RS 232C – praktyczne programowanie. Od Pascala i C++ do Delphi i Buildera. Wydanie III

Autor: Andrzej Daniluk
ISBN: 978-83-246-0778-5
Format: B5, stron: 504

Na uczelniach, w szkołach i biurach pojawia się coraz więcej zaawansowanych urządzeń komputerowych podłączanych przez port szeregowy. Czy koniecznie trzeba płacić wysokie stawki informatykom, aby wykorzystać pełnię możliwości tych nowoczesnych narzędzi? Na szczęście nie. Obsługa transmisji szeregowej przy użyciu standardu RS 232C może być na tyle łatwa, że uczniowie, studenci, nauczyciele, pracownicy naukowi czy inżynierowie mogą samodzielnie tworzyć potrzebne im oprogramowanie.

Dzięki książce „RS 232C – praktyczne programowanie. Od Pascala i C++ do Delphi i Buildera. Wydanie III" także i Ty szybko nauczysz się pisać programy sterujące urządzeniami podłączanymi przez port szeregowy. Dowiesz się, jak działa transmisja asynchroniczna oraz czym jest standard RS 232C. Poznasz interfejs RS 232C dla systemu Windows i nauczysz się go używać w środowiskach programistycznych Builder i Delphi, co pozwoli Ci pisać potrzebne oprogramowanie w języku Pascal lub C++. Najnowsze, poprawione wydanie zawiera jeszcze więcej przykładów, dzięki którym błyskawicznie będziesz mógł sprawdzić nabytą wiedzę w praktyce.

- Standard RS 232C
- Transmisja asynchroniczna
- Obsługa RS 232C w systemach MS-DOS i Windows
- Wykorzystanie elementów interfejsu Windows API w środowiskach Builder i Delphi
- Testowanie programów do obsługi transmisji szeregowej
- Tworzenie aplikacji wielowątkowych
- Narzędzia graficzne
- Przykładowe aplikacje i ich analiza
- Specyfikacje najważniejszych funkcji

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 5.

Programowa obsługa interfejsu RS 232C w Windows

Czwarte prawo Murphy'ego

> *Gdy dojdziesz do wniosku, że są cztery sposoby, na jakie może się nie powieść dane przedsięwzięcie, i zabezpieczysz się przed nimi, rychło pojawi się piąta możliwość.*
>
> *Murphy's Law and other reasons why things go wrong!*,
> Artur Bloch, Price Stern Sloan Inc. 1977.

Rozdział ten ma za zadanie zapoznać Czytelnika ze sposobami konstrukcji algorytmów realizujących transmisję szeregową w środowisku Windows, które charakteryzuje się pewnymi cechami niemającymi odpowiedników w MS-DOS. Poznanie i umiejętne wykorzystanie tych cech sprawi, iż problem obsługi interfejsów szeregowych z poziomu Windows — uważany powszechnie za trudny — przestanie być dla nas tajemnicą. Pokażemy, w jaki sposób należy tworzyć aplikacje służące do programowej obsługi łącza szeregowego RS 232C zarówno w C++, C++Builderze, jak i w Delphi. Wśród programistów istnieje zauważalny podział na osoby programujące głównie w Delphi oraz na preferujące Buildera lub ogólnie C++ dla Windows. Jednak zdaniem wielu osób uniwersalność jest jedną z tych cech, jakie powinny charakteryzować programistę. W rozdziale tym przybliżymy Czytelnikowi podobieństwa i różnice w sposobie konstrukcji algorytmów realizujących transmisję szeregową, pisanych w Delphi oraz Builderze.

W dalszej części książki będziemy się spotykać z typami danych, których poznanie i zrozumienie ma kluczowe znaczenie w projektowaniu aplikacji obsługujących urządzenia zewnętrzne. Zacznijmy od ich przypomnienia. W tabeli 5.1 przedstawiono porównanie podstawowych typów zmiennych wykorzystywanych w kompilatorach, które będą dla nas istotne. Większości z nich można używać zamiennie, pisząc zarówno w Delphi, jak i w C++Builderze.

Tabela 5.1. *Typy zmiennych stosowanych w Delphi oraz w C++Builderze*

Delphi	Rozmiar w bajtach	Znak +/–	Typ	C++Builder
ShortInt	1		Integer	signed char
SmallInt	2		Integer	short
LongInt	4		Integer	
Byte	1	Bez znaku	Integer	unsigned char
Word	2	Bez znaku	Integer	unsigned short
Integer	4		Integer	int
Cardinal	4	Bez znaku	Integer	unsigned int
Boolean	1	true/false		bool
ByteBool	1	true/false		unsigned char
		Bez znaku	Integer	
WordBool	2	true/false		unsigned short
		Bez znaku	Integer	
LongBool	4	true/false		
		Bez znaku	Integer	
AnsiChar	1	1 znak ANSI	Character	char
WideChar	2	1 znak Unicode	Character	wchar_t
Char	1	Bez znaku	Character	char
AnsiString	≈3GB	ANSIChar	AnsiString	AnsiString
String[n]	n = 1.255	ANSIChar	String	SmallString<n>
ShortString	255	ANSIChar	String	SmallString<255>
String	255 lub ≈3GB	ANSIChar	AnsiString	AnsiString
Single	4		Floating point number (liczba zmiennoprzecinkowa)	float
Double	8		Floating point number	double
Extended	10		Floating point number	long double
Real	4		Floating point number	double
Pointer	4		Generic pointer (wskaźnik ogólny, adresowy)	void *
PChar	4	Bez znaku	Pointer to characters	unsigned char *
PAnsiChar	4	Bez znaku	Pointer to ANSIChar	unsigned char *
Comp	8		Floating point number	Comp

Konstruując nasze programy, będziemy starali się jak najszerzej wykorzystywać standardowe zasoby Windows, w szczególności tzw. interfejs programisty Windows API (ang. *Application Programming Interface*). Jego umiejętne wykorzystanie umożliwi naszym aplikacjom błyskawiczne skonfigurowanie i uzyskanie dostępu do portu komunikacyjnego. Błędem jest twierdzenie, że sama — nawet bardzo dobra — znajomość języka pro-

gramowania wystarczy, żeby stworzyć poprawnie działający w Windows program. Otóż musimy zdawać sobie sprawę z faktu, o którym często się zapomina — niemożliwe jest napisanie udanej aplikacji mającej pracować w pewnym środowisku (czytaj — systemie operacyjnym) bez znajomości tego środowiska. Wiele już zostało powiedziane na temat dobrych i złych stron Windows, należy jednak pamiętać, że oferuje on nam swoją wizytówkę, ofertę współpracy, czyli API. Już nie wystarczy umiejętność wykorzystywania ulubionego kompilatora. Zasoby Delphi czy Buildera połączymy z zasobami systemu operacyjnego, a spoiwem będzie właśnie uniwersalne Windows API. Istnieje wiele warstw API używanych w zależności od potrzeb. W tym i dalszych rozdziałach zajmiemy się szeroko rozumianą warstwą komunikacyjną.

Windows API korzysta ze specjalnego systemu nazewnictwa zmiennych, z tzw. notacji węgierskiej wprowadzonej przez Karoja Szimoniego. Zgodnie z nią do rdzenia nazwy zadeklarowanej zmiennej dodaje się przedrostek (ang. *prefix*). Chociaż istnieją pod tym względem pewne rozbieżności pomiędzy nazewnictwem Microsoftu i Borlanda, to jednak zapis taki bardzo ułatwia szybkie ustalenie roli zmiennej w programie oraz jej typ. W następnych rozdziałach będziemy się starali — wszędzie gdzie jest to możliwe — zachowywać *samokomentujące się* nazewnictwo API (większość nazw API będziemy traktować jako nazwy własne). Z doświadczenia wiadomo, że stosowanie takiej konwencji bardzo pomaga w studiowaniu plików pomocy. Oczywiście moglibyśmy silić się na oryginalność, wprowadzając własne zmienne, zrozumiałe tylko dla piszącego dany program — wówczas przykłady musiałyby być zapisane jako wręcz humorystyczna mieszanina języków polskiego i angielskiego. Trzeba też przyznać, że byłby to bardzo skuteczny sposób zaciemnienia obrazu API. Zrozumienie znaczenia nazw tam stosowanych okaże się w przyszłości niezwykle cenne, gdyż API można czytać jak książkę. Aby pomóc Czytelnikom, którzy nie zetknęli się dotąd z tymi pojęciami, w tabeli 5.2 przedstawiono ogólne zasady tworzenia niektórych przedrostków.

Windows oferuje nam ponadto kilka typów danych, z których część tylko nieznacznie różni się sposobem zapisu w implementacjach Delphi i Buildera. Typy te mają najczęściej postać struktury lub klasy i są bardzo często wykorzystywane w warstwie komunikacyjnej programów.

Typy danych Windows

Nowoczesna idea programowania w Windows oparta na wykorzystaniu narzędzi programistycznych typu RAD, do których zaliczają się C++Builder oraz Delphi, pozwala programistom na maksymalne uproszczenie procesu tworzenia oprogramowania. Jednym z przykładów dążenia do zminimalizowania czasu tworzenia aplikacji jest zastosowanie w Windows pewnych bardzo zwartych w zapisie typów danych, które oczywiście mają swoje odpowiedniki w typach standardowych. W tabeli 5.3 zebrano najistotniejsze typy danych, którymi bardzo często posługują się programy Windows. Należy zdawać sobie sprawę z faktu, iż typy takie jak np. `LPVOID` i `LPSTR` nie są w dosłownym słowa tego znaczeniu typami nowymi, tzn. od początku stworzonymi na potrzeby aplikacji Windows, gdyż zostały zdefiniowane w plikach nagłówkowych za pomocą instrukcji `typedef` po to, aby uprościć zapis niektórych standardowych typów danych. W tabeli 5.3 przedstawiono wybrane typy danych, którymi posługuje się API Windows.

