

פסיקות (Interrupts)

פסיקה היא מצב של העברת שליטה בכפיה של ה-CPU לרוטינה. דרך אחרת לחשוב על פסיקה היא הפסקה בכוח של ריצה שוטפת של תוכנית והסתעפות לרוטינה אחרת.

פסיקה היא למעשה (או יכולה להיות) תגובה לאיתות של החומרה שמחוץ ל-CPU.

מדובר למעשה בעצירת התוכנית הנוכחית, ושפעול רוטינה אחרת על פניה. הרוטינה הזו תהיה בדרך כלל (אבל לא תמיד) רוטינה בלתי תלויה בתוכנית שמהלכה השוטף נעצרה. הרוטינה הזו "בלתי תלויה" בתוכנית המופסקת במובן הזה, שכותב התוכנית (בדרך כלל) אינו מכיר את הרוטינה החלופית, אינו מעוניין בעצירת התוכנית שלו ויכול להיות שהרוטינה החלופית אינה משרתת את מטרותיו כלל. בדרך כלל התוכניתן אפילו אינו מביא את האפשרות של פסיקה בחשבון. המהלך של פסיקה עשוי להתרחש, בדרך כלל, אחרי כל פקודת מכונה בתוכנית, ללא תאום כלשהוא איתה וללא אפשרות לחזות את נקודת הזמן או הפקודה מראש. כמובן שיוצא מזה שבדרך כלל, ה"רוטינות החלופיות" הללו, שבהמשך נקרא להם ISR-ים, נכתבות בצורה כזו שההפעלה שלהם לא תשפיע לרעה על התוכניות המופסקות, או שהשפעה שלילית כזו תהיה מינימלית. ה-ISR בדרך כלל ישתדל להחזיר את השליטה לתוכנית המופסקת במהירות המירבית, והתוכנית המופסקת תתחדש כמצב זהה לרגע שבה היא הופסקה.

אולי הדרך הטובה ביותר להבין את מושג הפסיקה הוא לראות איך הוא מתבטא ב-CPU:

כאשר תוכנית רגילה מתבצעת, הפקודה הבאה לביצוע הנקבעת על ידי הערך של הזוג CS:IP, מתעדכנת באופן שוטף לפי רצון התוכנית:

עבור פקודות בקרה, אותם פקודות שתפקידם לשנות את ה-IP או ה-CS:IP (כמו JMP, CALL), הערך החדש נקבע על פי ערכים שהפקודה מכילה.

עבור יתר הפקודות כמו ADD, MOV (ואילה הרוב) הפקודה העוקבת בזיכרון מיד אחרי הפקודה המתבצעת הנוכחית.

בשני המקרים הפקודה הבאה לביצוע נמצאת בתוך התוכנית המתבצעת.

פסיקה גורמת (או עשויה לגרום) שינוי הזוג CS:IP לערכים שהם מחוץ

לתוכנית המופסקת.

סוגי פסיקות

מבחינה לוגית, פסיקות נבדלות לכמה סוגים:

1. Hardware Interrupts - פסיקות חומרה - תגובה לאיתות רכיב חומרה במחשב שמחוץ ל-CPU. פסיקות מסוג זה הם בדרך כלל לצורכי קלט / פלט - למשל כתוצאה של לחיצת מקש במקלדת או הזזת העכבר.

2. Exceptions - חריגות - פסיקות הנובעות ממצב לא תקין במהלך הריצה של התוכנית הנוכחית. חלוקה באפס (divide overflow) היא דוגמא קלאסית לכך. דוגמאות אחרות הם נסיון לבצע פקודת מכונה לא חוקית, נסיון לגשת לכתובת לא קיימת בזיכרון, גישה לכונן דיסקטים שאין בו דיסקט וכו'.

3. Software Interrupts - פסיקות תוכנה - פסיקות ביוזמת התוכנית - בדרך כלל ע"י הפקודה INT. פסיקות מסוג זה הם בדרך כלל שימוש ברוטינות שירות הקיימות בזיכרון (אך מחוץ לתוכנית). זאת כבר ראינו: כאשר השתמשנו בספריית הרוטינות של DOS (INT 21h).

