Laboratorium Architektury Komputerów

Ćwiczenie 2

Przetwarzanie tekstów z wykorzystaniem różnych trybów adresowania

---

Reprezentacja tekstu w pamięci komputera

Początkowo komputery używane były do obliczeń numerycznych. Okazało się jednak, że doskonale nadają się także do edycji i przetwarzania tekstów. Wyłoniła się więc konieczność ustalenia w jakiej formie mają być przechowywane w komputerze znaki używane w tekstach. Ponieważ w komunikacji dalekopisowej (telegraficznej) ustalono wcześniej standardy kodowania znaków używanych w tekstach, więc sięgnięto najpierw do tych standardów. W wyniku różnych zmian i ulepszeń około roku 1968 w USA ustalił się sposób kodowania znaków znany jako kod ASCII (ang. American Standard Code for Information Interchange). Początkowo w kodzie ASCII każdemu znakowi przyporządkowano unikatowy 7-bitowy ciąg zer i jedynek, zaś ósmy bit służył do celów kontrolnych. Wkrótce zrezygnowano z bitu kontrolnego, co pozwoliło na rozszerzenie podstawowego kodu ASCII o nowe znaki, używane w alfabetach narodowych (głównie krajów Europy Zachodniej).

Ponieważ posługiwanie się kodami złożonymi z zer i jedynek jest kłopotliwe, w programach komputerowych kody ASCII poszczególnych znaków zapisuje się w postaci liczb dziesiętnych lub szesnastkowych. Znaki o kodach od 0 do 127 przyjęto nazywać *podstawowym kodem ASCII*, zaś znaki o kodach 128 do 255 *rozszerzonym kodem ASCII*. Podstawowy kod ASCII podano w dwóch tablicach: pierwsza tablica zawiera znaki sterujące o kodach 0 ÷ 31 i 127, druga tablica zawiera litery, cyfry i inne znaki.

Znak	Bin	Dec	Hex	Skrót
Null	0000 0000	0	00	NUL
Start Of Heading	0000 0001	1	01	SOH
Start of Text	0000 0010	2	02	STX
End of Text	0000 0011	3	03	ETX
End of Transmission	0000 0100	4	04	EOT
Enquiry	0000 0101	5	05	ENQ
Acknowledge	0000 0110	6	06	ACK
Bell	0000 0111	7	07	BEL
Back-space	0000 1000	8	08	BS
Horizontal Tab	0000 1001	9	09	HT
Line Feed	0000 1010	**10**	**0A**	**LF**
Vertical Tab	0000 1011	11	0B	VT
Form Feed	0000 1100	12	0C	FF
Carriage Return	0000 1101	**13**	**0D**	**CR**
Shift Out	0000 1110	14	0E	SO
Shift In	0000 1111	15	0F	SI

Data Link Escape	0001 0000	16	10	DLE
Device Control 1 (XON)	0001 0001	17	11	DC1
Device Control 2	0001 0010	18	12	DC2
Device Control 3 (XOFF)	0001 0011	19	13	DC3
Device Control 4	0001 0100	20	14	DC4
Negative Acknowledge	0001 0101	21	15	NAK
Synchronous Idle	0001 0110	22	16	SYN
End of Transmission Block	0001 0111	23	17	ETB
Cancel	0001 1000	24	18	CAN
End of Medium	0001 1001	25	19	EM
Substitute	0001 1010	26	1A	SUB
Escape	0001 1011	27	1B	ESC
File Separator	0001 1100	28	1C	FS
Group Separator	0001 1101	29	1D	GS
Record Separator	0001 1110	30	1E	RS
Unit Separator	0001 1111	31	1F	US
Delete	0111 1111	127	7F	DEL

Kody od 0 do 31 oraz kod 127 zostały przeznaczone do sterowania komunikacją dalekopisową. Niektóre z nich pozostały w informatyce, chociaż zatraciły swoje pierwotne znaczenie, inne zaś są nieużywane. Do tej grupy należy m.in. znak powrotu karetki (CR) o kodzie 0DH (dziesiętnie 13). W komunikacji dalekopisowej kod ten powodował przesunięcie wałka z papierem na skrajną lewą pozycję. W komputerze jest często interpretowany jako kod powodujący przesunięcie kursora do lewej krawędzi ekranu. Bardzo często używany jest także znak nowej linii (LF) o kodzie 0AH (dziesiętnie 10).

Znak	**Bin**	**Dec**	**Hex**
Spacja	0010 0000	32	20
!	0010 0001	33	21
"	0010 0010	34	22
#	0010 0011	35	23
$	0010 0100	36	24
%	0010 0101	37	25
&	0010 0110	38	26
'	0010 0111	39	27
(	0010 1000	40	28
)	0010 1001	41	29
*	0010 1010	42	2A
+	0010 1011	43	2B
,	0010 1100	44	2C
-	0010 1101	45	2D
.	0010 1110	46	2E
/	0010 1111	47	2F
0	0011 0000	48	30
1	0011 0001	49	31
2	0011 0010	50	32
3	0011 0011	51	33
4	0011 0100	52	34
5	0011 0101	53	35
6	0011 0110	54	36
7	0011 0111	55	37
8	0011 1000	56	38
9	0011 1001	57	39
:	0011 1010	58	3A
;	0011 1011	59	3B
<	0011 1100	60	3C
=	0011 1101	61	3D
>	0011 1110	62	3E
?	0011 1111	63	3F
@	0100 0000	64	40
A	0100 0001	65	41
B	0100 0010	66	42
C	0100 0011	67	43
D	0100 0100	68	44
E	0100 0101	69	45
F	0100 0110	70	46
G	0100 0117	71	47
H	0100 1000	72	48

I	0100 1001	73	49
J	0100 1010	74	4A
K	0100 1011	75	4B
L	0100 1100	76	4C
M	0100 1101	77	4D
N	0100 1110	78	4E
O	0100 1111	79	4F
P	0101 0000	80	50
Q	0101 0001	81	51
R	0101 0010	82	52
S	0101 0011	83	53
T	0101 0100	84	54
U	0101 0101	85	55
V	0101 0110	86	56
W	0101 0111	87	57
X	0101 1000	88	58
Y	0101 1001	89	59
Z	0101 1010	90	5A
[	0101 1011	91	5B
\	0101 1100	92	5C
]	0101 1101	93	5D
^	0101 1110	94	5E
_	0101 1111	95	5F
`	0110 0000	96	60
a	0110 0001	97	61
b	0110 0010	98	62
c	0110 0011	99	63
d	0110 0100	100	64
e	0110 0101	101	65
f	0110 0110	102	66
g	0110 0117	103	67
h	0110 1000	104	68
i	0110 1001	105	69
j	0110 1010	106	6A
k	0110 1011	107	6B
l	0110 1100	108	6C
m	0110 1101	109	6D
n	0110 1110	110	6E
o	0110 1111	111	6F
p	0111 0000	112	70
q	0111 0001	113	71
r	0111 0010	114	72
s	0111 0011	115	73
t	0111 0100	116	74
u	0111 0101	117	75
v	0111 0110	118	76
w	0111 0111	119	77

x	0111 1000	120	78
y	0111 1001	121	79
z	0111 1010	122	7A
{	0111 1011	123	7B
|	0111 1100	124	7C
}	0111 1101	125	7D
~	0111 1110	126	7E
Delete	0111 1111	127	7F