Tabela 5.2. *Ogólne zasady tworzenia przedrostków według notacji węgierskiej*

Przedrostek	Skrót angielski	Znaczenie
a	*array*	Tablica
b	*bool*	Zmienna logiczna `true` lub `false`
by	*byte unsigned char*	Znak (bajt)
cb	*count of bytes*	Liczba bajtów
ch	*char*	Znak
dw	*double word*	Podwójne słowo
evt	*event*	Zdarzenie
f	*flag*	Znacznik
fdw	*flag of double word*	Znacznik typu `dw`
fn	*function*	Funkcja
h	*handle*	Identyfikator (uchwyt)
i	*integer*	Typ całkowity 4-bajtowy
id	(*ID*) *identification*	Identyfikacja
in	*input*	Wejście, dane wejściowe
l	*long int*	Typ całkowity długi 4-bajtowy
lp	*long pointer*	Wskaźnik typu `long int`
lpc	*long pointer to C-string*	Wskaźnik typu `long int` do C-łańcucha
lpfdw	*long pointer to flag of dw*	Wskaźnik typu `lp` do znacznika typu `double word`
lpfn	*long pointer to function*	Wskaźnik typu `lp` do funkcji
n	*short or int*	Typ krótki lub całkowity
np	*near pointer*	Bliski wskaźnik (w środowisku 32-bitowym to samo co `lp`)
out	*output*	Wyjście, dane wyjściowe (przetworzone)
p	*pointer*	Wskaźnik (w środowisku 32-bitowym to samo co `lp`)
pfn	*pointer to function*	Wskaźnik do funkcji
que	*queue*	Kolejka, bufor danych
s (sz)	*string*	Łańcuch znaków
st	*struct*	Struktura
t	*type*	Typ
u	*unsigned*	Bez znaku
w	(*word*) *unsigned int*	Słowo
wc	*WCHAR*	Znak zgodny z Unicode

Tabela 5.3. *Niektóre typy danych stosowane w Windows*

Typ Windows	Znaczenie
BOOL	int z dwoma wartościami TRUE oraz FALSE
BYTE	unsigned char
DWORD	unsigned long
LPDWORD	unsigned long *
LONG	long
LPLONG	long *
LPCSTR	const char *
LPCTSTR	unsigned const char *
LPSTR	char *
LPVOID lub Pointer	void *
LPCVOID	const void *
UINT	unsigned int
WORD	unsigned short
DWORD32	32-bitowy typ całkowity bez znaku
DWORD64	64-bitowy typ całkowity bez znaku
INT	32-bitowy typ całkowity ze znakiem
INT32	32-bitowy typ całkowity ze znakiem
INT64	64-bitowy typ całkowity ze znakiem
LONG32	32-bitowy typ całkowity ze znakiem
LONG64	64-bitowy typ całkowity ze znakiem
LONGLONG	64-bitowy typ całkowity ze znakiem

Osobnym typem danych, bardzo często stosowanym w aplikacjach Windows, jest typ HANDLE. Jest on 32- lub 64-bitowym typem danych całkowitych oznaczającym tzw. *uchwyt* (ang. *handle*). Należy rozumieć, iż w rzeczywistości dane typu HANDLE nie obrazują jakichś tajemniczych *uchwytów* zakładanych na elementy aplikacji — są to po prostu 32- lub 64-bitowe liczby identyfikujące określony zasób aplikacji, systemu operacyjnego lub samego komputera. Z tego względu dane typu HANDLE często wygodniej i zręczniej jest określać mianem *identyfikatorów*, których wartości przechowywane są w określonym miejscu w pamięci. Cechą charakterystyczną identyfikatorów jest to, iż jeśli na początku programu inicjuje się je określonymi wartościami, w momencie zakończenia pracy aplikacji lub jej fragmentu należy przydzieloną im pamięć odpowiednio zwalniać. W tym celu wykorzystuje się funkcję API Windows:

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

z argumentem w postaci określonego identyfikatora.

Zaopatrzeni w powyższą terminologię pójdźmy dalej i zobaczmy, do czego mogą nam być przydatne poszczególne struktury oraz funkcje interfejsu programisty — Windows API.

Proces projektowania oprogramowania

Zanim przejdziemy do szczegółowego omawiania aspektów tworzenia programów obsługujących port szeregowy w Windows, należy wybrać jedną z metod projektowania tego rodzaju aplikacji.

Praktyka wskazuje, że dla pojedynczych użytkowników lub niewielkich organizacji dobrze sprawdza się metodologia oparta na programowaniu przyrostowym i iteracyjnym (ang. *iterative and incremental development*).

W dalszej części książki będziemy korzystać z metody projektowania iteracyjnego. Takie podejście do zagadnienia sprawi, iż tworzone aplikacje oprócz wysokiej sprawności działania będą jeszcze miały dwie bardzo ważne i nieczęsto spotykane w literaturze cechy. Będą mianowicie:

- w pełni rozbudowywalne,
- łatwe do samodzielnej modyfikacji nawet przez osoby dopiero poznające zasady programowania w środowiskach Buildera i Delphi.

Wszystkie prezentowane algorytmy będziemy się starali konstruować w ten sposób, aby pewne słynne twierdzenie wypowiedziane niegdyś przez Murphy'ego w omawianych programach nie miało zastosowania. Brzmi ono następująco:

Twierdzenie o komplikacji procedur

> *Każdą dowolnie skomplikowaną procedurę można skomplikować jeszcze bardziej. Twierdzenie odwrotne nie jest prawdziwe: nadzwyczaj rzadko się zdarza, aby skomplikowaną procedurę można było uprościć.*
>
> *Murphy's Law and other reasons why things go wrong!*, Artur Bloch, Price Stern Sloan Inc. 1977.

Wykorzystanie elementów Windows API w C++Builderze. Część I

Poznawanie tajników obsługi portu szeregowego w Windows rozpoczniemy, z czysto praktycznych względów, od pisania programów w C++Builderze. C++ ma składnię taką jak API, dlatego prościej nam będzie zapoznać się z budową funkcji oraz struktur oferowanych przez interfejs programisty. Ułatwi to też zrozumienie, w jaki sposób i w jakiej kolejności należy umieszczać je w programie.

Struktura DCB

Fundamentalne znaczenie ma struktura kontroli urządzeń zewnętrznych DCB (ang. *Device Control Block*). W Windows struktura DCB w pewnym sensie odpowiada funkcji 00h przerwania 14h BIOS-u. Udostępnia nam jednak nieporównywalnie większe możliwości programowej obsługi łącza szeregowego; umożliwia bezpośrednie programowanie rejestrów układu UART. W tabelach 5.4 oraz 5.5 przedstawiono specyfikację bloku kontroli urządzeń zewnętrznych DCB.

Większość pól tej struktury to pola jednobitowe. fDtrControl, fRtsControl są polami dwubitowymi. Aktualnie nieużywane w XP pole fDummy2 jest siedemnastobitowe. W perspektywie, wraz z wReserved oraz wReserved1, będzie wykorzystane na potrzeby innych protokołów komunikacyjnych. W Windows API blok kontroli urządzeń deklarowany jest w sposób następujący:

```
typedef struct _DCB {
        DWORD DCBlength;
        ...
} DCB;
```

Deklaracja ta tworzy nowe słowo kluczowe typu DCB (struktura). Zalecane jest, aby przed użyciem tej struktury jako parametru do elementu DCBlength wpisać wartość sizeof(DCB).

Strukturę tworzy zbiór logicznie powiązanych elementów, np. zmiennych lub (i) pól bitowych. Pole bitowe stanowi zbiór przylegających do siebie bitów, znajdujących się w jednym słowie. Adres struktury pobieramy za pomocą operatora referencji &, co umożliwia nam działania na jej składowych. Do struktury jako całości możemy odwołać się przez jej nazwę, zaś do poszczególnych jej elementów, czyli zmiennych oraz pól bitowych, przez podanie nazwy zmiennej reprezentującej strukturę oraz — po kropce — nazwy konkretnej zmiennej lub pola struktury, np.: dcb.fDtrControl = DTR_CONTROL_DISABLE. Operator składowych struktur "." jest lewostronnie łączny. Grupa związanych ze sobą zmiennych i pól bitowych traktowana jest jako jeden obiekt.

Zanim przejdziemy do praktycznego zastosowania poznanych pól struktury DCB, musimy zapoznać się z czterema podstawowymi funkcjami Windows API służącymi do programowej konfiguracji portów szeregowych. W dalszej części książki funkcji takich będzie przybywać, ale te przedstawione poniżej należy traktować jako najbardziej podstawowe.

Funkcja CreateFile()

Jest to funkcja służąca do utworzenia i otwarcia pliku lub urządzenia. Już sama nazwa wskazuje, że może być wykorzystywana nie tylko do obsługi portu szeregowego. Teraz jednak będzie nas interesować tylko to konkretne zastosowanie. Specyfikacja zasobów funkcji CreateFile() najczęściej używanych do operacji plikowych zamieszczona jest w dodatku A. Funkcja ta da nam 32- lub 64-bitowy identyfikator danego portu przechowywany pod właściwością HANDLE, do którego będą adresowane wszystkie komunikaty.

