תקציר מספר 6

פרוצדורות או תת-תוכניות

פרוצדורות (או תת-תוכניות, נהלים, פונקציות וכו) הם אמצעי תכנות חשוב. באופן בסיסי האמצעי הזה מאפשר לנו להפעיל קוד ביותר ממקום אחד בתוכנית מבלי להעתיק או לשכפל אותו במפורש. הדבר הזה מאפשר לנו, למשל, להכניס שינוי בתוכנית רק במקום אחד במידה ומתגלה טעות או צורך אחר בשינוי. אבל, כפי שכל תוכניתן יודע, החשיבות של פרוצדורות הוא מעבר לענין הטכני הזה: פרוצדורות משמשות "לשבירת" משימת תכנות מורכבת למרכיבים יותר פשוטים, לחלוקת משימות תכנות ליותר ממתכנת אחד, שימוש חוזר בקוד וכו'. לכל השימושים הללו, יש צורך לאפשר העברת אינפורמציה לפרוצדורות, וההתנהגותן מושפעות מהאינפרמציה הזו. האינפורמציה הזו שמועברת בדרך כלל ע"י מנגנון הפרמטרים.

כאן נתרכז באספקט הטכני של פרוצדורות: איך ברמת שפת המכונה ממשים אותן. מלבד הבנת המנגנון אנחנו נראה שלשימוש בפרוצדורות יש מחיר: ישנו ביצוע של פקודות מכונה שאינם חלק מביצוע המשימה של התוכנית, אלא רק לצורך מימוש האמצעי של הפרוצדורות.

התמיכה בחומרה לפרוצדורות

כפי שאני רואה את הדברים, התמיכה הישירה הכמעט יחידה לפרוצדורות במרבית המקרים הינה בפקודות מכונה להסתעפות וחזרה מהפרוצדורה. במחשב הזה גם התמיכה בחומרה למחסנית יכולה להחשב מעין תמיכה עקיפה בפרוצדורות.

פקודת המכונה CALL.

לפקודה CALL מספר ניכר של גירסאות, אבל השימוש העיקרי שלה הוא בצורה

```
CALL label
```

כאשר ה-label הוא מצביע לנקודת זכרון בתוך החלק הביצועי של התוכנית, ומוגדר בד"כ ע"י ההנחיה PROC שאותה נראה בהמשך. תפקידיה של פקודת המכונה CALL הם בעיקר שניים:

1. קודם כל לשמר את כתובת הפקודה העוקבת בזכרון במחסנית.
2. לשנות את הפקודה הבאה לביצוע לכתובת המוצבעת ע"י label.

הפקודה CALL היא איפוא פקודת הסתעפות בלתי מותנית. אחד ההבדלים

העקריים בינה לבין הפקודה JMP הוא שהיא משמרת אינפורמציה לאן לחזור (או אולי יותר נכון מאיפה להמשיך לאחר חזרה מהפרוצדורה). החלק הזה של הפקודה ממומש במהלך דומה לפקודה PUSH. השימור נעשה במחסנית בנקודה שהאוגרים SS:SP מצביעים עליה, ו-SP מופחת בהתאם.

ב-8086, לגבי 1, המשמעות היא לשמר את ערך ה-IP של הפקודה הבאה, או של ה-CS וה-IP של הפקודה הבאה, במקרה שהיעד נמצא בסגמנט אחר. ואכן יש לפקודה CALL שני סוגים:

1. CALL קרוב (NEAR) שבו יש שינוי רק של ה-IP במהלך ההסתעפות, כלומר הסתעפות שמראש מוגבל לנקודת זכרון באותו סגמנט. במקרה כזה יש צורך לשמר רק את ה-IP וזה גם מה שנעשה. אוגר ה-SP מופחת ב-2 באופן דומה לביצוע של פקודת PUSH.

2. CALL מרוחק (FAR) שבו יש אפשרות שינוי של ה-CS בנוסף ל-IP, כלומר הסתעפות לנקודה שעשויה להיות בסגמנט קוד אחר. במקרה כזה יש צורך לשמר גם את ה-CS וגם את ה-IP. הערך של ה-CS הוא שנדחף קודם. כלומר שאם אחרי ביצוע ה-CALL הערך של ה-IP בכתובת 101 - 100, אז הערך של ה-CS נמצא בכתובת 102 - 103. כמו תמיד, נשמר כאן הכלל של "המשמעותי פחות מקדים את המשמעותי יותר". אוגר ה-SP מופחת ב-4 באופן דומה לביצועים של פקודת PUSH.

יש כללים איזה משתי הגירסאות של CALL נבחר. נראה זאת בהמשך.

גירסה מיוחדת של CALL הוא ההסתעפות העקיפה: הפקודה מקבלת יעד בזכרון (DATA) אבל לא מסתעפת לשם, אלא שולפת כתובת לאן להתסעף. נדון בפקודה הזו בהזדמנות אחרת.

פקודת המכונה RET.

פקודת המכונה RET היא הפקודה הפוכה של CALL: היא מבצעת חזרה מפרוצדורה, תוך שימוש בכתובת החזרה ש-CALL שמר במחסנית. יותר נכון לומר שהפקודה RET, כאשר היא מבוצעת, מניחה שבראש המחסית, בנקודה ש-SS:SP מצביע עליה, נמצאת כתובת החזרה שנשמרה על ידי הפקודה CALL. באחריות תוכניתן האסמבלי לדאוג לכך שההנחה הזו תהיה נכונה: בחומרה אין שום דבר המבטיח זאת. במידה וההנחה הזו איננה נכונה, RET יפרש את מה שימצא בראש המחסנית ככתובת חזרה, ויסתעף לכתובת שרירותית, שמן הסתם לא מה שצריך.

בקיצור, הפקודה RET היא פקודת הסתעפות בלתי מותנית, שהיעד להסתעפות נמצא בראש המחסנית.