למעשה התאור שבראשית סיכום זה מתאים רק לפסיקות מסוג 1 ו-2 (פסיקות חומרה וחריגות). פסיקות מסוג 1 ו-2 שונים מאוד מפסיקות מסוג 3 (פסיקות תוכנה). פסיקות תוכנה הם כמעט כמו הסתעפות לרוטינה: המתכנת מכיר את הרוטינות הנקראות ופונה אליהם ביוזמתו.

מה מבדיל בין פסיקת תוכנה להסתעפות לרוטינה (call)?

יש שני הבדלים עיקריים:

1. הסתעפות לרוטינה היא לקוד הנמצא בתוך קוד ה-EXE של התוכנית הרצה.

2. פסיקת תוכנה נעשית דרך מבנה נתונים גלובלי של המחשב.

1. לגבי במערכות מוגנות (UNIX, WINDOWS NT) לא ניתן בעזרת CALL לעשות מה שעושים בפסיקת תוכנה: הסתעפות לקוד הנמצא מחוץ לקובץ ה-EXE. אולי יותר

נכון לומר שדואגים לכך שהדבר יהיה בלתי אפשרי. כאשר תוכנית מורצת, היא מועתקת לזיכרון ומקבלת שליטה. הסתעפויות CALL מוגבלות לנקודות זיכרון הנמצאים ב"תמונה" הזו של קובץ ה-EXE בזיכרון. כאשר תוכנית ב-C למשל קוראת לרוטינות סטנדרטיות כמו strcpy, sin, qsort (כל רוטינה שאינה קלט / פלט ואינה משפיעה / מסתיעה במערכת ההפעלה) מוכלל בתוך קובץ ה-EXE קוד בינארי (מתוך סיפריות סטנדרטיות) המישם את הפעולות הללו ונטען עמה לזיכרון. זה כמובן נכון גם עבור קוד מקורי שנכתב בתוכנית עצמה. רוטינות קלט / פלט כמו printf, fopen הינם קודים שנמצאים בקובץ ה-EXE אבל מסתעים במערכת ההפעלה ע"י פסיקות תוכנה.

תחת DOS אפשר לומר ש-CALL הוא כמעט תמיד להסתעפות לקוד בתוך קובץ ה-EXE למרות שאין מניעה לעשות אחרת. זה עניין של נוהג יותר מאשר עניין טכני. תחת DOS ניתן להסתעף לרוטינות חיצוניות כאילו ע"י פקודת CALL (ע"י שימוש בפוינטר לפונקציה) אבל בשפות עילית שימוש כזה ב-CALL הוא נדיר מאד ונעשה רק במצבים מיוחדים.

לגבי 2., קריאה לפסיקת תוכנה היא דרך משאב (מבנה נתונים) גלובלי - טבלת הפסיקות IV (שיתואר להלן). המשאב הזה מוגבל (ל-256 פוינטרים) השימוש בו דורש מיומנות ואחריות ומה שאולי חשוב מכל - במערכות מוגנות (UNIX, NT) הוא גם מוגן - תוכניות רגילות לא יכולות לשנות אותו.

מנגנון מימוש פסיקות

טבלת הפסיקות Interrupt Vector או IV.

במחשב PC קיימים 256 (מ-0 עד 255) מספרי פסיקות שניתן להקצות למטרות כאלו ואחרות, לא כולם מנוצלות. המספר 256 הינו מאפיין של המחשב. אין אפשרות לשנות זאת. קיים מערך של 256 פוינטרים מלאים (segment + offset) שהם הפוינטרים "לאן להסתעף" עבור 256 פסיקות אפשריות ממספרות 0..255. המערך טבלה הזו נקרא Interrupt Vector או IV. ב-8086 ובמצב real mode של ה-x86 המתקדמים יותר, הפוינטרים האלו הם כולם 32bit (16 ל-offset, 16 ל-segment) והמערך נמצא ב-1K הכתובות הפיזיות הראשונות (0..1023) של המחשב. עבור פסיקה מספר K, 4 הבתים בכתובות $K*4$ עד $K*4 + 3$ הינם פוינטר מלא - קודם offset (בתים בכתובות $K*4$, $K*4+1$) ואחר כך segment (בתים בכתובות $K*4+2$, $K*4+3$) - שהוא כתובת היעד "לאן להסתעף" בהתרחש פסיקה מספר K. היעד הזה הוא לרוטינה המיוחדת שתפקידה לבצע את המצופה מהפסיקה הזו, והרוטינה הזו נקראת רוטינת הטיפול בפסיקה Interrupt Service Routine או בקיצור ISR. לדוגמא, עבור פסיקה מספר 9 הפוינטר ל-ISR של הפסיקה נמצא בכתובות 36 - 39. ה-offset נמצא בכתובות 37 - 36, וה-segment בכתובות 38

ב-Protected mode יש הבדלים לגבי ה-IV לעומת המצב ב-Real mode. לא ניכנס לזה כאן.