Problem znaków narodowych

Z chwilą szerszego rozpowszechnienia się komputerów osobistych w wielu krajach wyłonił się problem kodowania znaków narodowych. Podstawowy kod ASCII zawiera bowiem jedynie znaki alfabetu łacińskiego (26 małych i 26 wielkich liter). Rozszerzenie kodu ASCII pozwoliło stosunkowo łatwo odwzorować znaki narodowe wielu alfabetów krajów Europy Zachodniej. Podobne działania podjęto także w odniesieniu do alfabetu języka polskiego. Działania te były prowadzone w sposób nieskoordynowany. Z jednej polscy producenci oprogramowania stosowali kilkanaście sposobów kodowania, z których najbardziej znany był kod Mazovia. Jednocześnie firma Microsoft wprowadziła standard kodowania znany jako Latin 2, a po wprowadzeniu systemu Windows zastąpiła go standardem Windows 1250. Dodatkowo jeszcze organizacja ISO (ang. International Organization for Standardization) wprowadziła własny standard (zgodny z polską normą) znany jako ISO 8859-2, który jest obecnie często stosowany w Internecie. Podana niżej tablica zawiera kody liter specyficznych dla języka polskiego w różnych standardach kodowania.

Znak	Mazovia	Latin 2	Windows 1250	ISO 8859-2	Unicode	UTF–8
ą	134 (86H)	165 (A5H)	185 (B9H)	177 (B1H)	(0105H)	C4H 85H
Ą	143 (8FH)	164 (A4H)	165 (A5H)	161 (A1H)	(0104H)	C4H 84H
ć	141 (8DH)	134 (86H)	230 (E6H)	230 (E6H)	(0107H)	C4H 87H
Ć	149 (95H)	143 (8FH)	198 (C6H)	198 (C6H)	(0106H)	C4H 86H
ę	145 (91H)	169 (A9H)	234 (EAH)	234 (EAH)	(0119H)	C4H 99H
Ę	144 (90H)	168 (A8H)	202 (CAH)	202 (CAH)	(0118H)	C4H 98H
ł	146 (92H)	136 (88H)	179 (B3H)	179 (B3H)	(0142H)	C5H 82H
Ł	156 (9CH)	157 (9DH)	163 (A3H)	163 (A3H)	(0141H)	C5H 81H
ń	164 (A4H)	228 (E4H)	241 (F1H)	241 (F1H)	(0144H)	C5H 84H
Ń	165 (A5H)	227 (E3H)	209 (D1H)	209 (D1H)	(0143H)	C5H 83H
ó	162 (A2H)	162 (A2H)	243 (F3H)	243 (F3H)	(00F3H)	C3H B3H
Ó	163 (A3H)	224 (E0H)	211 (D3H)	211 (D3H)	(00D3H)	C3H 93H
ś	158 (9EH)	152 (98H)	156 (9CH)	182 (B6H)	(015BH)	C5H 9BH
Ś	152 (98H)	151 (97H)	140 (8CH)	166 (A6H)	(015AH)	C5H 9AH
ź	166 (A6H)	171(ABH)	159 (9FH)	188 (BCH)	(017AH)	C5H BAH
Ź	160 (A0H)	141 (8DH)	143 (8FH)	172 (ACH)	(0179H)	C5H B9H
ż	167 (A7H)	190 (BEH)	191 (BFH)	191 (BFH)	(017CH)	C5H BCH
Ż	161 (A1H)	189 (BDH)	175 (AFH)	175 (AFH)	(017BH)	C5H BBH

Zadanie 2.1.: zmodyfikować dane programu przykładowego, który został podany w instrukcji ćw. 1 (str. 7) w taki sposób, by wszystkie litery tekstu powitalnego były wyświetlane poprawnie. *Wskazówka*: zastąpić litery specyficzne dla alfabetu języka polskiego podane w postaci zwykłych liter przez odpowiednie kody wyrażone w postaci liczbowej (dziesiętnej lub szesnastkowej).

Uniwersalny zestaw znaków

Kodowanie znaków za pomocą ośmiu bitów ogranicza liczbę różnych kodów do 256. Z pewnością nie wystarczy to do kodowania liter alfabetów europejskich, nie mówiąc już o alfabetach krajów dalekiego wschodu. Z tego względu od wielu lat prowadzone są prace na stworzeniem kodów obejmujących alfabety i inne znaki używane na całym świecie.

Prace nad standaryzacją zestawu znaków używanych w alfabetach narodowych podjęto na początku lat dziewięćdziesiątych ubiegłego stulecia. Początkowo prace prowadzone były niezależnie przez organizację ISO (International Organization for Standardization), jak również w ramach projektu *Unicode*, finansowanego przez konsorcjum czołowych producentów oprogramowania w USA. Około roku 1991 prace w obu instytucjach zostały skoordynowane, aczkolwiek dokumenty publikowane są niezależnie. Ustalono jednak, że tablice kodów standardu Unicode i standardu ISO 10646 są kompatybilne, a w wszelkie dalsze rozszerzenia są dokładnie uzgadniane.

Standard międzynarodowy ISO 10646 definiuje *Uniwersalny Zestaw Znaków* USC – ang. Universal Character Set. Standard zawiera znaki potrzebne do reprezentacji tekstów praktycznie we wszystkich znanych językach. Obejmuje nie tylko znaki alfabetu łacińskiego, greki, cyrylicy, arabskiego, ale także znaki chińskie, japońskie i wiele innych. Wszystko to dotyczy także standardu Unicode, który można uważać za implementację normy ISO 10646. Dla wygody dalszego opisu omawiany zestaw znaków będziemy określać terminem *Unicode* lub *Unikod*.

Unicode został przyjęty jako domyślny standard kodowania dla języków HTML i XML, stosowany jest w wielu systemach operacyjnych i językach programowania takich jak Java czy C#, a ponadto używany jest w nowych protokołach Internetu. Stanowi to podstawę do tworzenia oprogramowania o charakterze globalnym, dostępnego w wielu krajach niezależnie od używanego języka.