Tabela 5.4. *Zmienne struktury DCB reprezentujące dopuszczalne parametry ustawień portu szeregowego*

Typ	Zmienna	Znaczenie	Wartość, stała symboliczna
DWORD	DCBlength	Rozmiar struktury	Należy wpisać
DWORD	BaudRate	Określenie prędkości transmisji (b/s)	CBR_110 CBR_19200 CBR_300 CBR_38400 CBR_600 CBR_56000 CBR_1200 CBR_57600 CBR_2400 CBR_115200 CBR_4800 CBR_128000 CBR_9600 CBR_256000 CBR_14400
WORD	wReserved	Nieużywane	0
WORD	XonLim	Określenie minimalnej liczby bajtów w buforze wejściowym przed wysłaniem specjalnego znaku sterującego XON	Domyślnie: 65 535; w praktyce XonLim ustala się jako ½ rozmiaru deklarowanego wejściowego bufora danych
WORD	XoffLim	Określenie maksymalnej liczby bajtów w buforze wejściowym przed wysłaniem specjalnego znaku sterującego XOFF	Domyślnie: 65535; w praktyce XoffLim ustala się jako ¾ rozmiaru deklarowanego bufora wejściowego
BYTE	ByteSize	Wybór liczby bitów danych	5, 6, 7, 8
BYTE	Parity	Określenie kontroli parzystości	EVENPARITY — parzysta; MARKPARITY — bit parzystości stale równy 1; NOPARITY — brak kontroli; ODDPARITY — nieparzysta
BYTE	StopBits	Wybór bitów stopu	ONESTOPBIT — 1 bit stopu; ONE5STOPBITS — w przypadku słowa 5-bitowego bit stopu wydłużony o ½; TWOSTOPBITS — 2 bity stopu
char	XonChar	Określenie wartości znaku XON dla nadawania i odbioru (wysłanie znaku przywraca transmisję)	Standardowo (char) DC1, dziesiętnie: 17
char	XoffChar	Określenie wartości znaku XOFF dla nadawania i odbioru (wysłanie XOFF wstrzymuje transmisję do czasu odebrania znaku XON)	Standardowo (char) DC3, dziesiętnie: 19
char	ErrorChar	Określenie wartości znaku zastępującego bajty otrzymane z błędem parzystości	Opcjonalnie: 0 lub SUB
char	EofChar	Określenie wartości znaku końca otrzymanych danych	Opcjonalnie: 0
Char	EvtChar	Określenie wartości znaku służącego do sygnalizowania wystąpienia danego zdarzenia	Opcjonalnie: 0
WORD	wReserved1	Obecnie nieużywane	

Tabela 5.5. *Pola bitowe reprezentujące dopuszczalne wartości znaczników sterujących struktury DCB*

Typ	Pole bitowe	Właściwości pola	Wartość, znaczenie, stała symboliczna
DWORD	fBinary	Tryb binarny (Win API podtrzymuje jedynie ten tryb transmisji danych)	TRUE
DWORD	fParity	Umożliwia ustawienie sprawdzania parzystości — sposobu reakcji na bit parzystości	TRUE — kontrola parzystości włączona; FALSE — bit parzystości nie jest sprawdzany
DWORD	fOutxCtsFlow	Umożliwia ustawienie sprawdzania sygnału na linii CTS w celu kontroli danych wyjściowych	TRUE — jeżeli sygnał CTS jest nieaktywny, transmisja jest wstrzymywana do czasu ponownej aktywacji linii CTS; FALSE — włączenie sygnału na linii CTS nie jest wymagane do rozpoczęcia transmisji
DWORD	fOutxDsrFlow	Umożliwia ustawienie sprawdzania sygnału na linii DSR w celu kontroli danych wyjściowych	TRUE — jeżeli sygnał DSR jest nieaktywny, transmisja jest wstrzymywana do czasu ponownej aktywacji linii DSR; FALSE — włączenie sygnału na linii DSR nie jest wymagane do rozpoczęcia transmisji
DWORD	fDtrControl	Specyfikacja typu kontroli sygnału DTR	DTR_CONTROL_DISABLE / 0 — sygnał na linii DTR jest nieaktywny; DTR_CONTROL_ENABLE / 1 — sygnał na linii DTR jest aktywny; DTR_CONTROL_HANDSHAKE / 2 — włączenie potwierdzania przyjęcia sygnału DTR — potwierdzenie musi być odebrane na linii DSR. Używane w trybie półdupleksowym. Ewentualne błędy transmisji w tym trybie są usuwane przez funkcję EscapeCommFunction()
DWORD	fTXContinueOnXoff	Kontrola przerwania transmisji w przypadku przepełnienia bufora wejściowego i ewentualnie wystąpienia znaków XoffChar oraz XonChar	TRUE — wymuszanie kontynuowania transmisji nawet po wystąpieniu znaku XOFF i wypełnieniu wejściowego bufora danych powyżej XoffLim bajtów; FALSE — transmisja nie jest kontynuowana, dopóki bufor wejściowy nie zostanie opróżniony do pułapu XonLim bajtów i nie nadejdzie znak XON potwierdzenia dalszego odbioru
DWORD	fDsrSensitivity	Specyfikacja wykorzystania poziomu sygnału na linii DSR	TRUE — otrzymane bajty są ignorowane, o ile linia DSR nie jest w stanie wysokim; FALSE — stan linii DSR jest ignorowany

Tabela 5.5. *Pola bitowe reprezentujące dopuszczalne wartości znaczników sterujących struktury DCB — ciąg dalszy*

Typ	Pole bitowe	Właściwości pola	Wartość, znaczenie, stała symboliczna
DWORD	fInX	Programowe ustawienie protokołu XON-XOFF w czasie odbioru danych	TRUE — znak XoffChar jest wysyłany, kiedy bufor wejściowy jest pełny lub znajduje się w nim XoffLim bajtów; znak XonChar jest wysyłany, kiedy bufor wejściowy pozostaje pusty lub znajduje się w nim XonLim bajtów; FALSE — XON-XOFF w czasie odbioru nie jest ustawiony
DWORD	fRtsControl	Specyfikacja kontroli sygnału na linii RTS	RTS_CONTROL_DISABLE / 0 — sygnał na linii RTS jest nieaktywny; RTS_CONTROL_ENABLE / 1 — sygnał na linii RTS jest aktywny; RTS_CONTROL_HANDSHAKE / 2 — włączenie potwierdzania przyjęcia sygnału RTS (potwierdzenie musi być odebrane na linii CTS). Używane w trybie półdupleksowym. Sterownik podwyższa stan linii RTS, gdy wypełnienie bufora wejściowego jest mniejsze od ½. Stan linii RTS zostaje obniżony, gdy bufor wypełniony jest w ¾. Ewentualne błędy transmisji w tym trybie usuwane są przez funkcję EscapeCommFunction(); RTS_CONTROL_TOGGLE / 3 — linia RTS jest w stanie wysokim, jeżeli są bajty do transmisji i jest ona możliwa; po opróżnieniu bufora komunikacyjnego linia RTS pozostaje w stanie niskim
DWORD	fOutX	Programowe ustawienie protokołu XON-XOFF w czasie wysyłania danych	TRUE — transmisja zostaje przerwana po odebraniu znaku XoffChar i wznowiona po otrzymaniu znaku XonChar; FALSE — XON-XOFF w czasie wysyłania nie jest ustawiony
DWORD	fErrorChar	Umożliwia zastąpienie bajtów otrzymanych z błędem parzystości znakiem ErrorChar	TRUE — zastąpienie jest wykonywane, ponadto fParity musi być ustawione jako TRUE; FALSE — zastąpienie nie jest wykonane
DWORD	fNull	Odrzucenie odebranych nieważnych lub uszkodzonych bajtów	TRUE — nieważne bajty zostaną odrzucone przy odbiorze; FALSE — nieważne bajty nie będą odrzucane

Tabela 5.5. *Pola bitowe reprezentujące dopuszczalne wartości znaczników sterujących struktury DCB — ciąg dalszy*

Typ	Pole bitowe	Właściwości pola	Wartość, znaczenie, stała symboliczna
DWORD	fAbortOnError	Ustawienie wstrzymywania operacji nadawanie-odbiór przy wykryciu błędu transmisji	TRUE — wszelkie operacje nadawania i odbioru są wstrzymywane, zaś dalsza komunikacja nie jest możliwa, dopóki błąd nie zostanie usunięty przez wywołanie funkcji ClearCommError(); FALSE — nawet jeżeli wystąpi błąd, transmisja jest kontynuowana — błąd może być usunięty przez wywołanie funkcji ClearCommError()
DWORD	fDummy2	Zarezerwowane, nieużywane	

Ogólnie rzecz ujmując, przed rozpoczęciem czytania z portu szeregowego (lub innego urządzenia) należy o powyższym fakcie poinformować system operacyjny. Czynność tę określa się jako *otwieranie portu do transmisji*. Jednak zanim zaczniemy wykonywać jakiekolwiek operacje na porcie, system operacyjny musi sprawdzić, czy wybrany port komunikacyjny istnieje i czy w danym momencie nie jest już przypadkiem w jakiś sposób wykorzystywany. W przypadku uzyskania dostępu do portu system operacyjny przekazuje do aplikacji jego identyfikator. We wszystkich operacjach wejścia-wyjścia zamiast szczegółowej nazwy portu komunikacyjnego używa się właśnie jego identyfikatora.

Składnia CreateFile() wygląda następująco[1]:

```
HANDLE CreateFile(LPCTSTR lpFileName,
                  DWORD dwDesiredAccess,
                  DWORD ShareMode,
                  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                  DWORD dwCreationDistribution,
                  DWORD dwFlagsAndAttributes,
                  HANDLE hTemplateFile);
```

Niekiedy identyfikatory tego typu nazywa się uchwytami. Niestety, dosłowne przetłumaczenie angielskiego słowa *handle* jako uchwyt, np. *handle of drawer* — uchwyt, rączka szuflady, nie jest w pełni adekwatne. Właściwsze wydaje się utożsamianie *handle* z identyfikatorem (unikalną wartością zlokalizowaną w danym obszarze pamięci i skojarzoną z konkretnym portem komunikacyjnym, oknem czy plikiem). W potocznej angielszczyźnie *handle* może również oznaczać ksywę pozwalającą na szybką identyfikację danej osoby lub rzeczy. Koncepcja identyfikatorów nie jest niczym nowym, stosowano ją już w MS-DOS, jednak dopiero w Windows zyskała nową jakość.