מה גורם לפסיקה?

בפסיקות חומרה - הפסיקה נגרמת ע"י איתותים של רכיבי חומרה (כמו המקלדת למשל).

בפסיקות תוכנה - ע"י ביצוע של פקודת המכונה INT.

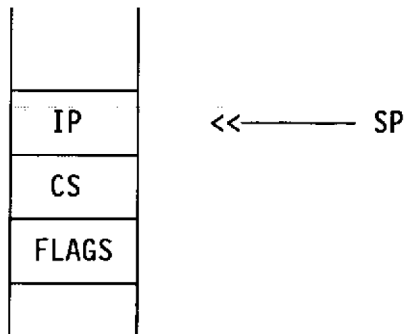
מנגנון מימוש הפסיקות

המנגנון מימוש הפסיקות משותף לפסיקות תוכנה וחומרה.

כאשר מתרחשת פסיקה מספר K מתרחש התהליך הבא:

- א. ה-CPU גומר את פקודת המכונה שהוא מבצע כרגע,
 - ב. מיד לאחר מכן הוא שומר את אוגר הדגלים (FLAGS) במחסנית (פעולת PUSH).
 - ג. הוא מאפס את הדגל Interrupt Flag (IF) באוגר הדגלים.
- על כך נרחיב מאוחר יותר. כרגע נציין שהדבר מונע התרחשות פסיקות נוספות, תוך שהמחשב עסוק בטיפול בנוכחית.
- ד. הוא מאפס את הדגל Trap Flag (TF) באוגר הדגלים.
- זהו דגל שקשור בעיקר למימוש debuggers. על כך נרחיב מעט בהמשך.
- ה. במחסנית נשמרים (במעין PUSH) "כתובת הפקודה הבאה" ה-CS הנוכחי ולאחר מכן ה-IP.

לפיכך תוכן ראש המחסנית היא כעת



ו. מתוך הכתובות $K*4$ עד $K*4+3$ של ה-IV נשלפים הערכים החדשים של ה-IP $(K*4, K*4+1)$ ו- $CS(K*4+2, K*4+3)$.

הסתיגות

לא תמיד הכתובת CS:IP הנשמרת במחסנית היא הכתובת של הפקודה הבאה לביצוע. עבור חריגות מסוימות (כמו ה-divide overflow, החלוקה באפס) הכתובת הנשמרת היא לא הפקודה הבאה לביצוע אלא דווקא הכתובת של הפקודה שהופסקה. השיקול כאן הוא כנראה להקל על איתור תקלות.

לסיכום, מתרחש כאן תהליך דומה במידה רבה לביצוע פקודה המכונה CALL: ההבדלים העיקריים הם מקור כתובת היעד (ה-IV), שמירת אוגר הדגלים ואיפוס ה-IF.

ה-Interrupt Flag (IF)

בתוך אוגר הדגלים קיים דגל ה-Interrupt Flag או IF, השולט על מנגנון הפסיקות. כאשר $IF = 0$, מנגנון פסיקות החומרה מושבת. ה-CPU מתעלם מבקשות פסיקות חומרה. כאשר $IF = 1$, מנגנון פסיקות החומרה פעיל. ה-IF לא משפיע על פסיקות התוכנה (הפקודה INT). הדבר נחוץ משום שיש מצבים שבהם התוכנה לא יכולה להרשות לעצמה התרחשות פסיקה, מצבים שעוד נראה.

פקודות מכונה שקשורות למנגנון הפסיקות

CLI - איפוס ה-IF ($IF = 0$).

STI - הדלקת ה-IF (IF = 1).

INT k - k תמיד מספר קבוע - גורם לפסיקת תוכנה k.

IRET - חזור מטיפול בפסיקה - שליפה (POP) מהמחסנית את ה-IP, CS, FLAGS מהמחסנית.
שחזור אוגר הדגלים (לערך שישנו במחסנית) ושינוי ה-CS:IP לערכים שנשלפו מהמחסנית.

