ROZDZIAŁ SIÓDMY: **STANDARDOWA BIBLIOTEKA UCR**

Większość języków programowania dostarcza kilku „wbudowanych” funkcji redukując wysiłek potrzebny do napisania programu. Tradycyjnie, programiści asemblerowi nie mieli dostępu do standardowego zbioru powszechnie używanych podprogramów dla swoich programów; w związku z tym, wydajność programistów asemblerowych była całkiem niska ponieważ stale „wymyślali koło” w każdym programie który napisali. Standardowa biblioteka UCR dla programistów 80x86 dostarcza takiego zbioru podprogramów .Rozdział ten omawia mały podzbiór podprogramów dostępnych w tej bibliotece. Po przeczytaniu tego rozdziału powinniśmy przestudiować dokumentację towarzyszącą podprogramom biblioteki standardowej.

7.0 WSTĘP

Rozdział ten dostarczy podstawowego wprowadzenia do funkcji dostępnych w Standardowej Bibliotece UCR dla programistów asemblerowych 80x86.Ten krótki wstęp obejmuje następujące tematy:

- Standardowa Biblioteka UCR dla Programistów Języka Asemblera
- Podprogramy zarządzania pamięcią
- Procedury wejściowe
- Procedury wyjściowe
- Konwersja
- Predefiniowane stałe i makra

7.1 WSTĘP DO STANDARDOWEJ BIBLIOTEKI UCR

„Standardowa biblioteka UCR dla Programistów Języka Asemblera 80x86” jest zbiorem asemblerowych podprogramów wzorowanych na standardowej bibliotece „C”. Pośród innych rzeczy, biblioteka standardowa zawiera procedury do operowaniu na danych wejściowych, wyjściowych, konwersji, kilku porównań i sprawdzeń, manipulowania łańcuchem, zarządzaniem pamięcią, zbiór operatorów znakowych, operacje zmiennoprzecinkowe, manipulowania listą, portami szeregowymi I/O ,współbieżność i współprogramy i dopasowanie do wzorca.

Ten rozdział nie będzie próbował opisać każdego podprogramu w bibliotece. Przede wszystkim Biblioteka jest stale zmieniana, więc taki opis szybko mógłby się stać przestarzały. Po drugie, kilka z tych podprogramów z biblioteki jest tylko dla zaawansowanych programistów, więc jest poza zasięgiem tego tekstu. W końcu, jest sto podprogramów w tej bibliotece. Zakładając, że opiszemy je tu wszystkie, byłoby poważnym zakłócenie dla prawdziwej pracy jaką mamy do wykonania – nauczenie się asemblera.

Dlatego też, ten tekst omawia kilka niezbędnych podprogramów, które działają przy najmniejszym wysiłku. Zauważmy, że pełna dokumentacja biblioteki, jak również kody źródłowe i kilka przykładowych plików znajduje się na dyskietce dołączonej do tego tekstu. Odnośny przewodnik znajduje się w dodatkach do tego tekstu. Możemy również znaleźć ostatnią wersję Standardowej Biblioteki UCR na wielu serwisach on-line, BBSach i wieku innych miejscach. Jest również dostępna przez anonimowe FTPy w Internecie.

Kiedy używamy Standardowej Biblioteki UCR, powinniśmy zawsze używać pliku SHELL.ASM dostarczanego jako „szkielet” nowego programu. Plik ten zakłada konieczne segmenty, dostarcza właściwych dyrektyw include i inicjuje dla nas konieczne podprogramy Biblioteki.

Nie powinniśmy próbować tworzyć nowego programu z podprogramów przypadkowych, chyba ,że jesteśmy bardzo dobrze zaznajomieni z wewnętrznymi działaniami Biblioteki Standardowej.

Zauważmy, że większość podprogramów Biblioteki Standardowej używa makr zamiast instrukcji call dla wywołania. Nie możemy, na przykład, bezpośrednio wywołać podprogramu putc. Faktycznie wywołujemy makro putc które zawiera wywołanie do procedury sl_putc („SL" - Standard Library).

Jeśli nie wybierzemy do używania pliku SHELL.ASM ,nasz program musi zawierać kilka instrukcji do uruchomienia biblioteki standardowej i zaspokojenia pewnych wymagań biblioteki standardowej. Dopóki korzystamy z doświadczeń programowania w asemblerze, powinniśmy zawsze używać pliku SHELL.ASM jako punktu startowego dla naszych programów.

7.1.1 PODPROGRAMY ZARZĄDZANIA PAMIĘCIĄ: MEMINIT,MALLOC I FREE

Biblioteka Standardowa dostarcza kilku podprogramów ,które zarządzają wolną pamięcią na stercie. Dają one programistom asemblerowym zdolność do dynamicznego alokowania pamięci podczas wykonywania programu i powrót tej pamięci do systemu kiedy program kiedy nie potrzebujemy dłużej programu. Poprzez dynamiczne alokowanie i zwalnianie bloków pamięci możemy wydajniej używać pamięci na PC.

Podprogram meminit inicjuje menadżera pamięci a my musimy wywołać go przed każdym podprogramem, który używa tego menadżera pamięci. Ponieważ wiele podprogramów Biblioteki Standardowej używa menadżera pamięci, powinniśmy wywoływać tą procedurę wcześniej w programie .Plik SHELL.ASM wykonuje takie wywołanie dla nas.