Podstawą systemu Unicode jest przypisanie każdemu znakowi wartości liczbowej określanej jako *punkt kodowy* (ang. code point), przy czym dodatkowo każdemu znakowi przyporządkowana jest także nazwa. Przykładowo, litera "ą" ma przypisany kod 0105 (zapis szesnastkowy), a oficjalna nazwa brzmi „LATIN SMALL LETTER A WITH OGONEK”.

Unikod nie określa kształtu znaku: ta sama litera "ą" może być drukowana w różnych postaciach, w zależności od użytego fontu (kroju): ą ą ą ą ą ... Innymi słowy, znak identyfikowany przez punkt kodowy jest jednostką abstrakcyjną, taką jak ww. „LATIN SMALL LETTER A WITH OGONEK”. Wizualną reprezentacją znaku, wyświetloną na ekranie lub wydrukowaną na papierze jest *glif* — Unikod nie definiuje wyglądu glifów, czyli nie określa precyzyjnie kształtu, rozmiaru i orientacji znaków wyświetlanych na ekranie. Z kolei repertuar glifów tworzy font (krój).

Przypuszcza się, że liczba różnych znaków, które używane są na świecie, wynosi ponad milion — wynika stąd konieczność przyjęcia sposobu kodowania tych znaków wykorzystujących co najmniej 21 bitów. Kierując się tym oszacowaniem, w standardzie Unikod udostępniono 1 114 112 punktów kodowych (w przedziale od 0 do 1 114 111). Punkty te tworzą *przestrzeń kodową* Unikodu. Aktualnie, w najnowszej wersji standardu (6.0) zdefiniowano 109 384 znaków. Większość powszechnie używanych znaków jest przyporządkowana punktom kodowym o wartościach nie przekraczających 65 535 — zbiór ten, obejmujący kody od 0 do 65535, oznaczany jest skrótem BMP (ang. Basic Multilingual Plane).

Punkty kodowe Unikodu zapisywane są w postaci liczb złożonych z 4, 5 lub 6 cyfr w zapisie szesnastkowym. Dość często spotykany jest zapis, w którym wartość liczbowa

poprzedzona jest znakami U+, co należy traktować jako informację, że wartość podana jest w zapisie szesnastkowym.

Wartości punktów kodowych są liczbami abstrakcyjnymi. Jeśli zamierzamy umieścić wartość punktu kodowego w pamięci komputera lub w pliku, to trzeba ustalić odpowiedni sposób kodowania dla używanego środowiska. Ponieważ niektóre wartości zajmują 21 bitów, wskazane byłoby zarezerwowanie na każdą wartość słowa 32-bitowego. Jeśli jednak uwzględnić podaną wyżej informację, że większość używanych znaków należy do zbioru BMP, gdzie punkty kodowe można zapisać na 16 bitach, to używanie słów 32-bitowych będzie powodować rozwlekłość kodowania. Problem staje się jeszcze bardziej wyraźny, jeśli uwzględnić, że w zwykłych tekstach dominują litery alfabetu łacińskiego, które w kodzie ASCII zajmują 7 bitów.

W świetle powyższych uwag można zauważyć, że kodowanie bezpośrednie (na 32 bitach) trzeba zastąpić bardziej efektywnymi sposobami kodowania — wśród stosowanych najczęściej spotykane jest kodowanie UTF–8 i UTF–16, rzadziej UTF–32 (ang. Unicode Transformation Format). Szczególnie ważne jest kodowanie UTF–8, które ze względu na swoje zalety zostało przyjęte m.in. jako domyślny standard dla dokumentów XML.

Kodowanie UTF–8

W podanych dalej opisach stosowana jest konwencja zapisu liczb szesnastkowych stosowana w asemblerze Intel: bezpośrednio po liczbie występuje litera H. Przykładowo, liczba 0105H (zapis asemblerowy) jest równoważna liczbie 0x0105 (zapis stosowany w języku C/C++).

Kodowanie w formacie UTF-8 oparte jest na następujących regułach:

1. Znaki Unicode o kodach 0000H do 007FH (czyli znaki kodu ASCII) są kodowane jako pojedyncze bajty o wartościach z przedziału 00H do 7FH. Oznacza to, że pliki zawierające wyłącznie 7-bitowe kody ASCII mają taką samą postać zarówno w kodzie ASCII jak i w UTF–8.
2. Wszystkie znaki o kodach większych od 007FH są kodowane jako sekwencja kilku bajtów, z których każdy ma ustawiony najstarszy bit na 1.
3. Pierwszy bajt w sekwencji kilku bajtów jest zawsze liczbą z przedziału C0H do FDH i określa ile bajtów następuje po nim. Wszystkie pozostałe bajty zawierają liczby z przedziału 80H do BFH.
4. Takie kodowanie pozwala, w przypadku utraty jednego z bajtów, na łatwe zidentyfikowanie kolejnej sekwencji bajtów (ang. resynchronization).
5. Kodowanie UTF–8 teoretycznie pozwala na utworzenie 6 bajtów, ale przypadku znaków Unicode generowane są maksymalnie cztery bajty, a w odniesieniu do znaków BMP generowane są co najwyżej trzy bajty.
6. Dla każdego znaku może być użyta tylko jedna, najkrótsza sekwencja bajtów.
7. Kolejność bajtów jest ustalona przez schemat kodowania.
8. Bajty 0xFE i 0xFF są nieużywane w kodzie UTF–8.
9. Opcjonalnie, dla wskazania, że ciąg bajtów zawiera znaki zakodowane w formacie UTF–8 stosuje się znacznik BOM (ang. Byte Order Mark — znacznik kolejności bajtów), który poprzedza właściwy tekst. W przypadku kodowania UTF–8 znacznik BOM zawiera trzy bajty i ma postać: EFH BBH BFH. Postać znacznika dla kodowania UTF–16 opisana jest dalej.

Podana niżej tablica określa sposób kodowania UTF–8 dla różnych wartości kodów znaków. Bity oznaczone *xxx* zawierają reprezentację binarną kodu znaku. Warto zwrócić uwagę, że liczba jedynek z lewej strony pierwszego bajtu jest równa liczbie bajtów w reprezentacji UTF–8.

Zakresy kodów		Reprezentacja w postaci UTF-8
od	do	
00H	7FH	0*xxxxxxx*
80H	7FFH	110*xxxxx* 10*xxxxxx*
800H	FFFFH	1110*xxxx* 10*xxxxxx* 10*xxxxxx*
10000H	10FFFFH	11110*xxx* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx*

Podstawową zaletą formatu UTF–8 jest zwarte kodowanie, w szczególności kody ASCII są nadal kodowane na jednym bajcie, a znacznie rzadziej używane znaki narodowe kodowane są za pomocą dwóch lub trzech bajtów — w rezultacie tekst w języku polskim zakodowany w formacie UTF–8 jest zazwyczaj o około 5% dłuższy od tekstu w formacie ISO 8859–2. Poniżej podano kody kilku liter alfabetu języka polskiego w formacie UTF–8.