Na tym etapie naszych rozważań tylko trzy parametry powyższej funkcji są istotne dla kompletnej konfiguracji portu szeregowego. Wyjaśnimy teraz ich znaczenie.

[1] Pełna specyfikacja funkcji CreateFile() została zamieszczona w dodatku A.

Pierwszy parametr, lpFileName, jest wskaźnikiem do zadeklarowanego ciągu znaków zakończonego zerem (zerowym ogranicznikiem), tzw. *null terminated string*, lub do C-łańcucha (dokładniej: do pierwszego znaku tego łańcucha), w którym przechowywana będzie nazwa (wartość) portu. Z poprzednich rozdziałów pamiętamy, że ogólnie przyjęte jest stosowanie nazewnictwa portów szeregowych jako COM*n* (nazwy COM*n* znajdują się na liście nazw zastrzeżonych), gdzie *n* oznacza numer portu. Deklaracja numeru portu szeregowego, np. 2., będzie więc przedstawiać się w sposób bardzo prosty:

```
LPCTSTR portName = "COM2";
```

lub, co jest równoważne:

```
unsigned const char  *portName = "COM2";
```

Można też zmienną, pod którą przechowywać będziemy numer portu, zadeklarować w sposób tradycyjny, używając typu char. Deklaracja taka będzie w pełni poprawna:

```
char  portName[5] = "COM2";
```

Parametr dwDesiredAccess typu DWORD umożliwia ustalenie rodzaju dostępu do portu szeregowego. Z praktycznego punktu widzenia najwygodniej jest ustalić rodzaj dostępu jako GENERIC_READ | GENERIC_WRITE (zapisuj do portu lub odczytuj z portu). Umożliwi nam to płynne wysyłanie i odbieranie komunikatów, co w pełni odpowiada półdupleksowemu wariantowi transmisji. Jeżeli zechcemy korzystać jedynie z trybu simpleksowego, do dwDesiredAccess wystarczy przypisać jeden z wybranych rodzajów dostępu.

Windows API posługuje się łańcuchami o długości większej niż 256 znaków. Aby przełamać to ograniczenie, zrezygnowano z zapamiętywania w pierwszym bajcie liczby określającej długość łańcucha znaków. W C-łańcuchach ostatnim znakiem, kończącym ciąg jest 0 (NULL lub heks. 00), którego nie należy mylić ze znakiem zero (48 lub heks. 30). Stąd nazwa *null terminated string*.C-łańcuchy osiągają długość 65535 znaków plus końcowy, tzw. NULL-bajt. Są one dynamicznie alokowane w pamięci, zaś ilość pamięci zajmowanej przez C-łańcuch jest automatycznie dostosowywana do jego długości, co w pełni odpowiada architekturze Windows.

Parametrowi dwCreationDistribution należy przypisać właściwość OPEN_EXISTING — otwórz istniejący (port). Pozostałym przyporządkujemy następujące wartości:

```
DWORD ShareMode = 0 (FALSE);
LPSECURITY_ATTRIBUTES lpSecurityAttributes = NULL;
DWORD dwFlagAndAttributes = 0 (FALSE);
HANDLE hTemplateFile = NULL.
```

Funkcja GetCommState()

Funkcja zwraca ustawienia portu ostatnio zapamiętane w strukturze DCB:

```
BOOL GetCommState(HANDLE hCommDev, LPDCB lpdcb),
```

gdzie:

`hCommDev` jest identyfikatorem danego portu, `CreateFile()` zwraca nam ten identyfikator, a `lpdcb` jest wskaźnikiem do struktury `DCB` zawierającej informację o aktualnych ustawieniach parametrów łącza szeregowego.

Funkcja `GetCommState()` (jak i wszystkie inne typu `BOOL`) zwraca wartość `TRUE` w przypadku pomyślnego jej wykonania, ewentualnie wartość `FALSE` w sytuacji przeciwnej.

Funkcja SetCommState()

Wybrany przez nas port szeregowy ostatecznie skonfigurujemy zgodnie ze specyfikacją struktury `DCB` za pomocą funkcji `SetCommState()`, która reinicjalizuje i uaktualnia wszystkie dostępne parametry w ustawieniach łącza szeregowego:

```
BOOL SetCommState(HANDLE hCommDev, LPDCB lpdcb)
```

Jednak tutaj parametr, na który wskazuje `lpdcb`, musi już zawierać informacje o nowych, wybranych przez nas parametrach ustawień portu komunikacyjnego. Należy też pamiętać, że funkcja `SetCommState()` nie zostanie wykonana pomyślnie, jeżeli posługując się strukturą `DCB`, element `XonChar` ustalimy identycznie z `XoffChar`.

Funkcja CloseHandle()

Przed zakończeniem działania aplikacji otwarty port szeregowy należy koniecznie zamknąć i zwolnić obszar pamięci przydzielony na jego identyfikator, korzystając z:

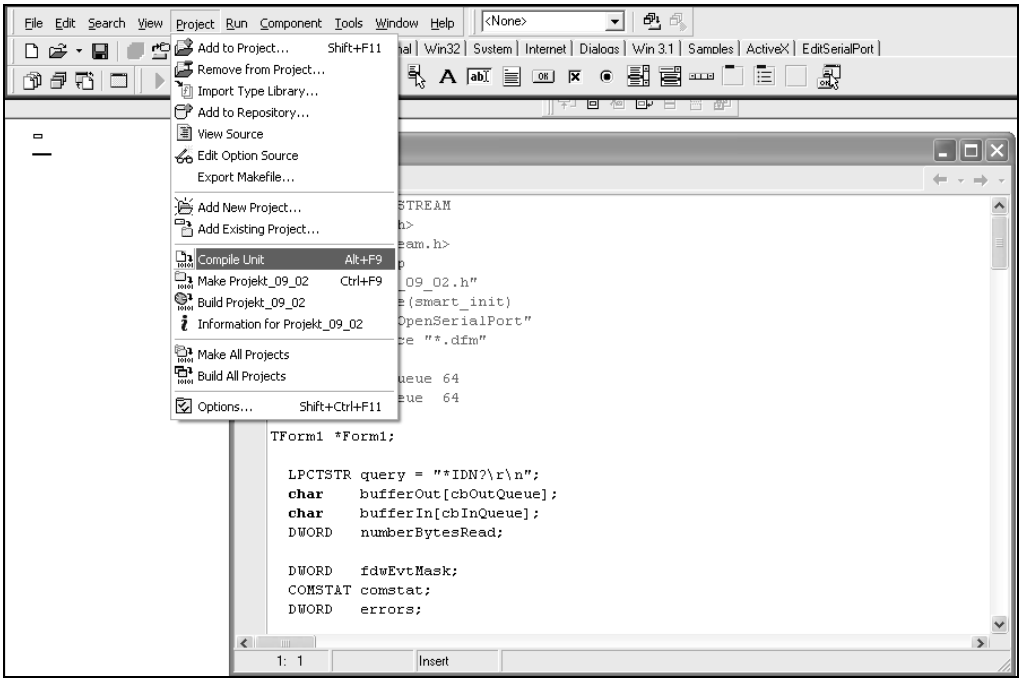
```
BOOL CloseHandle(HANDLE hCommDev)
```

We wszystkich przedstawionych powyżej funkcjach `hCommDev` w pełni identyfikuje dany port szeregowy, zawierając kompletną informację o tym, do którego łącza szeregowego będziemy wysyłać komunikaty. Ponieważ funkcje te mogą obsługiwać komunikaty wysyłane do wielu portów komunikacyjnych (jak również odbierane od wielu portów), zatem każdy otwarty i zainicjalizowany port szeregowy będzie identyfikowany właśnie za pomocą swojego własnego `hCommDev`. Nie należy przydzielać tego samego identyfikatora do dwóch różnych portów komunikacyjnych, tak samo jak nie należy z jednym portem kojarzyć dwóch różnych identyfikatorów.

Przy pisaniu aplikacji obsługujących łącze szeregowe należy koniecznie zamknąć port przed opuszczeniem programu. W razie korzystania z zegara systemowego przy obsłudze RS-a lub techniki programowania wielowątkowego trzeba pamiętać, że samo zamknięcie aplikacji nie powoduje automatycznego zamknięcia portu. Jego identyfikator dalej będzie przechowywany w pamięci.

W pewnych przypadkach aplikacja z niezamkniętym portem szeregowym może stać się programem rezydentnym i uniemożliwić powtórne otwarcie wybranego portu. Dobrym zwyczajem jest w pierwszej kolejności zaprojektowanie funkcji lub procedury obsługi zdarzenia zamykającego otwarty port. System operacyjny powinien być zawsze poinformowany o fakcie zamknięcia portu.

W praktyce zdarzają się jednak sytuacje, w których zamknięcie portu okaże się niemożliwe, np. z powodu jakiegoś błędu w algorytmie lub niewłaściwego sposobu wywołania danej funkcji. Mówimy wówczas, że program się załamał lub zawiesił. Ten problem powtarza się często w trakcie testowania programów komunikacyjnych. Nie należy wówczas od razu używać kombinacji klawiszy *Ctrl, Alt, Del*. W takich przypadkach wygodniej jest rozwinąć z głównego menu opcję *Project* oraz wybrać *Compile Unit*, tak jak pokazano to na rysunku 5.1. Nazwa działającej aplikacji powinna się pojawić na dolnym pasku zadań.