Podprogram malloc alokuje pamięć na stercie i zwraca wskaźnik do bloku, umieszczając go w rejestrach es:di. Przed wywołaniem malloc, musimy załadować rozmiar bloku (w bajtach) do rejestru cx. Przy powrocie, malloc ustawia flagę przeniesienia jeśli wystąpił błąd (niewystarczająca pamięć).Jeśli przeniesienie jest wyzerowane ,es:di wskazuje na blok bajtów, którego rozmiar wyszczególniliśmy:

```
mov     cx, 1024                ;przejęcie 1024 bajtów na stertę
malloc                          ;wywołanie MALLOC
jc      MllocError              ;jeśli błąd pamięci
mov     word ptr PNTR, DI       ; wskaźnik do zapisu bloku
mov     word ptr PNTR+2, ES
```

Kiedy wywołujemy malloc ,menadżer pamięci obiecuje, że blok, który nam dał jest wolny i wyzerowany i że nie realokuje tego bloku do momentu aż go wyraźnie nie zwolnimy. Zwracając blok pamięci z powrotem do menadżera pamięci, możemy (być może) użyć go ponownie w przyszłości, używając podprogramu free z Biblioteki. Free oczekuje, że podamy wskaźnik powrotny poprzez malloc:

```
les     di, PNTR                ;pobieranie wskaźnika do zwolnienia
free                            ;zwolnienie bloku
jc      BadFree
```

Jak zwykle przy większości podprogramów Biblioteki Standardowej, jeśli podprogram free ma kilka rodzajów trudności zwróci flagę przeniesienia aby zaznaczyć, ze wystąpił błąd

---

7.1.2 PODRPOGRAMY STANDARDOWEGO WEJŚCIA:GETC,GETS,GETSM

Biblioteka Standardowa dostarcza kilku podprogramów wejścia, są trzy, które w szczególności będziemy używali cały czas: getc (pobierz znak),gets (pobierz łańcuch) i getsm (pobierz łańcuch malloc).

Getc odczytuje pojedynczy znak z klawiatury i zwraca ten znak w rejestrze al. Zwraca ona stan końca pliku (EOF) w rejestrze ah (zero oznacza, że EOF nie wystąpił, jeden znaczy, że EOF wystąpił)Nie modyfikuje ona innych rejestrów. Jak zwykle, flaga przeniesienia zwraca stan błędu. Nie musimy przekazywać getc żadnej wartości w rejestrze. Getc nie potwierdza znaku wejściowego do wyświetlania na ekranie .Musimy wyraźnie wydrukować znak jeśli chcemy aby pojawił się na wyjściu monitora.

Następujący program przykładowy wykonuje nieustanną pętlę dopóki użytkownik nie naciśnie klawisza Enter:

```
;Notka: „CR" jest symbolem. ,który pojawia się w pliku nagłówkowym „consts.a". Jest to wartość 13,która jest
;kodem ASCII dla znaku powrotu karetki
Wait4Enter:     getc
                cmp     al., cl
                jne     Wait4Enter
```

Podprogram gets odczytuje całą linię tekstu z klawiatury. Przechowuje każdy kolejny znak linii wejściowej w tablicy bajtów której adres bazowy znajduje się w parze rejestrów es:di. Ta tablica musi mieć miejsce na przynajmniej 128 bajtów. Podprogram gets będzie odczytywał każdy znak i umieszczał go w tablicy z wyjątkiem dla znaku powrotu karetki. Gets kończy linię wejściową bajtem zerowym (który jest zgodny z podprogramem obsługi łańcucha Biblioteki Standardowej).Gets potwierdza każdy znak wypisywany na urządzeniu wyświetlającym, również operuje prostymi funkcjami edycyjnymi takimi jak backspace. Jak zwykle, gets zwraca ustawienie przepełnienia jeśli wystąpi błąd. Następujący przykład odczytuje linię tekstu ze

standardowego urządzenia wejściowego a potem zlicza liczbę wypisywanych znaków. Kod ten jest skomplikowany, zauważmy, że inicjuje licznik i wskaźnik do –1 uprzednio wprowadzając pętlę a potem bezpośrednio zwiększa je o jeden .Ustawia to licznik na zero i modyfikuje wskaźnik, żeby wskazywać pierwszy znak w łańcuchu. To uproszczenie tworzy kod mniej wydajny niż proste rozwiązanie:

```
DSEG            segment
MyArray         byte    128 dup (?)
DSEG            ends
CSEG            segment
                .
                .
                .
; Note: LESI is a macro (found in consts.a) that loads
; ES:DI with the address of its operand. It expands to the
; code:
;
;               mov di, seg operand
;               mov es, di
;               mov di, offset operand
;
; You will use the macro quite a bit before many Standard
; Library calls.
                lesi    MyArray             ;Get address of inp buf.
                gets                        ;Read a line of text.
                mov     ah, -1              ;Save count here.
                lea     bx, -1[di]          ;Point just before string.
CountLoop:      inc     ah                  ;Bump count by one.
                inc     bx                  ;Point at next char in str.
                cmp     byte ptr es:[bx], 0
                jne     CoutLoop
; Now AH contains the number of chars in the string.
```

Podprogram getsm również odczytuje łańcuch z klawiatury i zwraca wskaźnik do tego łańcucha w es:di. Różnica między gets a getsm jest taka, że nie musimy podawać adresu bufora wejściowego w es:di. Getsm automatycznie alokuje pamięć na stercie z wywołaniem malloc i zwraca wskaźnik do bufora w es:di .Nie zapomnijmy ,że musimy wywołać meminit na początku naszego programu jeśli używamy tego podprogramu. Plik szkieletowy SHELL.ASM wywołuje meminit za nas. Również, nie zapomnijmy wywołać free aby ściągnąć pamięć ze sterty.

```
Getsm           ;zwraca wskaźnik w ES:DI
-
-
free            ;Zwraca pamięć ze sterty
```

7.1.3 STANDARDOWE PODPROGRAMY WYJŚCIA:PUTC,PUTCR,PUTS,PUTH,PUTIPRINT I PRINTF

Biblioteka Standardowa dostarcza szerokiego wachlarza podprogramów wyjścia, dużo więcej niż zobaczymy tu. Podprogramy te są reprezentatywne dla podprogramów które znajdziemy w Bibliotece.

