametrów `af`, `type` oraz `protocol`.

```
SOCKET WSAAPI socket(
  __in  int af, //AF_BTH
  __in  int type, // SOCK_STREAM
  __in  int protocol // BTHPROTO_RFCOMM
);
```

3.2.10. Funkcja closesocket()

Funkcja zwalnia deskryptor `s` uprzednio przydzielony za pomocą `WSASocket()` lub `socket()`.

```
int closesocket(
  __in  SOCKET s
);
```

Prawidłowo wykonana funkcja zwraca `0`, w przeciwnym wypadku – `SOCKET_ERROR`. Kody pojawiających się błędów wykonania można diagnozować za pomocą funkcji `WSAGetLastError()`.

3.2.11. Funkcja getsockopt()

Funkcja `getsockopt()` pobiera parametry gniazda.

```
int getsockopt(
  __in     SOCKET s,
  __in     int level,
  __in     int optname,
  __out    char *optval,
  __inout  int *optlen
);
```

Parametr `s` jest deskryptorem gniazda zwróconym przez funkcje `WSASocket()`/ `socket()`. Parametr `level` opisuje poziom, na którym zdefiniowane są wykonywane operacje. Podczas programowania urządzeń Bluetooth najczęściej wykorzystywanymi są poziomy:

```
#define SOL_RFCOMM  BTHPROTO_RFCOMM
#define SOL_L2CAP   BTHPROTO_L2CAP
#define SOL_SDP     0x0101
```

Parametr `optname` jest nazwą opcji zdefiniowaną w obrębie poziomu:

```
#define SO_BTH_AUTHENTICATE 0x80000001
//optlen=sizeof(ULONG),
//optval=&(ULONG)TRUE/FALSE

#define SO_BTH_ENCRYPT      0x00000002
//optlen=sizeof(ULONG),
//optval=&(ULONG)TRUE/FALSE

#define SO_BTH_MTU          0x80000007
//optlen=sizeof(ULONG),optval=&mtu

#define SO_BTH_MTU_MAX      0x80000008
//optlen=sizeof(ULONG),optval=&max.mtu

#define SO_BTH_MTU_MIN      0x8000000a
//optlen=sizeof(ULONG),optval=&min.mtu
```

Wskaźnik optval wskazuje na bufor danych, w którym funkcja zwróci wartość odpowiadającą opcji optname. Wskaźnik optlen wskazuje na daną przechowującą rozmiar bufora optval, zaś po wywołaniu funkcji w miejscu wskazywanym przez optlen znajdzie się właściwa ilość danych umieszczonych w buforze.

Prawidłowo wykonana funkcja zwraca wartość zerową, w przeciwnym wypadku – SOCKET_ERROR. Kody pojawiających się błędów wykonania można diagnozować za pomocą funkcji WSAGetLastError().

3.2.12. Funkcja setsockopt()

Funkcja setsockopt() ustala parametry gniazda.

```
int setsockopt(
  __in  SOCKET s,
  __in  int level,
  __in  int optname,
  __in  const char *optval,
  __in  int optlen
);
```

Znaczenie argumentów oraz wartości zwracanych funkcji setsockopt() jest identyczne jak w przypadku getsockopt().

3.2.13. Przykłady

Na listingu 3.1 pokazano szkielet kodu obrazującego ogólne zasady stosowane podczas pracy z funkcjami Windows Socket API wykorzystywanymi w celu zdiagnozowania adresów oraz rodzaju będących w zasięgu urządzeń Bluetooth. Konstrukcja omawianego przykładu jest typowa dla tego typu programów. W pierwszej kolejności poprzez wywołanie funkcji WSAStartup() inicjowana jest biblioteka Windows Sockets w wersji 2.2. Następnie należy prawidłowo zadeklarować oraz zainicjować wskaźnik do struktury WSAQUERYSET. Struktura WSAQUERYSET ma wiele pól, jednak z punktu widzenia zapytań kierowanych do urządzeń Bluetooth istotne są dwa atrybuty: dwSize oraz dwNameSpace. Rozmiar struktury powinien być każdorazowo prawidłowo określony oraz w odniesieniu do urządzeń Bluetooth obszar nazw powinien być ustalony jako NS_BTH. Wszystkie inne atrybuty struktury WSAQUERYSET w tym przypadku nie są istotne. Znacznik dwControlFlag powinien jednoznacznie określać zasady, według których będące w zasięgu urządzenia mają być identyfikowane. Wywołanie funkcji WSALookupServiceBegin() rozpoczyna proces wyszukiwania będących w zasięgu urządzeń z włączoną opcją Bluetooth. Funkcja WSALookupServiceBegin() jedynie rozpoczyna proces wykrywania urządzeń nie zwracając jakichkolwiek o nich informacji. Aby zdiagnozować które urządzenia będące w zasięgu w rzeczywistości zostały wykryte należy

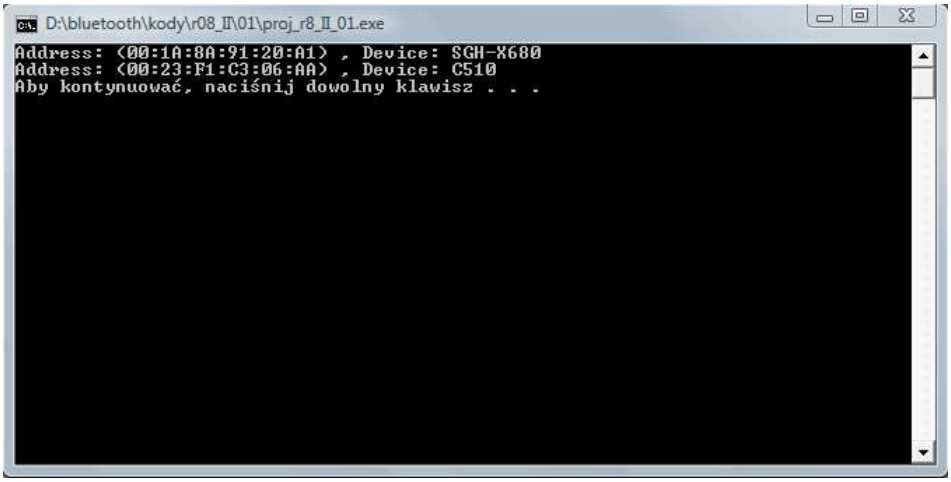
użyć `WSALookupServiceNext()`. W omawianym przykładzie, adres każdego będącego w zasięgu głównego modułu radiowego i zidentyfikowanego urządzenia Bluetooth jest konwertowany na łańcuch znaków za pomocą funkcji `WSAAddressToString()`. Po zidentyfikowaniu wszystkich będących w zasięgu urządzeń wywoływana jest funkcja `WSALookupServiceEnd()`. Na rysunku 3.2 pokazano omawiany program w trakcie działania.

Listing 3.1. Wyszukiwanie będących w zasięgu urządzeń z włączoną funkcją Bluetooth

```
#include <iostream>
#include <assert>
#pragma option push -a1
  #include <winsock2>
  #include "Ws2bth.h"
#pragma option pop

#pragma comment(lib, "ws2_32.lib")

using namespace std;

template <class T>
inline void releaseMemory(T &x)
{
    assert(x != NULL);
    delete x;
    x = NULL;
}
//---------------------------------------------------
void showError()
{
  LPVOID lpMsgBuf;
  FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
                 FORMAT_MESSAGE_FROM_SYSTEM |
                 FORMAT_MESSAGE_IGNORE_INSERTS,
                 NULL, WSAGetLastError(), 0, (LPTSTR)
                 &lpMsgBuf, 0, NULL );
                 fprintf(stderr, "\n%s\n", lpMsgBuf);
  free(lpMsgBuf);
  cin.get();
}
//---------------------------------------------------
int main()
{
   WORD wVersionRequested;
   WSADATA wsaData;
```

```
wVersionRequested = MAKEWORD(2,2);
if(WSAStartup(wVersionRequested, &wsaData) != 0) {
  showError();
}
WSAQUERYSET *lpqsRestrictions =  new
WSAQUERYSET[sizeof(WSAQUERYSET)];
memset(lpqsRestrictions, 0, sizeof(WSAQUERYSET));
lpqsRestrictions->dwSize = sizeof(WSAQUERYSET);
lpqsRestrictions->dwNameSpace = NS_BTH;
DWORD dwControlFlags = LUP_CONTAINERS;
dwControlFlags |= LUP_FLUSHCACHE | LUP_RETURN_NAME |
                  LUP_RETURN_ADDR;
HANDLE hLookup;
if(SOCKET_ERROR ==
  WSALookupServiceBegin(lpqsRestrictions,
                        dwControlFlags,
                        &hLookup)) {
  WSACleanup();
  return 0;
};
BOOL searchResult = FALSE;
while(! searchResult) {
  if(NO_ERROR ==
     WSALookupServiceNext(hLookup,dwControlFlags,
     &lpqsRestrictions->dwSize, lpqsRestrictions)){
     char buffer[40] = {0};
     DWORD bufLength = sizeof(buffer);
     WSAAddressToString(lpqsRestrictions->
     lpcsaBuffer->RemoteAddr.lpSockaddr,
     sizeof(SOCKADDR_BTH), NULL, buffer,
     &bufLength);
     printf("Address: %s , Device: %s\n", buffer,
            lpqsRestrictions->
            lpszServiceInstanceName);
  } else {
     int WSAerror = WSAGetLastError();
     if(WSAerror == WSAEFAULT) {
        releaseMemory(lpqsRestrictions);
        lpqsRestrictions = new
        WSAQUERYSET[sizeof(WSAQUERYSET)];
      } else
         if(WSAerror == WSA_E_NO_MORE) {
           searchResult = TRUE;
           }
           else {
```

```
                showError();
                searchResult = TRUE;
              }
            }
    }
    WSALookupServiceEnd(hLookup);
    releaseMemory(lpqsRestrictions);
    WSACleanup();
    system("PAUSE");
    return 0;
}
//---------------------------------------------------
```



Rysunek 3.2. Wyszukiwanie urządzeń Bluetooth i określanie ich adresów

Przedstawiona na przykładzie programu z listingu 3.1 metoda konwertowania na łańcuch znaków adresów zidentyfikowanych i będących w zasięgu urządzeń Bluetooth nie jest jedyną możliwą do zastosowania. Na listingu 3.2 zaprezentowano nieskomplikowany kod, w którym bezpośrednio wykorzystano zasoby bibliotecznej struktury `WSAQUERYSET` oraz samodzielnie zdefiniowanej przez użytkownika struktury `MY_BTH_DEVICE`. Nazwa oraz adres wykrytego urządzenia przechowywane są odpowiednio w polach `bthDevName` oraz `bthAddr` struktury `MY_BTH_DEVICE`.

Listing 3.2. Bezpośredni odczyt informacji zapisanych w polach struktury `WSAQUERYSET`

```
#include <iostream>
#include <assert>
#pragma option push -a1
  #include <winsock2>
  #include "Ws2bth.h"
```

```
#pragma option pop

#pragma comment(lib, "ws2_32.lib")

using namespace std;

typedef struct {
   char  bthDevName[256];
   BTH_ADDR bthAddr;
} MY_BTH_DEVICE;
//---------------------------------------------------
template <class T>
inline void releaseMemory(T &x)
{
    assert(x != NULL);
    delete x;
    x = NULL;
}
//---------------------------------------------------
void showError()
{
  LPVOID lpMsgBuf;
  FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
                 FORMAT_MESSAGE_FROM_SYSTEM |
                 FORMAT_MESSAGE_IGNORE_INSERTS,
                 NULL, WSAGetLastError(), 0, (LPTSTR)
                 &lpMsgBuf, 0, NULL );
                 fprintf(stderr, "\n%s\n", lpMsgBuf);
  free(lpMsgBuf);
  cin.get();
}
//---------------------------------------------------
int main()
{
   MY_BTH_DEVICE *bthDev;
   SOCKADDR_BTH *socAddrBTH;
   BTH_ADDR btAddr;

   WORD wVersionRequested;
   WSADATA wsaData;
   wVersionRequested = MAKEWORD(2,2);
   if(WSAStartup(wVersionRequested, &wsaData) != 0) {
     showError();
   }
   WSAQUERYSET *lpqsRestrictions =  new
```

```
            WSAQUERYSET[sizeof(WSAQUERYSET)];
memset(lpqsRestrictions, 0, sizeof(WSAQUERYSET));
lpqsRestrictions->dwSize = sizeof(WSAQUERYSET);
lpqsRestrictions->dwNameSpace = NS_BTH;
DWORD dwControlFlags = LUP_CONTAINERS;
dwControlFlags |= LUP_FLUSHCACHE | LUP_RETURN_NAME |
                  LUP_RETURN_ADDR;
HANDLE hLookup;
if(SOCKET_ERROR ==
   WSALookupServiceBegin(lpqsRestrictions,
        dwControlFlags, &hLookup)) {
  WSACleanup();
  return 0;
}
BOOL searchResult = FALSE;
while(! searchResult) {
  if(NO_ERROR ==
     WSALookupServiceNext(hLookup,dwControlFlags,
     &lpqsRestrictions->dwSize, lpqsRestrictions)){

    bthDev = new
             MY_BTH_DEVICE[sizeof(MY_BTH_DEVICE)];
    memset(bthDev,0,sizeof(bthDev));
    strcpy (bthDev->bthDevName, lpqsRestrictions->
            lpszServiceInstanceName);
    printf("\tDevice :%s", bthDev->bthDevName);
    socAddrBTH = (SOCKADDR_BTH *)lpqsRestrictions->
                 lpcsaBuffer->RemoteAddr.lpSockaddr;
    bthDev->bthAddr = socAddrBTH->btAddr;
    printf("\tAddress :%X\n", bthDev->bthAddr);
    //--alternatywny sposób--
    btAddr = ((SOCKADDR_BTH *)lpqsRestrictions->
        lpcsaBuffer->RemoteAddr.lpSockaddr)->btAddr;
    printf("Device Address is 0X%012X\n", btAddr);
    printf("%s\t0X%04X\t\t0X%08X\t0X%0X\n",
           lpqsRestrictions->
           lpszServiceInstanceName,
           GET_NAP(btAddr), GET_SAP(btAddr),
           lpqsRestrictions->dwNameSpace);
  } else {
     int WSAerror = WSAGetLastError();
     if(WSAerror == WSAEFAULT) {
        releaseMemory(lpqsRestrictions);
        lpqsRestrictions = new
             WSAQUERYSET[sizeof(WSAQUERYSET)];
```

```
         } else
            if(WSAerror == WSA_E_NO_MORE) {
              searchResult = TRUE;
              }
              else {
                 showError();
                 searchResult = TRUE;
              }
            }
   }
   releaseMemory(bthDev);
   WSALookupServiceEnd(hLookup);
   releaseMemory(lpqsRestrictions);
   WSACleanup();
   system("PAUSE");
   return 0;
}
//---------------------------------------------------
```

Na listingu 3.3 zaprezentowano jeden z możliwych sposobów wykorzystania w programie funkcji `WSASocket()`, `getsockopt()` oraz `closesocket()` w celu określenia usług udostępnianych przez będące w zasięgu głównego modułu radiowego zewnętrzne urządzenia z włączoną opcją Bluetooth. Na rysunku 3.3 pokazano wynik działania programu.