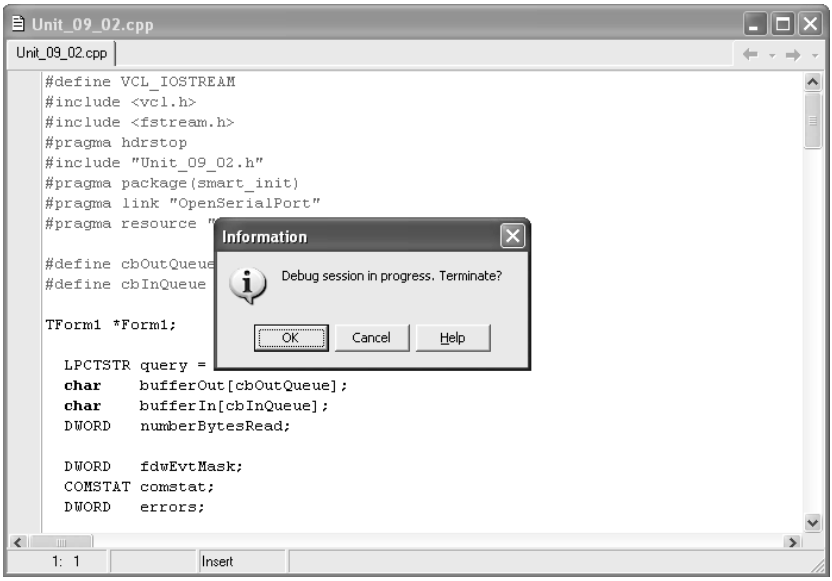
Rysunek 5.1. *Przykładowy sposób wstrzymywania działania aplikacji z otwartym portem szeregowym*

Po pojawieniu się informacji *Debug session in progress. Terminate?* (*Usuwanie sesji w toku. Zakończyć?*) (rysunek 5.2) należy nacisnąć przycisk *OK*.

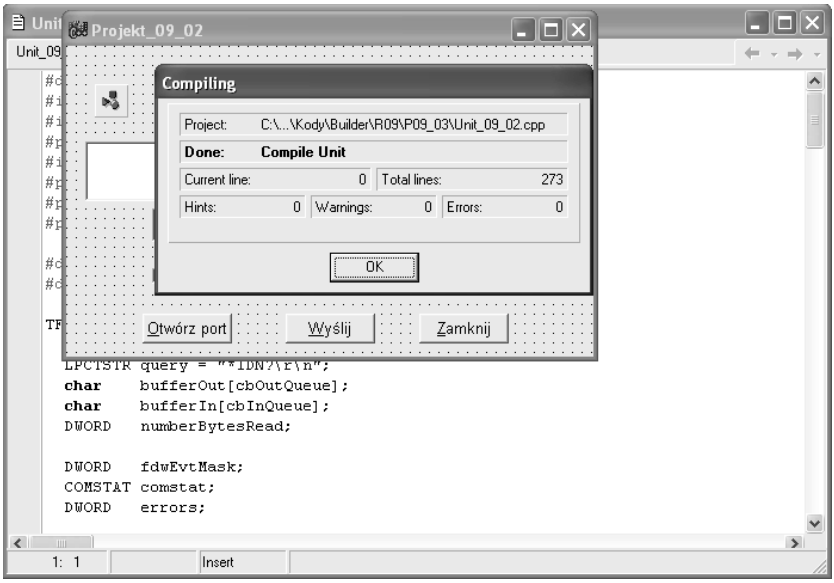
Po kolejnym komunikacie (rysunek 5.3) znów należy dokonać potwierdzenia.

Tak postępując, w większości przypadków odzyskamy program oraz odblokujemy łącze szeregowe. Sposób ten okaże się szczególnie przydatny przy kłopotach z aplikacją komunikacyjną korzystającą z komponentu typu `TTimer`, generującego zdarzenia w równych odstępach czasu.

Może oczywiście zdarzyć się sytuacja, w której nie będziemy w stanie powtórnie skompilować programu i samodzielnie prawidłowo zamknąć portu komunikacyjnego. Wówczas program należy usunąć z pamięci poleceniem menu *Run/Program Reset*.



Rysunek 5.2. *Okno dialogowe Debug session*



Rysunek 5.3. *Kompilacja projektu*

Testowanie portu szeregowego

Mając na uwadze wszystko, co powiedzieliśmy do tej pory, spróbujemy napisać w C++Builderze prosty program wykorzystujący przedstawione powyżej funkcje Windows API oraz niektóre zasoby struktury `DCB`.

Zadaniem naszej aplikacji będzie ustawienie wybranych parametrów danego portu szeregowego, otwarcie go oraz odczytanie nowych ustawień. W tym celu stwórzmy nową standardową aplikację (polecenie *File — New Application*). Niech jej formularz składa się z dwóch przycisków klasy `TButton`, pięciu komponentów klasy `TEdit` oraz pięciu `TLabel`.

Korzystając z inspektora obiektów (*Object Inspector*) oraz z karty własności (*Properties*), własność `Name` przycisku `Button1` zmieńmy na `CloseComm`, zaś jego własność `Caption` na *&Zamknij*. Podobnie własność `Name` przycisku `Button2` zmieńmy na `OpenComm`, zaś `Caption` na *&Otwórz port*. Własności `Caption` komponentów z klas `TLabel` zmieńmy odpowiednio na *Prędkość transmisji*, *Liczbę bitów danych*, *Parzystość*, *Bity stopu*, *Linia DTR*. Własności `Text` komponentów klasy `TEdit` wyczyśćmy.

Formularz naszej aplikacji, wyglądającej podobnie jak na rysunku 5.4, znajduje się na dołączonym CD w katalogu *\KODY\BUILDER\R05\P05_01*. Na listingu 5.1 pokazano kod głównego modułu omawianej aplikacji.

Rysunek 5.4. *Formularz główny projektu Projekt_05_01.bpr*

Listing 5.1. *Kod głównego modułu aplikacji testującej podstawowe parametry transmisji portu szeregowego*

```
#include <vcl.h>
#pragma hdrstop
#include "Unit_05_01.h"
#pragma package(smart_init)
#pragma resource "*.dfm"

TForm1 *Form1;

    HANDLE  hCommDev;  // identyfikator portu szeregowego
    // void *hCommDev;
    DCB     dcb;       // struktura kontroli portu
```

```
    LPCTSTR portName = "COM2";   // wskaźnik do nazwy portu
    // const char *portName  = "COM2";

    LPCTSTR sbuffer2 = "Uwaga!";
    LPCTSTR sbuffer1 = "Niewłaściwa nazwa portu lub port jest"
                       " aktywny.";
//---------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{

}
//----funkcja zamyka otwarty port szeregowy-----------------
BOOL __fastcall closeSerialPort(HANDLE hCommDev)
{
   if ((hCommDev == 0) || (hCommDev == INVALID_HANDLE_VALUE))
     return FALSE;
     else {
        CloseHandle(hCommDev);
        return TRUE;
     }
}
//-----zamknięcie portu i aplikacji--------------------------
void __fastcall TForm1::CloseCommClick(TObject *Sender)
{
     closeSerialPort(hCommDev);
     Application->Terminate();
}
//---otwarcie portu i ustawienie jego parametrów--------------
void __fastcall TForm1::OpenCommClick(TObject *Sender)
{
    hCommDev = CreateFile(portName, GENERIC_READ |
                          GENERIC_WRITE, 0, NULL,
                          OPEN_EXISTING, 0, NULL);

    if (hCommDev != INVALID_HANDLE_VALUE)
    // sprawdza, czy port jest otwarty prawidłowo
      {
        dcb.DCBlength = sizeof(dcb);  // aktualny rozmiar
                                      // struktury DCB
        GetCommState(hCommDev, &dcb); // udostępnienie aktualnych
                                      // parametrów DCB
        dcb.BaudRate = CBR_1200;      // prędkość transmisji
        dcb.fParity = TRUE;           // sprawdzanie parzystości
        dcb.Parity = NOPARITY;        // ustawienie parzystości
        dcb.StopBits = TWOSTOPBITS;   // bity stopu
        dcb.ByteSize = 7;             // bity danych
        dcb.fDtrControl = 1;          // np. kontrola linii DTR

        SetCommState(hCommDev, &dcb); // reinicjalizacja DCB

      }
      else
        {
         switch ((int)hCommDev)
           {
            case IE_BADID:
```

```
          // W przypadku błędnej identyfikacji portu
          // BADIDentify pokaż komunikat
               MessageBox(NULL, sbuffer1, sbuffer2, MB_OK);
             break;
           };
          }
//--sprawdzenie i wyświetlenie ustawionej prędkości------
    switch (dcb.BaudRate)
      {
        case CBR_9600:
            Edit1->Text = IntToStr(dcb.BaudRate);
            break;
        case CBR_1200:
            Edit1->Text = IntToStr(dcb.BaudRate);
            break;
        case CBR_300:
            Edit1->Text = IntToStr(dcb.BaudRate);
            break;
        case CBR_110:
            Edit1->Text = IntToStr(dcb.BaudRate);
            break;
      }
//--sprawdzenie i wyświetlenie ustawionych bitów danych-
    switch (dcb.ByteSize)
      {
        case 8:
            Edit2->Text = IntToStr(dcb.ByteSize);
            break;
        case 7:
            Edit2->Text = IntToStr(dcb.ByteSize);
            break;
        case 6:
            Edit2->Text = IntToStr(dcb.ByteSize);
            break;
        case 5:
            Edit2->Text = IntToStr(dcb.ByteSize);
            break;
      }
//--sprawdzenie i wyświetlenie ustawionej parzystości----
    switch (dcb.Parity)
      {
        case NOPARITY:
            Edit3->Text = "Brak";
            break;
        case ODDPARITY:
            Edit3->Text = "Nieparzysta";
            break;
        case EVENPARITY:
            Edit3->Text = "Parzysta";
            break;
        case MARKPARITY:
            Edit3->Text = "Znacznik: 1";
            break;
      }
//--sprawdzenie i wyświetlenie ustawionych bitów stopu---
    switch (dcb.StopBits)
```

```
      {
        case ONESTOPBIT:
            Edit4->Text = "1";
            break;
        case TWOSTOPBITS:
            Edit4->Text = "2";
            break;
        case ONE5STOPBITS:
            Edit4->Text = "1.5";
            break;
      }
//--sprawdzenie i wyświetlenie stanu linii DTR-----------
    switch (dcb.fDtrControl)
      {
        case DTR_CONTROL_DISABLE:
            Edit5->Text = "Nieaktywna";
            break;
        case DTR_CONTROL_ENABLE:
            Edit5->Text = "Aktywna";
            break;
        case DTR_CONTROL_HANDSHAKE:
            Edit5->Text = "Handshaking";
            break;
      }
}
//---------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender,
                                  TCloseAction &Action)
{
    Action=caFree;
}
//---------------------------------------------------------
```