Putc wyprowadza pojedynczy znak na urządzenie wyświetlające ,Wyprowadzony znak pojawia się w rejestrze al. Nie wpływa na inny rejestr, chyba, że wystąpi błąd na wyjściu (flaga przeniesienia oznacza błąd /brak błędu jak zwykle)..Zobacz dokumentację Biblioteki po więcej szczegółów.

Putcr wyprowadza „nową linię” (kombinację powrotu karetki CR /przesunięcia o jedną linię LF) na standardowe wyjście. Jest ona odpowiednio równoważna następującemu kodowi:

```
mov     al., cr     ;CR i LF są stałe
putc                ;pojawiają się w pliku
mov     al., lf     ;nagłówkowym const.a
putc
```

Podprogram puts (wprowadź łańcuch) drukuje łańcuch zakończony zerem który wskazuje es:di. Zauważmy, że puts automatycznie nie wyprowadza nowej linii po wydrukowaniu łańcucha. Musimy albo wprowadzić znak CR/LF na koniec łańcucha albo wywołać putcr po wywołaniu puts jeśli chcemy wydrukować nową linię po łańcuchu. Puts nie wpływa na żaden rejestr (chyba że wystąpi błąd).W szczególności, nie zmienia wartości rejestrów es:di. Następująca sekwencja kodu korzysta z tego faktu:

```
getsm           ;odczyt łańcucha
puts            ;drukuje go
putcr           ;druk nowej linii
```

```
        free            ;zwolnienie pamięci dla łańcucha
```

Ponieważ powyższy podprogram zachowuje es:di (z wyjątkiem oczywiście getsm),wywołanie free dealokuje zaalokowaną pamięć przez wywołanie getsm.

Podprogram puth drukuje wartość z rejestru al. jako dokładnie dwie heksadecymalne cyfry wliczając w to czołowy bajt zero jeśli wartość jest z zakresu 0..Fh.Następująca pętla odczytuje sekwencję klawiszy klawiatury i drukuje ich wartości ASCII do chwili kiedy użytkownik nie naciśnie klawisza ENTER

```
KeyLoop:        getc
                cmp     al., cr
                je      done
                puth
                putcr
                jmp     KeyLoop
done:
```

Podprogram puti drukuje wartość z ax jako 16 bitową wartość całkowitą ze znakiem. Następujący kod jest fragmentem wydruku sumy I i J :

```
        mov     ax, I
        add     ax, J
        puti
        putcr
```

Putu jest podobne do puti z wyjątkiem tego, że wyprowadza całkowitą wartość bez znaku zamiast całkowitej ze znakiem.

Podprogramy jak puti i putu zawsze wyprowadzają liczby używając minimalnej liczby możliwych pozycji drukowania .Na przykład ,puti używa trzech pozycji drukowania w łańcuchu drukującym wartość 123.Czasami.możemy chcieć zmusić te podprogramy wyjściowe do drukowania ich wartości używając stałej liczby pozycji drukowania, uzupełniając każdą ekstra pozycję spacją. Podprogramy putisize i putusize dostarczają takiej możliwości. Podprogramy te oczekują wartości liczbowych w ax i wyszczególnionej szerokości pola w cx. Drukują one liczbę w polu szerokości przynajmniej pozycji cx .Jeśli wartość w cx jest większa niż liczba drukowanych pozycji o wymaganych wartościach, podprogramy te wyrównują do prawej liczbę w polu cx drukowanej pozycji. Jeśli liczba w cx jest mniejsza niż liczba drukowanych pozycji wymaganych wartości. podprogramy te zignorują wartość w cx i użyją jednak wielu pozycji drukowania wymaganych liczb.

Podprogram print jest jednym z wielu, często wywoływanych procedur w bibliotece. Drukuje ona łańcuch zakończony zerem, który występuje bezpośrednio po wywołaniu print:

```
        print
        byte    „drukuj ten łańcuch na wyświetlaczu”,cr,lf,0
```

Powyższy przykład drukuje łańcuch „drukuj ten łańcuch na wyświetlaczu” poprzedzony przez nową linie. Zauważmy ,że print będzie drukował jakikolwiek znak bezpośrednio następujący po wywołaniu print aż do spotkania pierwszego bajtu zerowego. W szczególności, możemy drukować sekwencje nowej linii i każdy inny znak sterujący jak pokazano powyżej. Również zauważmy, że nie jesteśmy ograniczeni do drukowania jednej linii tekstu z podprogramem print:

```
        print
        byte            „To przykład podprogramu PRINT”,cr.lf
        byte            „drukującego kilka linii tekstu. ”,cr, lf
        byte            cr, lf
        byte            0
```

Uzyskamy cos takiego:

```
To przykład podprogramu PRINT
Drukującego kilka linijek tekstu.
```

Jest niezwykle ważne abyśmy nie zapominali o bajcie kończącym zerem. Podprogram print zaczyna wykonywanie pierwszej maszynowej instrukcji 80x86 z bajtem zakończonym zerem. jeśli zapomnimy wprowadzić bajt zakończony zerem po naszym łańcuchu, podprogram print chętnie pożre kolejne bajty instrukcji naszego łańcucha (wydrukuje je),chyba że znajdzie bajt zero (bajty zerowe są powszechne w programach asemblerowych).Będzie to przyczyną, że nasz program będzie się źle zachowywał a jest to duży błąd początkujących programistów, kiedy stosują podprogram print. Zawsze o tym pamiętaj.

Printf, podobnie jak jego imienniczka w „C”, dostarcza zdolności do formatowania danych wyjściowych dla pakietu Biblioteki Standardowej. Typowe wywołanie printf zawsze przybiera następującą formę:

```
        printf
        byte            „łańcuch formatowany:,0
        dword           operand1,operand2,.....opernadn
```

Łańcuch formatowany jest porównywalny do dostarczanego w języku „C". Dla większości znaków, printf po prostu drukuje znaki w łańcuchu formatowanym aż do momentu natrafienia na bajt zakończony zerem. Dwa wyjątki to znaki poprzedzone przez backslash ('\") i znak procent („%).Podobnie jak printf z C, printf Biblioteki Standardowej używa backslasha jako znaku sterującego i znaku procenta jako obowiązującego przy formatowaniu łańcucha.