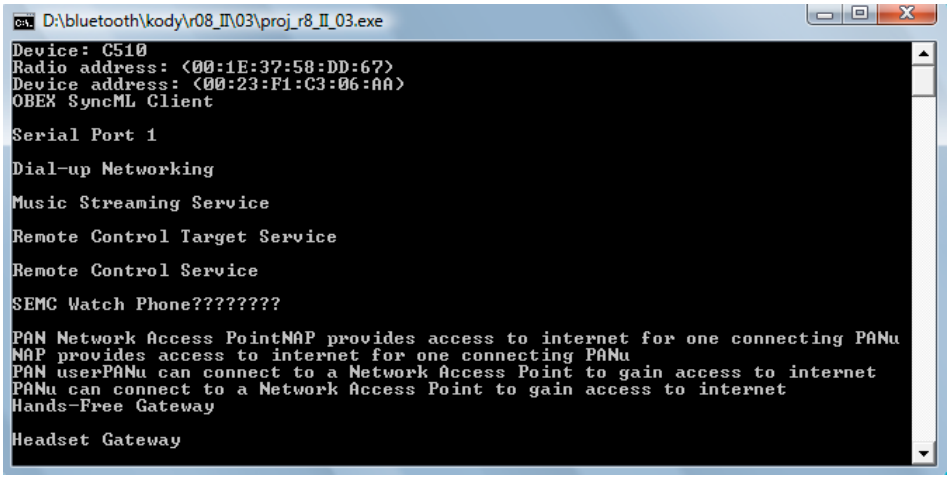
Listing 3.3. Utworzenie gniazda i odczyt usług udostępnianych przez urządzenie Bluetooth

```
#include <iostream>
#include <initguid>
#pragma option push -a1
  #include <winsock2>
  #include "Ws2bth.h"
#pragma option pop

#pragma comment(lib, "ws2_32.lib")

using namespace std;
//---------------------------------------------------
void showError()
{
  LPVOID lpMsgBuf;
  FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
                 FORMAT_MESSAGE_FROM_SYSTEM |
                 FORMAT_MESSAGE_IGNORE_INSERTS,
                 NULL, GetLastError(), 0, (LPTSTR)
```

```
                &lpMsgBuf, 0, NULL );
                fprintf(stderr, "\n%s\n", lpMsgBuf);
  free(lpMsgBuf);
  cin.get();
}
//--------------------------------------------------
int main()
{
  WORD wVersionRequested = MAKEWORD(2,2);
  WSADATA wsaData;
  if(WSAStartup(wVersionRequested, &wsaData) == 0){
    SOCKET s = WSASocket(AF_BTH, SOCK_STREAM,
                         BTHPROTO_RFCOMM, NULL,
                         NULL, NULL);

    if(s == INVALID_SOCKET) {
      WSACleanup();
      return 0;
    }
    WSAPROTOCOL_INFO WSAprotocolInfo;
    int WSAprotocolInfoSize = sizeof(WSAprotocolInfo);
    if(getsockopt(s, SOL_SOCKET, SO_PROTOCOL_INFO,
                  (char*)&WSAprotocolInfo,
                  &WSAprotocolInfoSize) != 0){
      WSACleanup();
      return 0;
    }

    WSAQUERYSET qsRestrictions;
    memset(&qsRestrictions, 0, sizeof(qsRestrictions));
    qsRestrictions.dwSize = sizeof(qsRestrictions);
    qsRestrictions.dwNameSpace = NS_BTH;
    HANDLE hLookup;
    DWORD dwControlFlags = LUP_RETURN_NAME |
                           LUP_CONTAINERS |
                           LUP_RETURN_ADDR |
                           LUP_FLUSHCACHE |
                           LUP_RETURN_TYPE |
                           LUP_RES_SERVICE;

    int WSALookup =
        WSALookupServiceBegin(&qsRestrictions,
                              dwControlFlags,
                              &hLookup);
```

```
if(WSALookup != SOCKET_ERROR) {
  while(WSALookup == 0) {
    char buffer[1000] = {0};
    DWORD dwBufferLength = sizeof(buffer);
    WSAQUERYSET *lpqsRestrictions =
                 (WSAQUERYSET*)&buffer;
    WSALookup = WSALookupServiceNext(hLookup,
                dwControlFlags, &dwBufferLength,
                lpqsRestrictions);
    if (WSALookup != NO_ERROR) {
       showError();
    }
    else {
      printf("Device: %s\n", lpqsRestrictions->
             lpszServiceInstanceName);
      CSADDR_INFO *pCSAddr = (CSADDR_INFO *)
                   lpqsRestrictions->lpcsaBuffer;
      WSAQUERYSET qsRestrictions;
      memset(&qsRestrictions, 0,
             sizeof(qsRestrictions));
      qsRestrictions.dwSize =
                     sizeof(qsRestrictions);
      GUID protocol = L2CAP_PROTOCOL_UUID;
      qsRestrictions.lpServiceClassId = &protocol;
      qsRestrictions.dwNameSpace = NS_BTH;
      char address[256];
      DWORD addressLength = sizeof(address);
      addressLength = sizeof(address);
      if(0 == WSAAddressToString(pCSAddr->
              LocalAddr.lpSockaddr,
              pCSAddr->LocalAddr.iSockaddrLength,
              &WSAprotocolInfo, address,
              &addressLength)) {
         printf("Radio address: %s\n", address);
      }
      addressLength = sizeof(address);
      if(0 == WSAAddressToString(pCSAddr->
              RemoteAddr.lpSockaddr,
              pCSAddr-> RemoteAddr.iSockaddrLength,
              &WSAprotocolInfo, address,
              &addressLength)) {
        printf("Device address: %s\n", address);
      }
      qsRestrictions.lpszContext = address;
      HANDLE hLookup;
```

```
          DWORD dwControlFlags = LUP_FLUSHCACHE |
                                 LUP_RETURN_NAME |
                                 LUP_RETURN_TYPE |
                                 LUP_RETURN_ADDR |
                                 LUP_RETURN_COMMENT;
          int WSALookup =
              WSALookupServiceBegin(&qsRestrictions,
                                    dwControlFlags,
                                    &hLookup);
          if (WSALookup != SOCKET_ERROR) {
            while (WSALookup == 0) {
              char buffer[2000] = {0};
              DWORD dwBufferLength = sizeof(buffer);
              WSAQUERYSET *lpqsRestrictions =
                           (WSAQUERYSET*)&buffer;
              WSALookup = WSALookupServiceNext(hLookup,
                          dwControlFlags, &dwBufferLength,
                          lpqsRestrictions);
              if(WSALookup != NO_ERROR) {
                showError();
              }
              else {
                printf("%s\n", lpqsRestrictions->
                       lpszServiceInstanceName);
                printf("%s\n", lpqsRestrictions->
                       lpszComment);
              }
            }//koniec while
            WSALookup = WSALookupServiceEnd(hLookup);
          }//koniec if
          else
            showError();
        }
      }// koniec while
      WSALookup = WSALookupServiceEnd(hLookup);
    }
    else
      showError();
    if (SOCKET_ERROR == closesocket(s))
       showError();
    WSACleanup(); }//if WSAStartup()
  system("PAUSE");
  return 0;
}
//---------------------------------------------------
```



Rysunek 3.3. Identyfikacja usług udostępnianych przez urządzenie Bluetooth

3.2.14. Funkcje służące do ustalania i zamykania połączeń

Do nawiązywania połączeń służą cztery funkcje: dla serwera – `WSAAccept()` lub `accept()`, oraz dla klienta – `WSAConnect()` lub `connect()`. Serwer aby mógł przyjmować połączenia powinien przed wywołaniem funkcji `WSAAccept()`/`accept()`, wywołać funkcję `listen()` w celu określenia wielkości kolejki zgłoszeń aplikacji klienckich. Funkcja `shutdown()` umożliwia zamykanie połączeń w celu odpowiedniego sterowania przepływem danych.

3.2.15. Funkcja WSAAccept()

Funkcja `WSAAccept()` służy do pobrania zgłoszenia z kolejki lub oczekiwania na takie zgłoszenie.

```
SOCKET WSAAccept(
  __in     SOCKET s,
  __out    struct sockaddr *addr,
  __inout  LPINT addrlen,
  __in     LPCONDITIONPROC lpfnCondition,
  __in     DWORD dwCallbackData
);
```

Parametr `s` jest deskryptorem gniazda, które jest już połączone. Wskaźnik `addr` wskazuje na strukturę typu `sockaddr` przechowującą adres połączenia. W domenie protokołów Bluetooth zamiast `sockaddr` używa się struktury `SOCKADDR_BTH` wykonując przy przekazywaniu wskaźnika rzutowanie na strukturę (`sockaddr*`). Wskaźnik `addrlen` wskazuje na daną typu `int` przechowującą rozmiar struktury z adresem połączenia (urządzenia). Funkcja

`WSAAccept()` warunkowo akceptuje połączenie bazując na danych powrotnych funkcji wskazywanej przez `lpfnCondition`. Dane, na podstawie których połączenie jest akceptowane lub odrzucane umieszczone są w argumencie `dwCallbackData`.

Prawidłowo wykonana funkcja zwraca nowy deskryptor gniazda, w przeciwnym wypadku – `INVALID_SOCKET`. Kody pojawiających się błędów wykonania można diagnozować za pomocą funkcji `WSAGetLastError()`.

3.2.16. Funkcja accept()

Użycie funkcji `accept()` jest równoważne z wywołaniem `WSAAccept()` z odpowiednimi wartościami parametrów `s`, `addr` oraz `addrlen`.

```
SOCKET accept(
  __in     SOCKET s,
  __out    struct sockaddr *addr,
  __inout  int *addrlen
);
```

3.2.17. Funkcja bind()

Funkcja `bind()` służy do powiązania gniazda z adresem/portem lokalnej maszyny i jest używana głównie przez aplikacje działające jako serwery.

```
int bind(
  __in  SOCKET s,
  __in  const struct sockaddr *name,
  __in  int namelen
);
```

Parametr `s` jest deskryptorem identyfikującym gniazda, które ma zostać powiązane z odpowiednim adresem wskazywanym przez wskaźnik `name`. Wskaźnik `name` wskazuje na strukturę typu `sockaddr` zawierającą żądany adres. W domenie protokołów Bluetooth zamiast `sockaddr` używa się struktury `SOCKADDR_BTH` wykonując przy przekazywaniu wskaźnika rzutowanie na strukturę (`const sockaddr*`). Parametr `namelen` przechowuje długość wskazywanego adresu. Aplikacja Bluetooth działająca jako serwer powinny używać struktury `SOCKADDR_BTH` jako argumentu funkcji `bind()`.

Listing 3.4. Przykład użycia funkcji `bind()`

```
//-----------------------------------------------------
SOCKET s;
SOCKADDR_BTH name;
int namelen = sizeof(name);
```

```
//...
name.addressFamily = AF_BTH;
name.btAddr = 0;
name.serviceClassId = NULL_GUID; //L2CAP_PROTOCOL_UUID
name.port = 0;//BT_PORT_ANY;
if(SOCKET_ERROR == bind(s,(const sockaddr*)&name,
sizeof(name))) {
//...
}
//-------------------------------------------------
```

Prawidłowo wykonana funkcja zwraca wartość zerową, w przeciwnym wypadku – SOCKET_ERROR. Kody pojawiających się błędów wykonania można diagnozować za pomocą funkcji WSAGetLastError().

3.2.18. Funkcja WSAConnect()

Funkcja WSAConnect() umożliwia związanie wcześniej utworzonego gniazda z odległym punktem końcowym zlokalizowanym pod określonym adresem zapisanym w strukturze typu sockaddr wskazywanej przez parametr name.

```
int WSAConnect(
  __in   SOCKET s,
  __in   const struct sockaddr *name,
  __in   int namelen,
  __in   LPWSABUF lpCallerData,
  __out  LPWSABUF lpCalleeData,
  __in   LPQOS lpSQOS,
  __in   LPQOS lpGQOS
);
```

Parametr s jest deskryptorem gniazda zwróconym przez funkcje WSASocket()/ socket(). Argument namelen jest rozmiarem struktury wskazywanej przez parametr name. W domenie protokołów Bluetooth zamiast struktury sockaddr używa się SOCKADDR_BTH wykonując przy przekazywaniu wskaźnika rzutowanie na wskaźnik do struktury (sockaddr*). Wskaźnik lpCallerData wskazuje na dane transmitowane w trakcie ustanawiania połączenia. Wskaźnik lpCalleeData wskazuje na dane zwrotne umieszczane w buforze po wywołaniu funkcji. Wskaźnik lpSQOS wskazuje na strukturę FLOWSPEC. Wskaźnik lpGQOS jest zarezerwowany. W trakcie ustanawiania połączenia blokującego z urządzeniem Bluetooth cztery ostanie argumenty funkcji mogą pozostać niewykorzystane (patrz listing 3.6).

Prawidłowo wykonana funkcja zwraca wartość `0`, w przeciwnym wypadku - `SOCKET_ERROR`. Kody pojawiających się błędów wykonania można diagnozować za pomocą funkcji `WSAGetLastError()`.

