

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 לאפשרויות 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.

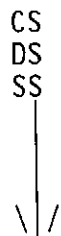
תוכנית שנטענת לזיכרון מקצה זיכרון ל-4 סוגי שימושים:

- 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 של הקוד, שטחי המשתנים הגלובליים/סטטיים כולם יחד יכולים לתפוס לכל היותר 64K.

תמונת הזיכרון של מודל tiny הוא כלהלן:



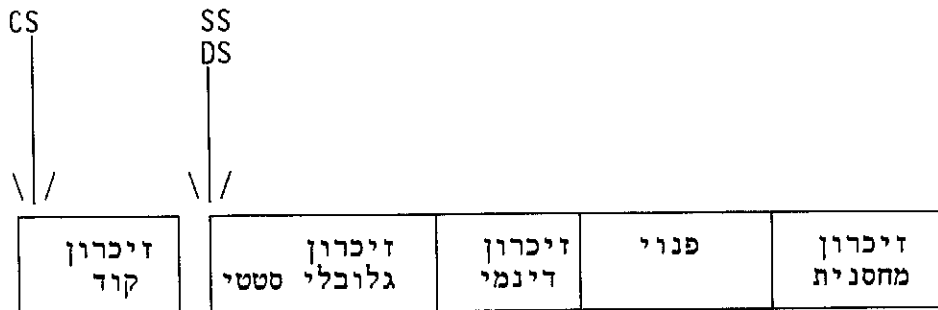
זיכרון מחסנית	פנוי	זיכרון דינמי	זיכרון גלובלי סטטי	זיכרון קוד
------------------	------	-----------------	--------------------------	---------------

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

מגדל Small

מודל Small מבוסס על העובדה שאפשר לשחרר חלק מהמגבלות על השימוש בשטח זיכרון של מודל Tiny מבלי להפסיד 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 לאופי הפעולה (הסתעפות או גישה לזיכרון למידע) לקבוע איזה אוגר סגמנט משתמשים. לפיכך תוכנית המקומפלת במודל 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

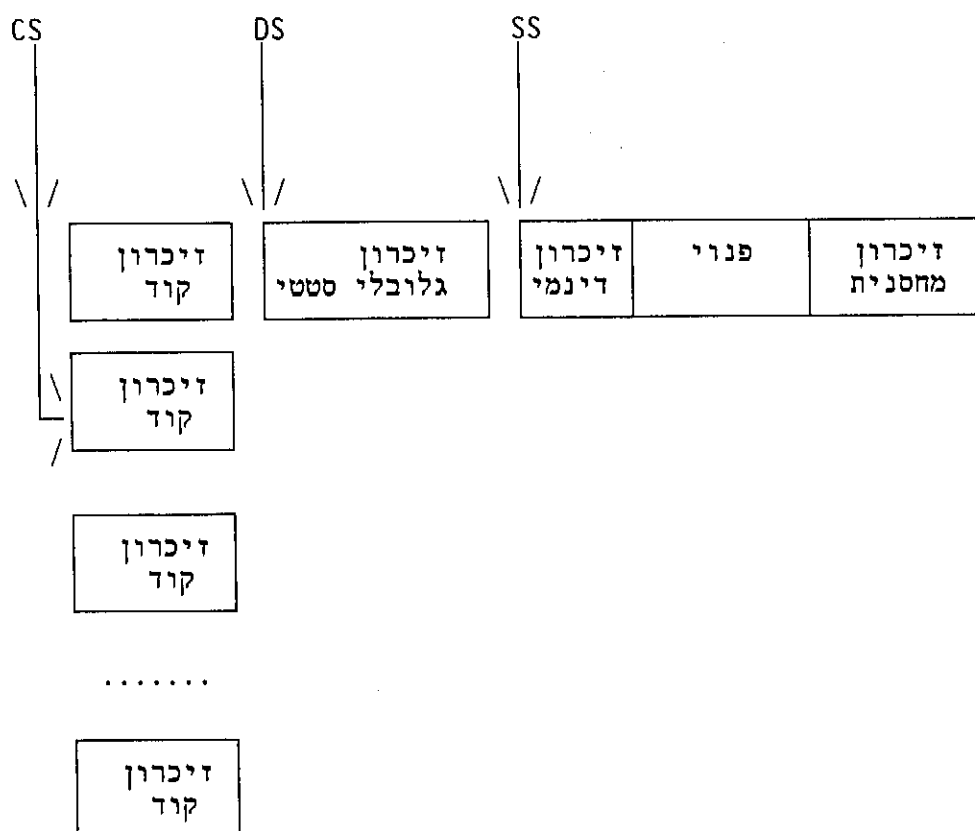
השניה. תמונת אוגרי הסגמנטים במודל Small נראית כך:



מודל Large

כמעט בקצה השני של האפשרויות הוא מודל Large. כאן מותרים 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), ואילו 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 ההסתעפויות 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 אחרת על שטח מחסנית או דינמי. תוכנית המקומפלט 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.

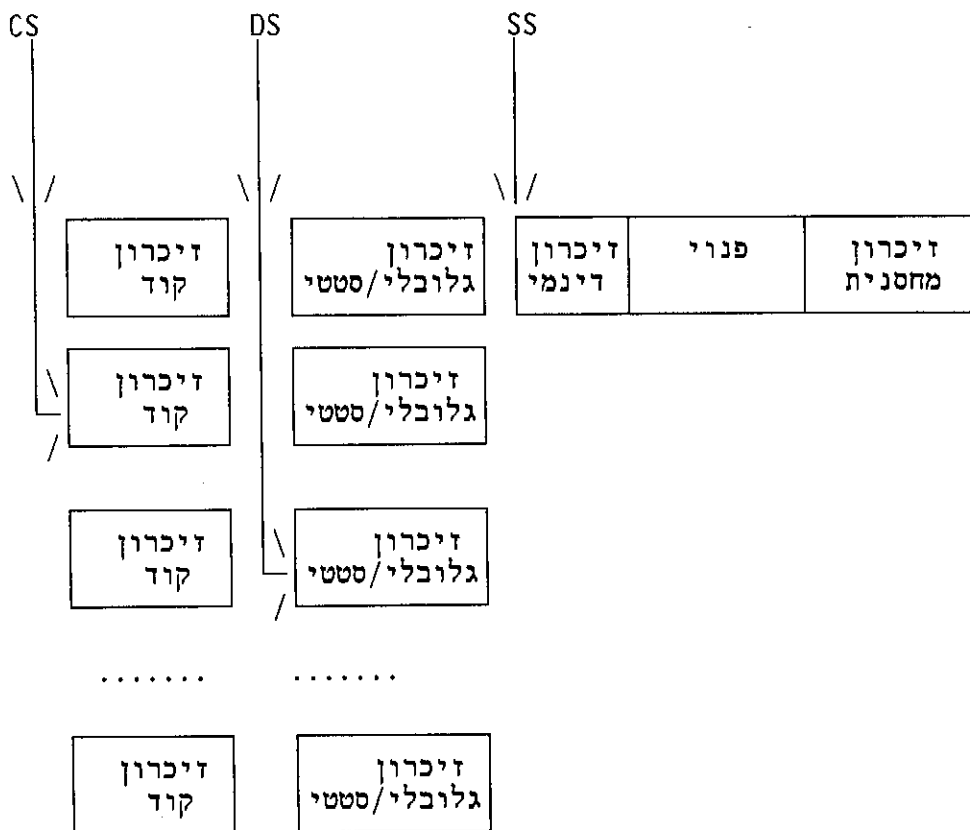
תמונת הזיכרון של מודל Large נראה עקרונית כך:



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. תוכנית 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:



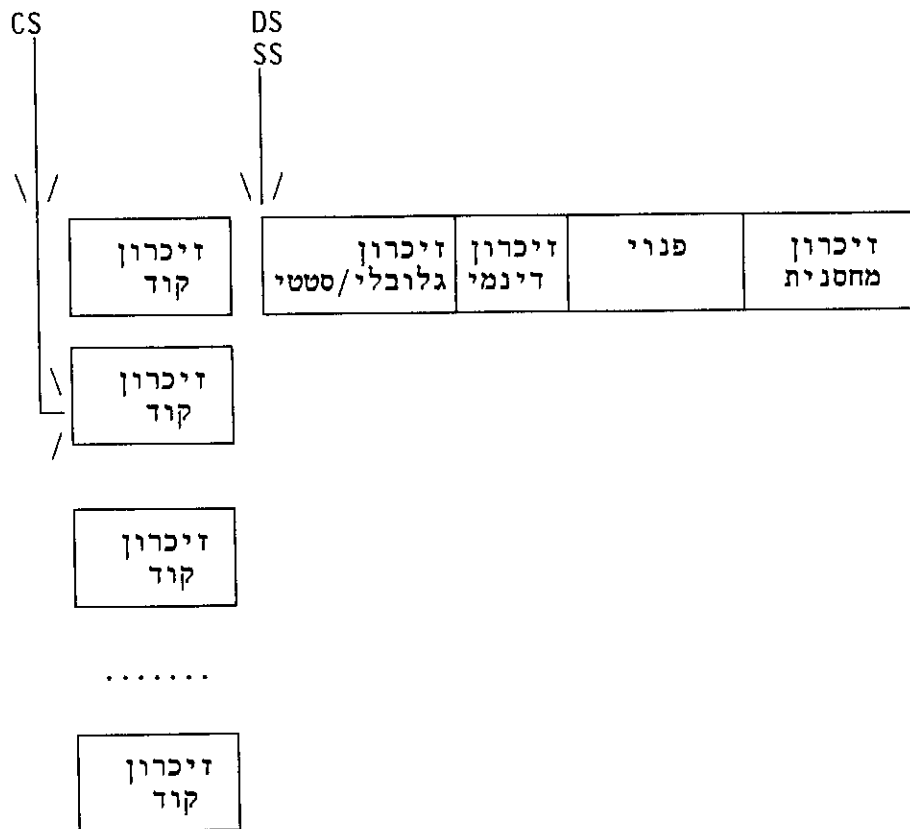
המודלים הנותרים

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

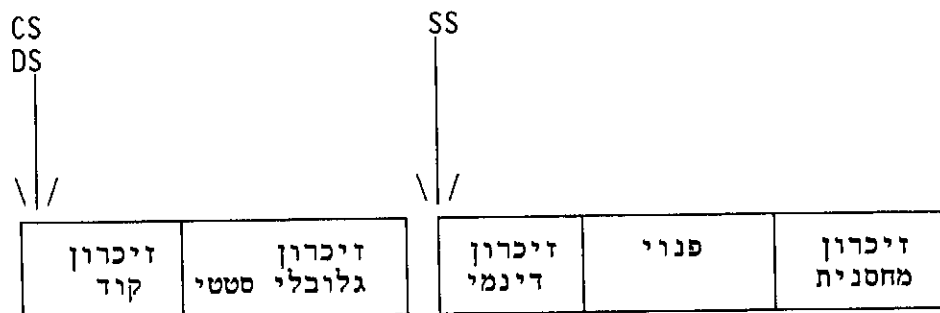
מודל Medium - שטח הקוד יכול להתפרס על מספר רצוי של סגמנטים כמו במודל Large אבל שטח המידע כולו (גלובלי/סטטי, דינמי, מחסנית) נכנס לסגמנט אחד כמו במודל Small. לפיכך כאן כל ההסתעפויות הם FAR, פוינטרים לפונקציה הם 32 ביט אבל הפוינטרים למידע הם 16 ביט.