Znak	a	ą	A	Ą
UTF-8	61H	C4H 85H	41H	C4H 84H

Kodowanie UTF–16

Kodowanie UTF–16 jest przeznaczone do reprezentacji znaków Unikodu w środowiskach lub kontekstach ukierunkowanych na słowa 16-bitowe. W przypadku znaków z grupy BMP (kody od 0H do FFFFH), kod UTF-16 jest identyczny z wartością punktu kodowego. Dla znaków z przedziału od 10000H do 10FFFFH (nie należących do BMP) stosuje się dwa słowa 16-bitowe, kodowane w niżej opisany sposób.

Wartość z przedziału od 10000H do 10FFFFH zostaje najpierw pomniejszona o 10000H. W rezultacie pojawia się wartość 20-bitowa, z której 10 najstarszych bitów wpisywana jest do pierwszego słowa (pole *xxxxxxxxxx*), a pozostałe 10 bajtów wpisywanych jest do drugiego słowa (pole *yyyyyyyyyy*), tak jak pokazano na rysunku.

110110 *xxxxxxxxxx*	110111 *yyyyyyyyyy*

Warto dodać, że w Unikodzie nie przyporządkowano jakiemukolwiek 16-bitowemu znakowi kodu zaczynającego od ciągu bitów 11011 — kody te zostały zarezerwowane do tworzenia omawianych par 16-bitowych (ang. surrogate pair). Innymi słowy, wartościom z przedziału <D800H, DFFFH> nie są przypisane żadne znaki.

Poniżej podano przykładowe kody znaków w zapisie szesnastkowym w standardzie Unicode

a	0061
A	0041
ą	0105
Ą	0104
b	0062
B	0042
c	0063

C	0043
ć	0107
Ć	0106
d	0064
D	0044
e	0065
E	0045

ę	0119
Ę	0118

Ze względu na stosowanie dwóch formatów przechowywania liczb znanych jako *little endian / big endian (*mniejsze niżej / mniejsze wyżej), stosowane są dodatkowe bajty BOM identyfikujące rodzaj kodowania. Postać tego znacznika podano w poniższej tabeli.

Dodatkowe bajty na początku strumienia (BOM — ang. byte order mark) identyfikujące rodzaj kodowania

Kodowanie	Znaki 16-bitowe	Znaki 32-bitowe
UTF-16 – little endian (mniejsze niżej)	FF FE	FF FE 00 00
UTF-16 – big endian (mniejsze wyżej)	FE FF	00 00 FE FF
UTF-8	EF BB BF	

Kodowanie UTF–32

Kodowanie UTF–32 stanowi najprostszy sposób reprezentacji znaku w Unikodzie. Każdy punkt kodowany jest przedstawiony w postaci liczby 32-bitowej (cztery bajty). Jeśli nie określono inaczej, stosuje się konwencję *big endian* (mniejsze wyżej), tzn. starszy bajt (bajty) przesyłany jest najpierw. Mimo że do kodowania używa się 32 bitów, to jednak ze względu na zgodność z innymi formatami, ograniczono zakres kodowanych wartości do przedziału od 0 do 10FFFFH.

Funkcja `MessageBox`

W systemie Windows krótkie teksty nie wymagające formatowania można wyświetlić na ekranie w postaci komunikatu — używana jest do tego funkcja `MessageBox`, której prototyp na poziomie języka C ma postać:

```
int MessageBox(HWND hWnd,
               LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
```

Szczegółowy opis znaczenia parametrów tej funkcji można znaleźć w dokumentacji systemu Windows (Win32 API), tutaj podamy tylko opis skrócony. Pierwszy parametr *hWnd* zawiera uchwyt okna programu, w którym zostanie wyświetlony komunikat. Jeśli wartość parametru wynosi `NULL` (lub 0 na poziomie kodu w asemblerze), to komunikat nie będzie skojarzony z oknem programu. Czwarty parametr *uType* określa postać komunikatu: w zależności od wartości tego parametru okienko komunikatu zawierać będzie jeden, dwa lub trzy przyciski wraz z odpowiednimi informacjami. W omawianym dalej przykładzie zastosujemy typowe okienko z jednym przyciskiem, które wymaga podania parametru o wartości `MB_OK` (0 na poziomie kodu w asemblerze).

Parametry drugi i trzeci są wskaźnikami (adresami) do obszarów pamięci, w którym znajdują się łańcuchy wyświetlanych znaków. Drugi parametr wskazuje na tekst stanowiący treść komunikatu, a trzeci wskazuje tytuł komunikatu. Oba łańcuchy znaków muszą być zakończone kodami o wartości 0. Jeśli w programie w języku C/C++ zdefiniowano stałą `UNICODE`, to łańcuchy znaków muszą być zakodowane w standardzie Unicode, w przeciwnym razie stosowany jest standard kodowania Windows 1250.

Na poziomie asemblera stosowane są odrębne funkcje dla obu systemów kodowania: `MessageBoxW@16` dla Unicode i `MessageBoxA@16` dla kodowania w standardzie

Windows 1250. Obie te funkcje wywoływane są wg konwencji StdCall, co oznacza, że parametry funkcji ładowane są na stos w kolejności od prawej do lewej, a usunięcie parametrów ze stosu realizowane jest przez kod zawarty wewnątrz funkcji (zob. opis ćw. 4). Sposób wykorzystania tych funkcji ilustruje podany niżej program w asemblerze.