Stworzyliśmy zatem bardzo prostą, wręcz „dydaktyczną" aplikację, ale taki właśnie był nasz cel. Możemy zauważyć, że obsługa tego programu sprowadza się do wywołania funkcji obsługi zdarzenia `OpenCommClick()`. Naciśnięcie przycisku *Otwórz port* powoduje automatyczne skonfigurowanie wybranego wcześniej portu szeregowego oraz odczytanie jego aktualnie wybranych ustawień. Dobrze byłoby, gdyby Czytelnik spróbował samodzielnie skonfigurować port z większą liczbą parametrów, a następnie je odczytał. Nabiera się przez to większej wprawy w manipulowaniu znacznikami struktury `DCB`. Zamknięcie portu i aplikacji nastąpi po wywołaniu funkcji obsługi zdarzenia `CloseCommClick()`, w której z kolei dokonujemy wywołania funkcji `CloseSerialPort()` zamykającej port szeregowy i aplikację. Przyglądając się kodowi funkcji obsługi zdarzenia `OpenCommClick()`, zauważymy, że tuż po wywołaniu `CreateFile()` zastosowaliśmy następującą instrukcję warunkową sprawdzającą, czy funkcja ta zwróciła prawidłowy identyfikator zadeklarowanego portu:

```
if (hCommDev != INVALID_HANDLE_VALUE)  {
   ...
  }
  else  {
     switch ((int)hCommDev)  {
```

```
        case IE_BADID: // W przypadku błędnej identyfikacji portu
                       // BADIDentify pokaż komunikat
        ...
        break;
      };
  }
```

Łatwo można się przekonać, że w przypadku błędnego przydzielenia identyfikatora dla portu COM*n*, funkcja `CreateFile()` zwraca wartość `INVALID_HANDLE_ VALUE` (niewłaściwa wartość identyfikatora) zdefiniowaną w Windows API. Jest to bardzo skuteczna metoda zabezpieczenia się przed próbą otwarcia nieistniejącego lub już otwartego portu (urządzenia). Zauważmy też, że aby odczytać aktualną wartość `hCommDev`, musieliśmy wymusić przekształcenie typów, używając operacji rzutowania `(int)hCommDev`. Każdy już się chyba przekonał, że identyfikator czy — jak kto woli — uchwyt typu `HANDLE` nie jest żadnym numerem bezpośrednio nadanym portowi komunikacyjnemu, lokalizuje jedynie unikalny obszar pamięci, do którego należy się odwołać, by uzyskać dostęp do danego urządzenia.

Raz otwartego portu komunikacyjnego nie można otworzyć powtórnie, podobnie jak nie uda się otworzyć już otwartego okna. Nie można też powtórnie skorzystać z obszaru pamięci, z którego właśnie korzystamy.

Jeżeli mimo wszystko port nie został otwarty prawidłowo, dobrze by było, gdyby aplikacja powiadomiła nas o tym fakcie. W tym celu można skorzystać z komunikatów Windows typu IE_ (ang. *Identify Error* — błąd identyfikacji portu) urządzenia lub jego ustawień. W tabeli 5.6 przedstawiono najczęściej otrzymywane od Windows komunikaty tego typu.

Tabela 5.6. *Najczęściej używane komunikaty błędnej identyfikacji ustawień portu szeregowego*

Identyfikacja komunikatu	Wartość	Znaczenie
`IE_BADID`	–1	Niewłaściwa identyfikacja urządzenia
`IE_BAUDRATE`	–12	Błędnie określona szybkość transmisji
`IE_BYTESIZE`	–11	Błędnie określona liczba bitów danych
`IE_DEFAULT`	–5	Niewłaściwie określone parametry domyślne urządzenia
`IE_HARDWARE`	–10	Odbiornik jest zablokowany
`IE_MEMORY`	–4	Niewłaściwie ustalony rozmiar buforów
`IE_NOPEN`	–3	Urządzenie nie jest otwarte do transmisji
`IE_OPEN`	–2	Urządzenie pozostaje otwarte

Struktura COMMPROP

W celu dokładniejszego zapoznania się z możliwościami testowania systemów komunikacyjnych dostępnych w Windows w tabeli 5.7 przedstawiono bardzo użyteczną strukturę oferowaną przez API. Zawarte w niej informacje mogą być wykorzystywane do pełnego odczytywania wszystkich istotnych parametrów interesującego nas łącza komunikacyjnego oraz usług potencjalnie przez nie oferowanych.

Tabela 5.7. *Zasoby struktury COMMPROP*

Typ	Element struktury	Znaczenie	Zawartość elementu, maska określająca włączony bit	
WORD	wPacketLength	Określa (w bajtach) rozmiar porcji pakietu danych	Należy odczytać, zależy też od typu sterownika	
WORD	wPacketVersion	Wersja struktury	Nr 2 w Win 9x, XP	
DWORD	dwServiceMask	Określenie maski bitowej wskazującej na typ aktualnie dostępnej usługi komunikacyjnej	SP_SERIALCOMM jest zawsze określone	
DWORD	dwReserved1	Zarezerwowane, nieużywane		
DWORD	dwMaxTxQueue	Maksymalny rozmiar wewnętrznego bufora wyjściowego nadajnika (w bajtach)	0 oznacza, że nie ustalono maksymalnej wartości	
DWORD	dwMaxRxQueue	Maksymalny rozmiar wewnętrznego bufora wejściowego odbiornika (w bajtach)	0 oznacza, że nie ustalono maksymalnej wartości	
DWORD	dwMaxBaud	Maksymalna dostępna prędkość transmisji w bitach na sekundę	BAUD_075	75 b/s
			BAUD_110	110 b/s
			BAUD_134_5	134.5 b/s
			BAUD_150	150 b/s
			BAUD_300	300 b/s
			BAUD_600	600 b/s
			BAUD_1200	1200 b/s
			BAUD_1800	1800 b/s
			BAUD_2400	2400 b/s
			BAUD_4800	4800 b/s
			BAUD_7200	7200 b/s
			BAUD_9600	9600 b/s
			BAUD_14400	14 400 b/s
			BAUD_19200	19 200 b/s
			BAUD_38400	38 400 b/s
			BAUD_56K	56K b/s
			BAUD_57600	57 600 b/s
			BAUD_115200	115 200 b/s
			BAUD_128K	128K b/s
			BAUD_USER	programowalne

Tabela 5.7. *Zasoby struktury COMMPROP — ciąg dalszy*

Typ	Element struktury	Znaczenie	Zawartość elementu, maska określająca włączony bit
DWORD	dwProvSubType	Typ usługi komunikacyjnej	PST_FAX — faks PST_LAT — protokół LAT (*Local — Area Transport*) PST_MODEM — modem PST_NETWORK_BRIDGE — niewyspecyfikowana sieć PST_PARALLELPORT — port równoległy PST_RS232 RS 232 PST_RS422 RS 422 PST_RS423 RS 423 PST_RS449 RS 449 PST_SCANNER — skaner PST_TCPIP_TELNET — protokół TCP/IP PST_UNSPECIFIED — brak specyfikacji PST_X25 — protokół X.25
DWORD	DwSettableParams	Specyfikacja maski bitowej identyfikującej parametry transmisji podlegające ewentualnym zmianom	SP_BAUD — prędkość SP_DATABITS — długość słowa danych SP_HANDSHAKING — kontrola przepływu danych SP_PARITY — parzystość SP_PARITY_CHECK — sprawdzanie parzystości SP_RLSD — sygnał RLSD SP_STOPBITS — bity stopu

W Windows API COMMPROP deklaruje się następująco:

```
typedef struct _COMMPROP {
        WORD wPacketLength;
        ...
} COMMPROP;
```

Powyższa deklaracja tworzy nowe słowo kluczowe typu COMMPROP (struktura).

Zbudujmy teraz aplikację, za pomocą której będziemy mogli selektywnie odczytywać stan poszczególnych masek bitowych udostępnianych przez COMMPROP.

