מודלים של זיכרון של Turbo C

בעידן של ה-8086 יעילות אלקטרונית היה עניין חשוב. כאשר מימשו קומפילרים באותה תקופה הדבר הביא, בין השאר, להתלבטות בכל הקשור לשימוש בהסתעפויות מסוג NEAR או FAR (ופוינטרים לפונקציה 32 ביט) על מנת להסתעף לתתי תוכניות וכן בזמן מימוש פוינטרים למידע האם להשתמש בפוינטרים מלאים 32 ביט (16 להיסט ו-16 לסגמנט) או לעבוד בפוינטרים 16 ביט (היסטים בלבד). יש כאן tradeoff בין מהירות לבין כלליות: בחירה באפשרויות המהירות יותר (הסתעפיות NEAR או פוינטרים 16 ביט) מגבילה את הקוד המשתמש בהם לסגמנט יחיד לכל סוג שימוש (קוד או מידע). על כך נרחיב בהמשך.

בקוד אסמבלי טהור, המתכנת יכול לממש רוטינות שלו כך שחלק מהרוטינות יהיו NEAR כשאפשר והאחרות FAR כשצריך. אבל כאן המתכנת לוקח על עצמו את האחריות. כאשר מנסים לממש מדיניות כזו בקומפילר הדבר הרבה יותר קשה, כי קשה לחזות מראש את כל התסריטים, ובפועל לא עשו את זה. מיקרוסופט ובעקבותם בורלנד מימשו מדיניות אחידה בכל הקשור למימוש קוד של קובצי המקור בכל פעולת קומפילציה מסוימת. יחד עם זאת, לפחות עבור C הם השאירו את האופציות פתוחות, כלומר המתכנת יכול לבחור איזה משתי האופציות בשתי הקטגוריות הוא צריך. הם קראו לאפשריות הללו מודלים של זיכרון (Memory Models). המתכנת יכול לבחור האם הוא לממש את ההסתעפיות ב-NEAR או ב-FAR ובאופן בלתי תלוי אם הפוינטרי למידע יהיו מלאים 32 ביט ע"י בחירה באופצית מודל הזיכרון המתאים.

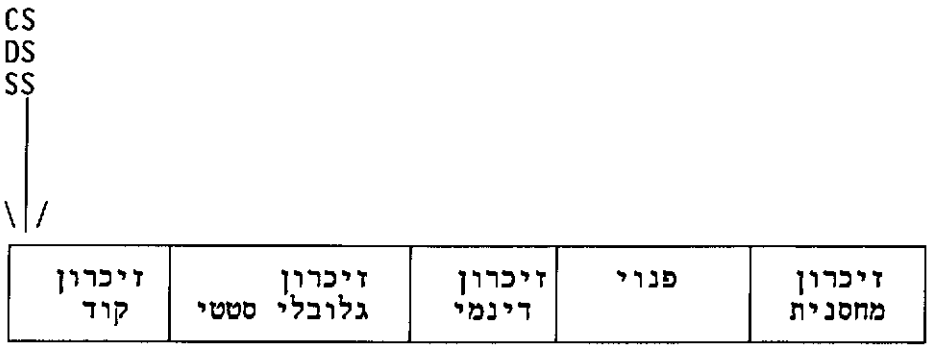
תוכנית שנטענת לזיכרון מקצה זיכרון ל-4 סוגי שימושים:

- שטח זיכרון לפקודות התוכנית - הקוד של התוכנית. השטח הזה נקבע בדיוק בזמן קומפילציה והוא תוצר של מספר הפקודות של התוכנית ואופיים.
- שטח זיכרון למשתנים גלובליים / סטטיים. גם השטח הזה נקבע בדיוק בזמן קומפילציה והוא תוצר של מספר המשתנים הגלובליים / סטטיים המוכרזים מראש בתוכנית.
- שטח זיכרון להקצאות דינמיות. הגודל המירבי של השטח הזה שתוכנית צריכה נקבע בפועל בזמן ריצה ואינו חייב להיות אחיד לכל הריצות של התוכנית. הוא נקבע על ידי בקשות ההקצאה (והשחרור) שיוזמת נתוכנית.
- שטח המחסנית. כמו שטח ההקצאה הדינמי, הגודל המירבי של השטח הזה שתוכנית צריכה נקבע בפועל בזמן ריצה ואינו חייב להיות אחיד לכל הריצות של התוכנית. הוא נקבע על ידי עומק הקריאות לפרוצדורות, הפרמטרים והמשתנים האוטומטיים שמוכרזות ע"י הפרוצדורות הללו.

הקומפילרים של מיקרוספט C ובעקבותיו טורבו C תומכים בשישה מודלים של זיכרון (אנחנו נתעכב בעיקר על ארבעת הראשונים המתוארים להלן):

מודל Tiny - כל התוכנית נדחסת לסגמנט אחד. כל אוגרי הסגמנטים מכילים בדיוק את אותו ערך ושומרים עליו לכל אורך הריצה של התוכנית. לפיכך כל אוגרי הסגמנטים מצביעים לאותו סגמנט ולפיכך כל ההסתעפויות יכולים להיות ובפועל הם NEAR וכל הפוינטרים 16 ביט. משמעות: כל שטחי הזיכרון, השטח של הקוד, שטחי המשתנים הגלובליים/סטטיים כולם יחד יכולים לתפוס לכל היותר 64K.

תמונת הזיכרון של מודל tiny הוא כלהלן:

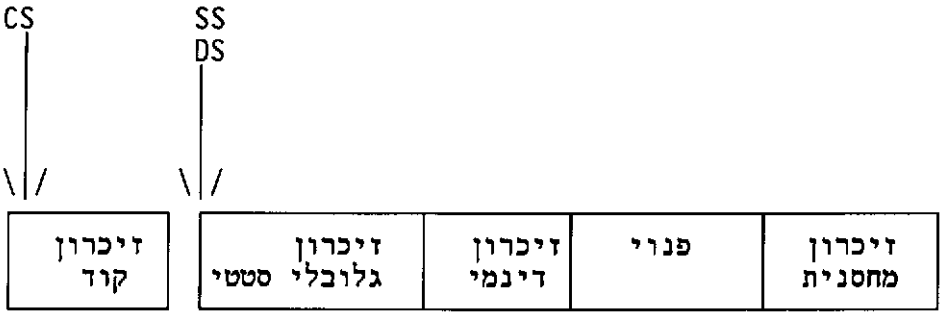


לפיכך תוכנית המקומפלת במודל Tiny תהיה המהירה ביותר האפשרית בכל הקשור להסתעפויות ושימוש בפוינטרים בהשוואה לאפשרויות הקימפול האחרות אבל מצד שני התוכנית המקומפלת במודל Tiny יכולה לנצל לכל היותר 64K וכל הקצאת שטח לאחד המטרות (מימוש פקודה למשל) הוא על חשבון השטח שניתן להקצות למטרות אחרות.