כמו CALL, גם ל-RET יש מספר גירסאות, אם כי פחות מ-CALL. ב-8086, שתי הגירסאות העיקריות הם:

RET קרוב (NEAR) ששולף רק שני בתים מהמחסנית ומציב אותם ב-IP. ה-SP מקודם ב-2.

RET מרוחק (FAR) ששולף 4 בתים מהמחסנית: שתי הראשונים מוצבים ל-IP ושתי האחרונים ל-CS. ה-SP מקודם ב-4.

<u>מבנה פרוצדורה באסמבלי</u>.

באופן כללי, פרוצדורה באסמבלי היא בעלת הצורה הבאה:

```
שם   PROC  סוג הפרוצדורה
.....  פקודות  .....
שם   ENDP
```

לדוגמא, הפרוצדודה הקצרה הבאה, שתפקידה לאפס את AX:

```
Ax_Zero  PROC NEAR
  MOV AX,0
  RET
Ax_Zero ENDP
```

כאשר PROC ו-ENDP הם הנחיות לאסמבלר (Directives) (בדומה ל-CODE. למשל). ל-PROC ו-ENDP אין, איפוא, איזו שהיא השתקפות בזכרון. הם בבחינת איפורמציה לתוכנית האסמבלי.

ההנחיה PROC אומרת לאסמבלר שמכאן ואילך מתחיל תאור של פרוצדורה, בשם מסוים. התאור של הפרוצדורה מסתים כאשר האסמבלר נתקל בהנחיה ENDP בעל שם זהה. הסוג מציין אם מדובר בפרוצדורה שהיא NEAR או FAR. באסמבלי <u>אין</u> דבר כזה כפרוצדורות מקוננות ("פרוצדורה בתוך פרוצדורה") דבר שקיים בשפות עיליות כמו פסקל, ADA, PLI (אבל לא ב-C ו-FORTRAN הישן, למשל).

ל-PROC יש מספר תפקידים:

קודם כל הוא מסמן נקודה בזכרון, כך שיהיה אפשר להתיחס אליו בפקודת CALL. בדוגמא שלעיל פקודת ה-CALL תהיה:

```
CALL Ax_Zero
```

הדבר השני הוא שהוא מגדיר סוג (NEAR, FAR) לפרוצדורה. לסוג יש שתי השלכות:

1. זה קובע את סוגי פקודות ה-CALL אליו.
2. זה קובע את סוגי פקודות ה-RET שבתוך הפרוצדורה.

בדוגמא שלעיל המשמעות של 1. היא שהפקודה

```
CALL Ax_Zero
```

פורשת פקודת CALL מסוג NEAR, משום ש-Ax_Zero מוגדרת בהנחית ה-PROC כפרוצדורת NEAR. אילו ההנחיה היתה, במקום המתואר לעיל, מהצורה:

```
Ax_Zero  PROC FAR
```

אז הפקודה

```
CALL Ax_Zero
```

היתה פורשת פקודת CALL מסוג FAR.

ההשלכה של 2. בדוגמא היא שהפקודה RET שבתוכה היא RET מסוג NEAR משום שזה סוג הפרוצדורה. אילו הפרוצדורה היתה מוגדרת

```
Ax_Zero  PROC FAR
```

אזי ה-RET היה מסוג FAR.

במידה ובהנחיית ה-PROC לא מופיע כלום בשדה הסוג, ברירת המחדל היא כנראה NEAR. לא כדאי לסמוך על זה.

<u>פרמטרים</u>

בחומרה אין, למעשה, תמיכה בפרמטרים במובן שמתכנתים בשפות עילית מבינים את המושג הזה. לעומת זאת, פרוצדורות ללא פרמטרים הם אמצעי תכנות כל כך מוגבל, שהוא חסר חשיבות. המימוש של מושג הפרמטרים הוא למעשה באחריות <u>התוכנה</u> (או אם תרצו, תוכניתן האסמבלי).

יש מספר גישות אפשריות למימוש פרמטרים.

1. העברת פרמטרים דרך האוגרים.

למעשה ראינו את האמצעי הזה - הקריאות ל-INT 21h הוא דוגמאות טובות לכך. הגישה הזו היא מוגבלת ביותר, בפרט במחשבי האינטל x86, שמספר האוגרים מצומצם ביותר.

2. העברת פרמטרים דרך שטחי זכרון.

בגישה הזו לכל פרוצדורה יש שטח זכרון במבנה מוגדר מראש (למעשה רשומה בדרך כלל), ששם התוכנית הקוראת מציבה את הפרמטרים שהפרוצדורה מצפה לקבל. השטח הזה יכול להיות שטח מידע של הפרוצדורה עצמה, אבל הוא יכול להיות שטח של התוכנית הקוראת, המעבירה פוינטר לשטח הזה לפרוצדורה (דרך האוגרים למשל). עד כמה שידוע לי, המחשבים המסחריים הראשונים בשנות החמישים והששים

עבדו בדר"כ בשיטה הזו, וכשל כך יש מחשבים / שפות שעובדות בצורה הזו עד היום. יחד עם זאת, יש לגישה הזו חסרונות משמעותיים. למשל יש כאן בעיה ברגע שרוצים לממש פרוצדורות רקורסיביות, אבל מה שהיא כנראה הבעיה העיקרית נעוצה במימוש מערכות רב תהליכיות. לזה לא ניכנס כאן. נאמר רק שבמערכות מודרניות, הסיכויים הם שלא משתמשים בגישה הזו יותר.

3. העברת פרמטרים דרך המחסנית.

זו הגישה היותר מקובלת היום, לפחות בכל הקשור לשפות עיליות. זו שיטה גמישה וכללית מאד, ומקלה על מימוש מערכות הפעלה מרובות תהליכים. היא לאו דווקא היעילה ביותר אלקטרונית.