Printf używa „\" do drukowania znaków specjalnych ,podobnie do, ale nie identycznie jak printf w C. Printf Biblioteki Standardowej wspiera następujące znaki specjalne:

- \r Drukowanie powrotu karetki ale nie przesunięcia o jedną linię
- \n Drukowanie znaku nowej linii (powrót karetki /przesunięcie o jedną linie)
- \b Drukowanie znaku backspace
- \t Drukowanie znaku tab
- \l Drukowanie znaku przesunięcia o jedną linię (ale nie powrotu karetki)
- \f Drukowanie znaku przesunięcia strony
- \\ Drukowanie znaku backslash
- \% Drukowanie znaku procenta
- \0xhh Drukowanie kodu ASCII hh, reprezentowanego przez dwie cyfry heksadecymalne

Użytkownicy C powinni zauważyć, że parę różnic pomiędzy Biblioteką Standardową a C. Po pierwsze użycie \% drukuje znak procenta wewnątrz formatowanego łańcucha, nie „%%".C nie pozwala używać \% ponieważ kompilator C przetwarzając „\%" podczas kompilacji programu (pozostawi pojedynczy„%" w kodzie obiektu) podczas gdy printf przetwarza łańcuch formatowany podczas wykonywania. Widzi pojedynczy „%" i traktuje go jako znak doprowadzający do formatowania. Printf Standardowej Biblioteki, z drugiej strony, przetwarza oba „\" i „%" w czasie wykonywania, dlatego też można rozpoznać „\%".

Łańcuchu w formie „\0xhh" muszą zawierać dokładnie dwie cyfry heksadecymalne. Bieżący podprogram printf nie jest dość silny aby operować sekwencjami formy „=oxh" która zawiera tylko pojedynczą cyfrę heksadecymalną. Zapamiętajmy, gdy odkryjemy, że printf będzie „odrąbywać" znaki po napisaniu wartości

Nie ma absolutnie żadnego powodu aby używać heksadecymalnego znaku sterującego z wyjątkiem „\0x00".Printf przechwytuje wszystkie znaki występujące po wywołaniu printf aż do bajtu zakończonego zerem (który jest po to abyśmy nie musieli stosować „\0x00" jeśli chcemy wydrukować znak null, printf nie wydrukuje takiej wartości).Printf Standardowej Biblioteki nie martwi się jak te znaki się tam znajdą W szczególności, nie jesteśmy ograniczeni do używania pojedynczego łańcucha po wywołaniu printf. To jest zupełnie poprawne:

```
printf
byte    „To jest łańcuch",13,10
byte    „jest w nowej lini",13,10
byte    :drukuje backspace na końcu tej linii:"
byte    8,13,10,0
```

Nasz kod będzie działał szybciej troszkę, jeśli unikniemy stosowania sekwencji znaków sterujących. Co ważniejsze, sekwencja znaków sterujących zajmuje przynajmniej dwa bajty. Możemy zakodować większość z nich jako pojedyncze bajty przez po prostu osadzenie kodu ASCII dla tego bajtu bezpośrednio do strumienia kodu. nie zapomnijmy, nie możemy wprowadzić bajtu zero do strumienia kodu. Bajt zerowy kończy łańcuch formatowany. zamiast tego użyjemy „\0x00".

Sekwencje formatowe zawsze zaczynają się od „%".Dla każdej sekwencji formatowej musimy dostarczyć daleki wskaźnik dla powiązania danej bezpośredniej występującej w łańcuchu formatowany. Np.

```
printf
byte    „%i, %1", 0
dword   i,j
```

Sekwencja formatowa przyjmuje ogólną formę „%s\cn^f" gdzie:

- % jest zawsze znakiem „%".Używamy „\%" jeśli chcemy wydrukować znak procenta
- s jest albo niczym albo znakiem minus („-„)
- „\c" jest również opcjonalny, może lub nie musi pojawiać się na pozycji formatowej ,"c" przedstawia każdy znak drukowalny
- „n" przedstawia łańcuch z jedną lub więcej cyfrą dziesiętną
- „^" jest znakiem karetki
- „f" przedstawia jeden ze znaków formatowania: i,d,x,h,u,c,s,ld,li,lx lub lu

Pozycje „s”,”\c”,”n” i „^” są opcjonalne, pozycje „%” i „f” musza być. Ponadto ,porządek tych pozycji w pozycjach formatowych jest bardzo ważny. Pozycja „\c”, na przykład nie może poprzedzać pozycji „s”. Podobnie ,znak „^” może następować za wszystkimi z wyjątkiem znaku „f”

Znaki formatowe: i,d,x,h,u,c,s,ld,li,lx i lu sterują formatem wyjściowym dla danej. Znaki formatowe i i d wykonują identyczne funkcje, mówią printf żeby drukował wartości jako 16 bitowe dziesiętne całkowite ze znakiem. znaki formatowe x i h instruują printf o drukowaniu wyszczególnionych wartości jako 16 bitowych lub 8 bitowych wartości heksadecymalnych .Jeśli wyspecyfikujemy u, printf wydrukuje wartość jako 16 bitową dziesiętną bez znaku. Używając c mówi printf aby drukował wartość jako pojedynczy znak. S mówi printf, że dostarczamy adres łańcucha zakończonego znakiem zera., printf drukuje ten łańcuch. Pozycje ld,li,lx i lu są długimi (32 bitowymi) wersjami d/l,x i u. Odpowiedni adres wskazuje na 32 bitową wartość, którą printf sformatuje i wydrukuje na standardowym wyjściu.