מודל Small

מודל Small מבוסס על העובדה שאפשר לשחרר חלק מהמגבלות על השימוש בשטח זיכרון של מודל Tiny <u>מבלי להפסיד</u> מאומה בכל הקשור למהירות הריצה של התוכנית. הדרך לעשות זאת הוא להקצות בדיוק סגמנט אחד לקוד וסגמנט אחר למידע, כל סגמנט ישמש אך ורק למטרה שהוא הוקצה לה, עדיין להשתמש בהסתעפויות NEAR ופוינטרים 16 ביט בלבד ולתת <u>לאופי הפעולה</u> (הסתעפות או גישה לזיכרון למידע) לקבוע איזה אוגר סגמנט משתמשים. לפיכך תוכנית המקומפלת בבמודל Small יכולה לנצל עד 128K זיכרון, אבל רק בצורה של 64K לקוד ו-64K למידע. לא ניתן להגדיל את הזיכרון לאחד המטרות על חשבון הצורה

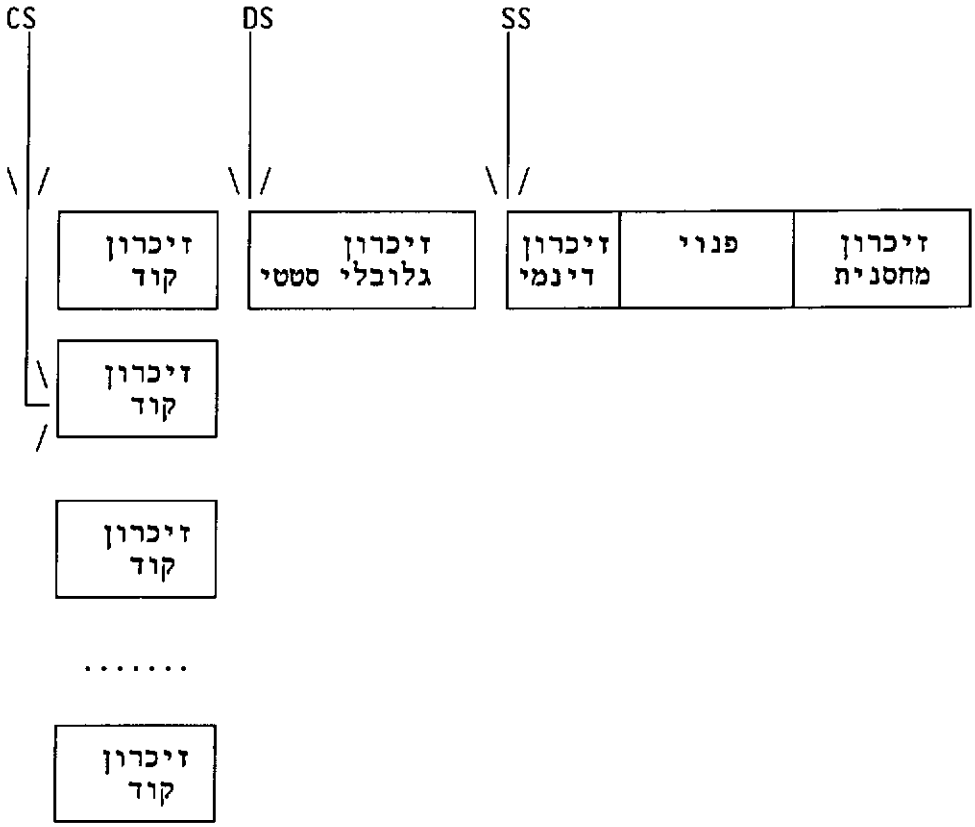
השניה. תמונת אוגרי הסגמנטים במודל Small נראית כך:



מודל Large

כמעט בקצה השני של האפשרויות הוא מודל Large. כאן מוותרים לחלוטין על נושא המהירות לטובת גודל או "כלליות" של התוכנית. במודל הזה הקוד יכול לתפוס כמה סגמנטים לפי הצורך (המגבלה היחידה היא ה-640K של כלל מערכת ה-DOS), ואליו לשטח הגלובלי / סטטי יש סגמנט יחיד משלו ולשטח הדינמי + מחסנית סגמנט יחיד משלו. במהלך ריצה של תוכנית במודל Large, אוגר ה-CS עשוי להשתנות בכל רגע, אך אוגרי ה-SS וה-DS שומרים על ערכם במהלך הריצה של התוכנית, אם כי DS ו-SS אינם שווים. כאן כל ההסתעפיות לפרצדורות הם FAR וכל הפוינטרים 32 ביט. הפוינטרים למידע חייבים להיות 32 ביט משום שכל פוינטר עשוי בהזדמנות אחת להצביע על שטח גלובלי/סטטי ובהזדמנות אחרית על שטח מחסנית או דינמי. תוכנית המקומפלת במודל Large היא כמעט ולא מוגבלת בגודל השטח המוקצה לקוד, אך שטח המידע של התוכנית מוגבל ל-128K כאשר 64K מתוכם יכולים להיות מנוצלים אך ורק למשתנים גלובליים/סטטיים ואילו רק עד 64K יכולים להיות מנוצלים לשטח דינמי + מחסנית. אי אפשר להגדיל את שטח המחסנית למעבר ל-64K על חשבון השטח הגלובלי /סטטי או להיפך. אך מתכנת המודע למגבלות הללו יכול לכתוב את התוכנית כך שיהיה למעשה חסכון באחד מסוגי השימוש לטובת השני. המקרה הנפוץ הוא ששטחי משתנים של פרוצדורות ימומשו גלובלית או סטטית, דבר שיקטין את הביקוש לשטח מחסנית על חשבון הסגמנט הגלובלי/סטטי. אגב, שיקולים מהסוג הזה קיימים גם במסגרות אחרות כמו Unix.

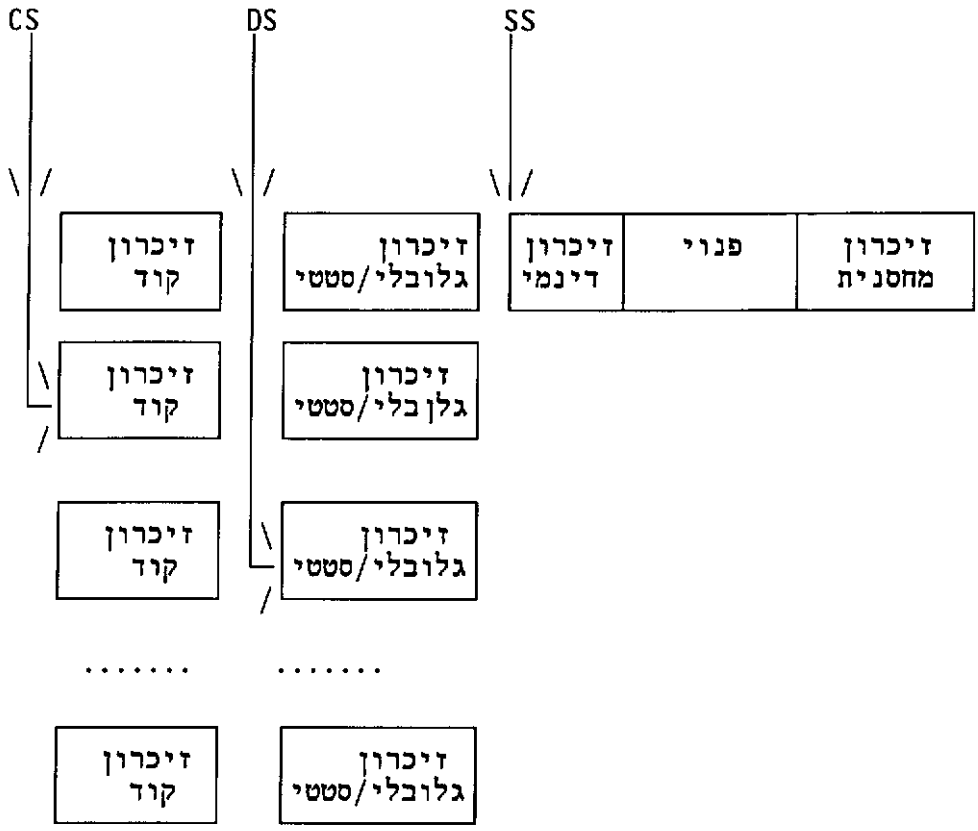
תמונת הזיכרון של מודל Large נראה עקרונית כך:



מודל Huge

מודל Huge הוא המודל הכללי או החזק ביותר, ועקרונית דומה למודל Large למעט שהשטח הגלובלי/סטטי אינו מוגבל לסגמנט יחיד. המחסנית לעומת זאת ממשיכה להיות מוגבלת לסגמנט אחד. לפיכך תוכנית המקומפלת במודל Huge היא כאילו לא מוגבלת הן בקוד והן במידע (שוב עד למגבלה הכוללת של 640K של ה-DOS). אולם ככל שמדובר בשימוש בזיכרון למידע, היכולת לנצל שטחי זיכרון גדולים מותנית בכך שהם יוכרזו כמשתנים גלובליים/סטטיים. תוכניתן הרוצה לכתוב תוכנית המנצלת שטחי זיכרון שהסה"כ שלהם עולה על 64K חייב את השטח העודף לתכנת כמשתנים גלובליים/סטטיים.