כאשר תוכניתן כותב רוטינה באסמבלי המשתלבת בתוכנה שנכתבת בשפה עילית, הוא חייב לציית למוסכמות הקריאה של השפה הזו והמימוש הספציפי שלה בקומפילר הנתון.

כאשר תוכנה נכתבת באסמבלי, התוכניתן יכול להעביר פרמטרים איך שהוא רוצה, אבל אם הוא מעונין להשתמש בסכמה כללית, מוטב שישתמש במוסכמות של שפת עילית זו או אחרת, ולא להמציא מחדש את הגלגל. פשוט כדאי להשתמש בסכמה ידועה ובדוקה, והדבר מאפשר גם שימוש ברוטינות הללו שפה העילית.

אנחנו נראה את סכמת העברת הפרמטרים של Turbo C++ for DOS, שישמש גם בתור דוגמא לסכמת העברת פרמטרים דרך המחסנית, וגם להבנה איך הקומפילר ממש את מנגנון הפרמטרים בשפה העלית.

<u>מוסכמות קריאת פרוצדורות של TURBO C</u>

לקומפילר של TURBO C יש פרמטר הנקרא מודל הזכרון. יש הבדלים מסוימים במוסכמות הקריאה של המודלים השונים. אנחנו נתרכז כאן במודל הזכרון SMALL.

מבחינת TURBO C, הפונקציה (פרוצדורה) הנקראת חייבת לקיים את התנאים הבאים:

הפונקציה הנקוראת חייבת לשמר את האוגרים:

BP ,SP, CS, DS, SS - תמיד

SI, DI - עשויים לשמש משתני אוגר (מעשית צריך לשמר אותם תמיד).

לשמר את האוגרים פירושו, שערכי האוגרים חייבים להיות <u>עם החזרה</u>

בדיוק אותם ערכים שהיו בהם עם הכניסה.

כמובן שבזמן ריצת הפונקציה, הם יכולים להשתנות.

כתיבת ערכי הפרמטרים למחסנית
ושיחרור שטח הפרמטרים ע"י קידום SP
באחריות התוכנית הקוראת.

העברת פרמטרים ע"י התוכנית הקוראית

C דוחף את הפרמטרים בסדר של ימין לשמאל,

הפרמטר השמאלי ביותר נדחף אחרון.

לאחר מכן כתובת חזרה לפקודה הבאה (על ידי ה-CALL עצמו).

החזרת ערכים

מידע חזרה הוא בעיקר דרך הפוינטרים.

תוצאות פונקציה - כאשר מדובר בתוצאות פונקציה שלמות (char, int, long)
ההחזרה היא דרך האוגר AX או דרך הזוג DX:AX.

תלוי באורך של התוצאה:

אם התוצאה היא 16 ביט או פחות char, int, unsigned int, פוינטר רגיל
וכו' - ההחזרה היא דרך האוגר AX.

במידה ומדובר בתוצאה 32 ביט, long, far pointer וכו' - DX:AX.

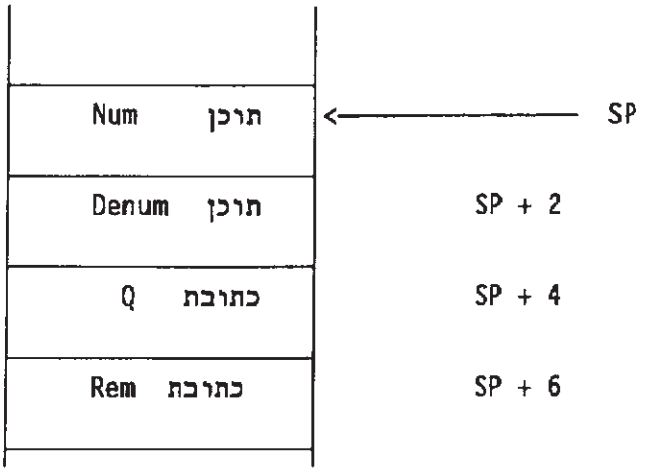
יש מוסכמות אחרות להחזיר רשומות (באורך לא ידוע) ומספרים ממשיים (float,
double, long double) לא נדון בזה כאן.

לדוגמא, אם ה-prototype של פונקצית אסמבלי היא:
int idiv_mod(int Num, int Denom, int *Q, int *Rem);
אזי בקריאה idiv_mod(Num, Denom, &Q, &Rem)

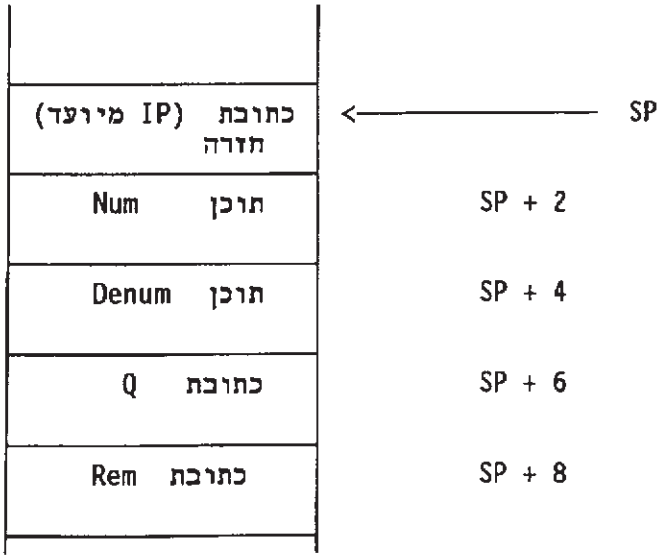
יתבצע דחיפת
כתובת של Rem,

כתובת Q,
תוכן Denom,
תוכן Num

במודל SMALL, כל הפיונטרים הם 16 ביט.
לפיכך רגע לפני ביצוע ה-CALL המחסנית תראה כך:



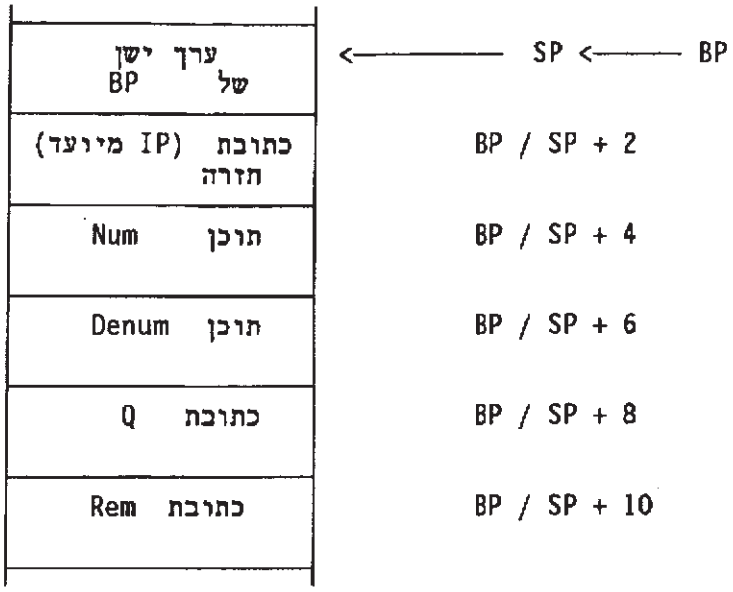
ברגע ביצוע ה-CALL, למחסנית נדחפת כתובת החזרה. במודל SMALL כל הפקודות ה-CALL הם NEAR. לפיכך לאחר ביצוע ה-CALL המחסנית תיראה:



כפי שנראה בהמשך, ההתחלה הסטנדרטית של כל פונקציית C הם הפקודות:

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

מצב המחסנית לאחר ביצוע הפקודות הללו הינו:


ערך ישן של BP
כתובת (IP מיועד) חזרה
תוכן Num
תוכן Denum
כתובת Q
כתובת Rem
SP
BP
BP / SP + 2
BP / SP + 4
BP / SP + 6
BP / SP + 8
BP / SP + 10

תוכניות דוגמא procs3.asm, procs4.asm, dos_prs1.asm

התוכניות הללו נועדו להמחיש שימוש בפרוצדורות בתוכניות אסמבלי טהורות ועוד כמה אספקטים.

מה שהתוכניות הראשיות עושות זה פשוט להדפיס את המחרוזת "Enter string:", לקבל מחרוזת כלשהיא מהמשתמש, להדפיס את המחרוזת "The string you entered:" ושוב להדפיס את המחרוזת שהמשתמש הכניס. לפיכך מה שרואים בפלט הן שלושת ההדפסות של התוכנית ועוד ההד (echo) של התוכנית הקולטת את הקלט מהמשתמש.

השימוש בפרוצדורות כאן מתבטא בביצוע פעולות הקלט וההדפסה בפרוצדורה מיעדת לכך. קליטת המחרוזת מהמשתמש נעשית ע"י פרוצדורה בשם Dos_Readln וכל שלושת ההדפסות נעשות ע"י פרוצדורה בשם Dos_Write.

המימוש של Dos_Readln מסתמך על רוטינת DOS

INT 21h, AH = 3Fh, BX = 0

שמשמעותה: "קרא שורת תוים מהמקלדת (עד ל-Enter) והעבר את תכנו (עד ל-CX תוים) לשטח זכרון המתחיל בכתובת DS:DX. החזר ב-AX את מספר התוים שהתקבלו בפועל.".

מי שמשתמש באופציה הזו למעשה צריך להכין שטח בגודל שהוא מעריך אמור להספיק לאכסון הקלט ולהעביר את הכתובת הזו לרוטינה. אולם, על מנת למנוע חריגה או "מריחה" של זכרון, הקורא לרוטינה גם מעביר את גודל השטח שהוא הכין, שישמש כגודל מירבי של תוים שיש להעביר לשטח הזה. במידה והקלט שהקליד מהמשתמש גדול מהגודל המירבי, רוטינת ה-DOS תעביר את הכמות המירבית של הנתונים ותשמור את השארית אצלה ויהיה אפשר לקרוא אותה בקריאות נוספות לרוטינה. הרוטינה הזו יכולה לקלוט נתונים לא רק מהמקלדת ובאיזה התקן מדובר נקבע ע"י הערך של BX, כאשר BX = 0 משמעותו המקלדת. לפיכך מיפרט העברת נתונים של הרוטינה הזו הינה:

קלט:

BX =	מספר ההתקן (0 עבור המקלדת)
DS:DX =	כתובת שטח הזכרון להעביר עליו את הקלט
CX =	גודל מירבי להעברה

פלט:

AX = מספר התוים שהוקלדו בפועל

המימוש של Dos_Write מסתמך על רוטינת DOS

INT 21h, AH = 40h, BX = 1

שמשמעותה: "הדפס מחרוזת תוים למסך שתוכנו נמצא בשטח זכרון המתחיל בכתובת DS:DX ושגודלו מצוין ב-CX". כמו הרוטינה הקודמת מדובר ברוטינה היכולה לגשת למספר התקנים ותוכן BX מצין איזו (BX = 1 משמעותה המסך). מאחר והתוכנית המדפיסה יודעת בדיוק כמה תוים היא רוצה להדפיס, אין את כל המנגנון של התמודדות עם שאלת הגודל. לפיכך מיפרט העברת נתונים של הרוטינה הזו הינה:

קלט:

BX = מספר ההתקן (1 עבור המסך)

DS:DX = כתובת שטח הזכרון ממנו לקרוא את המחרוזת להדפסה

CX = גודל מחרוזת ההדפסה

שתי הפרוצדורות Dos_Readln ו-Dos_Write מקבלות כפרמטרים את תוכני האוגרים DX ו-CX (למעשה גם את DS) ומעבירים את התכנים הללו ל-INT 21h יחד עם תוכני AH ו-BX שהן קובעות. Dos_Readln מחזיר כמות שהוא את תוכן ה-AL. אפשר לומר שהפרוצדורות חוסכות לקורא להם את הפקודות הקשורות להצבת ערכי ה-AH ו-BX והקריאה ל-INT 21h.