Tabela 5.7. *Zasoby struktury COMMPROP — ciąg dalszy*

Typ	Element struktury	Znaczenie	Zawartość elementu, maska określająca włączony bit
DWORD	dwProvCapabilities	Określa maskę bitową identyfikującą rodzaj funkcji udostępnianych przez usługę komunikacyjną (dostarczyciela usługi)	PCF_16BITMODE — tryb 16-bitowy PCF_DTRDSR — kontrola DTR-DSR PCF_INTTIMEOUTS — czas przeterminowania PCF_PARITY_CHECK — sprawdzanie parzystości PCF_RLSD — kontrola RLSD PCF_RTSCTS — kontrola RTS-CTS PCF_SETXCHAR — możliwość użycia protokołu XON/XOFF PCF_SPECIALCHARS — specjalny znak PCF_TOTALTIMEOUTS — kontrola czasu przeterminowania transmisji PCF_XONXOFF — podtrzymanie protokołu XON-XOFF
DWORD	dwSettableBaud	Specyfikacja maski bitowej umożliwiającej ustawienie prędkości transmisji	Tak samo jak w dwMaxBaud
WORD	wSettableData	Specyfikacja maski bitowej identyfikującej możliwe do użycia długości słowa danych	DATABITS_5 DATABITS_6 DATABITS_7 DATABITS_8 DATABITS_16 DATABITS_16X szczególna długość słowa danych
DWORD	dwCurrentTxQueue	Aktualny maksymalny rozmiar wewnętrznego bufora wyjściowego nadajnika (w bajtach)	0 oznacza, że wartość ta nie jest aktualnie dostępna
WORD	wSettableStopParity	Specyfikacja maski bitowej identyfikującej możliwe do użycia wartości bitów stopu i kontroli parzystości	STOPBITS_10 — 1 bit stopu STOPBITS_15 — 1,5 bitu STOPBITS_20 — 2 bity PARITY_NONE — brak PARITY_ODD — nieparzysta PARITY_EVEN — parzysta PARITY_MARK — 1 PARITY_SPACE — 0
DWORD	dwCurrentRxQueue	Aktualny maksymalny rozmiar wewnętrznego bufora wejściowego nadajnika (w bajtach)	0 oznacza, że wartość ta nie jest aktualnie dostępna

Tabela 5.7. *Zasoby struktury COMMPROP — ciąg dalszy*

Typ	Element struktury	Znaczenie	Zawartość elementu, maska określająca włączony bit
DWORD	dwProvSpec1	Specyfikacja formatu danych wymaganych przez daną usługę komunikacyjną	W zależności od dwProvSubType aplikacje powinny ignorować ten człon, chyba że zawierają szczegółowe informacje odnośnie do formatu danych wymaganych przez dostarczyciela usługi
DWORD	dwProvSpec2	Jak wyżej	
WCHAR	wcProvChar[1]	Jak wyżej	Jeżeli dwProvSubType przypisano PST_MODEM, musi nastąpić odwołanie do struktur MODEMDEVCAPS oraz MODEMSETTINGS[2]; dwProvSpec1 i dwProvSpec2 nie są wówczas używane

Wykorzystamy tu znany nam już proces maskowania z użyciem operatora iloczynu bitowego & (bitowe i). Program będzie odczytywał wartość wybranego elementu struktury, a następnie poprzez wybranie kolejnych masek będzie selektywnie sprawdzał, czy włączone są konkretne bity odpowiedzialne za pewne parametry transmisji. Omawiany projekt znajduje się na dołączonym CD w katalogu *\KODY\BUILDER\R05\P05_02*. Do testowania wybierzmy elementy: dwSettableParams, w XP reprezentowany na 32 bitach, oraz wSettableData i wSettableStopParity, reprezentowane na 16 bitach. Zastosujemy nieco odbiegający od przedstawionego wcześniej projekt formularza. Składać się on będzie z dwóch dobrze nam już znanych przycisków reprezentujących zdarzenia polegające na otwarciu oraz zamknięciu portu. Zastosowano ponadto dwa komponenty klasy TTrackBar, dwa TEdit oraz dwa typu TLabel, tak jak pokazuje to rysunek 5.5. Obsługa zdarzenia TrackBar1Change() polega na wybraniu interesującego nas elementu struktury COMMPROP oraz odczytaniu jego aktualnej wartości. Jeżeli zechcemy sprawdzić, czy włączony jest konkretny bit reprezentujący wybrany atrybut transmisji przechowywany w danym elemencie struktury, wystarczy przesunąć wskaźnik uruchamiający funkcję obsługi zdarzenia TrackBar2Change().

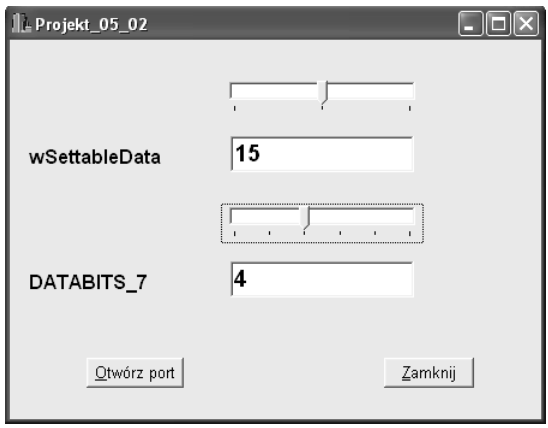
Funkcja GetCommProperties()

Funkcją, która zwraca aktualne właściwości portu komunikacyjnego identyfikowanego przez hCommDev, będzie:

```
BOOL GetCommProperties(HANDLE hCommDev, LPCOMMPROP lpCommProp);
```

lpCommProp jest wskaźnikiem do struktury COMMPROP, której format danych w ogólnym przypadku należy najpierw zainicjalizować, po uprzednim wpisaniu do pola wPacketLength aktualnego rozmiaru struktury:

[2] Miłośnikom modemów specyfikację tych struktur prezentujemy w dodatku B.



Rysunek 5.5. *Formularz główny uruchomionego projektu Projekt_R05_02.bpr*

```
commprop.wPacketLength = sizeof(COMMPROP);
//...
CommProp.dwProvSpec1 = COMMPROP_INITIALIZED;
```

Informacje tam zawarte mogą być pomocne przy odwoływaniu się do rodziny funkcji `SetCommState()`, `SetCommTimeouts()` lub `SetupComm()`.

Na listingu 5.2 pokazano kod głównego modułu omawianej aplikacji.

Listing 5.2. *Zawartość modułu Unit_R05_02.cpp*

```
#include <vcl.h>
#pragma hdrstop
#include "Unit_05_02.h"
#pragma package(smart_init)
#pragma resource "*.dfm"

TForm1 *Form1;

  HANDLE   hCommDev;          // identyfikator portu
  COMMPROP commprop;          // właściwości portu
  LPCTSTR portName = "COM2";  // wskaźnik do nazwy portu
                              // szeregowego

  LPCTSTR sbuffer1 = "Niewłaściwa nazwa portu lub port jest"
                     " aktywny.";
//---------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//-------funkcja zamyka otwarty port szeregowy------------------
BOOL __fastcall closeSerialPort(HANDLE hCommDev)
{
    if ((hCommDev == 0) || (hCommDev == INVALID_HANDLE_VALUE))
      return FALSE;
      else {
         CloseHandle(hCommDev);
         return TRUE;
```

```
      }
}
//-------zamknięcie portu i aplikacji----------------------------
void __fastcall TForm1::CloseCommClick(TObject *Sender)
{
     closeSerialPort(hCommDev);
     Application->Terminate();
}
//----otwarcie portu i ustawienie jego parametrów---------------
void __fastcall TForm1::OpenCommClick(TObject *Sender)
{
    hCommDev = CreateFile(portName, GENERIC_READ |
                          GENERIC_WRITE, 0, NULL,
                          OPEN_EXISTING, 0, NULL);

    if (hCommDev != INVALID_HANDLE_VALUE) {
     // sprawdza, czy port jest otwarty prawidłowo
       commprop.wPacketLength = sizeof(COMMPROP);
       commprop.dwProvSpec1 = COMMPROP_INITIALIZED;
      // inicjalizuje format danych usługi.
      // Port szeregowy jest zawsze dostępny
       memset(&commprop, 0, sizeof(COMMPROP));
       GetCommProperties(hCommDev, &commprop);
      }
      else {
         switch ((int)hCommDev) {
            case IE_BADID:
               ShowMessage(sbuffer1);
            break;
         };
      }
}
//------wybrane maski bitowe-------------------------------------
void __fastcall TForm1::TrackBar1Change(TObject *Sender)
{
  switch (TrackBar1->Position) {
     case 1: {
       TrackBar2->Max = 7;
       Label1->Caption = "dwSettableParams";
       Edit1->Text=IntToStr(commprop.dwSettableParams);
       break;
     }
     case 2: {
       TrackBar2->Max = 6;
       Label1->Caption = "wSettableData";
       Edit1->Text=IntToStr(commprop.wSettableData);
       break;
     }
     case 3: {
       TrackBar2->Max = 8;
       Label1->Caption = "wSettableStopParity";
       Edit1->Text=IntToStr(commprop.wSettableStopParity);
       break;
     }
  } // koniec switch
}
```

```
//---------------zawartość maski---------------------------
void __fastcall TForm1::TrackBar2Change(TObject *Sender)
{
  if (TrackBar1->Position == 1) {
    switch (TrackBar2->Position) {
      case 1: {
        Label2->Caption = "SP_PARITY";
        Edit2->Text=IntToStr(commprop.dwSettableParams & SP_PARITY);
        break;
      }
      case 2: {
        Label2->Caption = "SP_BAUD";
        Edit2->Text=IntToStr(commprop.dwSettableParams & SP_BAUD);
        break;
      }
      case 3: {
        Label2->Caption = "SP_DATABITS";
        Edit2->Text=IntToStr(commprop.dwSettableParams &
                             SP_DATABITS);
        break;
      }
      case 4: {
        Label2->Caption = "SP_STOPBITS";
        Edit2->Text=IntToStr(commprop.dwSettableParams &
                             SP_STOPBITS);
        break;
      }
      case 5: {
        Label2->Caption = "SP_HANDSHAKING";
        Edit2->Text=IntToStr(commprop.dwSettableParams &
                             SP_HANDSHAKING);
        break;
      }
      case 6: {
        Label2->Caption = "SP_PARITY_CHECK";
        Edit2->Text=IntToStr(commprop.dwSettableParams &
                             SP_PARITY_CHECK);
        break;
      }
      case 7: {
        Label2->Caption = "SP_RLSD";
        Edit2->Text=IntToStr(commprop.dwSettableParams & SP_RLSD);
        break;
      }
    } // koniec switch
  } // koniec if

  if (TrackBar1->Position == 2) {
    switch (TrackBar2->Position) {
      case 1: {
        Label2->Caption = "DATABITS_5";
        Edit2->Text=IntToStr(commprop.wSettableData & DATABITS_5);
        break;
      }
      case 2: {
        Label2->Caption = "DATABITS_6";
```

```
      Edit2->Text=IntToStr(commprop.wSettableData & DATABITS_6);
      break;
    }
    case 3: {
      Label2->Caption = "DATABITS_7";
      Edit2->Text=IntToStr(commprop.wSettableData & DATABITS_7);
      break;
    }
    case 4: {
      Label2->Caption = "DATABITS_8";
      Edit2->Text=IntToStr(commprop.wSettableData & DATABITS_8);
      break;
    }
    case 5: {
      Label2->Caption = "DATABITS_16";
      Edit2->Text=IntToStr(commprop.wSettableData & DATABITS_16);
      break;
    }
    case 6: {
      Label2->Caption = "DATABITS_16X";
      Edit2->Text=IntToStr(commprop.wSettableData &
                           DATABITS_16X);
      break;
    }
  } // koniec switch
} // koniec if

if (TrackBar1->Position == 3) {
  switch (TrackBar2->Position) {
    case 1: {
      Label2->Caption = "STOPBITS_10";
      Edit2->Text=IntToStr(commprop.wSettableStopParity &
                           STOPBITS_10);
      break;
    }
    case 2: {
      Label2->Caption = "STOPBITS_15";
      Edit2->Text=IntToStr(commprop.wSettableStopParity &
                           STOPBITS_15);
      break;
    }
    case 3: {
      Label2->Caption = "STOPBITS_20";
      Edit2->Text=IntToStr(commprop.wSettableStopParity &
                           STOPBITS_20);
      break;
    }
    case 4: {
      Label2->Caption = "PARITY_NONE";
      Edit2->Text=IntToStr(commprop.wSettableStopParity &
                           PARITY_NONE);
      break;
    }
    case 5: {
      Label2->Caption = "PARITY_ODD";
```

```
        Edit2->Text=IntToStr(commprop.wSettableStopParity &
                             PARITY_ODD);
        break;
      }
      case 6: {
        Label2->Caption = "PARITY_EVEN";
        Edit2->Text=IntToStr(commprop.wSettableStopParity &
                             PARITY_EVEN);
        break;
      }
      case 7: {
        Label2->Caption = "PARITY_MARK";
        Edit2->Text=IntToStr(commprop.wSettableStopParity &
                             PARITY_MARK);
        break;
      }
      case 8: {
        Label2->Caption = "PARITY_SPACE";
        Edit2->Text=IntToStr(commprop.wSettableStopParity &
                             PARITY_SPACE);
        break;
      }
    } // koniec switch
  } // koniec if
}
//---------------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender,
                                  TCloseAction &Action)
{
    Action=caFree;
}
//---------------------------------------------------------------
```