3.2.19. Funkcja connect()

Użycie funkcji `connect()` jest równoważne z wywołaniem `WSAConnect()` z odpowiednimi wartościami parametrów (`s`, `name`, `namelen`).

```
int connect(
  __in  SOCKET s,
  __in  const struct sockaddr *name,
  __in  int namelen
);
```

3.2.20. Funkcja listen()

Wywołanie funkcji `listen()` pozwala na przygotowanie utworzonego już gniazda (identyfikowanego przez deskryptor `s`) do odbierania zgłoszeń oraz umożliwia określenie rozmiaru kolejki oczekujących żądań (połączeń).

```
int listen(
  __in  SOCKET s,
  __in  int backlog
);
```

Argument `backlog` podaje maksymalną długość kolejki oczekujących połączeń. Wartość ta ograniczona jest stałą `SOMAXCONN`. Funkcja `listen()` powinna być wywoływana zaraz po funkcji `bind()`.

3.2.21. Funkcja getsockname()

Funkcja `getsockname()` zwraca nazwę lokalnego gniazda identyfikowanego przez deskryptor `s`.

```
int getsockname(
  __in     SOCKET s,
  __out    struct sockaddr *name,
  __inout  int *namelen
);
```

Wskaźnik `name` wskazuje na strukturę typu `SOCKADDR` zawierającą żądany adres/nazwę gniazda. Parametr `namelen` przechowuje długość wskazywanego adresu/nazwy. W domenie protokołów Bluetooth zamiast struktury `sockaddr`

używa się SOCKADDR_BTH wykonując przy przekazywaniu wskaźnika rzutowanie na wskaźnik do struktury (SOCADDR*).

Listing 3.5. Przykład użycia funkcji getsockname()

```
//-----------------------------------------------------
SOCKADDR_BTH name;
//...
listen(s, 1);
if(SOCKET_ERROR == getsockname(s, (SOCKADDR*)&name,
                               &namelen)) {
  //...
}
printf("nasłuch na porcie RFCOMM: %d\n", name.port);
//-----------------------------------------------------
```

Prawidłowo wykonana funkcja zwraca wartość 0, w przeciwnym wypadku – SOCKET_ERROR. Kody pojawiających się błędów wykonania można diagnozować za pomocą funkcji WSAGetLastError().

3.2.22. Funkcja shutdown()

Funkcja shutdown() służy do zamykania połączenia dając możliwość sterowania przepływem danych (o ile aplikacja używa funkcji blokujących, np. accept() lub connect()).

```
int shutdown(
  __in  SOCKET s,
  __in  int how
);
```

Parametr s jest deskryptorem gniazda. Znacznik how określa jaki rodzaj operacji nie będzie dostępny:
SD_RECEIVE (0) – gniazdo nie może już przyjąć żadnych komunikatów,
SD_SEND (1) – gniazdo nie może już wysyłać żadnych komunikatów,
SD_BOTH (2) – gniazdo nie może przyjmować, ani wysyłać komunikatów.

Prawidłowo wykonana funkcja zwraca wartość 0, w przeciwnym wypadku – SOCKET_ERROR. Kody pojawiających się błędów wykonania można diagnozować za pomocą funkcji WSAGetLastError().

3.2.23. Przykłady

Aby otworzyć kanał transmisyjny, punkt końcowy kanału powinien zostać odpowiednio skonfigurowany. Połączenie następuje, kiedy lokalny obiekt

L2CAP zażąda połączenia z zewnętrznym urządzeniem Bluetooth, lub jeżeli odebrano sygnał, że urządzenie sygnalizuje chęć nawiązania połączenia. Przed rozpoczęciem połączenia kanały transmisyjne powinny być odpowiednio skonfigurowane. Konfiguracja wymaga negocjacji między obiema stronami odpowiednich parametrów połączenia, do czasu, kiedy wszystkie parametry zostaną uzgodnione. Kanał transmisyjny zostanie zamknięty, jeśli jeden obiekt L2CAP wyśle do drugiego żądanie rozłączenia.

Na listingu 3.6 zaprezentowano kod nieskomplikowanego programu, za pomocą którego użytkownik może ustanowić połączenie z zewnętrznym urządzeniem Bluetooth. Przykład ten został skonstruowany w oparciu o kod z listingu 2.3 (patrz Rozdział 2). Po każdorazowym zidentyfikowaniu będącego w zasięgu urządzenia z włączoną funkcją Bluetooth, program sprawdza wartość logiczną znacznika `fConnected` będącego elementem struktury `BLUETOOTH_DEVICE_INFO`. Jeżeli komputer oraz wykryte urządzenie nie były do tej pory połączone, tworzone jest gniazdo połączeniowe, za pomocą funkcji `setsockopt()` ustalane są parametry gniazda oraz wywoływana jest funkcja `WSAConnect()` ustanawiająca połączenie z urządzeniem zlokalizowanym pod określonym adresem zapisanym w polu `btAddr` struktury `SOCKADDR_BTH`.

Listing 3.6. Ustanawianie połączenia z urządzeniem Bluetooth za pomocą funkcji `WSAConnect()`

```
#include <iostream>
#include <initguid>
#pragma option push -a1
  #include <winsock2>
  #include "Ws2bth.h"
  #include "BluetoothAPIs.h"
#pragma option pop

#pragma comment(lib, "ws2_32.lib")
#pragma comment(lib, "Bthprops.lib")

using namespace std;

//---------------------------------------------------
void showError()
{
  LPVOID lpMsgBuf;
  FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
                 FORMAT_MESSAGE_FROM_SYSTEM |
                 FORMAT_MESSAGE_IGNORE_INSERTS,
                 NULL, GetLastError(), 0, (LPTSTR)
                 &lpMsgBuf, 0, NULL );
                 fprintf(stderr, "\n%s\n", lpMsgBuf);
```

```
  free(lpMsgBuf);
  cin.get();
}
//-------------------------------------------------
BLUETOOTH_FIND_RADIO_PARAMS pbtfrp = {
  sizeof(BLUETOOTH_FIND_RADIO_PARAMS)
};

BLUETOOTH_RADIO_INFO pRadioInfo = {
  sizeof(BLUETOOTH_RADIO_INFO),  0,
};

BLUETOOTH_DEVICE_SEARCH_PARAMS pbtsp = {
  sizeof(BLUETOOTH_DEVICE_SEARCH_PARAMS),
  TRUE, TRUE, TRUE, TRUE, TRUE, 10 /*12.28 sek*/, NULL
};

BLUETOOTH_DEVICE_INFO pbtdi = {
  sizeof(BLUETOOTH_DEVICE_INFO), 0,
};

HANDLE phRadio = NULL;
HBLUETOOTH_RADIO_FIND bthRadioFind = NULL;
HBLUETOOTH_DEVICE_FIND hbthDeviceFind = NULL;
WORD wVersionRequested;
WSADATA wsaData;

int main() {

   wVersionRequested = MAKEWORD(2,2);
   if(WSAStartup(wVersionRequested, &wsaData) != 0) {
     showError();
   }

   bthRadioFind = BluetoothFindFirstRadio(&pbtfrp,
                  &phRadio);

   int radioNumber = 0;
   do {
     radioNumber++;
     BluetoothGetRadioInfo(phRadio, &pRadioInfo);
     wprintf(L"Master Radio %d:\n", radioNumber);
     wprintf(L"\tDevice Name: %s\n",
             pRadioInfo.szName);
     wprintf(L"\tAddress: \
```

```
        %02x:%02x:%02x:%02x:%02x:%02x\n",
        pRadioInfo.address.rgBytes[5],
        pRadioInfo.address.rgBytes[4],
        pRadioInfo.address.rgBytes[3],
        pRadioInfo.address.rgBytes[2],
        pRadioInfo.address.rgBytes[1],
        pRadioInfo.address.rgBytes[0]);
wprintf(L"\tSubversion: 0x%08x\n",
        pRadioInfo.lmpSubversion);
wprintf(L"\tClass: 0x%08x\n",
        pRadioInfo.ulClassofDevice);
wprintf(L"\tManufacturer: 0x%04x\n",
        pRadioInfo.manufacturer);

pbtsp.hRadio = phRadio;
memset(&pbtdi, 0, sizeof(BLUETOOTH_DEVICE_INFO));
pbtdi.dwSize = sizeof(BLUETOOTH_DEVICE_INFO);
hbthDeviceFind = BluetoothFindFirstDevice(&pbtsp,
                 &pbtdi);

int deviceNumber = 0;
do {
  deviceNumber++;
  wprintf(L"\tDevice %d:\n", deviceNumber);
  wprintf(L"\t\tName: %s\n", pbtdi.szName);
  wprintf(L"\t\tAddress: \
          %02x:%02x:%02x:%02x:%02x:%02x\n",
          pbtdi.Address.rgBytes[5],
          pbtdi.Address.rgBytes[4],
          pbtdi.Address.rgBytes[3],
          pbtdi.Address.rgBytes[2],
          pbtdi.Address.rgBytes[1],
          pbtdi.Address.rgBytes[0]);
  wprintf(L"\t\tClass: 0x%08x\n",
          pbtdi.ulClassofDevice);
  wprintf(L"\t\tConnected: %s\n", pbtdi.fConnected
          ? L"true" : L"false");
  wprintf(L"\t\tAuthenticated: %s\n",
          pbtdi.fAuthenticated ? L"true" :
          L"false");
  wprintf(L"\t\tRemembered: %s\n",
          pbtdi.fRemembered ? L"true" : L"false");
  //---
  if (pbtdi.fConnected == false) {
     // próba łączenia
```

```
          SOCKADDR_BTH socAddrBTH;
          memset(&socAddrBTH, 0, sizeof(socAddrBTH));
          socAddrBTH.addressFamily = AF_BTH;
          socAddrBTH.btAddr = pbtdi.Address.ullLong;
          socAddrBTH.port = 0;
          socAddrBTH.serviceClassId = L2CAP_PROTOCOL_UUID;
          SOCKET s = WSASocket(AF_BTH, SOCK_STREAM,
                     BTHPROTO_RFCOMM, NULL, NULL, NULL);
          if (s==INVALID_SOCKET) {
            showError();
          }
          else {
             ULONG optval = TRUE;
             if(setsockopt(s, SOL_RFCOMM,
                SO_BTH_AUTHENTICATE,
                (char*)&optval,
                sizeof(ULONG))==SOCKET_ERROR) {
               showError();
             }
             if(WSAConnect(s, (sockaddr*)&socAddrBTH,
                           sizeof(socAddrBTH),
                           NULL, NULL, NULL,
                           NULL)==SOCKET_ERROR) {
               showError();
             }
             else {
               wprintf(L"\t\tConnected: %s\n",
                       pbtdi.fConnected ? L"true" :
                       L"false");
               shutdown(s, SD_BOTH);
               if (SOCKET_ERROR == closesocket(s))
                 showError();
               }
           }
       }
     } while(BluetoothFindNextDevice(hbthDeviceFind,
            &pbtdi));
      BluetoothFindDeviceClose(hbthDeviceFind);
    } while(BluetoothFindNextRadio(&pbtfrp, &phRadio));
   BluetoothFindRadioClose(bthRadioFind);
   WSACleanup();
   system("PAUSE");
   return 0;
}
//---------------------------------------------------
```

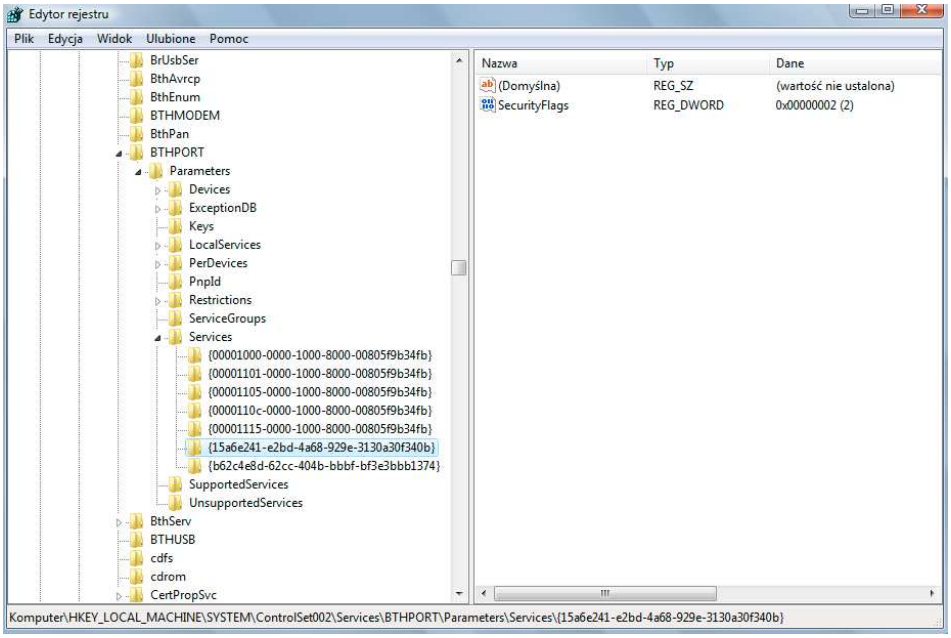
Na listingu 3.7 zaprezentowano przykład użycia sekwencji funkcji `bind()`, `listen()`, `getsockname()` oraz `WSASetService()` w celu zarejestrowania w rejestrze systemowym identyfikatora `GUID` nowo tworzonej usługi. Wynik działania programu należy zweryfikować edytując rejestr systemowy (np. za pomocą programu *regedit*). W podkluczu klucza tematycznego *HKEY_LOCAL_MACHINE\SYSYEM\...BRHPORT\Parameters\Services* powinien wystąpić zadeklarowany identyfikator `GUID` nowej usługi, tak jak pokazano to na rysunku 3.4.