אפשר להגיד שהפרוצדורות הללו הן דוגמא להעברת פרמטרים באמצעות אוגרים במובן הזה שהן מפרשות את תוכני האוגרים בזמן תחילת הריצה שלהן כפרמטרים או נתונים שמועברים/מנחים אותן.

אלמנט נוסף ברוטינות הללו היא העובדה שהן "משמרות אוגרים" BX במקרה של Dos_Readln ו-AX, BX במקרה של Dos_Write. זהו נוהג שהיום לפעמים נוהגים בו ולפעמים לא, שבו פרוצדורה לשימוש כללי משמרת ומשחזרת ערכי אוגרים שהיא משתמשת. לפעמים יש מוסכמות שתוכניות קוראות לרוטינות רוצות לסמוך על ערכי אוגרים שמלפני הקריאה. בפרוצדורות כאן השימור/שיחזור נעשה בדרך המקובלת של שימור במחסנית ע"י PUSH עם תחילת הריצה ושיחזור ע"י POP ממש לפני החזרה. שימור/שיחזור אוגרים נעשה בדר"כ בפרוצדורה הנקראית כי אם זה יעשה בקוד הקורא לפרוצדורה קוד השימור שיחזור ישוכפל כמספר הקריאות לפרוצדורה. כמובן שפעולות מהסוג הזה גורמות לפרוצדורות לרוץ יותר זמן.

תוכנית דוגמא procs3.asm

כתוכנית הראשית המסומנת ע"י הליבל Main אפשר לראות את שלוש הקריאות

```
CALL Dos_Write
```

המצינות את ביצוע שלוש ההדפסות השונות ע"י אותה פרוצדורה. פקודות שלפני הקריאה מבצעות את הצבת הערכים המנחים את הרוטינה. בנוסף יש את הפקודה

```
CALL Readln
```

המבצעת את קריאת הנתונים מהמקלדת.

פקודות ההצבה ל-CX מבוצעות ע"י מספר טכניקות. למשל בקריאה הראשונה ל-Dos_Write צריך לספק את אורך המחרוזת להדפסה PromptString. אפשרות אחת היא פשוט לספור את התוים אבל לזה יש מספר חסרונות (טעות בספירה וגם העובדה ששינוי המחרוזת מחיב עדכון פקודת ההשמה). דרך אחת לעקוף את הבעיה הזו הוא להחסיר את הליבל PromptString מהליבל שמגדיר את נקודת הזכרון שמיד אחריו (ResultMsg). ליבלים הם קבועים שאמנם לא ניתן לעשות את רוב פעולות החשבון (סכום, כפל חילוק) אבל כן ניתן להחסיר אותם. ההפרש בין ליבלים מוגדר כגודל השטח בבתים בין שתי נקודות הזכרון שהן מצינות. החישוב הזה נעשה בזמן האסמבלי והתוצאה נחשבת לקבוע. לפיכך הפקודה

```
MOV CX,ResultMsg - PromptString
```

ממומש כ-MOV של קבוע ל-CX, קבוע שהוא גודל המחרוזת.
הסמל Max_Str_Length מוגדר ע"י פקודת EQU בתחילת התוכנית והוא למעשה קבוע סימלי למספר 1000 כלומר לפקודה

```
Max_Str_Length  EQU 1000
```

יש משמעות דומה לפקודה בשפת C של

```
#define Max_Str_Length 1000
```

אשר להדפסת המחרוזת שהמשתמש הכניס, התוכנית פשוט משמרת במשתנה

InputLength את הערך המתקבל מ-INT 21h ושולפת אותו משם ל-CX כאשר הדבר נחוץ להדפסה.

יש לשים לב שהפעם הפקודה הראשונה לביצוע היא בלב התוכנית ואנחנו דואגים לכך ע"י ההנחיה

```
END Main
```

תוכניות דוגמא procs4.asm, dos_prsl.asm

ההבדל העקרי בין procs3.asm לבין שני קבצים אלו היא שכאן מדובר בתוכנית אחת המתוארת ביותר מקובץ אחד. procs4.asm מכילה רק את התוכנית הראשית ואילו dos_prsl.asm מכילה רק את הפרוצדורות, כאילו לתמוך בשימוש בפרוצדורות Dos_Readln ו-Dos_Write כפרוצדורות שרות שיכולות לשמש גם תוכניות אחרות. מלבד העובדה שמדובר בשני קבצים הקבצים הללו מקיימים תנאים נוספים:

- שימו לב של-dos_prsl.asm יש END משלו ללא ציון ליבל, כי כתוכנית שרות היא אינה מצינת פקודה ראשונה לביצוע.

- על מנת ששם של פרוצדורה או משתנה המוגדר באסמבלי יהיה גלובלי (יוכר מחוץ לקובץ שבו הוא מוגדר) יש להכריז עליו כ-PUBLIC. לפיכך יש בקובץ dos_prsl.asm את ההנחיות

```
PUBLIC Dos_Readln

  .....

PUBLIC Dos_Write
```

אפשר היה גם

```
PUBLIC Dos_Readln, Dos_Write
```

- על מנת שתוכנית בקובץ תכיר סמלים גלובליים המוגדרים בקובץ אחר יש להכריז עליו ע"י הנחית EXTRN יחד עם הסוג שלו. שים לב שזו הנחית

"EXTRN" ולא "EXTERN" (ללא "E" אמצעי). ההנחיה

```
EXTRN Dos_Readln:NEAR, Dos_Write:NEAR
```

מצינות שהסמלים הללו שם של פרוצדורות גלובליות מסוג NEAR.