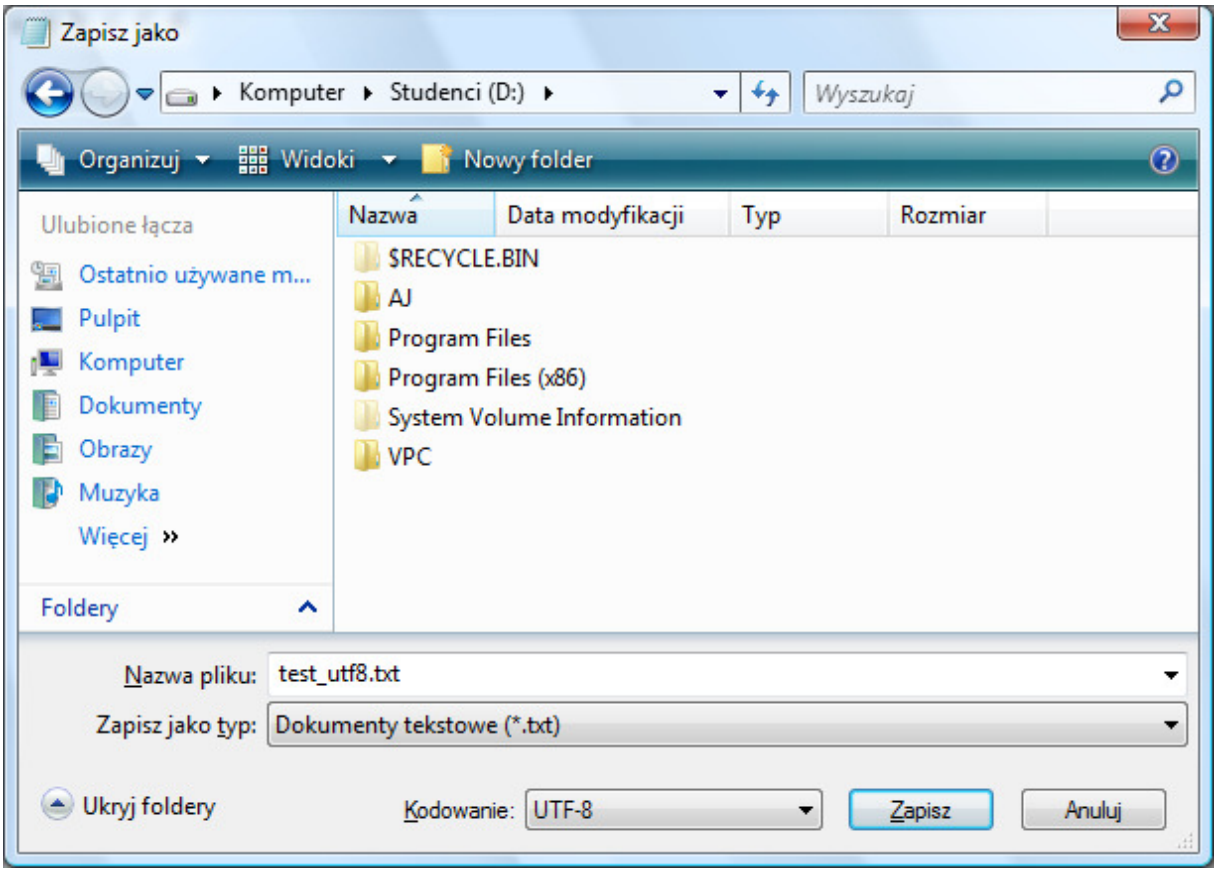
```
; Przykład wywoływania funkcji MessageBoxA i MessageBoxW
.686
.model flat
extern     _ExitProcess@4  : PROC
extern     _MessageBoxA@16 : PROC
extern     _MessageBoxW@16 : PROC
public     _main

.data
tytul_Unicode  db   'T',0,'e',0,'k',0,'s',0,'t',0,' ',0
               db   'w',0,' ',0
               db   's',0,'t',0,'a',0,'n',0,'d',0,'a',0,'r',0
               db   'd',0,'z',0,'i',0,'e',0,' ',0
               db   'U',0,'n',0,'i',0,'c',0,'o',0,'d',0,'e',0
               db   0,0

tekst_Unicode  db   'K',0,'a',0,'z',0,'d',0,'y',0
               db   ' ',0,'z',0,'n',0,'a',0,'k',0,' ',0
               db   'z',0,'a',0,'j',0,'m',0,'u',0,'j',0,'e',0
               db   ' ',0
               db   '1',0,'6',0,' ',0,'b',0,'i',0,'t',0,'o',0
               db   'w',0,0,0

tytul_Win1250  db   'Tekst w standardzie Windows 1250', 0
tekst_Win1250  db   'Kazdy znak zajmuje 8 bitow', 0

.code
_main:
           push    0       ; stała MB_OK

; adres obszaru zawierającego tytuł
           push    OFFSET tytul_Win1250

; adres obszaru zawierającego tekst
           push    OFFSET tekst_Win1250

           push    0       ; NULL
           call    _MessageBoxA@16


           push    0       ; stala MB_OK

; adres obszaru zawierającego tytuł
           push    OFFSET tytul_Unicode
```

```
; adres obszaru zawierającego tekst
            push    OFFSET tekst_Unicode

            push    0       ; NULL
            call    _MessageBoxW@16

            push    0       ; kod powrotu programu
            call    _ExitProcess@4
END
```

Zadanie 2.2.: Zmodyfikować dane programu podanego na str. 9-10 w taki sposób, by wszystkie litery tekstu były wyświetlane poprawnie. *Wskazówka*: zastąpić niektóre litery alfabetu łacińskiego przez odpowiednie kody wyrażone w postaci liczbowej (dziesiętnej lub szesnastkowej). Tabela kodów podana jest na str. 4.
Uwaga: w zapisie asemblerowym każda liczba szesnastkowa powinna zaczynać się od cyfry 0, 1, 2, ..., 9. Jeśli liczba szesnastkowa zaczyna się od cyfry A, B, ..., F, to należy przed liczbą należy wprowadzić dodatkowe 0, np. liczbę `F3H` w kodzie asemblerowym trzeba zapisać w postaci `0F3H`. Dodatkowe 0 nie ma wpływu na wartość liczby.

Zadanie 2.3. Za pomocą Notatnika (ang. *notepad*) w systemie Windows napisać tekst złożony z kilku wyrazów, w których występują także litery specyficzne dla alfabetu języka polskiego. Następnie zapamiętać ten tekst w pliku `test_utf8.txt` (wybrać opcję Plik / Zapisz jako…) w bieżącym katalogu na dysku D:\, przy czym należy wybrać kodowanie UTF-8, tak jak pokazano na poniższym rysunku.

Następnie uruchomić program Total Commander, odszukać utworzony plik `test_utf8.txt` i wyświetlić zawartość pliku poprzez naciśnięcie klawisza F3. W nowym oknie wybrać z menu Options wybrać Hex.
Zidentyfikować bajty BOM na początku pliku i porównać z podanymi na str. 6. Następnie porównać kody liter z kodami podanymi w tabeli na str. 4. Zapisać na kartce w postaci liczb szesnastkowych bajty BOM i kody UTF–8 dwóch liter specyficznych dla alfabetu języka polskiego.