Następujący przykład demonstruje te pozycje formatowe:

```
printf
byte    „I = %1, U= %u, HexC= %h,HexI =%x,C =%c „
dbyte   „S = %s”,13,10
byte    „L= %ld”,13,10,0
dword   i,u,c,i,c,s,l
```

Liczba dalekich adresów (wyszczególniony przez operand „dd” pseudo- opcodu) musi zgadzać się z liczbą pozycji formatowej „%” w łańcuchu formatowego. Printf liczy liczbę pozycji formatowych „%” w łańcuchu formatowanym i pomija wiele dalekich adresów będących za formatem łańcucha **.**Jeśli liczba pozycji nie zgadza się, adres powrotny dla printf będzie nieprawidłowy a program prawdopodobnie zawieszony lub będzie źle funkcjonował. Podobnie (jak podprogram print),łańcuch formatowany musi kończyć się bajtem zerowym. Adresy kolejnych pozycji łańcucha formatowanego musi wskazywać bezpośrednio na komórkę pamięci gdzie leży wyszczególniona dana.

Kiedy używamy powyższego formatu, printf zawsze drukuje wartości używając minimalnej liczby pozycji drukowania dla każdego operandu. Jeśli chcemy wyszczególnić minimalną szerokość pola, możemy zrobić to używając opcji formatowej „n”. Pozycja formatowa z formatu „%10d” drukuje wartość całkowitą dziesiętną używając przynajmniej dziesięć pozycji drukowania. Podobnie „%16” drukując łańcuch używając przynajmniej 16 pozycji drukowania. Jeśli wartość drukowania wymaga więcej niż wyszczególniona liczba pozycji drukowania, printf użyje tyle ile trzeba. Jeśli wartość drukowania wymaga mniej, printf zawsze będzie drukował wyszczególnioną liczbę ,uzupełniając wartość pustymi polami .Printf będzie drukował wartości wyrównane do prawej strony w polu drukowania (bez względu na typ danych).Jeśli chcemy drukować wartość wyrównaną do lewej strony w pliku wyjściowym, używamy znaku formatowego „-„ jako przedrostka w polu szerokości np.

```
printf
byte    „%-17s”,0
dword   łańcuch
```

W tym przykładzie, printf drukuje łańcuch używając 17 znaków pola szerokości wyrównanego do lewej strony w polu wyjściowym.

Domyślnie, printf wypełnia na pusto pole wyjściowe jeśli wartość drukowania wymaga mniej pozycji drukowania niż wyszczególnione przez pozycję formatową. Pozycja formatowa „\c” pozwala nam zmienić znak wypełnienia .Na przykład drukując wartości, wyrównane do prawej, używając „*”,jako znak wypełnienia, użyjemy pozycji formatowej „%\10d”.Drukowanie jako wyrównanego do lewej strony będziemy używać pozycji formatowej „%-\10d”.Zauważmy,że „-„ musi poprzedzać „\”.Jest to ograniczenie bieżącej wersji oprogramowania. Operandy muszą pojawiać się w tym porządku. Normalnie kolejny adres(y) łańcucha formatowego printf muszą być dalekimi wskaźnikami do aktualnej danej do drukowania.

Czasami, zwłaszcza kiedy alokujemy pamięć na stercie (używając malloc),możemy nie znać adresu obiektu, który chcemy drukować. Możemy mieć tylko wskaźnik do danej ,którą chcemy drukować. Opcja formatowania „^” mówi printf, że kolejny daleki wskaźnik łańcucha formatowego jest adres wskaźnika do danych zamiast adresu samej danej. ta opcja pozwala nam uzyskać dostęp do danej pośrednio.

Notka: w odróżnieniu od C, podprogram printf Biblioteki Standardowej nie wspiera danych wyjściowych zmienno przecinkowych. Wprowadzenie wartości zmiennoprzecinkowej do printf zwiększy rozmiar tego podprogramu o ogromna ilość. Ponieważ większość ludzi nie potrzebuje wartości zmienno przecinkowych ,nie będą się tu pojawiać. Jest oddzielny podprogram printf ,który zajmuje się operacjami zmiennoprzecinkowymi

Podprogram printf Biblioteki Standardowej jest złożoną bestią. Jednakże jest bardzo elastyczny i niezmiernie użyteczny. Powinniśmy spędzić trochę czasu na opanowanie jego głównych funkcji. Będziemy używać tego podprogramu dosyć często w naszych programach.

Pakiet standardowego wyjścia dostarcza wielu dodatkowych podprogramów oprócz tych omówionych tutaj. Po prostu nie ma tyle miejsca aby omówić je wszystkie w tym rozdziale .po więcej szczegółów można sięgnąć do dokumentacji Biblioteki Standardowej.

7.1.4 PODPROGRAMY FORMATOWANIA WYJŚCIOWEGO:PUTISIZE,PUTUSIZE,PUTLSIZE I PUTULSIZE

Podprogramy wyjściowe puti, putu i putl łańcuchów liczbowych używają minimalnej liczby koniecznych pozycji drukowania .Na przykład, puti używa trzech znaków pozycji do drukowania wartości –12.Czasami,możemy potrzebować wyszczególnić różne pola szerokości więc możemy wyrównywać kolumny liczb lub osiągać inne zadania formatowania. Chociaż możemy użyć printf do osiągnięcia tego celu, printf ma dwie główne wady – drukuje tylko wartości w pamięci (tj. nie może drukować wartości w rejestrze) a pole szerokości wyszczególnione dla printf musi być stałe. Podprogramy putisize, putusize i putlsize przezwyciężają te ograniczenia.