תמונת הזיכרון של תוכנית המקומפלת במודל Huge נראית כך:



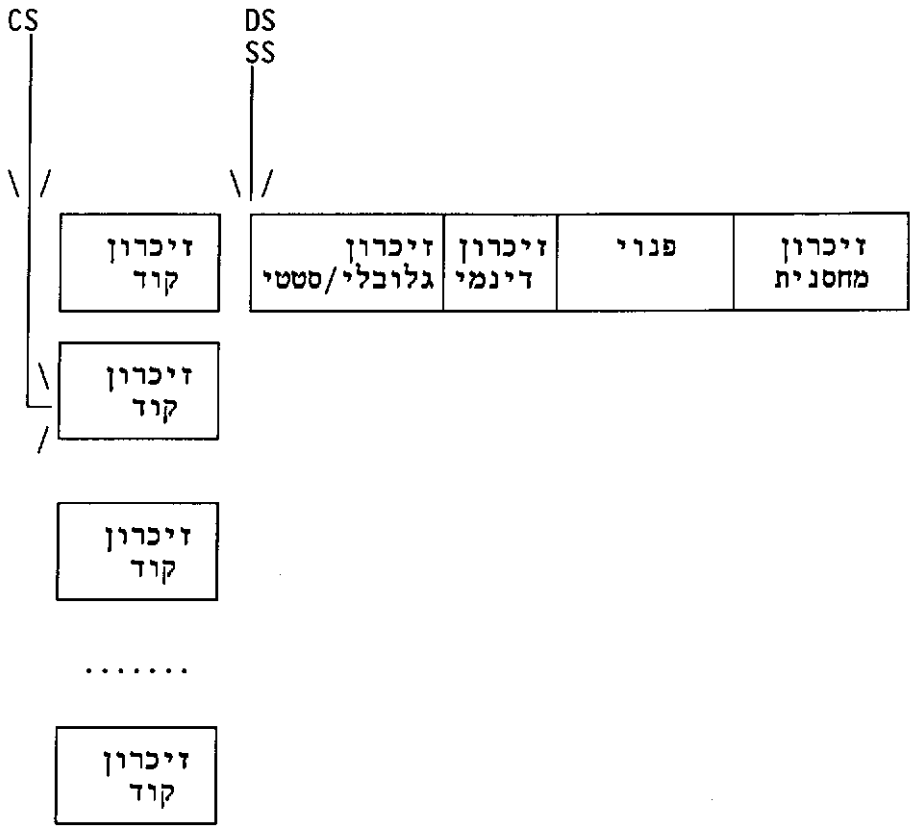
<u>המודלים הנותרים</u>

המודלים האחרים הם פשוט מודלי ביניים בין Small ל-Large. משום כך התעכבות עליהם לא תחכים אותנו בהרבה, אבל בקצרה:

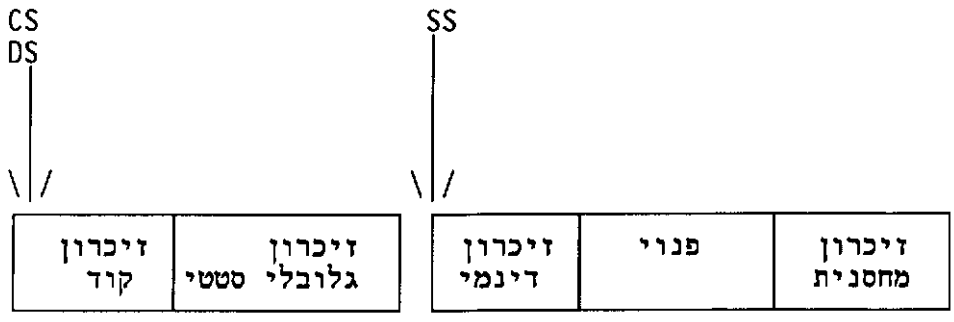
מודל Medium - שטח הקוד יכול להתפרס על מספר רצוי של סגמנטים כמו במודל Large אבל שטח המידע כולו (גלובלי/סטטי, דינמי, מחסנית) נכנס לסגמנט אחד כמו במודל Small. לפיכך כאן כל ההסטעפויות הם FAR, פוינטרים <u>לפונקציה</u> הם 32 ביט אבל הפוינטרים <u>למידע</u> הם 16 ביט.

מודל Compact - שטח הקוד והשטח הגלובלי/סטטי הם יחד בסגמנט אחד ואילו השטח הדינמי והמחסנית נמצאים בסגמנט נפרד. לפיכך גודל הקוד של התוכנית מגביל את גודל השטח הדינמי/סטטי ולהיפך אבל גודל השטח הגלובלי/סטטי לא מגביל את השטח הדינמי והמחסנית. במודל הזה כל ההסתעפויות הם NEAR, הפוינטרים לפונקציה הם 16 ביט. לעטמת זאת פוינטרים <u>למידע</u> הם32 ביט.

תמונת הזיכרון של מודל Medium:



מודל הזיכרון של מודל Compact:



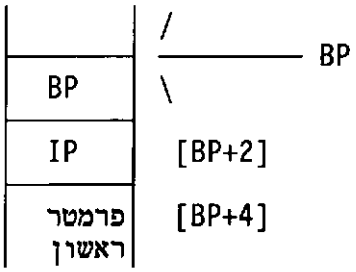
תוכניתן הכותב תוכנית אסמבלי הנקרא מתוכנית טורבו C חייב להיות מודע לאיזה מודל הוא כותב את התוכנית, גם אם למען האמת הוא הוא לא צריך להבין לעומק את תמונת הזיכרון של המודל. תוכנית אסמבלי שנכתבת למודל Small לא תעבוד למודל Large. העובדות שתוכניתן האסמבלי צריך להיות מודע להן הן רק סוגי ההסתעפויות של המודל (FAR, NEAR) וגודל הפוינטרים למידע (16 או 32 ביט).

<u>סוג ההסתעפויות</u> - אם המודל משתמש בהסתעפויות NEAR בכדי להסתעף לפרוצדורה (מודלים Tiny, Small, Compact) אזי הרוטינה הנקראית מ-C חייבת להיות מוגדרת גם היא NEAR, ובהנחה שהפקודות הראשונות של הרוטינה הם

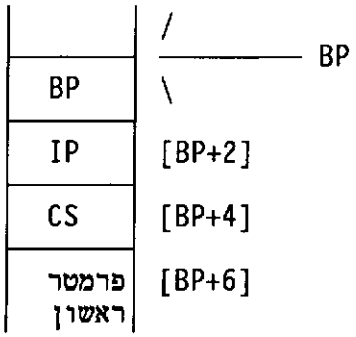
```
PUSH BP
MOV BP,SP
```

אזי הפרמטר הראשון המועבר אל הרוטינה נמצא בכתובת [BP+4].