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 למידע הם 32 ביט.

תמונת הזיכרון של מודל Medium:



מודל הזיכרון של מודל Compact:



כתיבת תוכניות כאסמבלי - ההבדלים בין המודלים

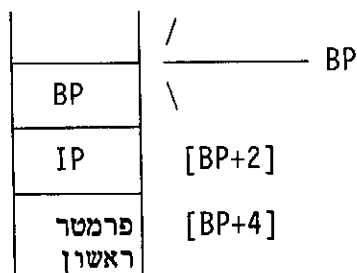
תוכניתן הכותב תוכנית אסמבלי הנקרא מתוכנית טורבו C חייב להיות מודע לאיזה מודל הוא כותב את התוכנית, גם אם למען האמת הוא לא צריך להבין לעומק את תמונת הזיכרון של המודל. תוכנית אסמבלי שנכתבת למודל Small לא תעבוד למודל Large. העובדות שתוכניתן האסמבלי צריך להיות מודע להן הן רק סוגי ההסתעפויות של המודל (FAR,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

סוג ההסתעפויות - 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 בהסתעפויות מסוג FAR (מודלים Medium, Large, Huge), הפרמטר הראשון יהיה ב-[BP+6] משום שתמונת המחסנית עכשיו תהיה

	/	
BP	\	BP
IP		[BP+2]
CS		[BP+4]
פרמטר ראשון		[BP+6]

בנוסף לכך, גודל הפוינטרים לפונקציה 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 בכתיבות הנמוכות, ערך חדש ל-CS בכתיבות המרוחקות יותר). לדוגמא, נניח שכתובת [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 נוסף

CALL WORD PTR [BP+10]

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

CALL DWORD PTR [BP+10]

סוג הפוינטרים למידע - במידה ופרוצדורת האסמבלי נקראית מתוכנית C המקומפלת במודל המממש פוינטרים למידע 16 ביט (מודלים Tiny, Small ו-Medium) כל פוינטר למידע הוא 16 ביט ומכיל היסט בלבד אם למשל בכתיבת [BP+10] יש פוינטר כזה, שימוש בו יעשה ע"י פקודות נוסף

MOV BX,[BP+10]

MOV AX,[BX]

לעומת זאת, במידה ופרוצדורת האסמבלי נקראית מתוכנית C המקומפלת במודל המממש 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 נוסף

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

כלומר שימוש אופייני של פוינטר כזה יכול להשמה לאוגר סגמנט ושימוש ב-Segment Override.

התייחסות לזיכרון המחסנית דרך ה-BP

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

הכרזת משתנים בסגמנט ה-DATA

בכל המודלים למעט מודל Huge ערך ה-DS לא משנה את ערכו במהלך התוכנית. לפיכך בכל תוכנית אסמבלי המשתלבת בקוד C והמקומפלט בכל המודלים למעט מודל Huge ניתן להכריז על משתנים (סטטיים) תחת ה-DATA. או ה-DATA_ ולהתייחס בהם כתוכנית 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 ה-Huge, תוכנית אסמבלי הנקראת מקוד C במקומפל במודל Huge בן צריכה להתייחס לנושא הזה, נניח באמצעות האופרטור SEG.

חשוב לזכור: 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 כזו:

BP ישן	← BP
IP	[BP+2]
CS	[BP+4]
תוכן Num	[BP+6]
תוכן Denum	[BP+8]
כתובת Q	[BP+10]
כתובת Rem	[BP+14]

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 מוערכות 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("\nEnter 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:\>

```

165

```

; 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 Rem
    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

```

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:

BP ישן	← BP
IP	[BP+2]
CS	[BP+4]
Num תוכן	[BP+6]
Denum תוכן	[BP+10]
Q כתובת	[BP+14]
Rem כתובת	[BP+18]

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 מוערכות 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("\nEnter 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

```