Podobnie jak ich odpowiedniki puti, putu i putl, te podprogramy drukują wartości całkowite ze znakiem, całkowite bez znaku i 32 bitowe wartości całkowite ze znakiem .Oczekują wartości do drukowania w rejestrze ax (putisize i putusize) lub parze rejestrów dx:ax(putlsize).Oczekują również minimalnego pola szerokości w rejestrze. Aczkolwiek printf, jeśli wartość w rejestrze cx jest mniejsza niż liczba pozycji drukowania ta liczba w rzeczywistości potrzebuje drukować, putisize ,putusize i putlsize ignorują tą wartość w cx i drukuje wartość używając minimalną konieczną liczbę pozycji drukowania.

7.1.5 PODPROGRAMY ROZMAIRU PÓL WYJŚCIOWYCH: ISIZE,USIZE I LSIZE

Raz na jakiś czas, możemy chcieć znać liczbę pozycji drukowania wartości wymaganej przed właściwym drukowaniem tej wartości .Na przykład, możemy chcieć obliczyć maksymalną szerokość zbioru liczb więc możemy drukować je w formacie kolumnowym automatycznie modyfikować szerokość pola dla największej liczby w zbiorze. Podprogramy isize, usize i lsize zrobią to za nas.

Podprogram isize oczekuje wartości całkowitej ze znakiem w rejestrze ax. Zwraca minimalną szerokość pola tej wartości (zawierającą pozycję dla znaku minus, jeśli to konieczne) w rejestrze ax. Usize oblicza rozmiar wartości całkowitej bez znaku w ax i zwraca minimalną szerokość pola w rejestrze ax. Lsize oblicza minimalną szerokość wartości całkowitej ze znakiem w dx:ax (zawierającą pozycję dla znaku minus, jeśli to konieczne) i zwraca tą szerokość w rejestrze ax.

7.1.6 PODPROGRAMY KONWERSJI :ATOx I xTOA

Biblioteka Standardowa dostarcza kilku podprogramów do konwersji między łańcuchem a wartościami liczbowymi. Są to atoi, atoh, atou ,itoa, htoa, wtoa i utoa (plus inne).Podprogramy ATOx konwertują łańcuch ASCII w stosownym formacie do wartości liczbowej i zostawia tą wartość w ax lub al. .Podprogramy ITOx konwertują wartość w al. ./ax do łańcucha cyfr i przechowuje ten łańcuch w buforze którego adres jest w es:di. Jest kilka wariantów każdego podprogramu który operuje w różnych przypadkach .Następny paragraf opisuje każdy podprogram.

Podprogram atoi zakłada, że es:di wskazuje na łańcuch zawierający cyfry całkowite (i być może znak minus).Konwertuje ten łańcuch na wartość całkowitą i zwraca wartość całkowitą do ax. Po powrocie es:di wskazuje jeszcze na początek łańcucha. Jeśli es:di nie wskazuje na łańcuch cyfr na wejściu lub jeśli wystąpiło przepełnienie, atoi zwraca ustawienie flagi przeniesienia. Atoi zachowuje wartość pary rejestrów es:di. Wariant atoi,atoi2 również konwertuje łańcuch ASCII na wartość całkowitą z wyjątkiem tego, że nie zachowuje wartości w rejestrze di. Podprogram atoi2 jest szczególnie użyteczny jeśli musimy skonwertować sekwencję liczb pojawiającą się w tym samym łańcuchu Każde wywołanie atoi2 pozostawia rejestr di wskazując na pierwszy znak poza łańcuchem cyfr. Możemy łatwo przeskoczyć każdą spację ,przecinek lub inne znaki ograniczające dopóki nie dotrzemy do następnej liczby w łańcuchu. Potem możemy wywołać atoi2 do konwersji tego łańcucha na liczbę. Możemy powtórzyć to działanie dla każdej liczby w linii.

Atoh pracuje podobnie jak podprogram atoi, z wyjątkiem tego, że oczekuje łańcucha zawierającego cyfry heksadecymalne .Po powrocie ax zawiera skonwertowaną 16 bitową wartość i flagę przeniesienia oznaczającą błąd. brak błędu. Podobnie jak atoi, podprogram atoh zachowuje wartości w parze rejestrów es:di .Możemy wywołać atoh2 ,jeśli chcemy aby rejestr es:di wskazywał na pierwszy znak poza końcem łańcucha cyfr heksadecymalnych.

Atou konwertuje łańcuch ASCII cyfr dziesiętnych w zakresie 0..65,536 do wartości całkowitej i zwraca tą wartość w ax. Z wyjątkiem tego, że nie jest dozwolony, ten podprogram zachowuje się jak atoi. jest również podprogram atou2 który nie przechowuje wartości w rejestrze di; di wskazuje na pierwszy znak poza łańcuchem cyfr dziesiętnych.

Ponieważ nie ma podprogramów geti, geth lub getu dostępnych w Bibliotece Standardowej ,będziemy musieli stworzyć je sami. Poniższy kod demonstruje jak odczytać wartość całkowitą z klawiatury:

```
print
```

```
        byte        „Wprowadź wartość całkowitą:”,0
        getsm
        atoi        ;konwersja łańcucha na wartość całkowitą w AX
        free        ;powrót alokowanej pamięci przez getsm
        print
        byte        „Wprowadziłeś” . 0
        puti        ;drukuj wartość zwróconą przez ATOI
        putcr
```

Podprogramy itoa utoa, htoa i wtoa są logicznymi odwróceniami podprogramu atox. Konwertują one wartości liczbowe do całkowitych, bez znakowych i heksadecymalnych łańcuchów. Jest kilka wariantów tych podprogramów w zależności od tego czy chcemy automatycznie alokować pamięć dla łańcucha lub czy chcemy je zachować w rejestrze di.