שימו לב שעם ביצוע ה-IRET, ה-IF וה-TF ישתחזרו באופן אוטומטי לערכם המקורי לפני הפסיקה עם שיחזור הדגלים.

ה-Trap Flag (TF)

ה-TF הוא דגל מיוחד באוגר הדגלים שכאשר הוא דלוק, ה-CPU מתפקד בצורה מיוחדת מאד: הוא מבצע פסיקה (למעשה חריגה) של פסיקה מספר 1 אחרי כל ביצוע של פקודת מכונה. פסיקה מספר 1 נקראית בשל כך פסיקת ה-single step. שים לב, שמכיון שה-TF מכובה אוטומטית ע"י מנגנון הפסיקות, התופעה הזו לא תתרחש ב-ISR של פסיקה 1 עצמה. מנגנון זה נועד בעיקר למימוש תוכנות debugger-ים תחת DOS. ה-debugger היה לוקח לעצמו את השליטה על פסיקה מספר 1 ואחרי ביצוע של כל פקודת מכונה של התוכנית הנבדקת ה-ISR שלו של פסיקה 1 היה בודק את מצב ה-CPU והתוכנית ולפי זה היה מחליט איזה פעולות לנקות.

ל-TF אין פקודות מיוחדות להדליקה / כבוי שלו, כפי שיש ל-IF (CLI, STI) למשל. ההדלקה או כיבוי הדגל נעשה בעקיפין בעזרת הפקודות POPF או IRET. בדרך כלל הדבר יעשה על בסיס הערך הנוכחי של אוגר הדגלים.

קוד אופיני להדלקת ה-TF הוא כלהלן:

```
PUSHF
POP AX
OR AX,100000000B
PUSH AX
POPF
```

קוד אופיני לכבוי ה-TF הוא כלהלן:

```
PUSHF
POP AX
AND AX,111111011111111B
PUSH AX
POPF
```

ה-ROM-BIOS

את ה-ROM-BIOS אפשר לכנות "תוכנה באדיבות היצרן".

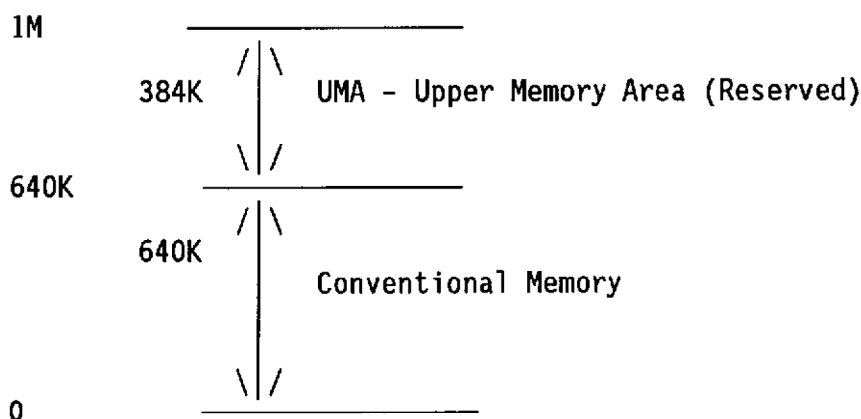
ראשי התיבות של ROM-BIOS הוא:

Read Only Memory, Basic Input Output System

מדובר ברוטינות אסמבלי (שעקרונית כל אחד יכול לכתוב), אשר מממשות בצורת ISR-ים. זו בעצם סיפריית רוטינות טיפול בפסיקות.

המיקום שלהם מבחינת כתובות זיכרון הוא ב-384K של זיכרון בין ה-640K הראשונים ל-1M או ה-Upper Memory Area או ה-UMA. אותו שטח מיוחד ששמור ל-SYSTEM כלומר לא האפליקציה וגם לא מערכות ההפעלה פועלים שם.