סוגים אחרים שידועים לי שניתן להכריז עליהם כ-EXTRN:

NEAR	פרוצדורה גלובלית
FAR	פרוצדורה גלובלית
BYTE	משתנה גלובלי
WORD	משתנה גלובלי
DWORD	משתנה גלובלי
FWORD	משתנה גלובלי
PWORD	משתנה גלובלי
QWORD	משתנה גלובלי
TBYTE	משתנה גלובלי
UNKNOWN	לא ידוע

- הסיבה שהפרוצדורות Dos_Readln ו-Dos_Write מוגדרות FAR בקובץ procs3.asm ו-NEAR ב-dos_prsl.asm היה רק להמחיש שברמת הטקסט אין הבדל בשימוש בשני סוגי הפרוצדורות ואין קשר בין הסוגי הפרוצדורות לשאלה אם התוכנית היא בקובץ אחד או יותר.

- על מנת ליצור קובץ EXE מתוכנית הנמצאת במספר קובצי asm יש לעשות tasm לחוד לכל אחד מהקבצים ו"לסכם" אותם ב-tlink. לפיכך הנוהל במקרה שלנו הוא:

```
tasm procs4.asm
  .....
tasm dos_prsl.asm
  ......
tlink procs4.obj + dos_prsl.obj
```

```
;
;  procs3.asm - Demonstrate use of local procedures
;
     .MODEL SMALL
     .STACK 100h
     .DATA
Max_Str_Lngth EQU 1000
PromptString DB 'Enter string:',13,10
ResultMsg DB 'The string you entered:',13,10
InputString  DB  Max_Str_Lngth  DUP (?)
InputLength DW ?
                                        ;
      .CODE
Dos_Readln  PROC  FAR
     ; CALL INT 21h with AH = 3Fh, BX = 0
     ; Read from stndin until eoln, ...
     ; ... but no more than CX chars into buffer
     ; Input expected:
     ;     DX = Buffer OFFSET
     ;     CX = Max buffer size
     ; Output:
     ;     AX = Number of chars read.
     ;
      PUSH BX                           ; Preserve service register
      MOV AH,3Fh                        ; DOS read from handle function #
      MOV BX,0                          ; Standard input handle
      INT 21h                           ; Get the string
      POP BX                            ; Restore service register
      RET
Dos_Readln  ENDP
                                        ;
                                        ;
Dos_Write  PROC  FAR
     ; CALL INT 21h with AH = 40h, BX = 1
     ; Write to stndout CX chars from buffer
     ; Input expected:
     ;     DX = Buffer OFFSET
     ;     CX = Number of chars to print.
     ;
      PUSH AX                           ; Preserve service register
      PUSH BX                           ; Preserve service register
      MOV AH,40h                        ; DOS write from handle function #
      MOV BX,1                          ; Standard output handle
      INT 21h                           ; Print the string
      POP BX                            ; Restore service register
      POP AX                            ; Restore service register
      RET
Dos_Write ENDP
```

```
Main:
      MOV AX,@DATA
      MOV DS,AX                        ; Set DS  to point to data segment
                                       ;
      MOV DX,OFFSET PromptString       ; Set DX to point to string
                                       ; Set number of chars to print
      MOV  CX,ResultMsg - PromptString
      CALL Dos_Write                   ; Print the string
      MOV CX,Max_Str_Lngth             ; Read up to maximum number of chars
      MOV DX,OFFSET InputString        ; Store the string here
      CALL Dos_Readln                  ;
      MOV  InputLength,AX              ; Preserve input length
                                       ;
      MOV DX,OFFSET ResultMsg          ; Set DX to point to string
                                       ; Set number of chars to print
      MOV  CX,InputString - ResultMsg
      CALL Dos_Write                   ; Print the string
      MOV DX,OFFSET InputString        ; Set DX to point to string
      MOV  CX,InputLength              ; Set number of chars to print
      CALL Dos_Write                   ;
                                       ;
      MOV AH,4Ch                       ; DOS terminate program function #
      INT 21h                          ; Terminate the program
      END Main
```

```
E:\>tasm procs3

Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International

Assembling file:   procs3.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  413k

E:\>tlink procs3

Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

E:\>procs3

Enter string:
A1B2C3D4E5F6
The string you entered:
A1B2C3D4E5F6

E:\>
```

```
E:\>tasm procs3

Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International

Assembling file:   procs3.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  413k

E:\>tlink procs3

Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

E:\>procs3

Enter string:
A1B2C3D4E5F6
The string you entered:
A1B2C3D4E5F6

E:\>
```

```
;
;  procs4.asm - Demonstrate use of external procedures
;
     .MODEL SMALL
     .STACK 100h
     .DATA
Max_Str_Lngth EQU 1000
PromptString DB 'Enter string:',13,10
ResultMsg DB 'The string you entered:',13,10
InputString  DB  Max_Str_Lngth  DUP (?)
InputLength DW ?
      .CODE
       EXTRN Dos_Readln:NEAR,Dos_Write:NEAR
Begin:
      MOV AX,@DATA
      MOV DS,AX                        ; Set DS  to point to data segment
                                       ;
      MOV DX,OFFSET PromptString       ; Set DX to point to string
                                       ; Set number of chars to print
      MOV  CX,ResultMsg - PromptString
      CALL Dos_Write                   ; Print the string
      MOV CX,Max_Str_Lngth             ; Read up to maximum number of chars
      MOV DX,OFFSET InputString        ; Store the string here
      CALL Dos_Readln                  ;
      MOV  InputLength,AX              ; Preserve input length
                                       ;
      MOV DX,OFFSET ResultMsg          ; Set DX to point to string
                                       ; Set number of chars to print
      MOV  CX,InputString - ResultMsg
      CALL Dos_Write                   ; Print the string
      MOV DX,OFFSET InputString        ; Set DX to point to string
      MOV  CX,InputLength              ; Set number of chars to print
      CALL Dos_Write                   ;
                                       ;
      MOV AH,4Ch                       ; DOS terminate program function #
      INT 21h                          ; Terminate the program
      END Begin
```

```
E:\>tasm procs4.asm
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International

Assembling file:   procs4.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  389k


E:\>tasm dos_prs1.asm

Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International

Assembling file:   dos_prs1.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  390k


E:\>tlink procs4.obj + dos_prs1.obj
Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

E:\>procs4
Enter string:
ABCD123
The string you entered:
ABCD123

E:\>
```

```
E:\>tasm procs4.asm
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International

Assembling file:   procs4.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  389k


E:\>tasm dos_prs1.asm

Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International

Assembling file:   dos_prs1.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  390k


E:\>tlink procs4.obj + dos_prs1.obj
Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

E:\>procs4
Enter string:
ABCD123
The string you entered:
ABCD123

E:\>
```