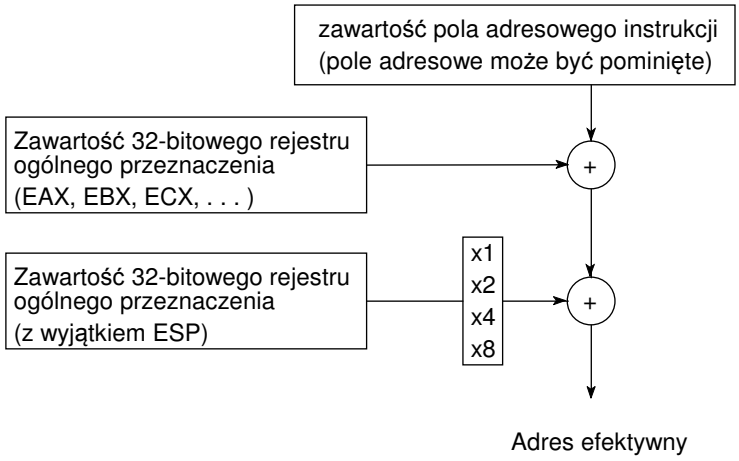
Tryby adresowania

W wielu problemach informatycznych mamy do czynienia ze zbiorami danych w formie różnego rodzaju tablic, które można przeglądać, odczytywać, zapisywać, sortować itd. Na poziomie rozkazów procesora występują powtarzające się operacje, w których za każdym razem zmienia się tylko indeks odczytywanego lub zapisywanego elementu tablicy. Takie powtarzające się operacje koduje się w postaci pętli. W przypadku operacji na elementach tablic muszą być dostępne mechanizmy pozwalające na dostęp do kolejnych elementów tablicy w trakcie kolejnych obiegów pętli. Ten właśnie problem rozwiązywany jest za pomocą różnych rodzajów trybów adresowania.

Współczesne procesory udostępniają wiele trybów adresowania, dostosowanych do różnych problemów programistycznych. Między innymi pewne tryby adresowania zostały opracowane specjalnie dla odczytywania wielobajtowych liczb, inne wspomagają przekazywanie parametrów przy wywoływaniu procedur i funkcji.

Znaczna część rozkazów procesora wykonujących operacje arytmetyczne i logiczne jest dwuargumentowa, co oznacza, że w kodzie w rozkazie podane są informacje o położeniu dwóch argumentów, np. odjemnej i odjemnika w przypadku odejmowania. Wynik operacji przesyłany jest zazwyczaj w miejsce pierwszego argumentu. Prawie zawsze jeden z argumentów znajduje się w jednym z rejestrów procesora, a drugi argument znajduje się w komórce pamięci, albo także w rejestrze procesora.

Omawiane tu tryby adresowania dotyczą przypadku, gdy jeden z argumentów operacji znajduje się w komórce pamięci. Wprowadzenie adresowania indeksowego lub bazowo-indeksowego, powoduje, że adres danej, na której ma być wykonana operacja, obliczany jest jako suma zawartości pola adresowego rozkazu (instrukcji) i zawartości jednego lub dwóch rejestrów 32-bitowych. Algorytm wyznaczania adresu w procesorach rodziny x86 ilustruje poniższy rysunek.



Przykładowo, jeśli chcemy obliczyć sumę elementów tablicy składającej się z liczb 16-bitowych (czyli dwubajtowych), to zwiększając w każdym obiegu pętli zawartość rejestru indeksowego o 2 powodujemy, że kolejne wykonania tego samego rozkazu dodawania ADD spowodują za każdym razem dodanie kolejnego elementu tablicy.

Dodatkowo może być stosowany tzw. współczynnik skali (x1, x2, x4, x8), co ułatwia uzyskiwanie adresu wynikowego.

Wyznaczanie adresu z użyciem indeksowania stanowi jeden z etapów wykonywania rozkazu przez procesor — jej wynikiem jest *adres efektywny* (zob. rysunek) rozkazu, czyli adres komórki pamięci zawierającej daną, na której zostanie wykonana operacja, np. mnożenie. Dość często pole adresowe instrukcji (rozkazu) jest pominięte, co oznacza, że adres efektywny określony jest wyłącznie przez zawartość podanego rejestru indeksowego. Przykładowo, jeśli rejestr indeksowy (np. EBX) zawierać będzie liczbę 724, to procesor odczyta wartość danej z komórki o adresie 724. Podkreślamy, że procesor wykonana wymaganą operację nie na liczbie 724, ale na liczbie która zostanie odczytana z komórki pamięci o adresie 724.

W zapisie asemblerowym, symbole rejestru, w którym zawarty jest adres komórki pamięci umieszcza się w nawiasach kwadratowych. Przykładowo, rozkaz

```
mov  edx, [ebx]
```

powoduje przesłanie do rejestru EDX wartości (np. liczby całkowitej) pobranej z komórki pamięci operacyjnej o adresie znajdującym się w rejestrze EBX. Zauważmy, że w omawianym przykładzie pole adresowe rozkazu nie występuje, a adres efektywny równy jest po prostu liczbie zawartej w rejestrze EBX.

Porównywanie liczb całkowitych bez znaku

W trakcie kodowania programów występuje często konieczność porównywania zawartości rejestrów i komórek pamięci. Na razie uwagę skupimy na porównywaniu liczb bez znaku (porównywanie liczb ze znakiem działa tak samo, ale używane są inne rozkazy skoku).

Porównanie zawartości rejestru i zawartości komórki lub porównanie zawartości dwóch rejestrów lub porównanie z liczbą wymaga zawsze użycia dwóch rozkazów:

- rozkazu `CMP`, który porównuje zawartości;
- rozkazu skoku (np. `JE`, `JA`, itp.), który w zależności od wyniku porównania zmienia naturalny porządek wykonywania programu (poprzez wykonanie skoku) albo pozostawia ten porządek niezmieniony, tak że procesor wykonuje dalej rozkazy w naturalnej kolejności.

Rozpatrzmy fragment programu, którego zadaniem jest sprawdzenie czy liczba zawarta w 8-bitowym rejestrze CL jest większa od 12.

```
     cmp     cl, 12     ; porównanie
     ja      idz_dalej  ; skok, gdy liczba w CL większa od 12
- - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - -
idz_dalej:
- - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - -
```

Jeśli liczba w rejestrze CL jest większa od 12, to procesor przeskoczy kolejne rozkazy i będzie kontynuował wykonywanie programu od miejsca oznaczonego etykietą `idz_dalej`. Jeśli liczba w rejestrze CL jest równa 12 lub jest mniejsza od 12, to skok nie zostanie wykonany i procesor będzie wykonywał dalsze rozkazy w naturalnym porządku (w podanym przykładzie rozkazy symbolicznie zaznaczono znakami `- - - - - - -`).

Jeśli trzeba zbadać czy liczba w rejestrze CL jest mniejsza od 12, to postępowanie jest podobne: zamiast rozkazu `ja` (skrót od ang. *jump if above*) należy użyć rozkazu `jb` (skrót od ang. *jump if below*). Jeśli sprawdzamy czy zawartości są równe, to używamy rozkazu `je` (skrót od ang. *jump if equal*).

Do porównywania liczb bez znaku i liczb ze znakiem używa się nieco innych rozkazów sterujących (rozkazów skoku). Mnemoniki tych rozkazów zestawiono w poniższej tablicy.