מרחב הכתובות 0 - 1M הראשונים ב-PC נראים כך:



הרוטינות של ה-BIOS מהווים סיפריית רוטינות שמממשות פונקציות קלט / פלט בסיסיות. רובם ISR-ים של פסיקות תוכנה אך חלקם ISR-ים של פסיקות חומרה. רוטינות ה-BIOS מאפשרות לכותבי מערכות ההפעלה (DOS במקרה שלנו) לכתוב קוד שמבצע קלט / פלט בסיסי מבלי לגשת ישירות לחומרה. אחד היתרונות בכך, שלמרות שיש מספר גדול של יצרני PC ולמרות שיש הבדלים בין היצרנים, DOS עובד על כולם. הכותבים של DOS לא היו צריכים להביא בחשבון את ההבדלים בין היצרנים. בכדי ש-PC יהיה תואם, הוא לא חייב להיות זהה ליצרנים האחרים, מספיק שרוטינות ה-BIOS שלו (השונות מיצרן ליצרן) יעמדו בסטנדרטים מסוימים. אלמלא היה הדבר כך, לא רק שהיו בעיות בין יצרנים, אלא גם בעיות בין מודלים שונים של אותו יצרן (386, 286, 8086...). בכדי לקבל תחושה מה היה קורה אלמלא היה BIOS, שימו לב מה קורה כאשר מתקינים מדפסת או מודם או כרטיס רשת חדש למחשב: צריך להתקין תוכנה מיחדת הספציפית ליצרן של המכשיר החדש (הנקרא Device driver). מערכות הפעלה היום מסופקות עם דיסקים של סיפריות של

drivers של מאות יצרנים ומכשירים. זה היה המצב עבור כל סוג של קלט / פלט, אלמלא ה-BIOS.

יש לשים לב: בניגוד ל-INT 21h, שהם חלק ממערכת ההפעלה (DOS) רוטינות ה-BIOS אינם חלק משום מערכת ההפעלה. הם חלק מהמחשב והם משרתים כל מערכת הפעלה שמעוניין בשירותם. מערכת הפעלה אינה חייבת, כמובן, להשתמש בהם, חלקם או כולם, ואכן מערכות הפעלה כמו LINUX או NT בודאי מתעלמות מחלק מהם, משיקולים שונים.

רשימת חלקית של פסיקות חשובות

מספר פסיקה	שם	סוג הפסיקה	אחריות
0	Divide overflow	חריגה	DOS
5	Print-Screen	חומרה	BIOS
8	Timer	חומרה	BIOS
9	Keyboard	חומרה	BIOS
10h (16)	Video	תוכנה	BIOS
13h (19)	Floppy	תוכנה	BIOS
16h (22)	Keyboard	תוכנה	BIOS
18h (27)	Crtl-Break	חומרה	DOS
21h (33)	Function Request	תוכנה	DOS

הערות:

פסיקה מספר 21h הוא כמובן פסיקת התוכנה שהשתמשנו עד עכשיו ל-קלט/פלט דרך DOS (INT 21h).

פסיקות 9 ו-16h שניהם פסיקות מקלדת (Keyboard). זו אינה כפילות. פסיקה 9 היא פסיקת החומרה של המקלדת (מתרחשת עם כל לחיצה / שחרור של מקש). לעומת זאת פסיקה 16h היא פסיקת התוכנה של המקלדת - רוטינה שתוכניות קוראות לה ביוזמתם ע"י הפקודה INT 16h - על מנת לקבל קלט מהמקלדת, בדומה ל-INT 21h. למעשה INT 21h אופציה AH=1 משתמש ב-INT 16h בכדי לבצע את המשימה שלו.

צריך להיות ברור, למשל, שה-ISR של פסיקת ה-Ctrl-Break הוא פסיקה שהטיפול בה הוא באחריות DOS, שכן מדובר בחזרה למערכת ההפעלה, ורק מערכת ההפעלה יכולה לדעת לאן חוזרים. ה-BIOS לא יכול לדעת. כנ"ל לגבי Divide Overflow.

לעומת זאת, ה-ISR-ים של פסיקות Keyboard, Timer וכו' יכולות להיות BIOS, כי מדובר בגישה להתקני חומרה, שהם חלק מהמחשב בלי קשר לשאלה, איזה מערכת

הפעלה מותקנת.