Itoa konwertuje 16 bitowe wartości całkowite ze znakiem w ax do łańcucha i przechowuje znaki tego łańcucha poczynając od lokacji es:di. Kiedy wywołujemy itoa ,musimy zapewnić, że es:di wskazuje na tablicę znaków dość dużą do przetrzymywania łańcuchów wynikowych. Itoa wymaga maksymalnie siedmiu bajtów dla tej konwersji :pięciu cyfr ,znaku i bajtu zakończonego zerem. Itoa zachowuje wartości w parze rejestrów es:di, więc na powrót es:di wskazuje na początek łańcucha stworzonego przez itoa.

Czasami możemy nie chcieć przechowywać wartości w rejestrze di kiedy wywołujemy podprogram itoa. Na przykład, jeśli chcemy stworzyć pojedynczy łańcuch zawierający kilka skonwertowanych wartości, byłoby miło gdyby itoa opuścił di wskazujący na koniec łańcucha zamiast na początek. Podprogram itoa2 robi to dla nas; pozostawia rejestr di wskazując na bajt zakończony zerem na końcu łańcucha. Rozważmy następujący segment kodu który stworzy łańcuch zawierający reprezentację ASCII dla trzech zmiennych całkowitych Int1,Int2 i Int3:

```
;zakładamy, że es:di już wskazuje na lokację początkową przechowująca skonwertowane wartości całkowite
        mov     ax, Int1
        itoa2                   ;konwersja Int1 do łańcucha
        mov     byte ptr es:][di], ‘ ‘
        inc     di
,Konwersja drugiej wartości
        mov     ax, Int2
        itoa2
        mov     byte ptr es:[di]
        inc di
;Konwersja trzeciej wartości
        mov ax, Int3
        itoa2
;w tym miejscu di wskazuje na koniec łańcucha zawierającego skonwertowane wartości .Szczęśliwie jeszcze
;wiemy gdzie zaczyna się łańcuch więc możemy nim manipulować!
```

Inny wariant podprogramu itoa, itoam, nie wymaga inicjacji pary rejestrów es:di .Podprogram ten wywołuje malloc automatycznie alokując pamięć. Zwraca wskaźnik do skonwertowanego łańcucha na stercie w parze rejestrów es:di. Kiedy skończymy z łańcuchem, powinniśmy wywołać free aby zwrócić pamięć ze sterty.

```
;następujący fragment kodu konwertuje wartość całkowitą w AX do łańcucha i drukuje ten łańcuch. Oczywiście,
możemy zrobić to samo z użyciem PUTI, ale ten kod demonstruje jak wywołać itoam
        itoam                   ;konwertuje wartość całkowitą do łańcucha
        puts                    ;drukuje łańcuch
        free                    ;zwraca pamięć ze sterty
```

Podprogramy utoa,utoa2 i utoam pracują dokładnie tak jak itoa,itoa2 i itoam., z wyjątkiem tego, że konwertują wartości całkowite bez znaku w ax do łańcucha. Zauważmy, że utoa i utoa2 wymagają sześciu bajtów ponieważ nigdy nie przetwarzają znaku ze znakiem.

Wtoa,wtoa2 i wtoam konwertują 16 bitową wartość w ax do łańcucha z dokładnie czterema znakami heksadecymalnymi plus bajt zakończony zerem. W przeciwnym razie, zachowują się dokładnie jak itoa,itoa2 i i itoam. Zauważmy, że te podprogramy przetwarzają zero początkowe więc wartość jest zawsze długa na cztery cyfry.

Podprogramy htoa,htoa2 i htoam są podobne do wtoa,wtoa2 i wtoam. Jednakże ,podprogramy htoax konwertują ośmio bitową wartość w al. do łańcucha z dwoma znakami heksadecymalnymi plus bajt zakończony zerem.

Biblioteka Standardowa dostarcza kilku innych podprogramów konwersji poza tymi wymienionymi w tej sekcji.

7.1.7 PODPROGRAMY TESTUJĄCE ZNAKI DLA PRZYANALEŻNOSCI DO ZBIORU

Biblioteka Standardowa UCR dostarcza wiele podprogramów ,które testują znak w rejestrze al. aby sprawdzić czy należy do pewnego zbioru znaków. Te podprogramy wszystkie zwracają stan we fladze zera .Jeśli warunek jest prawdziwy, ustawiają flagę zera (więc możemy przetestować warunki instrukcją je). Jeśli warunek jest fałszywa, zerują flagę zera (testujemy instrukcją jne),Te podprogramy to:

* IsAlNum- Sprawdza czy al. zawiera znaki alfanumeryczne
* IsXDigit- Sprawdza al. czy zawiera znaki cyfr heksadecymalnych
* IsDigit- Sprawdza al. czy zawiera znaki cyfr dziesiętnych
* IsAlpha Sprawdza al. czy zawiera znaki alfabetu
* IsLower Sprawdza al. czy zawiera znaki małych liter
* Is Upper Sprawdza al. czy zawiera znaki dużych liter

7.1.8 PODPROGRAMY KONWERSJI ZNAKÓW: TOUPPER,TOLOWER

Podprogramy ToLower i ToUpper sprawdzają znak w rejestrze al. Skonwertują znak w al. do właściwej wielkości znaku.

Jeśli al. zawiera znak alfabetyczny małej litery, ToUpper skonwertuje go do odpowiedniego znaku dużej litery. Jeśli al zawiera jakiś inny znak, ToUpper zwróci go niezmienionym.

Jeśli al zawiera znak alfabetyczny dużej litery, ToLower skonwertuje go do odpowiedniego znaku małej litery .Jeśli wartość nie jest znakiem alfabetycznym dużej litery ToLower pozostawi go niezmienionym.

7.1.9 GENEROWANIE LICZB LOSOWYCH: RANDOM,RANDOMIZE

Podprogram Random Standardowej Biblioteki generuje sekwencje pseudo – losowych liczb. Zwraca wartość losową w rejestrze ax w każdym wywołaniu. Możemy potraktować tą wartość jako wartość ze znakiem lub bez, ponieważ Random manipuluje wszystkimi 16 bitami rejestru ax.