זאת משום שתמונת המחסנית היא



אם לעומת זאת המודל משתמש בהסתעפיות מסוג FAR (מודלים Medium, Large, Huge), הפרמטר הראשון יהיה ב- [BP+6] משום שתמונת המחסנית עכשיו תהיה



בנוסף לכך, גודל <u>הפוינטרים לפונקציה</u> בתוכנית (נניח שהם מועברים כפרמטרים, או משתנים גלובליים) תלויים רק בסוגי ההסתעפויות של המודל. אם סוג ההסתעפויות היו NEAR אז הפוינטר לפונקציה ממומש 16 ביט בלבד (ערך חדש ל-IP) ואם סוג ההסתעפויות היא FAR אזי הפוינטר לפונקציה היא 32 ביט (ערך חדש ל-IP בבכתובות הנמוכות, ערך חדש ל-CS בכתובות המרוחקות יותר). לדוגמא, נניח שבתובת [BP+10] נמצא פוינטר לפונקציה, אם מדובר במודל שמשתמש בהסתעפויות NEAR אז הפוינטר לפונקציה יתפוס 2 בתים שהם ערך חדש ל-IP והסתעפויות דרכו יעשו בפקודה נוסך

```
CALL WORD PTR [BP+10]
```

לעומת זאת, אם מדובר במודל המשתמש בהסתעפויות FAR אזי בכתובות [BP+10] - [BP+11] ישב ערך חדש ל-IP ובכתובות [BP+12] - [BP+13] ישב ערך חדש ל-CS. הסתעפות דרך הפוינטר יהיה דרך פקודה נוסך

```
CALL DWORD PTR [BP+10]
```

<u>סוג הפוינטרים למידע</u> - במידה ופרוצדרת האסמבלי נקראית מתוכנית C המקומפלת במודל הממומש פוינטרים למידע 16 ביט (מודלים Tiny, Small ו-Medium) כל פוינטר למידע הוא 16 ביט ומכיל היסט בלבד אם למשל בכתובת [BP+10] יש פוינטר כזה, שימוש בו יעשה ע"י פקודות נוסך

```
MOV BX,[BP+10]
MOV AX,[BX]
```

לעומת זאת, במידה ופרוצדרת האסמבלי נקראית מתוכנית C המקומפלת במודל הממומש פוינטרים 32 ביט (מודלים Compact, Large ו-Huge) כל פוינטר למידע

הוא 32 ביט. אם למשל בכתובת [BP+10] יש פוינטר כזה, הוא מכיל היסט בכתובות [BP+11] - [BP+10] ומספר סגמנט בכתובות [BP+12] - [BP+13]. השימוש בו יעשה ע"י פקודות נוסך

```
MOV BX,[BP+10]
MOV ES,[BP+12]
MOV AX,ES:[BX]
```

כלומר שימוש אופיני של פוינטר כזה יכלול השמה לאוגר סגמנט ושימוש ב-Segment Override.

<u>התיחסות לזיכרון המחסנית דרך ה-BP</u>

בכל המודלים האוגר SS לא משנה את ערכו, המחסנית הוא תמיד סגמנט אחד, ולפיכךהמתכנת אינו צורך לדאוג לכך שתוכן אוגר ה-SS יצביע לסגמנט "הנכון". מהסיבה הזו בהתיחסיות לזיכרון המחסנית יכולות להעשות תמידבעזרת האוגר ה-BP ללא צורך ב-Segment Override נוסח :SS וללא צורך בעדכון של תוכן אוגר ה-SS לפני כן.

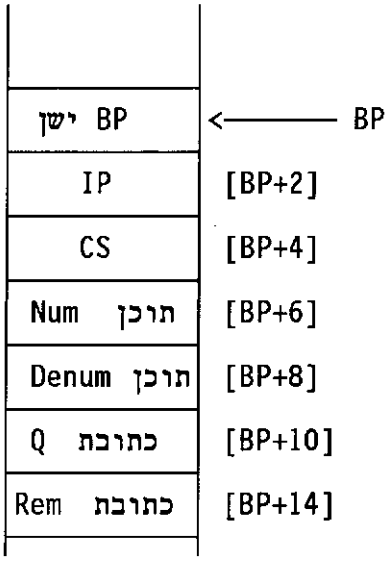
<u>הכרזת משתנים בסגמנט ה-DATA.</u>

בכל המודלים למעט מודל Huge ערך ה-DS לא משנה את ערכו במהלך התוכנית. לפיכך בכל תוכנית אסמבלי המשתלבת בקוד C והמקומפלת בכל המודלים למעט מודל Huge ניתן להכריז על משתנים (סטטיים) תחת ה-DATA. או ה-DATA_ ולהתיחס בהם בתוכנית ללא חשש שה-DS לא בהכרח מצביע לסגמנט הנכון. אין צורך לעדכן את ערכו של ה-DS ואין צורך ב-Segment Override. יוצא דופן הוא מודל הה-Huge, תוכנית אסמבלי הנקראית מקוד C במקומפל במודל Huge <u>כן</u> צריכה להתיחס לנושא הזה, נניח באמצעות האופרטור SEG.

<u>חשוב לזכור</u>: כאשר פוינטר (למידע או לפונקציה) מועבר כפרמטר, הגודל שלו קובע את מיקומו של הפרמטר הבא אם ישנו כזה. במידה ופוינטר מועבר במחסנית בכתובת [BP+10] והפוינטר הוא 16 ביט, הפרמטר הבא נמצא בכתובת [BP+12]. לעומת זאת אם הוא פוינטר ל-32 ביט הפרמטר הבא נמצא בכתובת [BP+14]. לפיכך יש כאן השפעה נוספת של המודל על מיקום הפרמטרים, שצריך להביא בחשבון. במודלים Compact ו-Medium יש הבדל בהשפעה אם מדובר בפוינטר לפונקציה או למידע. במודלים האחרים זה פשוט יותר: ב-Tiny ו-Small הפונטרים מכל סוג הם 16 ביט, ב-Large ו-Huge הפוינטרים מכל סוג 32 ביט.

תוכניות דוגמא call_id1.c, idiv_mo7.asm

בתוכנית המשולבת call_id1.c, idiv_mo1.asm ההנחה ב-idiv_mo1.asm היתה שהקימפול הוא במודל Small. בקובץ idiv_mo7.asm הפונקציה idiv_mod ממומשת עבור קימפול במודל Large. תמונת המחסנית היא עכשיו כדו:



מבנה המחסנית השתנה בשני הקשרים: הפרמטר הראשון הוא ב-[BP+6] במקום ב-[BP+4] ומאחר והפונטרים עכשיו 32 ביט, הפרמטר "כתובת Rem" נמצא במרחק 4 בתים מתחילת הפרמטר "כתובת Q". בנוסף לכך כל שימוש בפוינטרים מחייב מוערבות של אוגר סגמנט. בתור אוגר סגמנט נעשה שימוש ב-ES, והוא נטען בחצי המרוחק של שני הפונטרים (BP+12 ו-BP+16). בפוינטר מלא ההיסט תמיד מקדים את מספר הסגמנט.