שימוש עקיף של ISR-ים של ה-BIOS

לפעמים מערכת הפעלה עומדת בפני דילמה: היא צריכה לקחת לעצמה את הטיפול בפסיקה מסוימת (פסיקת חומרה בדרך כלל) אך עדיין מעוניינת להסתמך על ה-ISR של ה-BIOS בכדי לתקשר עם החומרה. דרך אפשרית להתמודד עם הדילמה הזו היא לקרוא ל-ISR ה-BIOS מתוך ה-ISR החדש. הדרך הפשוטה ביותר לעשות זאת היא ע"י שימוש ב-CALL של פוינטר לפונקציה מסוג FAR. נראה דוגמא לכך בהמשך. באופן כללי השימוש באמצעי הזה נראה כך:

```
PUSHF
CALL משתנה 32 ביט
```

כאשר המשתנה 32 ביט מכיל את הכתובת המלאה של ה-ISR של ה-BIOS.

מימוש רוטינות טיפול בפסיקה ISR

בכל הקשור לכתיבת פסיקות תוכנה, אין הרבה הבדל בין פסיקת תוכנה לפרוצדורה. ההבדל היחיד המתחייב מהעובדה שמדובר בפסיקת תוכנה הוא, שהחזרה היא דרך הפקודה IRET (ולא RET).

בכתיבת פסיקת חומרה המצב שונה מאד. פסיקת חומרה מפסיקה (בדרך כלל) תוכנית שאינה קשורה לפסיקה. זה בודאי יכול לקרות לכל ISR חומרה. לכן ה-ISR צריך לדאוג לכך שהתוכנית המופסקת לא תושפע ע"י הפסיקה - התנהגותה צריכה להיות זהה למקרה שבו לא היתה פסיקה. זה כמובן למעט מקרים נדירים. בפועל הקריטריון הזה מתבטא, בראש ובראשונה, בשימור ערכים של כל האוגרים כולל אוגר הדגלים. זאת משום שהתוכנית המופסקת מסתמכת על ערכי האוגרים (מפקודת מכונה אחת לשניה) ואין אפשרות להמנע מכך. מאחר ו-IP, CS, ואוגר הדגלים נשמרים במחסנית בזמן הפסיקה עצמה, ה-IRET הוא זה שמשחזר אותם יחד, בפועל, עם אוגר ה-SP. יתר האוגרים חייבים להשמר ולהשתחזר ע"י צמדים של פקודות PUSH ו-POP בתחילת הטיפול בפסיקה ובסופה.