Rodzaj porównywanych liczb	liczby bez znaku	liczby ze znakiem
skocz, gdy większy	`ja (jnbe)`	`jg (jnle)`
skocz, gdy mniejszy	`jb (jnae, jc)`	`jl (jnge)`
skocz, gdy równe	`je (jz)`	`je (jz)`
skocz, gdy nierówne	`jne (jnz)`	`jne (jnz)`
skocz, gdy większy lub równy	`jae (jnb, jnc)`	`jge (jnl)`
skocz, gdy mniejszy lub równy	`jbe (jna)`	`jle (jng)`

W nawiasach podano mnemoniki rozkazów o tych samych kodach — w zależności konkretnego porównania można bardziej odpowiedni mnemonik, np. rozkaz `JAE` używamy do sprawdzania czy pierwszy operand rozkazu cmp (liczby bez znaku) jest większy lub

równy od drugiego; jeśli chcemy zbadać pierwszy operand jest niemniejszy od drugiego, to używamy rozkazu JNB — rozkazy JAE i JNB są identyczne i są tłumaczone na ten sam kod.

W przypadku, gdy porównania dotyczą znaków w kodzie ASCII drugi argument można podać w postaci znaku ASCII ujętego w apostrofy, np.

```
cmp     dl, 'b'
```

Powyższy zapis jest wygodniejszy niż porównywanie wartości liczbowych:

```
cmp     dl, 62H
```

albo

```
cmp     dl, 98
```

Wszystkie trzy podane wyżej formy zapisu rozkazu CMP są całkowicie równoważne — tłumaczone są na ten sam kod maszynowy.

Omawiany tu rozkaz CMP jest w istocie odmianą rozkazu odejmowania. Zwykłe odejmowanie wykonuje się za pomocą rozkazu SUB, natomiast rozkaz CMP także wykonuje odejmowanie, ale nigdzie nie wpisuje jego wyniku. Oba omawiane rozkazy, prócz właściwego odejmowania, ustawiają także znaczniki w rejestrze stanu procesora (w rejestrze znaczników EFLAGS).

Z punktu widzenia techniki porównywania liczb bez znaku istotne znaczenie mają znaczniki (pojedyncze bity) w rejestrze stanu procesora (w rejestrze znaczników):

- ZF (znacznik zera, ang. *zero flag*) — ustawiany w stan 1, gdy wynik ostatnio wykonanej operacji arytmetycznej lub logicznej wynosi 0, w przeciwnym razie do znacznika wpisywane jest 0;
- CF (znacznik przeniesienia, ang. *carry flag*), ustawiany w stan 1, gdy w trakcie dodawania wystąpiło przeniesienie wychodzące poza rejestr albo w trakcie odejmowania wystąpiła prośba o pożyczkę z pozycji poza rejestrem, w przeciwnym razie do znacznika wpisywane jest 0.

W takim ujęciu rozkaz porównywania CMP, na podstawie wartości obu porównywanych argumentów, odpowiednio ustawia znaczniki ZF i CF, natomiast następujący po nim rozkaz skoku (np. JE) analizuje stan tych znaczników i wykonuje skok albo nie (w zależności od stanu tych znaczników). Przykładowo, do sprawdzenia czy zawartości rejestrów 16-bitowych DX i SI są jednakowe możemy użyć poniższej sekwencji rozkazów:

```
cmp     dx, si      ; porównanie zawartości rejestrów
je      zaw_rowne   ; skok, gdy zawartości są jednakowe
```

Podany tu rozkaz CMP oblicza różnicę zawartości rejestrów DX i SI, jednak nie wpisuje obliczonej różnicy — ustawia natomiast znaczniki, między innymi ustawia znacznik ZF. Jeśli liczby w rejestrach DX i SI są równe, to ich różnica wynosi 0, co spowoduje wpisanie 1 do znacznika ZF. Kolejny rozkaz JE testuje stan znacznika ZF:

- jeśli ZF = 1, to skok jest wykonywany;
- jeśli ZF = 0, to skok nie jest wykonywany.

Przykład programu przekształcającego tekst

Podany poniżej program wczytuje wiersz z klawiatury i wyświetla go ponownie wielkimi literami. W przypadku znaków z podstawowego zbioru ASCII (kody o wartościach mniejszych od 128) zamiana kodu małej litery na kod wielkiej litery wymaga tylko odjęcia wartości 20H od kodu małej litery. Przed wykonaniem odejmowania trzeba jednak sprawdzić czy pobrany znak jest małą literą — sprawdzenie to wykonuje poniższy fragment programu (analizowany kod znaku został wcześniej wpisany do rejestru `DL`):

```
cmp     dl, 'a'
jb      dalej   ; skok, gdy znak nie wymaga zamiany
cmp     dl, 'z'
ja      dalej   ; skok, gdy znak nie wymaga zamiany
sub     dl, 20H ; zamiana na wielkie litery
```

I tak jeśli kod znaku jest mniejszy od kodu litery `a` lub jest większy od kodu litery `z`, to zamiana jest niepotrzebna — w tych przypadkach następuje skok do rozkazu poprzedzonego etykietą `dalej`. W przeciwnym razie kod zawarty w rejestrze `DL` zostaje zmniejszony o 20H (rozkaz `sub dl, 20H`).