Listing. 3.7. Rejestracja identyfikatora GUID nowo tworzonej usługi

```
#include <iostream>
#include <initguid>
#pragma option push -a1
  #include <winsock2>
  #include "Ws2bth.h"
#pragma option pop

#pragma comment(lib, "ws2_32.lib")

using namespace std;

//['{15A6E241-E2BD-4A68-929E-3130A30F340B}'] //Ctrl+Shift+G
DEFINE_GUID(SAMPLE_UUID, 0x15A6E241, 0xE2BD, 0x4A68,
                         0x92, 0x9E, 0x31, 0x30, 0xA3, 0x0F,
                         0x34, 0x0B);
//---------------------------------------------------
void showError()
{
  LPVOID lpMsgBuf;
  FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
                 FORMAT_MESSAGE_FROM_SYSTEM |
                 FORMAT_MESSAGE_IGNORE_INSERTS,
                 NULL, GetLastError(), 0, (LPTSTR)
                 &lpMsgBuf, 0, NULL );
                 fprintf(stderr, "\n%s\n", lpMsgBuf);
  free(lpMsgBuf);
  cin.get();
}
//---------------------------------------------------
int main() {
   SOCKET s;
   SOCKADDR_BTH socAddrBTH;
   int socAddrBTHlength = sizeof(socAddrBTH);
```

```
WORD wVersionRequested;
WSADATA wsaData;
wVersionRequested = MAKEWORD( 2, 2 );

if( WSAStartup( wVersionRequested, &wsaData ) != 0 )
    showError();

s = socket(AF_BTH, SOCK_STREAM, BTHPROTO_RFCOMM);
if(SOCKET_ERROR == s)
    showError();

socAddrBTH.addressFamily = AF_BTH;
socAddrBTH.btAddr = 0;
socAddrBTH.port = BT_PORT_ANY;
if(SOCKET_ERROR == bind(s, (const sockaddr*)
         &socAddrBTH,sizeof(SOCKADDR_BTH)))
    showError();

listen(s, 1);

if(SOCKET_ERROR == getsockname(s,
  (SOCKADDR*)&socAddrBTH,&socAddrBTHlength))
    showError();

printf("nasłuchiwanie na porcie RFCOMM: %d\n",
       socAddrBTH.port);

CSADDR_INFO CSAddr;
CSAddr.iProtocol = BTHPROTO_RFCOMM;
CSAddr.iSocketType = SOCK_STREAM;
CSAddr.LocalAddr.lpSockaddr=(LPSOCKADDR) &socAddrBTH;
CSAddr.LocalAddr.iSockaddrLength=sizeof(socAddrBTH);
CSAddr.RemoteAddr.lpSockaddr=(LPSOCKADDR) &socAddrBTH;
CSAddr.RemoteAddr.iSockaddrLength=sizeof(socAddrBTH);

WSAQUERYSET qsRestrictions = { 0 };
qsRestrictions.dwSize = sizeof(qsRestrictions);
qsRestrictions.dwNameSpace = NS_BTH;
qsRestrictions.lpszServiceInstanceName =
               "My Bluetooth Service";
qsRestrictions.lpszComment = "Service description...";
qsRestrictions.lpServiceClassId =
               (LPGUID) &SAMPLE_UUID;
qsRestrictions.dwNumberOfCsAddrs = 1;
qsRestrictions.lpcsaBuffer = &CSAddr;
```

```
    if(SOCKET_ERROR == WSASetService(&qsRestrictions,
                        RNRSERVICE_REGISTER,0))
       showError();

    if (SOCKET_ERROR == closesocket(s))
        showError();
    WSACleanup();
    system("PAUSE");
    return 0;
}
//--------------------------------------------------------
```



Rysunek 3.4. Edytor rejestru systemowego. Identyfikator GUID nowo zarejestrowanej usługi

3.3. Podsumowanie

W obecnym rozdziale zostały omówione podstawowe zasoby systemowe związane z biblioteką WinSock, które mogą być pomocne w trakcie konstrukcji programów służących do detekcji i identyfikacji urządzeń Bluetooth. Zaprezentowane zostały przykłady praktycznego wykorzystania omawianych funkcji i struktur. Konstrukcje przykładowych programów starano się przedstawić w sposób na tyle przejrzysty, by Czytelnik nie miał żadnych problemów z samodzielną ich modyfikacją i dostosowaniem do swoich wymagań sprzętowych i programowych. Starano się również zadbać o czytelność kodu, stosując oryginalne nazewnictwo dla zmiennych i funkcji

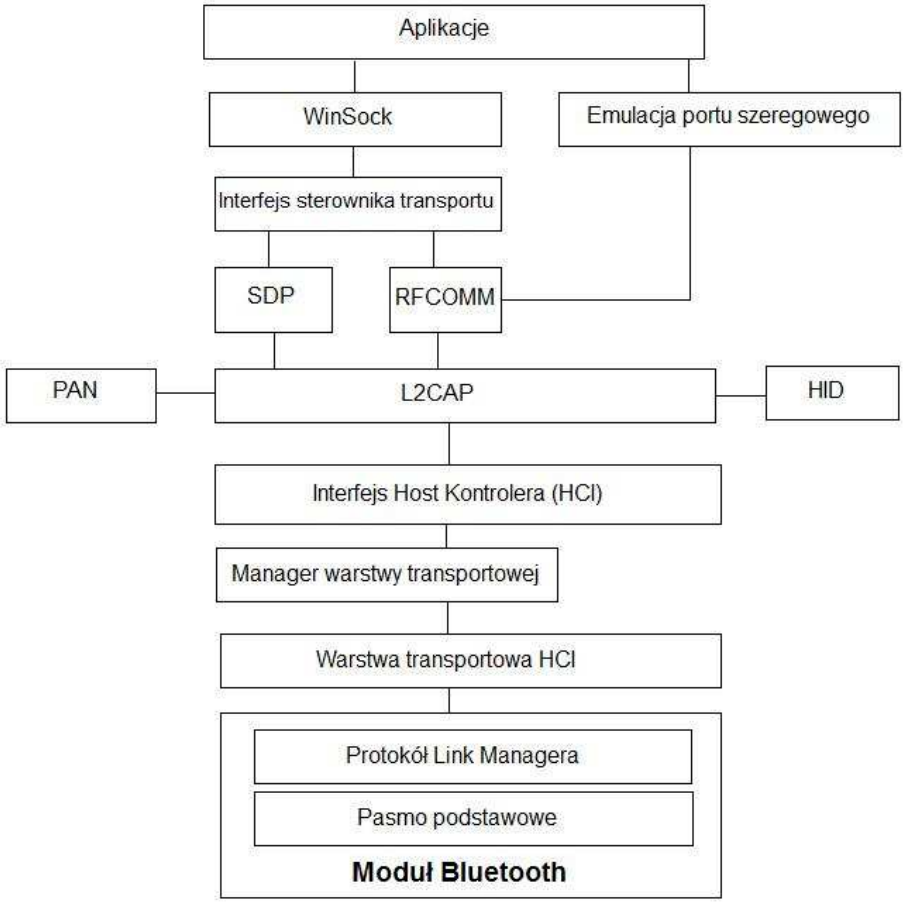
wiernie odzwierciedlające ich rolę w programie. Więcej na temat zasobów biblioteki WinSock można znaleźć w bogatej literaturze przedmiotu [17, 18] oraz w dokumentacji Windows Sockets.

ROZDZIAŁ 4

TRANSMISJA DANYCH

4.1. Aplikacje Bluetooth

Warstwa aplikacji Bluetooth dotyczy oprogramowania, które znajduje się powyżej grupy protokołów pośredniczących i grupy protokołów transportowych. Warstwy tej grupy schematycznie pokazane są na rysunku 4.1 [19].



Rysunek 4.1. Ogólny podział stosu protokołów Bluetooth

Ważną cechą standardu Bluetooth jest to, że oprócz aplikacji dedykowanych na potrzeby technologii, istnieje możliwość korzystania ze starszego oprogramowania konstruowanego pod kątem korzystania z mechanizmów transmisji danych różnych niż Bluetooth. Jest to możliwe dzięki zdefiniowaniu przez SIG takich protokołów jak RFCOMM. W praktyce oznacza to, iż programy pierwotnie zaprojektowane do pracy z wykorzystaniem standardowego portu szeregowego [7] będą mogły po niewielkich modyfikacjach korzystać z Bluetooth, dzięki oferowanym przez protokół RFCOMM mechanizmom emulacji portu szeregowego.

4.2. Uzyskiwanie dostępu do wirtualnego portu szeregowego

Protokół transportowy RFCOMM emuluje standardowy port szeregowy RS-232C [7]. Przed rozpoczęciem korzystania z urządzenia zaopatrzonego w protokół transportowy RFCOMM należy o powyższym fakcie poinformować system operacyjny. Czynność tę określa się jako *otwieranie (lub odblokowywanie) portu do transmisji*. Jednak zanim zaczniemy wykonywać jakiekolwiek czynności, system operacyjny musi zidentyfikować sterownik urządzenia aktualnie przyłączonego do wybranego portu wirtualnego. W przypadku uzyskania dostępu do sterownika urządzenia system operacyjny przekazuje do aplikacji jego identyfikator. We wszystkich operacjach wejścia-wyjścia zamiast szczegółowej ścieżki dostępu do portu wirtualnego urządzenia z funkcją Bluetooth używa się właśnie jego identyfikatora.

4.2.1. Funkcja CreateFile()

Funkcja służy do otwarcia pliku reprezentującego urządzenie (plik sterownika urządzenia). Funkcja zwraca 32 (lub 64)-bitowy identyfikator danego urządzenia przechowywany pod właściwością `HANDLE`, do którego będą adresowane wszystkie komunikaty oraz inne komendy.
Składnia `CreateFile()` wygląda następująco:

```
HANDLE WINAPI CreateFile(IN LPCTSTR lpFileName,
              DWORD dwDesiredAccess,
              IN DWORD dwShareMode,
              IN LPSECURITY_ATTRIBUTES
              lpSecurityAttributes,
              IN DWORD dwCreationDisposition,
              IN DWORD dwFlagsAndAttributes,
              IN HANDLE hTemplateFile);
```

Pierwszy parametr, `lpFileName`, jest wskaźnikiem do zadeklarowanego ciągu znaków zakończonego zerem (zerowym ogranicznikiem), tzw. *null terminated string* (dokładniej do pierwszego znaku tego łańcucha), w którym przechowywana będzie pełna nazwa symboliczna portu komunikacyjnego (COMn) [7]. Parametr `dwDesiredAccess` — specyfikacja rodzaju dostępu do obiektu. Parametr ten może być kombinacją następujących wartości:
`GENERIC_ALL` — przyznanie aplikacji praw do zapisu, odczytu i uruchamiania pliku.
`GENERIC_EXECUTE` — dostęp do uruchamiania pliku.
`GENERIC_READ` — dostęp do odczytu z pliku lub urządzenia.
`GENERIC_WRITE` — dostęp do zapisu do pliku lub urządzenia.
`0` — bez dostępu np., tylko tworzenie nowego pliku.
Parametr `dwShareMode` wyszczególnia, w jaki sposób dany obiekt (lub plik) może być współdzielony:

`0` — obiekt nie może być współdzielony (ang. *exclusive access*).
`FILE_SHARE_DELETE` — współdzielenie z operacjami usuwania.
`FILE_SHARE_READ` — tryb współdzielenia z operacjami czytania.
`FILE_SHARE_WRITE` — tryb współdzielenia z operacjami zapisu.

Opcjonalnie używany parametr `lpSecurityAttributes` jest wskaźnikiem do struktury `SECURITY_ATTRIBUTES` zawierającej deskryptor zabezpieczeń obiektu.

```
typedef struct _SECURITY_ATTRIBUTES {
    DWORD  nLength;
    LPVOID lpSecurityDescriptor;
    BOOL   bInheritHandle;
} SECURITY_ATTRIBUTES;
```

gdzie, `nLength` to rozmiar struktury w bajtach, `lpSecurityDescriptor` jest wskaźnikiem do deskryptora zabezpieczeń obiektu. Jeżeli ustalono `NULL`, zostanie wybrana wartość domyślna. Pole `bInheritHandle` wyszczególnia, czy zwracany przez `CreateFile()` identyfikator jest dziedziczony przy tworzeniu nowego procesu. Wartość `TRUE` oznacza, że nowy proces dziedziczy ten identyfikator.

Parametr wejściowy `dwCreationDistribution` funkcji `CreateFile()` określa rodzaje operacji wykonywanych na pliku i może być reprezentowany przez następujące stałe symboliczne: `CREATE_NEW` — utworzenie nowego pliku. Funkcja nie będzie wykonana pomyślnie, jeżeli plik już istnieje, `CREATE_ALWAYS` — utworzenie nowego pliku niezależnie od tego, czy już istnieje. Jeżeli plik istnieje, nowy zostanie zapisany na istniejącym, `OPEN_EXISTING` — otwarcie istniejącego pliku. Jeżeli plik nie istnieje, funkcja nie będzie wykonana pomyślnie, `OPEN_ALWAYS` — otwarcie istniejącego pliku. Jeżeli takowy nie istnieje, zostanie stworzony identycznie jak za pomocą `CREATE_NEW` oraz `TRUNCATE_EXISTING` — tuż po otwarciu plik jest okrojony do rozmiaru 0 bajtów. Wymagane jest wcześniejsze jego utworzenie przynajmniej z rodzajem dostępu `GENERIC_WRITE`. Funkcja nie będzie wykonana pomyślnie, jeżeli plik nie istnieje.

Parametr wejściowy `dwFlagsAndAttributes` funkcji `CreateFile()` określa atrybuty i znaczniki wykorzystywane podczas wykonywania operacji na plikach. Atrybuty mogą być reprezentowany przez następujące stałe symboliczne: `FILE_ATTRIBUTE_OFFLINE` — dane zawarte w pliku nie są bezpośrednio udostępniane, `FILE_ATTRIBUTE_READONLY` — plik tylko do odczytu, `FILE_ATTRIBUTE_SYSTEM` — plik jest częścią systemu operacyjnego lub jest używany wyłącznie przez system operacyjny, `FILE_ATTRIBUTE_TEMPORARY` — plik jest używany do czasowego przechowywania danych. Powinien być usunięty, jeżeli nie jest wykorzystywany.