Dla przykładu rozpatrzmy parametr `dwSettableParams` typu `DWORD`, w XP reprezentowany na 32 bitach. Odczytując odpowiednią wartość, przekonaliśmy się, że cała zawarta tam informacja zapisana jest na 7 pierwszych bitach `dwSettableParams`.

Użyliśmy operatora `&`, aby sprawdzić, czy włączone są poszczególne bity reprezentujące atrybuty związane z konkretnymi parametrami komunikacyjnymi. Patrząc na wartości w postaci binarnej, łatwo zorientujemy się, jaki jest aktualny stan logiczny poszczególnych bitów zawartych w tej zmiennej i za co są one odpowiedzialne, tak jak pokazano to w zawartości tabeli 5.8.

W analogiczny sposób możemy przetestować wszystkie maski bitowe udostępniane przez `COMMPROP`, odpowiadające właściwym pozycjom konkretnych bitów. Jeżeli jako rezultat iloczynu bitowego wartości elementu struktury z maską określającą włączony bit otrzymamy wartość 0, oznaczać to będzie, że testowany bit jest wyłączony i dany parametr komunikacyjny nie jest aktualnie dostępny. Lepiej znający temat Czytelnicy zapewne już zorientowali się, jakie niespodzianki oferuje nam ta struktura. Manipulowanie bitami jest tu sprawą dobrania odpowiednich operatorów przesuwania, maskowania i dopełniania. Można np. skasować bity `SP_PARITY` i `SP_BAUD` w elemencie `dwSettableParams`:

```
CommProp.dwSettableParams &= ~(SP_PARITY | SP_BAUD);
```

Tabela 5.8. *Maski bitowe elementu dwSettableParams struktury COMMPROP*

	Wartość	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
dwSettableParams	**127**	**0**	***1***	***1***	***1***	***1***	***1***	***1***	***1***
Maska	**Rezultat maskowania**								
SP_PARITY	1	0	0	0	0	0	0	0	*1*
SP_BAUD	2	0	0	0	0	0	0	*1*	0
SP_DATABITS	4	0	0	0	0	0	*1*	0	0
SP_STOPBITS	8	0	0	0	0	*1*	0	0	0
SP_HANDSHAKING	16	0	0	0	*1*	0	0	0	0
SP_PARITY_CHECK	32	0	0	*1*	0	0	0	0	0
SP_RLSD	64	0	*1*	0	0	0	0	0	0

Podobnie w jakimś fragmencie aplikacji można zastosować warunek:

```
if ((CommProp.dwSettableParams & (SP_PARITY | SP_BAUD)) == 0)  {
   ...
}
```

który będzie prawdziwy, gdy oba bity będą skasowane. Jednak osobom mniej zaawansowanym w operacjach bitowych odradzałbym jakiekolwiek próby ingerowania w zawartość COMMPROP.

Struktura COMMCONFIG

Struktura COMMCONFIG zawiera informacje o stanie konfiguracji danego urządzenia komunikacyjnego. Tabela 5.9 prezentuje jej zasoby.

Windows API COMMCONFIG deklaruje następująco:

```
typedef struct _COMM_CONFIG {
        DWORD dwSize;
        ...
} COMMCONFIG, *LPCOMMCONFIG;
```

Powyższa deklaracja tworzy dwa nowe słowa kluczowe typu COMMCONFIG (struktura) oraz LPCOMMCONFIG (wskaźnik do struktury).

Funkcje GetCommConfig() i SetCommConfig()

Aktualną konfigurację łącza komunikacyjnego odczytamy, korzystając z funkcji API:

```
BOOL GetCommConfig(HANDLE hCommDev, LPCOMMCONFIG lpCC,
                   LPDWORD lpdwSize);
```

gdzie lpCC wskazuje na strukturę COMMCONFIG, zaś lpdwSize jest wskaźnikiem do zmiennej określającej rozmiar struktury.

Tabela 5.9. *Specyfikacja struktury COMMCONFIG*

Typ	Element struktury	Znaczenie	Zawartość
DWORD	dwSize	Rozmiar struktury w bajtach	Należy wpisać
WORD	wVersion	Wersja struktury	Należy odczytać
WORD	wReserved	Zarezerwowane	
DCB	dcb	Struktura kontroli portu szeregowego	Patrz DCB
DWORD	dwProviderSubType	Identyfikacja typu dostarczanej usługi komunikacyjnej, a tym samym wymaganego formatu danych	Patrz COMMPROP
DWORD	dwProviderOffset	Określenie offsetu dla danych wymaganych przez dostarczyciela usługi komunikacyjnej; offset (tzw. przesunięcie) określony jest zwykle w stosunku do początku struktury	0, jeżeli nie określono typu danych
DWORD	dwProviderSize	Rozmiar danych (w bajtach) wymaganych przez usługę komunikacyjną (dostarczyciela usługi)	Zależnie od typu usługi
WCHAR	wcProviderData[1]	Dane dostarczane wraz z usługą (ponieważ przewidywane jest w przyszłości uzupełnienie struktury, aplikacja powinna używać dwProviderOffset w celu określenia położenia wcProviderData)	Jeżeli ustalono typ usługi: PST_RS232 lub PST_PARALLELPORT, człon ten jest pomijany; jeżeli ustalono PST_MODEM, należy odwołać się do MODEMSETTINGS

Bieżącą konfigurację portu komunikacyjnego zapiszemy za pomocą:

```
BOOL SetCommConfig(HANDLE hCommDev, LPBYTE lpCC, DWORD dwSize);
```

gdzie lpCC jest wskaźnikiem do COMMCONFIG, zaś dwSize określa (w bajtach) rozmiar struktury wskazywanej przez lpCC. Przed przekazaniem tej struktury jako parametru należy do elementu dwSize wpisać wartość równą sizeof(COMMCONFIG).

Funkcja CommConfigDialog()

Funkcja wyświetla okno dialogowe zawierające aktualne ustawienia parametrów portu szeregowego.

```
BOOL CommConfigDialog(LPTSTR lpszName,  HWND hWnd,
                      LPCOMMCONFIG lpCC);
```

lpszName jest wskaźnikiem do łańcucha znaków określającego nazwę portu, hWnd jest identyfikatorem właściciela aktualnie wyświetlanego okna dialogowego, a lpCC wskazuje na strukturę COMMCONFIG. Użycie tej funkcji, np. w kontekście obsługi wybranego zdarzenia, może wyglądać następująco:

```
//---------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
```

```
    LPCOMMCONFIG commconfig;
    CommConfigDialog("COM2", NULL, commconfig);
}
//---------------------------------------------------------------
```