Przekodowanie znaków narodowych jest bardziej skomplikowane. W przypadku funkcji `read` wczytywane znaki kodowane są standardzie Latin 2. Trzeba więc dla każdej małej litery należącej do alfabetu narodowego odszukać w tablicy kodów Latin 2 odpowiedni kod wielkiej litery. Najprostsza metoda rozwiązania tego problemu polega na wielokrotnym wykonywaniu rozkazu porównywania `cmp` z kodami różnych liter. Po stwierdzeniu zgodności w miejsce kodu małej litery wprowadza się odpowiedni kod wielkiej litery. Omawiana tu zamiana stanowi treść zadania 2.4., natomiast zamiana liter alfabetu łacińskiego (podstawowy zbiór ASCII) została pokazana w poniższym przykładzie programu.

```
; wczytywanie i wyświetlanie tekstu wielkimi literami
; (inne znaki się nie zmieniają)

.686
.model flat
extern  _ExitProcess@4 : PROC
extern  __write : PROC  ; (dwa znaki podkreślenia)
extern  __read  : PROC  ; (dwa znaki podkreślenia)
public  _main

.data
tekst_pocz     db   10, 'Proszę napisać jakiś tekst '
               db   'i nacisnac Enter', 10
koniec_t       db   ?
magazyn        db   80 dup (?)
nowa_linia          db   10
liczba_znakow  dd   ?
```

```
.code
_main:

; wyświetlenie tekstu informacyjnego

; liczba znaków tekstu
          mov     ecx,(OFFSET koniec_t) - (OFFSET tekst_pocz)
          push    ecx

          push    OFFSET tekst_pocz  ; adres tekstu
          push    1     ; nr urządzenia (tu: ekran - nr 1)
          call    __write  ; wyświetlenie tekstu początkowego

          add     esp, 12    ; usuniecie parametrów ze stosu

; czytanie wiersza z klawiatury
          push    80 ; maksymalna liczba znaków
          push    OFFSET magazyn
          push    0  ; nr urządzenia (tu: klawiatura - nr 0)
          call    __read ; czytanie znaków z klawiatury
          add     esp, 12    ; usuniecie parametrów ze stosu
; kody ASCII napisanego tekstu zostały wprowadzone
; do obszaru 'magazyn'

; funkcja read wpisuje do rejestru EAX liczbę
; wprowadzonych znaków
          mov     liczba_znakow, eax
; rejestr ECX pełni rolę licznika obiegów pętli
          mov     ecx, eax
          mov     ebx, 0          ; indeks początkowy

ptl:      mov     dl, magazyn[ebx] ; pobranie kolejnego znaku
          cmp     dl, 'a'
          jb      dalej   ; skok, gdy znak nie wymaga zamiany
          cmp     dl, 'z'
          ja      dalej   ; skok, gdy znak nie wymaga zamiany
          sub     dl, 20H ; zamiana na wielkie litery

; odesłanie znaku do pamięci
          mov     magazyn[ebx], dl
dalej:    inc     ebx     ; inkrementacja indeksu
          loop    ptl     ; sterowanie pętlą

; wyświetlenie przekształconego tekstu
          push    liczba_znakow
          push    OFFSET magazyn
          push    1
          call    __write  ; wyświetlenie przekształconego
tekstu
          add     esp, 12  ; usuniecie parametrów ze stosu
```

```
        push    0
        call    _ExitProcess@4      ; zakończenie programu

END
```

Zadanie 2.4. Uruchomić podany wyżej program przykładowy, a następnie przebudować go w taki sposób by zamiana małych liter na wielkie obejmowała litery specyficzne dla języka polskiego (ą, Ą, ć, Ć, ę, ...).

Wskazówka: po dopisaniu fragmentu programu, w którym nastąpi zamiana liter specyficznych dla języka polskiego (ą, ć, ę, ...), asembler będzie sygnalizował przekroczenie rozmiaru pętli:

```
error A2075: jump destination too far : by 24 byte(s)
```

Błąd ten wynika z formatu rozkazu `loop`, który może być stosowany do sterowania pętlą o rozmiarze nie przekraczającym 127 bajtów. Jeśli liczba bajtów zajmowanych przez pętlę jest większa od 127, to zamiast rozkazu `loop ptl` należy zastosować dwa poniższe rozkazy:

```
        dec     ecx
        jnz     ptl
```

Zadanie 2.5. Zmodyfikować podany wyżej program przykładowy w taki sposób, by tekst wczytany z klawiatury został wyświetlony w postaci komunikatu za pomocą funkcji `MessageBoxA`. W wyświetlanym komunikacie mogą wystąpić wszystkie litery alfabetu języka polskiego.

Zadanie 2.6. Rozbudować podany wyżej program przykładowy w taki sposób, by tekst wczytany z klawiatury został wyświetlony także w postaci komunikatu za pomocą funkcji `MessageBoxW`. W wyświetlanym komunikacie mogą wystąpić wszystkie litery alfabetu języka polskiego.
Wskazówka: przed wywołaniem funkcji `MessageBoxW` należy przygotować odrębną tablicę z tekstem w postaci ciągu znaków 16-bitowych formacie Unicode. W przypadku znaków z podstawowego kodu ASCII (kody o wartościach mniejszych od 128) wystarczy tylko znak 8-bitowy rozszerzyć do 16 bitów poprzez dopisanie 8 bitów zerowych z lewej strony. W przypadku znaków narodowych trzeba odpowiednio przekodować znak 8-bitowy na 16-bitowy. W trakcie zapisywania znaków 16-bitowych do tablicy z tekstem dla funkcji `MessageBoxW` należy po każdym zapisie zwiększać rejestr indeksowy o 2, przy czym nie powinien być to ten sam rejestr, który używany jest odczytywania znaków 8-bitowych.

Zadanie 2.7. Uruchomić podany wyżej program przykładowy, a następnie przebudować go w taki sposób by tekst wynikowy wyświetlany był w oknie konsoli wielkimi literami w kolorze jasnozielonym. Cały program powinien być zakodowany w języku asemblera (nie mogą występować fragmenty w języku C).

Wskazówki:

1. Zmiana koloru tekstu wyświetlanego w oknie konsoli wymaga wykonania poniższego kodu podanego w języku C.

```
HANDLE uchwyt;
uchwyt = GetStdHandle(STD_OUTPUT_HANDLE);
SetConsoleTextAttribute(uchwyt, FOREGROUND_GREEN |
                                    FOREGROUND_INTENSITY);
```

2. Powyższy kod w języku C należy zastąpić równoważnym kodem w języku asemblera. Przyjąć, że zmienne typu `HANDLE` zapisywane są w słowach 32-bitowych. Utworzony kod należy wprowadzić do podanego wyżej programu w asemblerze przed wywołaniem funkcji `write`. Przy zamianie na kod asemblerowy można się kierować przykładem podanym na wykładzie dotyczącym funkcji `GetComputerName`. Zwrócić uwagę, że funkcje systemowe Windows kodowane są zgodnie ze standardem *stdcall* i same usuwają parametry ze stosu (inaczej niż w przypadku funkcji `write`).

3. Wartość liczbowa przypisana stałej `STD_OUTPUT_HANDLE` podana jest w pliku nagłówkowym `WinBase.h`, a wartości przypisane stałym `FOREGROUND_GREEN` i `FOREGROUND_INTENSITY` podane są w pliku `WinCon.h` (podkatalog `Microsoft SDKs`). W środowisku MS Visual Studio wartości omawianych stałych można też uzyskać poprzez dołączenie do projektu modułu (pliku) tymczasowego w języku C (poprzez Source File / Add / New Item). Na początku tego modułu należy wpisać:

```
#include <windows.h>
```

   a w następnych wierszach podać nazwy stałych: `STD_OUTPUT_HANDLE`, `FOREGROUND_GREEN`, `FOREGROUND_INTENSITY`. Wartość stałej uzyskuje się poprzez kliknięcie prawym klawiszem myszki na wartość stałej i wybranie opcji Go To Definition. Wartości stałych można też odnaleźć na stronie internetowej `http://msdn.microsoft.com`

4. Wygodnie jest zdefiniować wartości stałych za pomocą dyrektywy `EQU` podanej w początkowej części programu w asemblerze, np.

```
FOREGROUND_INTENSITY     EQU  0008H
```