Znaczniki reprezentowane są następująco: FILE_FLAG_WRITE_THROUGH — zawartość pliku zostaje zapisana pośrednio poprzez bufor FILE_FLAG_OPEN_NO_RECALL — używane dane powinny pozostać na zdalnym dysku, FILE_FLAG_OPEN_REPARSE_POINT — pozwala na podłączenie do lokalnego, pustego katalogu na partycji NTFS dowolnego katalogu znajdującego się na lokalnym lub zdalnym dysku. Znacznik FILE_FLAG_OVERLAPPED stosowany jest w przypadku asynchronicznych operacji realizowanych przez dłuższy czas przez funkcje ReadFile(), WriteFile(), ConnectNamedPipe() i TransactNamedPipe(). W tym kontekście musi nastąpić odwołanie do struktury OVERLAPPED zawierającej informacje używane w asynchronicznych operacjach wejścia-wyjścia.

Prawidłowo wywołana Funkcja CreateFile() zwraca identyfikator pliku sterownika urządzenia. Jeżeli plik został już otwarty przed wywołaniem funkcji z CREATE_ALWAYS lub OPEN_ALWAYS przypisanymi do dwCreationDistribution, funkcja GetLastError() zwróci wartość ERROR_ALREADY_EXISTS. Jeżeli plik sterownika nie istnieje GetLastError() — zwraca 0. W przypadku, gdy funkcja CreateFile() nie została wykonana pomyślnie, należy oczekiwać wartości INVALID_HANDLE_VALUE.

4.2.2. Funkcja CloseHandle()

Przed zakończeniem działania programu otwarty plik sterownika urządzenia z funkcją Bluetooth należy zamknąć i zwolnić obszar pamięci przydzielony na jego identyfikator, korzystając z funkcji:

```
BOOL WINAPI CloseHandle(IN HANDLE hObject);
```

Parametr wejściowy hObject zwracany jest przez funkcję CreateFile() i w pełni identyfikuje aktualnie używany sterownik urządzenia.

4.2.3. Funkcja ReadFile()

Zasadniczą częścią kodu realizującego cykliczny odczyt danych pochodzących z urządzenia Bluetooth będzie funkcja SDK API:

```
BOOL ReadFile(IN HANDLE  hCommDev,
              OUT LPVOID  lpBuffer,
              IN DWORD  nNumberOfBytesToRead,
              OUT LPDWORD  lpNumberOfBytesRead,
              IN OUT LPOVERLAPPED  lpOverlapped);
```

Użycie jej w programie zapewni odczytanie wszelkich danych przychodzących do urządzenia identyfikowanego przez hCommDev. Parametr lpBuffer jest wskaźnikiem do bufora danych, przez który będziemy odczytywać wszelkie informacje, nNumberOfBytesToRead określa liczbę bajtów do odebrania, zaś

lpNumberOfBytesRead wskazuje na liczbę bajtów rzeczywiście odebranych. Aby nie dopuścić do przekroczenia rozmiaru bufora danych wejściowych, liczba bajtów faktycznie odebranych może być mniejsza niż nNumberOfBytesToRead, dlatego funkcja umieszcza ją w zmiennej lpNumberOfBytesRead, stanowiącej przedostatni parametr. W ten sposób działa mechanizm ochrony dla danych odbieranych. Wskaźnik lpOverlapped wskazuje na strukturę OVERLAPPED i powinien być wykorzystywany w trakcie pisania programów obsługujących transmisję asynchroniczną.

4.2.4. Funkcja WriteFile()

Zasadniczą częścią kodu wysyłającego dane jest zdefiniowana w Windows SDK API funkcja:

```
BOOL WriteFile(IN HANDLE  hCommDev,
               IN LPCVOID  lpBuffer,
               IN DWORD  nNumberOfBytesToWrite,
               OUT LPDWORD  lpNumberOfBytesWritten,
               IN OUT LPOVERLAPPED  lpOverlapped);
```

Powyższa funkcja może zapisywać dane do dowolnego urządzenia (pliku) jednoznacznie wskazanego przez identyfikator hCommDev. Dane wyjściowe umieszczane są buforze danych identyfikowanym przez wskaźnik lpBuffer. Deklaracja LPCVOID lpBuffer odpowiada klasycznej deklaracji wskaźnika ogólnego (adresowego) stałej, czyli: const void *Buffer. Rozmiar bufora ustala się w zależności od potrzeb, zasobów pamięci komputera oraz pojemności bufora danych urządzenia zewnętrznego. Następny parametr nNumberOfBytesToWrite określa liczbę bajtów do wysłania, zaś wskaźnik lpNumberOfBytesWritten wskazuje liczbę bajtów rzeczywiście wysłanych. Aby nie doszło do przekroczenia rozmiaru bufora danych wyjściowych, liczba bajtów faktycznie wysłanych może być mniejsza niż nNumberOfBytesToWrite, dlatego funkcja umieszcza ją w zmiennej lpNumberOfBytesWritten stanowiącej przedostatni parametr. W ten sposób działa mechanizm ochrony dla wysyłanych danych. Ostatni parametr lpOverlapped jest wskaźnikiem struktury OVERLAPPED. Zawiera ona informacje o dodatkowych metodach kontroli transmisji, polegających na sygnalizowaniu aktualnego położenia pozycji wskaźnika transmitowanego pliku. Większość elementów tej struktury jest zarezerwowana przez system operacyjny. Jeżeli jednak chcielibyśmy skorzystać z jej usług, należałoby w funkcji CreateFile() parametrowi dwFlagsAndAttributes przypisać znacznik FILE_FLAG_OVERLAPPED. W trakcie realizacji transmisji synchronicznej wskaźnik lpOverlapped powinien być ignorowany (powinien mieć przypisaną wartość NULL).

4.2.5. Struktura OVERLAPPED

W większości spotykanych sytuacji wirtualny port szeregowy przypisany do wybranego urządzenia z funkcją Bluetooth może być odblokowany w normalnym trybie działania synchronicznego. W celu odblokowania portu wirtualnego dla asynchronicznych operacji odczytu i zapisu danych, powinien zostać utworzony obiekt na bazie struktury kontrolującej asynchroniczne operacje I/O (wejścia – wyjścia):

```
typedef struct _OVERLAPPED {
    DWORD  Internal;
    DWORD  InternalHigh;
    DWORD  Offset;
    DWORD  OffsetHigh;
    HANDLE hEvent;
} OVERLAPPED;
```

Znaczenie poszczególnych pól tej struktury jest następujące:
`Internal` — zarezerwowane dla systemu operacyjnego. Człon ten staje się istotny, gdy funkcja `GetOverlappedResult()` zwróci rezultat wykonanej asynchronicznych operacji I/O w przypadku pliku, potoku lub urządzenia zewnętrznego.
`InternalHigh` — zarezerwowane dla systemu. Specyfikuje długość przesyłanych danych. Staje się istotny, gdy funkcja `GetOverlappedResult()` zwraca wartość `TRUE`.
`Offset` — określa wskaźnik położenia pliku przeznaczonego do transferu. Traktowany jest jako offset wyznaczony w stosunku do początku pliku.
`OffsetHigh` — część bardziej znacząca offsetu.
`hEvent` — określenie sposobu oznaczenia końca transferu danych.

Dla każdego żądania wykonania asynchronicznej operacji przez funkcje: `ReadFile()` i `WriteFile()`, należy używać osobnego obiektu struktury `OVERLAPPED`, przekazywanego do tych funkcji jako ostatni argument (patrz listing 4.1).

W trakcie wykonywania asynchroniczny operacji I/O zdarzają się sytuacje, w których powinno się diagnozować czasy przeterminowania dla odczytu lub(i) zapisu danych. Windows SDK API definiuje funkcję często wykorzystywaną w tym celu:

```
DWORD WaitForSingleObject(IN HANDLE hEvent,
                          IN DWORD dwMilliseconds);
```

W omawianej funkcji `dwMilliseconds` w zależności od kontekstu jej użycia określa w milisekundach czas przeterminowania (ang. *time out*) lub czas oczekiwania na zdarzenie (ang. *break time*). Funkcja zwraca upływający

przedział czasu nawet, jeżeli stan obiektu nie jest w żaden sposób sygnalizowany. Jeżeli parametr ten równy jest zero, funkcja natychmiast testuje stan obiektu. W przypadku, gdy zostanie przypisana mu wartość INFINITE (nieskończoność), stan obiektu nie będzie testowany. Parametr hEvent jest identyfikatorem określonego obiektu zdarzenia. Z reguły należy mu przydzielić odpowiednią wartość, korzystając z funkcji SDK API:

```
HANDLE CreateEvent(IN OUT LPSECURITY_ATTRIBUTES
                   lpEventAttributes,
                   IN BOOL bManualReset,
                   IN BOOL bInitialState,
                   IN OUT LPCTSTR lpName);
```

gdzie, lpEventAttributes jest wskaźnikiem do struktury zabezpieczeń obiektu SECURITY_ATTRIBUTES określającej, czy zwracany identyfikator może być dziedziczony przez procesy potomne. Jeżeli przypiszemy mu wartość NULL, identyfikator taki nie będzie dziedziczony. Parametr bManualReset określa, czy i kiedy występuje automatyczne lub ręczne zwolnienie stworzonego obiekty zdarzenia. Jeżeli przypiszemy mu wartość FALSE, Windows automatycznie zwolni obiekt w przypadku zakończenia danego procesu lub występowania zdarzenia. Parametr lpName jest wskaźnikiem do łańcucha liczącego co najwyżej MAX_PATH znaków i zakończonego zerowym ogranicznikiem specyfikującego konkretną nazwę obiektu zdarzenia.

Funkcja WaitForSingleObject() w przypadku niepomyślnego wykonania zwraca wartość WAIT_FAILED. Jeżeli zostanie wykonana pomyślnie, należy spodziewać się następujących wartości:

WAIT_ABANDONED — wyspecyfikowany jest obiekt wzajemnego wykluczania (ang. *mutex*), tj. sekcji krytycznej współdzielonej przez wiele procesów, który nie został zwolniony przez odnośny wątek.

WAIT_OBJECT_0 — aktualny stan wyspecyfikowanego obiektu jest sygnalizowany.

WAIT_TIMEOUT — czas przeterminowania wybranej operacji upłynął i aktualny stan obiektu nie będzie sygnalizowany.

Jeżeli w funkcji CreateEvent() parametrowi bManualReset zostanie przypisana wartość TRUE, należy skorzystać w funkcji Windows SDK API:

```
BOOL ResetEvent(IN HANDLE hEvent);
```

zwalniającej możliwość sygnalizowania stanu obiektu zdarzenia.

4.3. Transmisja asynchroniczna

Funkcje WriteFile() oraz ReadFile() mogą być wykorzystywane w trakcie realizacji zarówno transmisji asynchronicznej jak i synchronicznej.

Transmisja asynchroniczna jest traktowana jako szczególna metoda przesyłania danych. Z tego względu Windows API udostępnia grupę funkcji przeznaczonych do obsługi tylko i wyłączne asynchronicznego trybu przesyłania informacji. Do grupy tej zaliczamy tzw. funkcje rozszerzone: `WriteFilex()`oraz `ReadFileEx()`. Warto zwrócić uwagę na fakt, iż w odróżnieniu od `WriteFile()` i `ReadFile()` funkcje `WriteFileEx()` oraz `ReadFileEx()` nie posługują się jawnie zaimplementowanym mechanizmem ochrony danych wysyłanych i odbieranych. Zamiast tego wykorzystują wskaźnik do funkcji:

```
VOID CALLBACK
FileIOCompletionRoutine(IN DWORD dwErrorCode,
                  IN DWORD dwNumberOfBytesTransfered,
                  IN LPOVERLAPPED lpOverlapped);
```

gdzie, `dwNumberOfBytesTransfered` jest liczbą transferowanych bajtów (w przypadku wystąpienia błędów w transmisji parametr ten równy jest zero), `dwErrorCode` reprezentuje status wykonania operacji wejścia/wyjścia i może przyjąć następujące wartości:
`0` — operacja wejścia/wyjścia została wykonana prawidłowo.
`ERROR_HANDLE_EOF` – funkcja `ReadFileEx()` próbuje odczytać dane zawarte poza znacznikiem końca pliku (`EOF`).

Na listingu 4.1 zaprezentowano jeden z możliwych sposobów wykorzystania omawianych funkcji. Funkcje te w programie głównym używane są pośrednio poprzez wywołanie `readBluetoothDataEx()`.

Listing 4.1. Fragment kodu programu asynchronicznie pobierającego dane wejściowe

```
//...
BYTE bufferIn[256]; //przykładowy bufor danych
wejściowych
DWORD status = ERROR_SUCCESS;
//-----------------------------------------------------
void CALLBACK FileIOCompletionRoutine(const DWORD
     errorCode,const DWORD numberOfBytesTransfered,
     OVERLAPPED* overlapped)
{
   status = errorCode;

   if(ERROR_SUCCESS == status) {
     //...
     // do bufora są pobierane z kanału
     // transmisyjnego asynchronicznie dane wejściowe
      if (!ReadFileEx(hCommDev, &bufferIn,
```

```
                        sizeof(bufferIn), overlapped,
                        FileIOCompletionRoutine))
            status = GetLastError();
      }
   }
   //-----------------------------------------------------
   DWORD readBluetoothDataEx(HANDLE, void*)
   {
      OVERLAPPED *overlapped = NULL;
      if(overlapped == NULL){
          overlapped = new OVERLAPPED;
          overlapped->Offset = 0;
          overlapped->OffsetHigh = 0;
      }
      if(!ReadFileEx(hCommDev, bufferIn,
                     sizeof(bufferIn), overlapped,
                     FileIOCompletionRoutine)) {
         status = GetLastError();
      }
      else {
           //...
          }
      }
      delete overlapped;
      return status;
   }
   //-----------------------------------------------------
   int main()
   {
      //...
      readBluetoothDataEx(hCommDev, bufferIn);
      //...
      CloseHandle(hCommDev);
      system("PAUSE");
      return 0;
   }
   //-----------------------------------------------------
```

Więcej na ten temat funkcji `xxxEx()` można przeczytać w dokumentacji SDK Windows. Liczne przykłady praktycznego wykorzystania wymienionych zasobów systemu w trakcie konstruowania programów komunikacyjnych zamieszczone są także w pozycji [10]. Dostosowanie algorytmów prezentowanych w pracy [10] na potrzeby realizacji bezprzewodowej transmisji danych wymaga jedynie niewielkich zmian uwzględniających specyfikę standardu Bluetooth. Praktyczne wykorzystanie funkcji rozszerzonych w

aplikacjach Bluetooth polecamy Czytelnikom jako niezwykle pożyteczne ćwiczenia do samodzielnego wykonania.