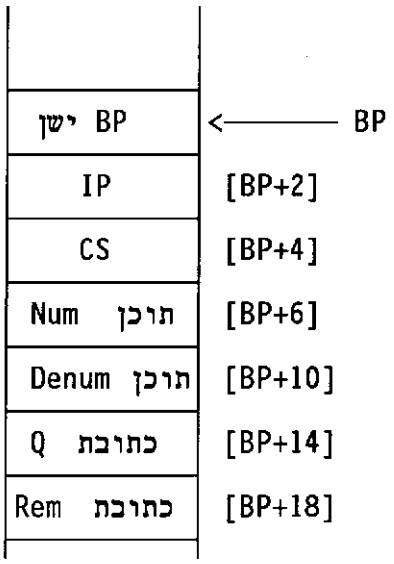
```
/* call_id1.c - call assembler subroutine idiv_mod.asm from C program */

#include <stdio.h>

 extern  int idiv_mod(int Num, int Denom,  int *Q,  int *Rem);

 void main()
    {
      int Num, Denom, Q, Rem, No_Zero_Divide;

      printf("\Enter Numerator, Denominator\n:");
      scanf("%d %d",&Num, &Denom);
      No_Zero_Divide = idiv_mod(Num,Denom,&Q,&Rem);
      if  (No_Zero_Divide)
          printf("\n %d div %d = %d, mod(%d,%d) = %d\n",
                    Num, Denom, Q, Num, Denom, Rem);
      else
         printf("\nError: Zero Divide.\n");

     } /* main */
```

```
E:\>tcc -ml call_id1.c idiv_mo7.asm
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
call_id1.c:
idiv_mo7.asm:
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland
International

Assembling file:   idiv_mo7.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  398k

Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

        Available memory 4115728

E:\>CALL_ID1.EXE
Enter Numerator, Denominator
:7000   666

 7000 div 666 = 10, mod(7000,666) = 340

E:\>
```

```
;  idiv_mo7.asm - MODEL LARGE assembler implementation of
;                     C-callable function idiv_mod.
;
     .MODEL LARGE
     .CODE
     ; Implementation of C callable function ...
     ;     ...  int idiv_mod(int Num, int Denom,  int *Q,  int *Rem)
     ;                          [BP+6]    [BP+8]     [BP+10]    [BP+14]
     ;  Compute Q := |_ Num / Denom _|  ,Rem := MOD(Num, Denom)
     ;  function  idiv_mod  returns 0 if Denom = 0 (illegal ..
     ;  ... division by zero), 1 otherwise
     ;
      PUBLIC _idiv_mod
_idiv_mod  PROC FAR
      PUSH BP          ; Preserve BP
      MOV BP,SP        ; Set BP to point to Parameter area
      PUSH SI          ; Preserve register variables
      PUSH DI          ;
      PUSH ES
;
      MOV SI,[BP+8]   ; SI := Denom
      CMP SI,0      ; Denom = 0 ?
      JNE Cont         ; No, continue regular operation
            ; Yes, Denom = 0
          MOV AX,0     ; Return value := 0
          JMP Done              ; Skip following code
Cont:       ; Denom <> 0
      MOV AX,[BP+6]          ; AX := Num
      CWD                    ; DX:AX := AX
      IDIV SI                ; AX := DX:AX / SI,  DX := MOD(AX,SI)
      MOV  DI,[BP+10]        ; DI := Offset Q
      MOV  ES,[BP+12]        ; ES := Seg Q
      MOV  ES:[DI],AX        ; *Q := AX
      MOV  DI,[BP+14]        ; DI := Offset Rem
      MOV  ES,[BP+16]        ; ES := Seg Q
      MOV  ES:[DI],DX        ; *Rem := DX
      MOV  AX,1    ; Ensure return value := 1
Done:
;
      POP ES     ;
      POP DI     ; Restore register variables
      POP SI     ;
      POP  BP    ; Restore BP register
      RET
_idiv_mod  ENDP
      END
```

תוכנית דוגמא call_id2.c, idiv_m10.asm

בתוכנית המשולבת call_id2.c, idiv_mo6.asm ההנחה ב-idiv_mo6.asm היתה שהקימפול הוא במודל Small. בקובץ idiv_m10.asm הפונקציה idiv_mod32 ממומשת עבור קימפול במודל Large. תמונת המחסנית היא עכשיו כזו:



מבנה המחסנית השתנה בשני הקשרים: הפרמטר הראשון הוא ב-[BP+6] במקום ב-[BP+4] ומאחר והפוינטרים עכשיו 32 ביט, הפרמטר "כתובת Rem" נמצא במרחק 4 בתים מתחילת הפרמטר "כתובת Q". בנוסף לכך כל שימוש בפוינטרים מחייב מוערבות של אוגר סגמנט. בתור אוגר סגמנט נעשה שימוש ב-ES, והוא נטען בחצי המרוחק של שני הפונטרים (BP+16 ו-BP+20). בפוינטר מלא ההיסט תמיד מקדים את מספר הסגמנט.

```
/* call_id2.c - call assembler subroutine idiv_mod32.asm from C program */

#include <stdio.h>

 extern  int idiv_mod32(long int Num, long int Denom,
                  long int *Q, long int *Rem);

 void main()
    {
      long int Num, Denom, Q, Rem;
      int  No_Zero_Divide;

      printf("\Enter Numerator, Denominator\n:");
      scanf("%ld %ld",&Num, &Denom);
      No_Zero_Divide = idiv_mod32(Num,Denom,&Q,&Rem);
      if  (No_Zero_Divide)
          printf("\n %ld div %ld = %ld, mod(%ld,%ld) = %ld\n",
                   Num, Denom, Q, Num, Denom, Rem);
      else
         printf("\nError: Zero Divide.\n");

     } /* main */
```

```
E:\asm>tcc -ml call_id2.c idiv_m10.asm
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
call_id2.c:
idiv_m10.asm:
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland
International

Assembling file:   idiv_m10.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  431k

Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

        Available memory 4149984

E:\asm>call_id2.exe
Enter Numerator, Denominator
:700000 111110

 700000 div 111110 = 6, mod(700000,111110) = 33340

E:\asm>
```

```
;  idiv_m10.asm - Assembler implementation of
;                          C-callable function idiv_mod32.
;
     .MODEL LARGE
     .CODE
     .386
     ; Implementation of C callable function ...
     ;     ...  int idiv_mod32(long int Num, long int Denom,
     ;                              [BP+6]          [BP+10]
     ;  long int *Q,     long int *Rem)
     ;       [BP+14]         [BP+18]
     ;  Compute Q := |_ Num / Denom _|   ,Rem := MOD(Num, Denom)
     ;  function  idiv_mod32  returns 0 if Denom = 0 (illegal ..
     ;  ... division by zero), 1 otherwise
     ;
      PUBLIC _idiv_mod32
_idiv_mod32  PROC FAR
      PUSH BP          ; Preserve BP
      MOV BP,SP        ; Set BP to point to Parameter area
      PUSH SI          ; Preserve register variables
      PUSH DI          ;
      PUSH ES
;
      MOV ESI,[BP+10]   ; ESI := Denom
      CMP ESI,0      ; Denom = 0 ?
      JNE Cont         ; No, continue regular operation
            ; Yes, Denom = 0
          MOV AX,0     ; Return value := 0
          JMP Done              ; Skip following code
Cont:       ; Denom <> 0
      MOV EAX,[BP+6]         ; EAX := Num
      CDQ                   ; EDX:EAX := EAX
      IDIV ESI               ; EAX := EDX:EAX / ESI,  EDX := MOD(EAX,ESI)
      MOV  DI,[BP+14]        ; DI := Offset Q
      MOV  ES,[BP+16]        ; ES := Seg Q
      MOV  ES:[DI],EAX          ; *Q := AX
      MOV  DI,[BP+18]       ; DI := Offset Rem
      MOV  ES,[BP+20]        ; ES := Seg Rem
      MOV  ES:[DI],EDX          ; *Rem := DX
      MOV  AX,1    ; Ensure return value := 1
Done:
;
      POP ES     ;
      POP DI     ; Restore register variables
      POP SI     ;
      POP  BP               ; Restore BP register
      RET
_idiv_mod32  ENDP
      END
```

תוכניות דוגמא runswap1.c, swaps.asm, swap11.asm, swap12.asm

תוכניות הדוגמא המצוינות ממחישות את עיקרי ההבדלים בתכנות באסמבלי במודלים Small ו-Large. תתי התוכניות swap16, swap32 ו-swap_n מבצעות החלפה של ערכי משתנים (בגדלים שונים) של התוכנית הקוראת לפי פוינטרים שהוכנית הקוראית מעבירה.