Możemy użyć instrukcji div lub i div do zmuszenia do przetwarzania random w wyszczególnionym zakresie. Podzielenie wartości losowej zwraca jakąś liczbę n a reszta tego dzielenia będzie wartością z zakresu 0..n-1.Na przykład, obliczenie liczby losowej z zakresu 1..10,może wymagać kodu jak następuj:

```
random          ;pobiera losową liczbę z  zakresu 0..65,536
sub     dx, dx  ;powielenie zera do 16 bitów
mov     bx, 10  ;chcemy wartości z zakresu 1..10
div     bx      ;reszta idzie do dx!
Inc     dx      ;konwersja 0..9 do 1..10
```

;w tym punkcie, liczba losową z zakresu 1..10 znajduje się w rejestrze dx.

Podprogram random zawsze zwraca tą samą sekwencję wartości kiedy program ładuje się z dysku i wykonuje. Random używa wewnętrznej tablicy wartości ziarnistej która przechowuje część swojego kodu. Ponieważ wartości te są stałe i zawsze ładują do pamięci z programu, algorytm którego używa random zawsze stworzy tą samą sekwencję wartości kiedy program zawiera go ładując z dysku i zaczynając program. Może to nie wyglądać „losowo”, ale, faktycznie ,jest to miła cecha ponieważ jest bardzo trudno przetestować program który naprawdę używa wartości losowych. Jeśli generator liczb losowych tworzy tą samą sekwencję liczb, żaden test uruchomiony w tym programie nie będzie powtarzany

Niestety, jest wiele przykładów programów które możemy chcieć napisać (np. gry) gdzie posiadanie powtarzalnego wyniku nie jest do przyjęcia. Dla takich aplikacji możemy wywołać podprogram randomize. Randomize używa bieżącej wartości zegara systemowego do generowania prawie losowej sekwencji startowej. Więc jeśli potrzebujemy (prawie) unikalnej sekwencji liczb losowych, w każdym czasie kiedy program zaczyna się wykonywać, wywołujemy podprogram randomize przed każdym wywołaniem podprogramu random. Zauważmy, że jest mały profit z wywołania podprogramu randomize więcej niż jeden raz w programie. Raz random ustala punkt startowy random, dalsze wywołanie randomize nie poprawi jakości (przypadkowości) liczb przez niego generowanych.

7.1.10 STAŁE,MAKRA I INNE RÓŻNOŚCI

Kiedy stosujemy plik nagłówkowy „stdlib.a”, możemy zdefiniować pewne makra (zobacz Rozdział Ósmy opis makr) i powszechnie stosujemy stałe. Zawiera się to następująco:

```
NULL    =       0       ;jakiś pewien kod ASCII
BELL    =       07      ;znak dzwonka(??????chyba)
bs      =       08      ;znak backspace
tab     =       09      ;znak tabulatora
lf      =       0ah     ;znak przesunięcia do nowej linii
cr      =       odh     ;powrót karetki
```

W dodatku do powyższych stałych,”stdlib.a” również definiuje kilka użytecznych makr zawierających ExitPgm,lesi i ldxi.Makra te zawierają następujące instrukcje:

```
;ExitPgm – zwraca sterowanie do MS-DOS
ExitPgm     macro
            mov     ah, 4ch         ;opcod zakończenia programu DOSowskiego
            int     21h             ,wywołanie DOS
            endm
;LESI ADR – ładuje ES:DI adresem wyszczególnionym przez operand
 lesi       macro   adrs
            mov     di, ser adrs
            mov     es, di
            mov     si, offset adrs
            endm

;LDXI ADRS – ładuje DX:SI adresem wyszczególnionym przez operand
ldxi        macro   adrs
            mov     dx, seg adrs
            mov     si, offset adrs
            endm
```

Makra lesi i ldxi są zwłaszcza użyteczne dla ładowania adresów do es:di lub dx:si przed wywołaniem kilku podprogramów standardowej biblioteki.

7.1.11 PLUS WIĘCEJ!

Biblioteka Standardowa zawiera wiele ,wiele podprogramów których ten rozdział nie omawiał .Większość można znaleźć w dokumentacji Biblioteki Standardowej. Podprogramy omówione w tym rozdziale są podprogramami, których będziemy używali najczęściej.

7.5 PODSUMOWANIE

Rozdział ten wprowadził kilka dyrektyw asemblera i pseudo opcodów wspieranych przez MASM. Omówił również krótko podprogramy w Standardowej Bibliotece UCR dla Programistów Języka Asemblera 80x86.Nie znaczy to, że rozdział ten omówił komplet tego co MASM lub Biblioteka Standardowa nam oferują. Dostarczył tylko dość informacji dla rozpoczęcia działań.

Aby pomóc w pisaniu programów asemblerowych z minimalnym zamieszaniem, tekst ten korzysta w znacznym stopniu z kilku podprogramów Biblioteki Standardowej. Chociaż rozdział ten z pewnością nie omówił wszystkich jej podprogramów, omówił wiele często używanych.

7.6 PYTANIA

1. Jakiego pliku powinniśmy używać na początku naszego programu kiedy piszemy kod używający Standardowej Biblioteki UCR?
2. Jaki podprogram alokuj pamięć na stercie
3. Jakiego podprogramu będziemy używać do drukowania pojedynczego znaku?
4. Jaki podprogram pozwala nam drukować łańcuch stałych znakowych na wyświetlaczu?
5. Biblioteka Standardowa nie dostarcza podprogramu do odczytu wartości całkowitych od użytkownika. Opisz, jak zastosować podprogramy GETS i ATOI do wykonania tego zadania.
6. Jak jest różnica między podprogramem GETS a GETSM?
7. Jaka jest różnica między podprogramem ATOI a ATOI2?
8. Co robi podprogram ITOA? Opisz wartości wejściowe i wyjściowe