4.4. WinSock

Biblioteka WinSock udostępnia cztery podstawowe funkcje służące do wymiany danych pomiędzy urządzeniami za pośrednictwem gniazd. Należy pamiętać, iż po skończonej transmisji danych bezwzględnie należy poinformować o tym fakcie urządzenie zewnętrzne poprzez wywołanie funkcji `shutdown()` oraz `closesocket()` (patrz Rozdział 3).

4.4.1. Funkcja send()

Funkcja wysyła dane wskazywane przez `buf` do gniazda określonego deskryptorem `s`.

```
int send(
  __in  SOCKET s,
  __in  const char *buf,
  __in  int len,
  __in  int flags
);
```

Parametr `len` określa rozmiar (w bajtach) danych przeznaczonych do wysłania, zmienna `flags` jest znacznikiem opisującym opcje wysyłania danych (standardowo 0). Wartością zwracaną przez funkcję jest liczba wysłanych bajtów. W przypadku błędnego wykonania funkcja zwraca wartość `SOCKET_ERROR`. Szczegółowe kody błędów mogą być diagnozowane za pomocą funkcji `WSAGetLastError()`. Poniżej pokazano nieskomplikowany przykład praktycznego wykorzystania sekwencji funkcji `send()`, `closesocket()`, `shutdown()` oraz `WSACleanup()`.

Listing 4.2. Fragment kodu programu z funkcją wysyłającą dane

```
//-----------------------------------------------------
result = send(s, bufferOut, strlen(bufferOut), 0);
if (result == SOCKET_ERROR) {
    wprintf(L"błąd wysłania danych: %d\n",
            WSAGetLastError());
    closesocket(s);
    WSACleanup();
    return 1;
}
printf("wysłano bajtów: %d\n", result);
//...
```

```
result = shutdown(s, SD_SEND);
if (result == SOCKET_ERROR) {
   wprintf(L"błąd wykonania shutdown: %d\n",
           WSAGetLastError());
   closesocket(s);
   WSACleanup();
   return 1;
}
//-----------------------------------------------------
```

4.4.2. Funkcja sendto()

Funkcja wysyła dane wskazywane przez `buf` do gniazda określonego deskryptorem `s` i przechowuje informacje o strukturze opisującej odbiorcę.

```
int sendto(
  __in  SOCKET s,
  __in  const char *buf,
  __in  int len,
  __in  int flags,
  __in  const struct sockaddr *to,
  __in  int tolen
);
```

Wskaźnik `buf` wskazuje na bufor z danymi odebranymi, parametr `len` określa rozmiar bufora (w bajtach), `flags` jest znacznikiem określającym sposób odbierania danych, wskaźnik `to` wskazuje na strukturę z adresem gniazda do którego dane są wysyłane, `tolen` jest rozmiarem (w bajtach) tego wskaźnika.

4.4.3. Funkcja recv()

Funkcja odbiera dane przychodzące z gniazda określonego deskryptorem `s`. Dane odebrane umieszczane są w buforze wskazywanym przez `buf`.

```
int recv(
  __in   SOCKET s,
  __out  char *buf,
  __in   int len,
  __in   int flags
);
```

Parametr `len` określa rozmiar (w bajtach) danych przeznaczonych do wysłania, zmienna `flags` jest znacznikiem opisującym opcje wysyłania danych (standardowo 0). Wartością zwracaną przez funkcję jest liczba wysłanych bajtów. W przypadku błędnego wykonania funkcja zwraca wartość

SOCKET_ERROR. Szczegółowe kody błędów mogą być diagnozowane za pomocą funkcji WSAGetLastError().

Listing 4.3. Fragment kodu programu z funkcją odbierającą dane

```
//-------------------------------------------------
char bufferIn[256];
if (recv(s, bufferIn, sizeof(bufferIn), 0)) ==
    SOCKET_ERROR)
{
  // komunikat o błędzie
}
//-------------------------------------------------
```

4.4.4. Funkcja recvfrom()

Funkcja odbiera pakiet danych przychodzący z gniazda określonego deskryptorem s i przechwytuje informacje o nadawcy. Dane odebrane umieszczane są w buforze wskazywanym przez buf.

```
int recvfrom(
  __in         SOCKET s,
  __out        char *buf,
  __in         int len,
  __in         int flags,
  __out        struct sockaddr *from,
  __inout_opt  int *fromlen
);
```

Wskaźnik buf wskazuje na bufor z danymi odebranymi, parametr len określa rozmiar bufora (w bajtach), flags jest znacznikiem określającym sposób odbierania danych, wskaźnik from wskazuje na strukturę z adresem gniazda od którego dane są odbierane, fromlen jest rozmiarem (w bajtach) tego wskaźnika.

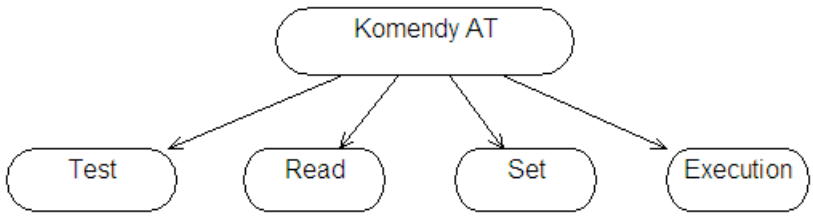
Listing 4.4. Przykładowe użycie funkcji recvfrom()

```
//-------------------------------------------------
WSADATA wsaData;
SOCKET s;
sockaddr_in recvAddr;
//...
char bufferIn[256];
sockaddr_in senderAddr;
int senderAddrSize = sizeof(senderAddr);
//...
```

```
result = bind(s, (SOCKADDR *)& recvAddr,
              sizeof(recvAddr));
if (result != 0) {
   wprintf(L"błędne wykonanie bind %d\n",
           WSAGetLastError());
   return 1;
}
//...
wprintf(L"dane odbierane...\n");
result = recvfrom(s, bufferIn, strlen(bufferIn), 0,
                  (SOCKADDR *) & senderAddr,
                  &senderAddrSize);
if (result == SOCKET_ERROR) {
    wprintf(L"błędne wykonanie recvfrom %d\n",
            WSAGetLastError());
}
//-------------------------------------------------
```

4.5. Komendy AT

Komendy AT to zestaw poleceń, które po raz pierwszy zastosowała w swoich urządzeniach (w celu ujednolicenia obsługi sprzętu, z którym miał współpracować komputer) znana z produkcji modemów firma Hayes. Pierwotnie polecenia te miały służyć jedynie do sterowania pracą modemów analogowych. Jednak wraz z upowszechnieniem się technologii GSM bardzo szybko zostały zaadoptowane do obsługi modemów wbudowanych w telefony komórkowe. Współcześnie każde urządzenie bazujące na technologii GSM posiada wbudowany interpreter komend AT i wykonuje je zgodnie z normą przyjętą przez producentów. Oznacza to, iż implementacje komend AT dla konkretnych urządzeń mogą nieznacznie różnić się pomiędzy sobą, co nie zmienia faktu, iż zarówno składnia komend oraz wynik ich realizacji są znormalizowane [20]. Na rysunku 4.2 pokazano ogólną klasyfikacje komend AT.



Rysunek 4.2. Klasyfikacja poleceń AT

Tak jak pokazano to na rysunku 4.2 komendy AT dzielą się na cztery podstawowe grupy:

- Polecenia typu *Test* (testowe) – służą do sprawdzania, czy dana komenda jest obsługiwana przez urządzenie, czy też nie.
  składnia: `AT<polecenie>=?`

- Polecenia typu *Read* (zapytania) – służą do uzyskiwania informacji na temat aktualnych ustawień urządzenia zewnętrznego.
  składnia: `AT<polecenie>?`

- Polecenia typu *Set* (zestawy poleceń) – służą do modyfikowania wybranych parametrów ustawień urządzenia zewnętrznego.
  składnia: `AT<polecenie>=wartość1, wartość2, …, wartośćN`

- Polecenia typu *Execution* (wykonywalne) – służą do przesyłania rozkazów wykonania konkretnej operacji przez urządzenie zewnętrzne.
  składnia: `AT<polecenie>=parametr1, parametr2, …, parameteN`

Zgodnie ze standardem, każde polecenie rozpoczyna się od prefiksu AT i kończy znakiem powrotu karetki CR (13 lub \r). Komenda nie będzie realizowana dopóty, dopóki urządzenie GSM nie odbierze znaku CR. Przyjęcie komendy do realizacji przez urządzenie potwierdzane jest znakiem nowej linii LF (10 lub \n). Więcej informacji na temat komend AT można znaleźć w publikacji J. Bogusza [20] oraz na stronie internetowej [21].

Na listingu 4.5 zaprezentowano przykładowy program kontrolujący w sposób asynchroniczny operacje wysyłania za pośrednictwem wirtualnego portu szeregowego poleceń ATI (typ urządzenia) oraz AT+CCLK? (zapytanie o aktualna datę i czas), a następnie pobierania informacji zwrotnych od urządzenia z funkcją Bluetooth. Należy zwrócić uwagę, iż do poprawnego działania pokazanego algorytmu transmisji danych wymagane jest, aby urządzenie podrzędne było wcześniej poprawnie zestawione i uwierzytelnione (patrz Rozdział 2). Transmisja danych programowana jest za pomocą funkcji API SDK Windows [7].

Listing 4.5. Przykład asynchronicznej transmisji danych poprzez wirtualny port szeregowy pomiędzy głównym modułem radiowym urządzenia nadrzędnego (komputer) a pozostającym w zasięgu urządzeniem podrzędnym

```
#include <iostream>
#include <windows>
#pragma hdrstop

#define cbInQueue 1024
#define cbOutQueue 1024

  using namespace std;
```

```
  void* hCommDev;
  DCB   dcb;
  COMMTIMEOUTS commTimeouts;

  //-prototypy funkcji--------
  void closeSerialPort();
  int readSerialPort(void *buffer, unsigned long
                     numberOfBytesToRead);
  int writeSerialPort(void *buffer, unsigned long
                      numberOfBytesToWrite);
  bool openSerialPort(const char* portName);
  bool setCommTimeouts(unsigned long
       ReadIntervalTimeout, unsigned long
       ReadTotalTimeoutMultiplier,unsigned long
       ReadTotalTimeoutConstant,unsigned long
       WriteTotalTimeoutMultiplier,unsigned long
       WriteTotalTimeoutConstant);
  bool setTransmissionParameters(unsigned long
       BaudRate, int ByteSize, unsigned long
       fParity,  int  Parity, int StopBits);
//---------------------------------------------
int main()
{
  openSerialPort("COM9");
  setTransmissionParameters(CBR_9600, 8, true,
                            ODDPARITY, ONESTOPBIT);
  setCommTimeouts(0xFFFFFFFF, 10, 0, 10, 0);
  char bufferIn[24];
  char bufferOut[64] = {0};

  char *text;
  //text = "ATI\r";  //przykładowe komendy AT
  text = "AT+CCLK?\r";
  strcpy(bufferIn, text);
  writeSerialPort(bufferIn, strlen(bufferIn));

  cout << "Otrzymano bajtow: " << \
  readSerialPort(bufferOut, sizeof(bufferOut)) \
  << endl;
  cout << bufferOut;

  closeSerialPort();
  system("PAUSE");
  return 0;
```

```
}
//----ciała funkcji---------------------------
bool openSerialPort(const char* portName)
{
  hCommDev = CreateFile(portName,GENERIC_READ |
             GENERIC_WRITE, 0,
             NULL,OPEN_EXISTING,
             FILE_ATTRIBUTE_NORMAL |
             FILE_FLAG_OVERLAPPED, NULL);
  if(hCommDev==INVALID_HANDLE_VALUE){
     cout <<"Błąd otwarcia portu " <<portName << "\
            " lub port jest aktywny.\n";
    return false;
  }
    else SetupComm(hCommDev, cbOutQueue,
                   cbOutQueue);
      return true;
}
//-----------------------------------------------
bool setTransmissionParameters(unsigned long
     BaudRate, int ByteSize, unsigned long
     fParity, int Parity, int StopBits)
{
  dcb.DCBlength = sizeof(dcb);
  GetCommState(hCommDev, &dcb);
  dcb.BaudRate =BaudRate;
  dcb.ByteSize = ByteSize;
  dcb.Parity =Parity ;
  dcb.StopBits =StopBits;
  dcb.fBinary=true;
  dcb.fParity=fParity;
  //...
  if(SetCommState(hCommDev, &dcb)==0){
     cout << "Błąd wykonania funkcji "\
              " SetCommState()\n";
     CloseHandle(hCommDev);
     return false;
  }
  return true;
}
//-----------------------------------------------
bool setCommTimeouts(unsigned long
     ReadIntervalTimeout, unsigned long
     ReadTotalTimeoutMultiplier, unsigned long
     ReadTotalTimeoutConstant, unsigned long
```