ההבדלים הבאים לידי ביטוי בדוגמאות הללו הן:
- בעוד ששני הפוינטרים המועברים כפוינטרים במודל Small נמצאים בכתובות [BP+4] ו-[BP+6] בתוכניות במודל Large הם נמצאים בכתובות [BP+6] ו-[BP+10].

- בעוד שכאשר משתמשים בפוינטרים במודל Small מתיחסים להיסט כאילו הוא כל הכתובת, בתוכניות במודל Large יש לדאוג לטעון לאוגר סגמנט את מרכיב הסגמנט של הפוינטר בטרם נעשה בו שימוש.

התוכנית swap11.asm נכתבה בסגנון 8086 כאשר אוגר הסגמנט ES היה בבחינת אוגר הסגמנט הפנוי היחיד למטרות מהסוג של שימוש בפוינטר מלא. התוכנית swap12.asm מנצלת את העובדה שהחל מה-386 קיימים למטרות מסוג זה 2 אוגרי סגמנט נוספים: FS ו-GS.

```
/* runswap1.c - run swaps */

extern void swap16( int *x, int *y);
extern void swap32(long int *x, long int *y);
extern void swap_n( void *x, void *y, int n);

int main ()
{
 int x=10, y=99;
 long int u=333333, v=777777;
 char str1[10]="Hello", str2[10]="World!";

 printf("x = %d, y = %d\n", x, y);
 swap16(&x, &y);
 printf("x = %d, y = %d\n", x, y);

 printf("u = %ld, v = %ld\n", u, v);
 swap32(&u, &v);
 printf("u = %ld, v = %ld\n", u, v);

 printf("str1 = %s, y = %s\n", str1, str2);
 swap_n(str1, str2, 10);
 printf("str1 = %s, str2 = %s\n", str1, str2);

 return 0;

} /* main */
```

```
E:\>tcc  runswap1.c swaps.asm
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
runswap1.c:
swaps.asm:
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland
International

Assembling file:   swaps.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  398k

Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

        Available memory 4129872

E:\ACADEMIC\ASM>runswap1.exe
x = 10, y = 99
x = 99, y = 10
u = 333333, v = 777777
u = 777777, v = 333333
str1 = Hello, y = World!
str1 = World!, str2 = Hello

E:\>
```

```
;
;  swaps.asm - swap program model small
;
   .MODEL SMALL
   .CODE
   .386
;
;  extern void swap16( int *x, int *y);
;                      [BP+4]  [BP+6]
;
_swap16 PROC NEAR
   PUBLIC _swap16
    PUSH BP
    MOV BP,SP
    PUSH SI
    PUSH DI
;
   MOV SI,[BP+4]
   MOV DI,[BP+6]
   MOV AX,[SI]
   XCHG  AX,[DI]
   MOV [SI],AX
;
    POP DI
    POP SI
    POP BP
    RET
_swap16 ENDP
;
;  extern void swap32( long int *x, long int *y);
;                      [BP+4]        [BP+6]
;
_swap32 PROC NEAR
   PUBLIC _swap32
    PUSH BP
    MOV BP,SP
    PUSH SI
    PUSH DI
;
   MOV SI,[BP+4]
   MOV DI,[BP+6]
   MOV EAX,[SI]
   XCHG  EAX,[DI]
   MOV [SI],EAX
;
    POP DI
    POP SI
    POP BP
    RET
_swap32 ENDP
```

```
;
; extern void swap_n( void *x, void *y, int n);
;                       [BP+4]    [BP+6]  [BP+8]
;
;
_swap_n PROC NEAR
   PUBLIC _swap_n
    PUSH BP
    MOV BP,SP
    PUSH SI
    PUSH DI
;
   MOV SI,[BP+4]
   MOV DI,[BP+6]
   MOV CX,[BP+8]
   JCXZ ToRet1
Aloop1:
   MOV AL,[SI]
   XCHG  AL,[DI]
   MOV [SI],AL
   INC SI
   INC DI
;
   LOOP Aloop1
;
ToRet1:
    POP DI
    POP SI
    POP BP
    RET
_swap_n ENDP
     END
```

```
/* runswap1.c - run swaps */

extern void swap16( int *x, int *y);
extern void swap32(long int *x, long int *y);
extern void swap_n( void *x, void *y, int n);

int main ()
{
 int x=10, y=99;
 long int u=333333, v=777777;
 char str1[10]="Hello", str2[10]="World!";

 printf("x = %d, y = %d\n", x, y);
 swap16(&x, &y);
 printf("x = %d, y = %d\n", x, y);

 printf("u = %ld, v = %ld\n", u, v);
 swap32(&u, &v);
 printf("u = %ld, v = %ld\n", u, v);

 printf("str1 = %s, y = %s\n", str1, str2);
 swap_n(str1, str2, 10);
 printf("str1 = %s, str2 = %s\n", str1, str2);

 return 0;

} /* main */
```

```
E:\>tcc -ml runswap1.c swap11.asm
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
runswap1.c:
swap11.asm:
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland
International

Assembling file:   swap11.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  397k

Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

        Available memory 4129616

E:\ACADEMIC\ASM>runswap1.exe
x = 10, y = 99
x = 99, y = 10
u = 333333, v = 777777
u = 777777, v = 333333
str1 = Hello, y = World!
str1 = World!, str2 = Hello

E:\>
```

```
;
;  swap12.asm - swap program model large
;
   .MODEL LARGE
   .CODE
   .386
;
;  extern void swap16( int *x, int *y);
;                      [BP+6]  [BP+10]
;
_swap16 PROC FAR
   PUBLIC _swap16
    PUSH BP
    MOV BP,SP
    PUSH GS
    PUSH FS
    PUSH SI
    PUSH DI
;
   MOV SI,[BP+6]
   MOV GS,[BP+8]
   MOV DI,[BP+10]
   MOV FS,[BP+12]
;
   MOV AX,GS:[SI]
   XCHG  AX,FS:[DI]
   MOV GS:[SI],AX
;
    POP DI
    POP SI
    POP FS
    POP GS
    POP BP
    RET
_swap16 ENDP
;
;  extern void swap32( long int *x, long int *y);
;                      [BP+6]        [BP+10]
;
_swap32 PROC FAR
   PUBLIC _swap32
    PUSH BP
    MOV BP,SP
    PUSH GS
    PUSH FS
    PUSH SI
    PUSH DI
;
   MOV SI,[BP+6]
   MOV GS,[BP+8]
   MOV DI,[BP+10]
   MOV FS,[BP+12]
   MOV EAX,GS:[SI]
   XCHG  EAX,FS:[DI]
   MOV GS:[SI],EAX
;
    POP DI
    POP SI
    POP FS
    POP GS
    POP BP
    RET
_swap32 ENDP
```

```
;
; extern void swap_n( void *x, void *y, int n);
;                     [BP+6]    [BP+10]  [BP+14]
;
;
_swap_n PROC FAR
  PUBLIC _swap_n
   PUSH BP
   MOV BP,SP
   PUSH GS
   PUSH FS
   PUSH SI
   PUSH DI
;
  MOV SI,[BP+6]
  MOV GS,[BP+8]
  MOV DI,[BP+10]
  MOV FS,[BP+12]
  MOV CX,[BP+14]
;
  JCXZ ToRet1
Aloop1:
  MOV AL,GS:[SI]
  XCHG  AL,FS:[DI]
  MOV GS:[SI],AL
  INC SI
  INC DI
;
  LOOP Aloop1
;
ToRet1:
   POP DI
   POP SI
   POP FS
   POP GS
   POP BP
   RET
_swap_n ENDP
    END
```

```
;
;  swap11.asm - swap program model large
;
   .MODEL LARGE
   .CODE
   .386
;
;  extern void swap16( int *x, int *y);
;                      [BP+6]  [BP+10]
;
_swap16 PROC FAR
   PUBLIC _swap16
    PUSH BP
    MOV BP,SP
    PUSH ES
    PUSH SI
    PUSH DI
;
   MOV SI,[BP+6]
   MOV DI,[BP+10]
;
   MOV ES,[BP+8]
   MOV AX,ES:[SI]
   MOV ES,[BP+12]
   XCHG  AX,ES:[DI]
   MOV ES,[BP+8]
   MOV ES:[SI],AX
;
    POP DI
    POP SI
    POP ES
    POP BP
    RET
_swap16 ENDP
;
;  extern void swap32( long int *x, long int *y);
;                      [BP+6]        [BP+10]
;
_swap32 PROC FAR
   PUBLIC _swap32
    PUSH BP
    MOV BP,SP
    PUSH ES
    PUSH SI
    PUSH DI
;
   MOV SI,[BP+6]
   MOV DI,[BP+10]
;
   MOV ES,[BP+8]
   MOV EAX,ES:[SI]
   MOV ES,[BP+12]
   XCHG EAX,ES:[DI]
   MOV ES,[BP+8]
   MOV ES:[SI],EAX
;
    POP DI
    POP SI
    POP ES
    POP BP
    RET
_swap32 ENDP
```

```
;
; extern void swap_n( void *x, void *y, int n);
;                       [BP+6]    [BP+10]  [BP+14]
;
;
_swap_n PROC FAR
   PUBLIC _swap_n
    PUSH BP
    MOV BP,SP
    PUSH ES
    PUSH SI
    PUSH DI
;
   MOV SI,[BP+6]
   MOV DI,[BP+10]
   MOV CX,[BP+14]
;
   JCXZ ToRet1
Aloop1:
   MOV ES,[BP+8]
   MOV AL,ES:[SI]
   MOV ES,[BP+12]
   XCHG  AL,ES:[DI]
   MOV ES,[BP+8]
   MOV ES:[SI],AL
   INC SI
   INC DI
;
   LOOP Aloop1
;
ToRet1:
    POP DI
    POP SI
    POP ES
    POP BP
    RET
_swap_n ENDP
     END
```