תוכניות דוגמא call_idl.c, idiv_mo4.asm, idiv_mo5.asm

התוכניות הללו משמשות כמובן כדוגמא לקריאה לפרוצדורת אסמבלי מתוך תוכנית C.

ניהול המחסנית של התוכניות תוארו קודם. להלן יתר התיעוד של התוכנית.

מה שהתוכנית המשולבת, call_idl.c עם כל אחד מתוכניות ה-asm, idiv_mo4.asm או idiv_mo5.asm, עושה היא לקבל 2 מספרים שלמים (לאו דווקא חיוביים) ולחשב את החלוקה ללא שארית ושארית החלוקה של 2 המספרים הללו ולהדפיס אותם. התוכנית מוגנת מפני בקשה לחלוקה באפס.

מימוש החלוקה נעשה ע"י פקודת המכונה IDIV, ואפשר לראות מהריצות שמבחינתה של פקודת המכונה הזו היא ששארית החלוקה של 105 ב- 44- הוא 17, שארית החלוקה של 105- ב- 44 הוא 17- ושארית החלוקה של 105- ב- 44- הוא 17-. מתכנת הכותב תוכנית המחשבת שארית חלוקה של מספרים שעשויים להיות שליליים חייב לבדוק אם המוסכמות הללו מקובלים עליו.

קימפול תכנית המשלבת קבצי מקור ב-C ואסמבלי ניתן לעשות על ידי תוכנת tcc.exe (בתנאי שיש לך גם את tasm.exe) או bcc.exe . עקרונית כותבים את רשימת הקבצים שממנה מורכבת התוכנית כאשר התוכנית הראשית חייבת להיות ראשונה. במקרה שלנו, קימפול התוכנית המורכבת מהקבצים call_idl.c ו-idiv_mo4.asm יהיה:

```
tcc call_idl.c idiv_mo4.asm
```

תוצאת הקימפול של השורה הזו (בתנאי שאין שגיאות) תהיה לפי השם של הקובץ הראשון, כלומר יווצר call_idl.exe.

אם רוצים להכין את הקובץ לדיבוג ב-Turbu Debugger אזי פשוט מוסיפים את האופציה "v-" כלומר

```
tcc -v call_idl.c idiv_mo4.asm
```

התוכניות idiv_mo4.asm ו-idiv_mo5.asm

ההבדל בין שתי הקבצים הללו שב-idiv_mo5.asm אני משתמש באוגרים SI ו-DI, לכן אני חייב לשמר אותם בקובץ הזה ולשחזר אותם לפני החזרה, מה שאין צורך ב-idiv_mo4.asm.

נקודות שיש לשים לב אליהם:

- כל שם המוגדר בתוכניות C או שהתוכנית מתיחסת אליו, באסמבלי חייבים להתיחס אליו עם קו תחתי ("_") מוביל. מהסיבה הזו בקבצי האסמבלי שמה של הרוטינה הנקראית מ-C היא "_idiv_mod".

- _idiv_mod צריכה לחלק מספר 16 ביט במספר 16 ביט, אבל הפקודה IDIV מחלקת 32 ביט (DX:AX) ב-16 ביט. על מנת לבצע את המשימה עלינו "להרחיב" את המספר ב-AX לתוך DX. אילו היה מדובר במספרים חסרי סימן היה מדובר כאן בהצבת 0 ל-DX, אבל במספרים עם סימן צריך להביא בחשבון את הסימן של AX: אם הוא חיובי, צריך להציב 0 ל-DX, ואם הוא שלילי, צריך להציב 1 לכל הביטים של DX. הדבר הוא למעשה הצבת ביט הסימן של AX לכל הביטים של DX. יש פקודת מכונה שעושה בשבילנו בדיוק את זה: CWD. גרסאות דומות של הפקודה הזו:

CBW		AX	ל-	AL את הרחב
CWD		DX:AX	ל-	AX את הרחב
CWDE		EAX	ל-	AX את הרחב
CWQ		EDX:EAX	ל-	EAX את הרחב

- שימו לב למבנה:

```
      MOV AX,0
      JMP Done

       ....

      MOV AX,1
Done:
       .....
      POP BP
      RET
```