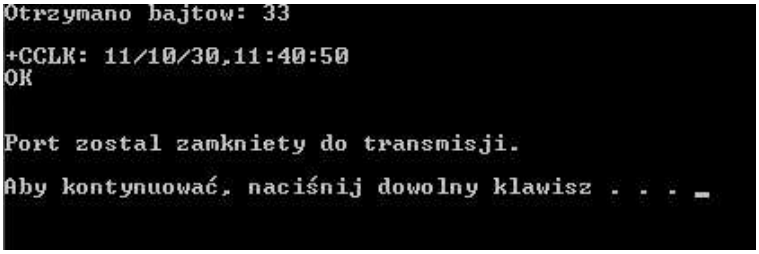
```
     WriteTotalTimeoutMultiplier, unsigned long
     WriteTotalTimeoutConstant)
{
  if(GetCommTimeouts(hCommDev, &commTimeouts)==0)
     return false;
  commTimeouts.ReadIntervalTimeout=
               ReadIntervalTimeout;
  commTimeouts.ReadTotalTimeoutConstant =
               ReadTotalTimeoutConstant;
  commTimeouts.ReadTotalTimeoutMultiplier =
               ReadTotalTimeoutMultiplier;
  commTimeouts.WriteTotalTimeoutConstant =
               WriteTotalTimeoutConstant;
  commTimeouts.WriteTotalTimeoutMultiplier =
               WriteTotalTimeoutMultiplier;

  if (SetCommTimeouts(hCommDev, &commTimeouts)==
      0){
     cout << "Błąd wykonania funkcji" \
              " SetCommTimeouts()\n";
     CloseHandle(hCommDev);
     return false;
  }
  return true;
}
//-----------------------------------------------
int writeSerialPort(void *buffer,
                    unsigned long
                    numberOfBytesToWrite)
{
   BOOL           result;
   unsigned long  numberOfBytesWritten = 0;
   unsigned long  errors;
   unsigned long  lastError;
   unsigned long  bytesSent = 0;
   COMSTAT        comStat;
   OVERLAPPED     overlapped;

   result = WriteFile(hCommDev, buffer,
            numberOfBytesToWrite,
            &numberOfBytesWritten, &overlapped);
   if (!result) {
      if(GetLastError() == ERROR_IO_PENDING) {
         while(!GetOverlappedResult(hCommDev,
               &overlapped,
```

```
                &numberOfBytesWritten, FALSE )) {
             lastError = GetLastError();
             if(lastError == ERROR_IO_INCOMPLETE){
                numberOfBytesWritten += bytesSent;
                continue;
             }
             else {
               ClearCommError(hCommDev, &errors,
                              &comStat) ;
               break;
             }
          }
          numberOfBytesWritten += bytesSent;
       }
       else {
          ClearCommError(hCommDev, &errors,
                         &comStat) ;
       }
    }
    else
       numberOfBytesWritten += bytesSent;
  FlushFileBuffers(hCommDev);
  return numberOfBytesWritten;
}
//-----------------------------------------------
int readSerialPort(void *buffer, unsigned long
numberOfBytesToRead)
{
   BOOL          result;
   COMSTAT       comStat ;
   unsigned long errors;
   unsigned long bytesRead = 0;
   unsigned long numberOfBytesRead = 0;
   unsigned long lastError;
   OVERLAPPED overlapped;

   ClearCommError(hCommDev, &errors, &comStat ) ;
   bytesRead =  numberOfBytesToRead;
   if(bytesRead > 0) {
     result = ReadFile(hCommDev, buffer,
                       bytesRead, &bytesRead,
                       &overlapped) ;
     if(!result) {
       if(GetLastError() == ERROR_IO_PENDING) {
          while(!GetOverlappedResult(hCommDev,
```

```
                &overlapped,&bytesRead, TRUE)) {
              lastError = GetLastError();
              if(lastError == ERROR_IO_INCOMPLETE)
              {
                 numberOfBytesRead += bytesRead;
                 continue;
              }
              else {
                 ClearCommError(hCommDev, &errors,
                                &comStat);
              }
               break;
           }
           numberOfBytesRead += bytesRead;
        }
      }
      else numberOfBytesRead += bytesRead;
    }
     else
        ClearCommError(hCommDev, &errors,
                       &comStat);

    return numberOfBytesRead;
}
//---------------------------------------------
void closeSerialPort()
{
  if (CloseHandle(hCommDev))
    cout << "\n\nPort został zamknięty do" \
            " transmisji.\n\n";
  return;
}
//---------------------------------------------
```

```
Otrzymano bajtow: 25

Samsung SGH-X680
OK


Port zostal zamkniety do transmisji.

Aby kontynuować, naciśnij dowolny klawisz . . .
```

Rysunek 4.3. Odpowiedź urządzenia zewnętrznego na odebraną komendę ATI



Rysunek 4.4. Odpowiedź urządzenia zewnętrznego na odebraną komendę AT+CCLK?

Na listingu 4.6 zaprezentowano przykładowy program wysyłający do telefonu GSM zestawionego z komputerem rozkaz `ATD <numer>` dzwonienia pod wybrany numer. Transmisja danych programowana jest za pomocą funkcji z biblioteki WinSock.

Listing 4.6. Rozkaz dzwonienia ATD pod wybrany numer

```
#include <iostream>
#include <initguid>
#pragma option push -a1
  #include <winsock2>
  #include "Ws2bth.h"
  #include "BluetoothAPIs.h"
#pragma option pop
using namespace std;
//---------------------------------------------------
void showError()
{
  LPVOID lpMsgBuf;
  FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
                 FORMAT_MESSAGE_FROM_SYSTEM |
                 FORMAT_MESSAGE_IGNORE_INSERTS,
                 NULL, GetLastError(), 0,
                 (LPTSTR) &lpMsgBuf, 0, NULL );
                 fprintf(stderr, "\n%s\n", lpMsgBuf);
  free(lpMsgBuf);
  cin.get();
}
//---------------------------------------------------
int main() {

  WORD wVersionRequested;
  WSADATA wsaData;
  int result;

  wVersionRequested = MAKEWORD(2,2);
```

```
if(WSAStartup(wVersionRequested, &wsaData) != 0) {
   showError();
}

SOCKET s;
SOCKADDR_BTH socAddrBTH;
int socAddrBTHlength = sizeof(socAddrBTH);

memset(&socAddrBTH, 0, sizeof(socAddrBTH));
socAddrBTH.addressFamily = AF_BTH;
socAddrBTH.btAddr = (BTH_ADDR)0x001a8a9120a1;
socAddrBTH.port = BT_PORT_ANY;
socAddrBTH.serviceClassId =
           SerialPortServiceClass_UUID;
s = socket(AF_BTH, SOCK_STREAM, BTHPROTO_RFCOMM );
if(s == SOCKET_ERROR) {
 wprintf(L"błędne wykonanie socket %d\n",
         WSAGetLastError());
 //return 1;
}
if(SOCKET_ERROR==connect(s, (SOCKADDR*) &socAddrBTH,
                         socAddrBTHlength)) {
   wprintf(L"błędne wykonanie connect %d\n",\
           WSAGetLastError());
   return 1;
}

char komenda[] = "ATD 123456789;\r";
//komenda ATD <numer>[;]
//dzwoni pod wybrany numer
//[;]oznacza połączenie głosowe

//send(s , komenda , sizeof(komenda), 0);

result = send(s, komenda , strlen(komenda), 0);
if (result == SOCKET_ERROR) {
  wprintf(L"błąd wysłania danych: %d\n",
          WSAGetLastError());
  closesocket(s);
  WSACleanup();
  return 1;
}
printf("wysłano bajtów: %d\n", result);

Sleep(10000); // próba łączenia przez ok. 10 sek.
```

```
    result = shutdown(s, SD_SEND);
    if (result == SOCKET_ERROR) {
      wprintf(L"błąd wykonania shutdown: %d\n",
              WSAGetLastError());
      closesocket(s);
      WSACleanup();
      return 1;
    }
     WSACleanup();
     system("PAUSE");
     return 0;
}
//-------------------------------------------------
```

4.6. Podsumowanie

Obecny rozdział należy traktować jako uzupełnienie dwóch poprzednich. Zawarto w nim opis praktycznych metod wykorzystywania w działających programach zasobów systemowych odpowiedzialnych za realizację transmisji danych w standardzie Bluetooth. Omawiane kody zostały przedstawione w formach sekwencyjnych i proceduralnych w ten sposób, aby Czytelnicy nie zaznajomieni z zasadami programowania zorientowanego obiektowo mogli bez trudu wykorzystać je dla własnych potrzeb. Przedstawione algorytmy są również podatne na wszelkiego rodzaju modyfikacje i uzupełnienia w zależności od własnych potrzeb i aktualnych wymagań. Więcej na temat protokołów RFCOMM oraz L2CAP można znaleźć w literaturze przedmiotu oraz na stronach: http://www.palowireless.com/infotooth/tutorial/rfcomm.asp#Multiple Emulated Serial Ports. Również na stronie firmy Microsoft http://msdn.microsoft.com/enus/library/windows/desktop/ms740149(v=vs.85).aspx znajdują się liczne przykłady praktycznego wykorzystania w działających programach funkcji z biblioteki WinSock.

DODATEK A

PROGRAMY WIELOWĄTKOWE

Wątek (ang. *thread*) definiowany jest jako odrębny przebieg aplikacji. Każda aplikacja pisana dla Windows (a także Linuksa) może zawierać wiele wątków, każdy z własnym stosem, własnym identyfikatorem oraz kopią rejestrów procesora. W komputerach wieloprocesorowych poszczególne procesory są w stanie wykonywać odnośny wątek w sposób niezależny. W komputerach jednoprocesorowych otrzymujemy wrażenie jednoczesnego (współbieżnego) wykonywania wielu wątków, chociaż w rzeczywistości w danym przedziale czasu procesor jest w stanie wykonać tylko jeden wątek.

Proces definiowany jest jako wykonujący się program w postaci kolekcji wielu wątków, pracujących we wspólnej przestrzeni adresowej procesora. Każdy proces musi zawierać przynajmniej jeden wątek główny (ang. *main thread*). Wątki należące do tego samego procesu mogą współdzielić różne zasoby aplikacji (lub systemu), takie jak otwarte pliki lub uruchomione inne aplikacje, oraz odwoływać się do prawidłowo wybranego adresu pamięci w przestrzeni adresowej procesora.

W pierwszym przybliżeniu współbieżność odrębnych procesów może być realizowana na jednym z trzech poziomów:

- sprzętowym (komputer posiadający architekturę wieloprocesorową),
- systemowym,
- aplikacji (podział czasu procesora pomiędzy różne elementy tej samej aplikacji).

W dalszej części niniejszego uzupełnienia zostaną omówione niektóre aspekty współbieżności realizowanej na poziomie systemu operacyjnego oraz aplikacji.

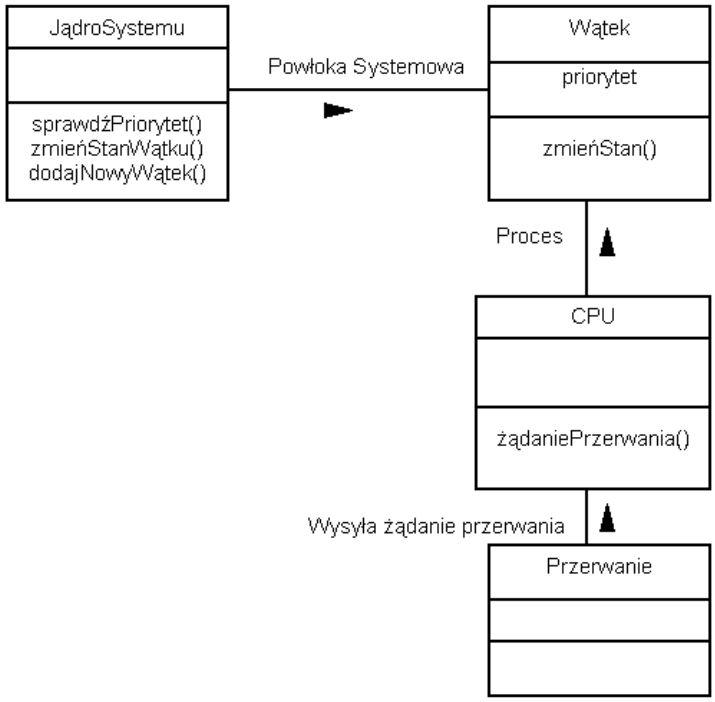
System operacyjny zarządza wątkami na podstawie ich priorytetu. Wątki o nadanym wyższym priorytecie mają pierwszeństwo w wykonaniu przed wątkami o priorytecie niższym. Na poziomie tego samego priorytetu wątki zarządzane są w taki sposób, aby każdy z nich był w stanie się wykonać. System może wstrzymać wykonanie wątku (sytuację taką nazywa się wywłaszczeniem), aby przekazać czas procesora na rzecz innego wątku. W systemie operacyjnym Windows definiowane są trzy podstawowe kategorie określające stan wątku:

- wątek wykonujący się,
- wątek gotowy,
- wątek blokowany.

Każdy wątek jest w stanie się wykonać, pod warunkiem, że w danej chwili posiada dostęp do rejestrów procesora. System operacyjny współdzieli tyle wykonujących się wątków, ile ma procesorów — po jednym wątku na procesor. Wątek pozostaje w stanie wykonania do momentu, kiedy wstrzyma się w oczekiwaniu na jakąś konkretną operację. Wtedy zostaje wywłaszczony przez system operacyjny, aby umożliwić wykonanie innemu wątkowi lub sam zawiesza swoje wykonanie. Wątek jest gotowy do wykonania, jeżeli się nie wykonuje i nie jest blokowany. Wątek gotowy może wywłaszczyć wątek wykonujący się o tym samym priorytecie, ale nie wątek o priorytecie wyższym. Wątek jest blokowany, jeżeli oczekuje na wykonanie konkretnej operacji. Zawsze można jawnie zablokować wątek przez jego zawieszenie (ang. *suspend*).

Zawieszony wątek będzie oczekiwał w nieskończoność (ang. *infinite*), do momentu jego wznowienia (ang. *resume*).

Jądro systemu Windows działa w trybie z wywłaszczaniem. Korzystając z zasady reifikacji danych, na rysunku A.1 pokazano uproszczony diagram klas dla systemu posiadającego jądro działające w trybie z wywłaszczeniem [22].



Rysunek A.1. Uproszczony schemat dla systemu operacyjnego działającego w trybie z wywłaszczeniem

Projektując aplikację zawierającą elementy wielowątkowości, programista powinien szukać odpowiedzi na pytanie: kiedy wykonywanie określonego wątku powinno być zawieszone lub wznowione? Należy zadbać również o to, aby wątki wykonujące się w programie jak najmniej czasu spędzały w stanie zawieszenia (zablokowania) i jak najwięcej w stanie wykonywania się.