address. For example, given the pointer 2F84:0532, we convert that to the absolute address 2FD72, which we then normalize to 2FD7:0002. Here are a few more pointers with their normalized equivalents:

```
0000:0123    0012:0003
0040:0056    0045:0006
500D:9407    594D:0007
7418:D03F    811B:000F
```

Now you know that huge pointers are always kept normalized. Why is this important? Because it means that for any given memory address, there is only one possible huge address—segment:offset pair—for it. And that means that the == and != operators return correct answers for any huge pointers.

In addition to that, the >, >=, <, and <= operators are all used on the full 32-bit value for huge pointers. Normalization guarantees that the results there will be correct also.

Finally, because of normalization, the offset in a huge pointer automatically wraps around every 16 values, but—unlike far pointers—the segment is adjusted as well. For example, if you were to increment 811B:000F, the result would be 811C:0000; likewise, if you decrement 811C:0000, you get 811B:000F. It is this aspect of huge pointers that allows you to manipulate data structures greater than 64K in size.

There is a price for using huge pointers: additional overhead. Huge pointer arithmetic is done with calls to special subroutines. Because of this, huge pointer arithmetic is significantly slower than that of far or near pointers.

Turbo C's Six Memory Models

Avoiding overhead—except when you want it—is just what Turbo C allows you to do. There are six different memory models you can choose from: tiny, small, medium, compact, large, and huge. Your program requirements determine which one you pick. Here's a brief summary of each:

Tiny: As you might guess, this is the smallest of the memory models. All four segment registers (CS, DS, SS, ES) are set to the same address, so you have a total of 64K for all of your code, data, and arrays. Near pointers are always used. Use this when memory is at an absolute premium. Tiny model programs can be converted to .COM format by linking with the /t option.

Small: The code and data segments are different and don't overlap, so you have 64K of code and 64K of static data. The stack and extra segments start at the same address as the

data segment. Near pointers are always used. This is a good size for average applications.

Medium: Far pointers are used for code, but not for data. As a result, static data is limited to 64K, but code can occupy up to 1 Mb. This is best for large programs that don't keep much data in memory.

Compact: The inverse of medium: Far pointers are used for data, but not for code. Code is then limited to 64K, while data has a 1-Mb range. This choice is best if your code is small but you need to address a lot of data.

Large: Far pointers are used for both code and data, giving both a 1-Mb range. It is needed only for very large applications.

Huge: Far pointers are used for both code and data. Turbo C normally limits the size of all static data to 64K; the huge memory model sets aside that limit, allowing static data to occupy more than 64K.

The following illustrations (Figures 12.2 through 12.7) show how memory in the 8086 is apportioned for the six Turbo C memory models.