המבנה הזה מבטיח שעם החזרה לקוד הקורא AX מכיל את הערך הנכון של תוצאת הפונקציה.

```
/* 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 call_id1.c idiv_mo4.asm
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
call_id1.c:
idiv_mo4.asm:
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International

Assembling file:   idiv_mo4.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  395k

Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

        Available memory 4111504

E:\>call_id1.exe
Enter Numerator, Denominator
:105 44

 105 div 44 = 2, mod(105,44) = 17

E:\>call_id1.exe
Enter Numerator, Denominator
:105 -44

 105 div -44 = -2, mod(105,-44) = 17

E:\>call_id1.exe
Enter Numerator, Denominator
:-105 44

 -105 div 44 = -2, mod(-105,44) = -17

E:\>call_id1.exe
Enter Numerator, Denominator
:-105 -44

 -105 div -44 = 2, mod(-105,-44) = -17

E:\>
```

```
;  idiv_mo4.asm - Assembler implementation of
;                          C-callable function idiv_mod.
;
;
     .MODEL SMALL
     .CODE
     ; Implementation of C callable function ...
     ;     ...  int idiv_mod(int Num, int Denom,  int *Q,  int *Rem)
     ;                          [BP+4]    [BP+6]      [BP+8]    [BP+10]
     ;
     ;  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 NEAR
      PUSH BP          ; Preserve BP
      MOV BP,SP        ; Set BP to point to Parameter area
      MOV CX,[BP+6]    ; CX := Denom
      CMP CX,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+4]          ; AX := Num
      CWD                    ; DX:AX := AX
      IDIV CX                ; AX := DX:AX / CX,   DX := MOD(AX,CX)
      MOV  BX,[BP+8]         ; BX := Offset Q
      MOV  [BX],AX           ; *Q := AX
      MOV  BX,[BP+10]        ; BX := Offset Rem
      MOV  [BX],DX           ; *Rem := DX
      MOV  AX,1    ; Ensure return value := 1
Done:
      POP  BP                ; Restore BP register
      RET
_idiv_mod  ENDP
      END
```

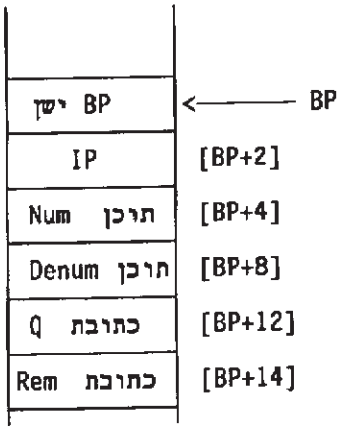
```
;  idiv_mo5.asm - Assembler implementation of
;                          C-callable function idiv_mod.
;
     .MODEL SMALL
     .CODE
     ; Implementation of C callable function ...
     ;     ...  int idiv_mod(int Num, int Denom,  int *Q,  int *Rem)
     ;                          [BP+4]    [BP+6]     [BP+8]    [BP+10]
     ;  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 NEAR
      PUSH BP          ; Preserve BP
      MOV BP,SP        ; Set BP to point to Parameter area
      PUSH SI          ; Preserve register variables
      PUSH DI          ;
;
      MOV SI,[BP+6]   ; 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+4]          ; AX := Num
      CWD                    ; DX:AX := AX
      IDIV SI                ; AX := DX:AX / SI,   DX := MOD(AX,SI)
      MOV  DI,[BP+8]         ; DI := Offset Q
      MOV  [DI],AX           ; *Q := AX
      MOV  DI,[BP+10]        ; DI := Offset Rem
      MOV  [DI],DX           ; *Rem := DX
      MOV  AX,1    ; Ensure return value := 1
Done:
;
      POP DI     ; Restore register variables
      POP SI     ;
      POP  BP                ; Restore BP register
      RET
_idiv_mod  ENDP
      END
```

תוכנית דוגמא call_id2.c, idiv_mo6.asm

בתוכנית המשולבת call_id1.c, idiv_mo1.asm כל הפרמטרים היו 2 בתים כי זה הגודל של הן int וכן פוינטרים כהקשר הזה של קומפילציה של טורבו C. נראה שזה לא תמיד כך בהמשך הקורס. כאשר מחשבים את המיקום של פרמטרים צריך תמיד להביא בחשבון את גודל הפרמטרים, שכמובן לא יהיו תמיד 2 בתים. זה יקרה למשל אם היינו עובדים עם פרמטרים long int במקום int. זה מה שקורה בדוגמא הבאה, התוכנית המשולבת call_id2.c, idiv_mo6.asm. ההגדרה של idiv_mod32 הינה:

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

מאחר ו-Num ו-Denom הם עכשיו 32 ביט, תמונת המחסנית היא עכשיו כזו:



בנוסף לכך שמבנה הפרמטרים במחסנית משתנה, ה-idiv_mod32 צריכה לבצע אריטטיקה של 32 ביט.

```
/* 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:\>tcc call_id2.c idiv_mo6.asm
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
call_id2.c:
idiv_mo6.asm:
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland
International

Assembling file:    idiv_mo6.ASM
Error messages:     None
Warning messages:   None
Passes:             1
Remaining memory:   431k

Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

        Available memory 4150128

E:\>CALL_ID2.EXE
Enter Numerator, Denominator
:-700000 66666

 -700000 div 66666 = -10, mod(-700000,66666) = -33340

E:\>
```

```
;  idiv_mo6.asm - Assembler implementation of
;                          C-callable function idiv_mod32.
;
     .MODEL SMALL
     .CODE
     .386
     ; Implementation of C callable function ...
     ;     ...  int idiv_mod32(long int Num, long int Denom,
     ;                              [BP+4]          [BP+8]
     ;  long int *Q,     long int *Rem)
     ;       [BP+12]     [BP+14]
     ;  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 NEAR
      PUSH BP          ; Preserve BP
      MOV BP,SP        ; Set BP to point to Parameter area
      PUSH SI          ; Preserve register variables
      PUSH DI          ;
;
      MOV ESI,[BP+8]   ; SI := 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+4]         ; EAX := Num
      CDQ                   ; EDX:EAX := EAX
      IDIV ESI               ; EAX := EDX:EAX / ESI,   EDX := MOD(EAX,ESI)
      MOV  DI,[BP+12]        ; DI := Offset Q
      MOV  [DI],EAX          ; *Q := AX
      MOV  DI,[BP+14]       ; DI := Offset Rem
      MOV  [DI],EDX          ; *Rem := DX
      MOV  AX,1    ; Ensure return value := 1
Done:
;
      POP DI     ; Restore register variables
      POP SI     ;
      POP  BP                ; Restore BP register
      RET
_idiv_mod32  ENDP
      END
```