Podstawową funkcją Windows API, tworzącą nowy watek, jest:

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES
                    lpThreadAttributes,
                    DWORD dwStackSize,
                    LPTHREAD_START_ROUTINE
                    lpStartAddress,
                    LPVOID lpParameter,
```

```
                    DWORD dwCreationFlags,
                    LPDWORD lpThreadId);
```

Na listingu A.1 pokazano kod głównego modułu projektu tworzącego dwa wątki: jeden do wysyłania, drugi do odbioru danych.

Listing A.1. Przykład wykorzystania funkcji `CreateThread()`

```
#include <assert>
#include <iostream>
//...

using namespace std;

   //Deklaracje zmiennych globalnych

unsigned long threadFuncSend(void* parameter);
unsigned long threadFuncReceive(void* parameter);
//-----------------------------------------------------
int main()
{
   bufferIn=(char*)HeapAlloc(GetProcessHeap(),
                             HEAP_ZERO_MEMORY,
                             strlen(bufferOut)+1);

   //Konfiguracja portu wirtualnego lub gniazda
   //Patrz np. listingi 4.1, 4.2

   hEvent=CreateEvent(NULL, TRUE, FALSE,
                      "FILE_EXISTS");
   assert(hEvent);

   // tworzymy dwa wątki
   //1. do wysyłania danych
   hThread[0] = CreateThread(NULL, 0,
        (LPTHREAD_START_ROUTINE)threadFuncSend,
         NULL, 0, &threadID1);
   //2. do czytania danych
   hThread[1] = CreateThread(NULL, 0,
        (LPTHREAD_START_ROUTINE)threadFuncReceive,
         NULL, 0, &threadID2);

   // sygnalizacja wątkom tego, że dane są gotowe
   if(SetEvent(hEvent))
     WaitForMultipleObjects(2, hThread, TRUE, 100);
```

```
   CloseHandle(hEvent);

   CloseHandle(hThread[0]);
   CloseHandle(hThread[1]);
   HeapFree(GetProcessHeap(),0,bufferIn);
   cin.get();
   return 0;
}
//-----------------------------------------------------
unsigned long threadFuncSend(void* parameter)
{
  //Pobranie identyfikatora do istniejącego
  //obiektu zdarzenia
  //OpenEvent
  void* hEvent = OpenEvent(SYNCHRONIZE, FALSE,
                           "FILE_EXISTS");
  assert(hEvent);
  //Oczekiwanie na jego pojawienie się
  WaitForSingleObject(hEvent, INFINITE);

  //Wysyłanie danych do portu wirtualnego
  //lub gniazda
  //...

  return TRUE;
}
//-----------------------------------------------------
unsigned long threadFuncReceive(void* parameter)
{
  //Pobranie identyfikatora do istniejącego
  //obiektu zdarzenia
  //OpenEvent
  void* hEvent = OpenEvent(SYNCHRONIZE, FALSE,
                           "FILE_EXISTS");
  //Oczekiwanie na jego pojawienie się
  assert(hEvent);
  WaitForSingleObject(hEvent, INFINITE);

  //Odbiór danych z portu wirtualnego lub gniazda
  //...

  return TRUE;
}
//-----------------------------------------------------
```

Wzajemne wykluczenie (ang. *mutual exclusion*) jest sekcją krytyczną, która może być współdzielona przez wiele procesów i może działać pomiędzy wieloma procesami. Programy próbują tworzyć wzajemne wykluczenie pod specyficzną nazwą `lpName`, wykorzystując w tym celu funkcję API Windows:

```
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES
                   lpMutexAttributes,
                   BOOL bInitialOwner, LPCTSTR lpName);
```

Pierwszy proces, któremu uda się utworzyć obiekt wzajemnego wykluczenia, staje się serwerem. Jeżeli wzajemne wykluczenie już istnieje, proces staje się klientem względem serwera. Na listingu A.2 pokazano szkielet przykładu, w którym tworzone jest wzajemne wykluczanie współdzielone przez wszystkie procesy. Funkcja zwraca wartość prawdziwą, jeżeli proces jest serwerem, lub wartość fałszywą, jeżeli jest klientem. Ponieważ serwer zawsze staje się właścicielem wykluczenia, dlatego też zawsze należy go zwolnić, zanim zostanie ono przechwycone przez klienta. Zwolnienie wzajemnego wykluczenia o podanym identyfikatorze wykonywane jest poprzez funkcję API:

```
BOOL ReleaseMutex(HANDLE hMutex);
```

Listing A.2. Przykład wykorzystania funkcji `CreateMutex()`

```
#include <iostream>
//...
using namespace std;

//Deklaracje zmiennych globalnych

unsigned long threadFuncSend(void* parameter);
unsigned long threadFuncReceive(void* parameter);
//-----------------------------------------------------
int main()
{
   bufferIn=(char*)HeapAlloc(GetProcessHeap(),
                   HEAP_ZERO_MEMORY,
                   strlen(bufferOut)+1);

   //Konfiguracja portu wirtualnego lub gniazda
   //Patrz np. listingi 4.1, 4.2

   hMutex=CreateMutex(NULL, TRUE, "FILE_EXISTS");

   // tworzymy dwa wątki
```

```
   //1. do wysyłania danych
   hThread[0] = CreateThread(NULL, 0,

(LPTHREAD_START_ROUTINE)threadFuncSend,
                        &hMutex, 0, &threadID1);
   //2. do czytania danych
   hThread[1] = CreateThread(NULL, 0,

(LPTHREAD_START_ROUTINE)threadFuncReceive,
                        &hMutex, 0, &threadID2);

   ReleaseMutex(hMutex);
   WaitForMultipleObjects(2, hThread, TRUE, 10);

   CloseHandle(hMutex);
   CloseHandle(hThread[0]);
   CloseHandle(hThread[1]);
   HeapFree(GetProcessHeap(),0,bufferIn);
   cin.get();
   return 0;
}
//-------------------------------------------------------
unsigned long threadFuncSend(void* parameter)
{
  void* hMutex = &parameter;
  WaitForSingleObject(hMutex, INFINITE);
  //Wysyłanie danych do portu wirtualnego
  //lub gniazda
  //...
  ReleaseMutex(hMutex);
  return TRUE;
}
//-------------------------------------------------------
unsigned long threadFuncReceive(void* parameter)
{
  void* hMutex = &parameter;
  WaitForSingleObject(hMutex, INFINITE);
  //Odbiór danych z portu wirtualnego lub gniazda
  //...
  ReleaseMutex(hMutex);
  return TRUE;
}
//-------------------------------------------------------
```

Semafor (ang. *semaphore*) działa jak bramka kontrolująca ilość wątków wykonujących dany fragment kodu. Nowy semafor tworzony jest w funkcji:

```
HANDLE CreateSemaphore(LPSECURITY_ATTRIBUTES
                       lpSemaphoreAttributes,
                       LONG lInitialCount,
                       LONG lMaximumCount,
                       LPCTSTR lpName);
```

Wątek tworzący semafor specyfikuje wartość wstępną i maksymalną licznika wywołań. Inne wątki uzyskują dostęp do semafora za pomocą funkcji `OpenSemaphore()` i po zakończeniu pracy w sekcji krytycznej wątek zwalnia semafor za pomocą funkcji `ReleaseSemaphore()`, tak jak pokazano to na listingu A.3.

Listing A.3. Przykład wykorzystania funkcji `CreateSemaphore()`

```
#include <iostream>
//...

using namespace std;

//Deklaracje zmiennych globalnych

unsigned long threadFuncSend(void* parameter);
unsigned long threadFuncReceive(void* parameter);
//-----------------------------------------------------
int main()
{
   bufferIn=(char*)HeapAlloc(GetProcessHeap(),
                  HEAP_ZERO_MEMORY,
                  strlen(bufferOut)+1);

   //Konfiguracja portu wirtualnego lub gniazda
   //Patrz np. listingi 4.1, 4.2

   hSemaphore=CreateSemaphore(NULL, 0, 1,
                              "FILE_EXISTS");

   // tworzymy dwa wątki
   //1. do wysyłania danych
   hThread[0] = CreateThread(NULL, 0,
          (LPTHREAD_START_ROUTINE)threadFuncSend,
           &hSemaphore, 0, &threadID1);
```

```
    //2. do czytania danych
    hThread[1] = CreateThread(NULL, 0,
        (LPTHREAD_START_ROUTINE)threadFuncReceive,
         &hSemaphore, 0, &threadID2);


    ReleaseSemaphore(hSemaphore, 1, NULL);
    WaitForMultipleObjects(2, hThread, TRUE, 10);

    CloseHandle(hSemaphore);
    CloseHandle(hThread[0]);
    CloseHandle(hThread[1]);
    HeapFree(GetProcessHeap(),0,bufferIn);
    cin.get();
    return 0;
}
//-------------------------------------------------------
unsigned long threadFuncSend(void* parameter)
{
  void* hSemaphore =
        OpenSemaphore(SEMAPHORE_ALL_ACCESS, 1,
                      "FILE_EXISTS");
  WaitForSingleObject(hSemaphore, INFINITE);
  //Wysyłanie danych do portu wirtualnego
  //lub gniazda
  //...
  ReleaseSemaphore(hSemaphore ,1 ,NULL);
  return TRUE;
}
//-------------------------------------------------------
unsigned long threadFuncReceive(void* parameter)
{
  void* hSemaphore =
        OpenSemaphore(SEMAPHORE_ALL_ACCESS, 1,
                      "FILE_EXISTS");
  WaitForSingleObject(hSemaphore, INFINITE);
  //Odbiór danych z portu wirtualnego lub gniazda
  //...
  ReleaseSemaphore(hSemaphore, 1,NULL);
  return TRUE;
}
//-------------------------------------------------------
```

DODATEK B

ZESTAWY BIBLIOTEK DLA PROGRAMISTÓW

W podręczniku opisano zasoby systemowe zarówno Windows SDK, jak i biblioteki WinSock pomocne w samodzielnym programowaniu bezprzewodowej transmisji danych w standardzie Bluetooth. Programiści powinni być świadomi faktu, że oprócz przedstawionych zasobów istnieje szereg innych narzędzi pomocnych w samodzielnym konstruowaniu aplikacji Bluetooth. Poniżej przedstawiono kilka najbardziej wydajnych bibliotek programistycznych. Zasady ich wykorzystanie zasadniczo nie różną się od tego, co zostało opisane w podręczniku.

Widcomm

Widcomm jest biblioteką umożliwiającą uzyskiwanie dostępu w systemach Windows do bardzo wielu protokołów transportowych. Informacje szczegółowe dostępne są na stronie: http://widcomm-bluetooth.software.informer.com/.

BlueZ

Bazująca na języku C biblioteka BlueZ oferuje dostęp do stosu protokołów Bluetooth w systemach GNU/Linux. Biblioteka ta nie jest instalowana domyślnie. Informacje szczegółowe dostępne są na stronie: http://www.bluez.org/.

PyBlueZ

PyBlueZ jest biblioteką przeznaczoną dla programistów piszących w języku Python. Informacje szczegółowe dostępne są pod adresem: http://code.google.com/p/pybluez/.

BlueCove

Biblioteka BlueCove oferuje wygodne API dla programistów Javy. Szczegółowe informacje dostępne są pod adresem: http://code.google.com/p/bluecove/.

Wiele użytecznych przykładów praktycznego wykorzystania niektórych zasobów oferowanych przez wymienione wyżej biblioteki można znaleźć na stronie: http://people.csail.mit.edu/albert/bluez-intro/.

BIBLIOGRAFIA

[1] B. A. Miller, Ch. Bisdikian, *Bluetooth Revealed: The Insider's Guide to an Open Specification for Global Wireless Communications* (2nd Edition), Prentice Hall PTR 2001; wydanie polskie *Bluetooth*, Helion 2003.

[2] J. Bray and Ch. F. Sturman, *Bluetooth – Connect without Cables*, Prentice Hall 2000.

[3] W. H. Tranter, T. S. Rappaport, B. D. Woerner (Editor), J. H. Reed (Editor), *Wireless Personal Communications: Bluetooth Tutorial and Other Technologies*, Kluwer Academic Publishers 2000.

[4] D. McMichael Gilster, *Bluetooth End to End*, Wiley 2002.

[5] http://www.palowireless.com/infotooth/tutorial.asp

[6] https://www.bluetooth.org/Building/HowTechnologyWorks/ProfilesAndProtocols/Overview.htm

[7] A. Daniluk, *RS 232C – praktyczne programowanie. Od Pascala i C++ do Delphi i Buildera*. Wydanie III, Helion 2007.

[8] K. Łoziak, M. Sikora, M. Wągrowski, *Profile aplikacji standardu Bluetooth*, Telekomunikacja Cyfrowa – Technologie i Usługi, Tom 5, s. 16-22, 2003.

[9] D. A. Gratton, *Bluetooth Profiles: The Definitive Guide*, Prentice Hall 2002.

[10] A. Daniluk, *USB. Praktyczne programowanie z Windows API w C++*, Helion 2009.

[11] Texas Instruments CC2540/41 *Bluetooth Low Energy Software Developer's Guide v1.2 Document*, Texas Instruments, Inc., 2010-2012.

[12] J. Bogusz, *Moduły Bluetooth Rayson Technology*, Elektronika Praktyczna 12/2010; http://ep.com.pl/files/2065.pdf

[13] http://www.palowireless.com/bluetooth/

[14] Ch. Gehrmann, *Bluetooth Security White Paper*, http://grouper.ieee.org/groups/1451/5/Comparison%20of%20PHY/Bluetooth_24Security_Paper.pdf

[15] N. Mavrogiannopoulos, *On Bluetooth security*, http://members.hellug.gr/-nmav/papers/other/Bluetooth%20security.pdf

[16] I. M. B. Nogales, *Bluetooth Security Features*, http://www.urel.feec.-vutbr.cz/ra2008/archive/ra2006/abstracts/085.pdf

[17] A. Dumas, *Programming WinSock*, Sams Publishing 1994.

[18] Dreamtech Software Team, *WAP, Bluetooth, and 3G Programming: Cracking the Code*, Wiley 2001.

[19] http://msdn.microsoft.com/en-us/library/ms890956.aspx

[20] J. Bogusz, *Programowanie i obsługa modułów GSM*, Elektronika Praktyczna 8/2002; http://ep.com.pl/files/8073.pdf

[21] http://en.wikipedia.org/wiki/ Hayes_command_set

[22] A. Daniluk, *C++Builder Borland Developer Studio 2006. Kompendium programisty*, Helion 2006.

SKOROWIDZ

R

S

T

U

W

Z