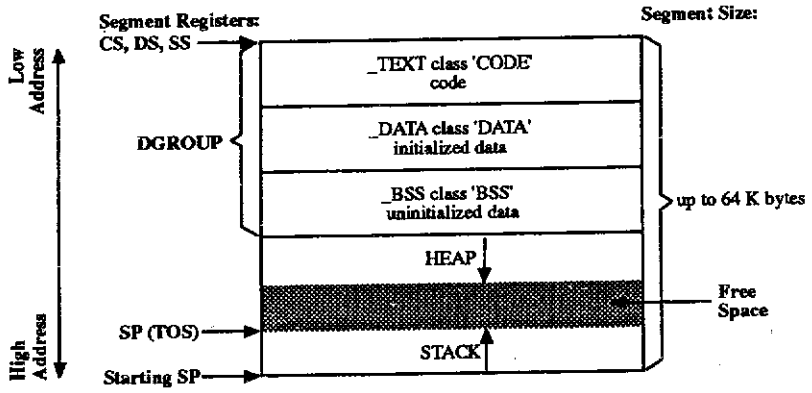


Figure 12.2: Tiny Model Memory Segmentation

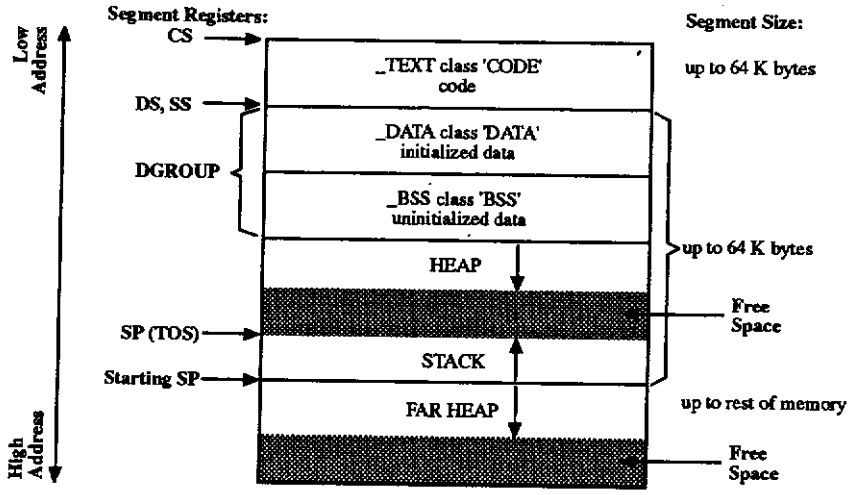


Figure 12.3: Small Model Memory Segmentation

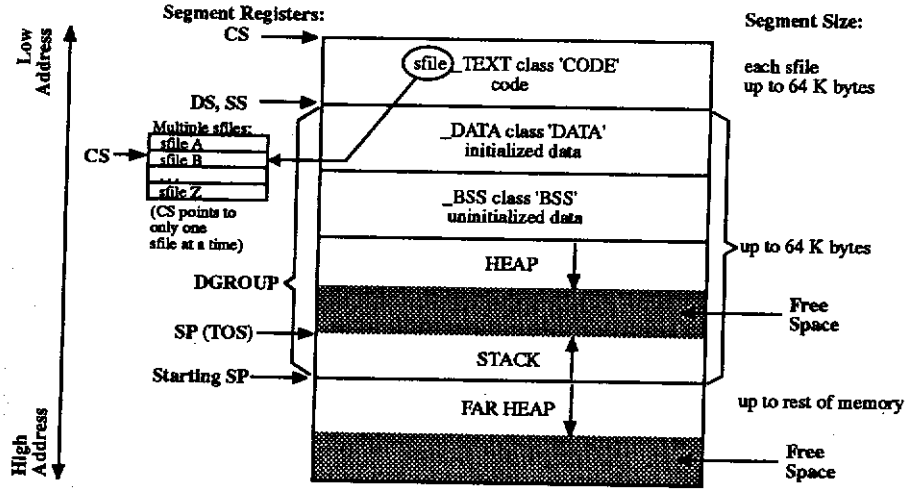


Figure 12.4: Medium Model Memory Segmentation

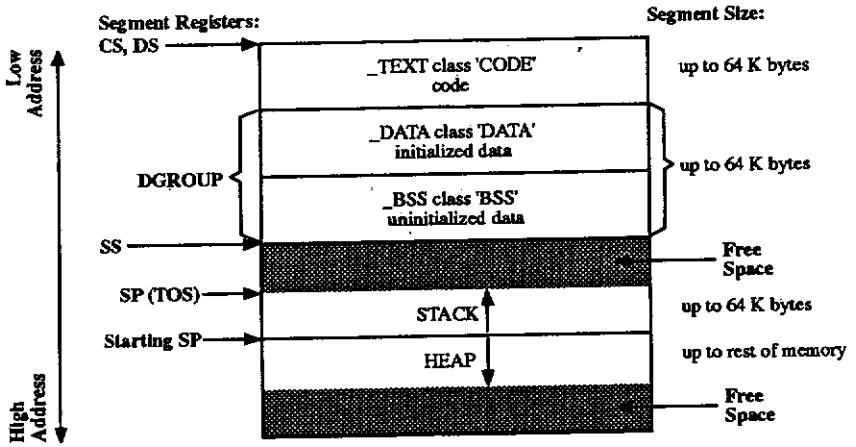


Figure 12.5: Compact Model Memory Segmentation

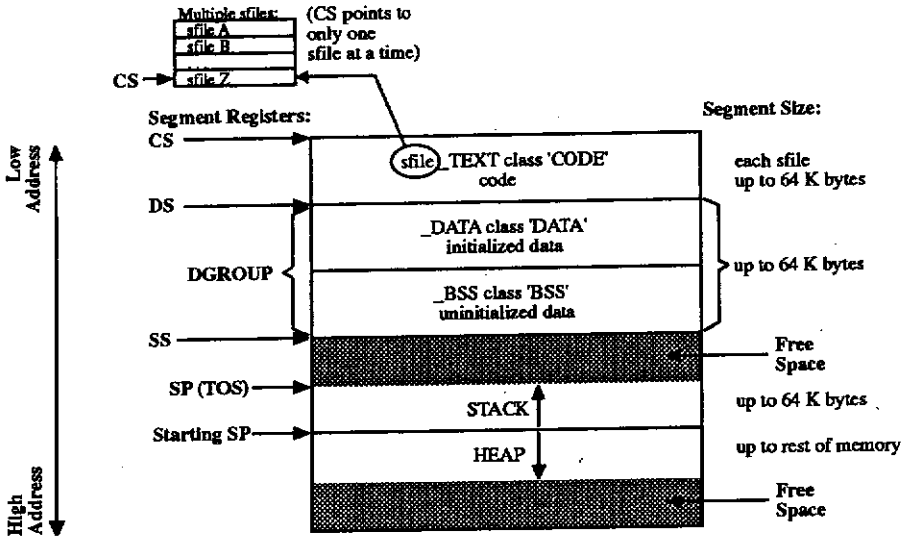


Figure 12.6: Large Model Memory Segmentation

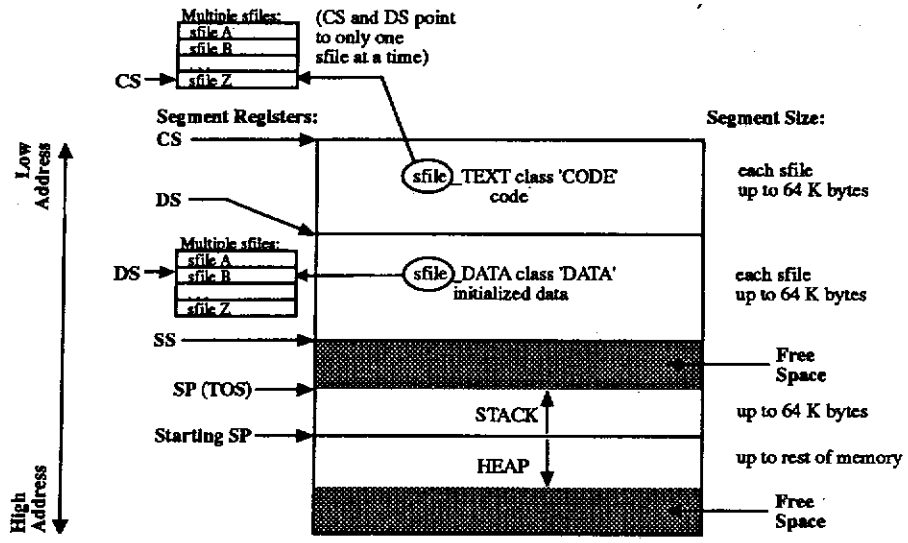


Figure 12.7: Huge Model Memory Segmentation

Table 12.1 summarizes the different models and how they compare to one another. The models are often grouped according to whether their code or data models are *small* (64K) or *large* (1 Mb); these groups correspond to the rows and columns in Table 12.1. So, for example, the models tiny, small, and compact are known as *small code models* because, by default, code pointers are near; likewise, compact, large, and huge are known as *large data models* because, by default, data pointers are far. Note that this is also true for the huge model—the default data pointers are far, not huge. If you want huge data pointers, you must declare them explicitly as huge.