Address					
SEG:OFF	20-bit	Type	Purpose	Description	
00E3:3072	03EA2	Intel	Divide Overflow	This interrupt occurs when a divide overflow takes place. The interrupt vector varies with the DOS version.	
0600:08ED	068ED	Intel	Single-step	This interrupt simulates single-step execution. IBM uses the Trace command in DEBUG to accomplish this task.	
F000:E2C3 (F000:F85F - XT)	FE2C3	BIOS	Non-Maskable Interrupt	This interrupt cannot be prevented. It calls the BIOS NEAR procedure NMI_INT and results from memory errors on the system board (PARITY CHECK 1) or add-on boards (PARITY CHECK 2).	
0600:08E6	068E6	Intel	Set Breakpoint	This interrupt stops the processing at a particular address.	
0070:0147	00847	Intel	Interrupt If Overflow	This interrupt activates an INTO instruction return (IRET).	
F000:FF54	FFF54	BIOS	Print Screen	This interrupt prints the screen under program control. The FAR procedure called is PRINT_SCREEN and the address 0050:0000 contains the status.	
—	—	—	—	Not used.	
—	—	—	—	Not used.	
F000:FEA5	FFEAS	BIOS	Timer Interrupt	This routine handles the timer interrupt from channel 0 of the 8253 timer. There are approximately 18.2 interrupts/second. This handler maintains a count of the number of times it was called since power up. The FAR procedure is TIMER_INT and it calls an interrupt 1C H, in turn, which can contain a user routine.	
F000:E987	FE987	BIOS	Keyboard Interrupt	This routine is FAR procedure KB_INT. It continues to address F000:EC32 and constitutes the keyboard interrupt. INT 16H is the keyboard I/O routine and is much more flexible.	
—	—	—	—	Not used.	
—	—	—	—	Not used.	
—	—	—	—	Not used.	
—	—	—	—	Not used.	
F000:EF57	FEF57	BIOS	Floppy Diskette	This FAR procedure, DISK_INT, handles the diskette interrupt.	
0070:0147	00847	DOS	INTO	This interrupt activates the same call as TYPE 4.	
F000:F063	FF063	BIOS	Video Parameters	This set of routines contained in the NEAR procedure VIDEO_INT provides the CRT interface. The use of this interrupt involves many options which are discussed in a subsequent table.	
F000:F84D	FF84D	BIOS	Equipment	This procedure looks for the number of printers, any game I/O, the number of RS-232C cards, number of diskette drives, video mode, and RAM size.	
F000:F841	FF841	BIOS	Memory Size	Determines the memory size from data.	
F000:EC59 (C800:0256 - XT)	FEC59	BIOS	Diskette I/O	This procedure calls a series of routines that accomplish diskette I/O. Since a number of parameters are involved, this routine will be discussed separately.	
F000:E739	FE739	BIOS	Communications Adapter	This procedure lets the user input/output data from the RS-232C communications port.	
F000:F859	FF859	BIOS	Cassette I/O	This interrupt is used to control cassette I/O.	
F000:E82E	FE82E	BIOS	Keyboard I/O	This interrupt manipulates AX to read the keyboard. It will be discussed separately.	
F000:FEFD2	FEFD2	BIOS	Printer I/O	This routine provides communication with the printer. It uses the AX and DX registers to setup parameters. We will discuss it later.	
F600:000	F6000	BIOS	Cassette BASIC	This interrupt calls cassette BASIC.	
F000:E6F2	FE6F2	BIOS	Bootstrap	Track 0 sector 1 of Drive A is read into the boot location. Control is transferred there.	
F000:FE6E	FFE6E	BIOS	Time-Of-Day	This routine allows the clock to be set/read. CX contains the high portion of the count and DX the low portion.	
0070:0140	00840	DOS	Ctrl-Break	This interrupt results when a keyboard interrupt is used.	
F000:FF53	FFF53	BIOS	Dummy Return	This interrupt simply calls an IRET instruction.	
F000:F0A4	FF0A4	BIOS	Video Parameters	This is simply a table of byte values and routines for setting up various graphics parameters.	
0000:0522	00522	DOS	Floppy Table		
00E3:0B07	01937	DOS	Graphics Table	Used with DOS 3.0	
PSP:0000	—	DOS	Program Terminate	This interrupt is issued by DOS to exit from a program. It is the first address in the Program Segment Prefix area.	
relocatable	—	DOS	Function Request	This interrupt has many options and will be discussed later.	
PSP:000A	—	DOS	Terminate Address	Control transfers to the address specified at this interrupt location when the program terminates. Do not issue this address directly.	
PSP:000E	—	DOS	Ctrl-Break Exit Address	This interrupt is issued in response to a Ctrl-Break from the standard input.	

(Continued)

(Continued)

TABLE 5.2 (Continued)

Int.	Address		20-bit	Type	Purpose	Description
	SEG:OFF					
24	PSP:0012	—	DOS	Critical Error Handler Vector	This interrupt is called when a critical error occurs within DOS such as a disk error.	
25	relocatable	—	DOS	Absolute Disk Read	This interrupt transfers control to the device driver for a read.	
26	relocatable	—	DOS	Absolute Disk Write	This interrupt transfers control to the device driver for a write.	
27	relocatable	—	DOS	Terminate But Stay Resident	This vector is used by programs to remain resident after DOS regains control.	
28-2E		RESERVED FOR DOS				
2F	relocatable	—	DOS	Multiplex Interrupt	This interrupt defines a general interface between 2 processes. Each handler is assigned a specific number in AH and the function of the handler in AL.	
30-3F		RESERVED FOR DOS				

Table 5.2 is rather extensive and contains most of the interrupt vectors for the IBM PC and XT [2]. Most of these interrupts will not be useful from a programming viewpoint; however, the interrupts indicated in the description as affecting such things as I/O can make the programmer's job much simpler. Three types of interrupts are discussed: Intel, BIOS, and DOS. The Intel interrupts are common to most Intel processors in the 8086 family. The BIOS interrupts are specific to the IBM Basic Input Output System and the DOS interrupts are, of course, specific to the IBM Disk Operating System [3]. The BIOS interrupts are explained in the BIOS program listing discussed above, and the DOS interrupts are described in the *DOS Technical Reference manual* [4].

5.1.1 Interrupts 0-0FH

Interrupts 0-4 are not particularly useful to the assembler programmer because they tend to call system oriented functions. INT 5 can be used to print the contents of the screen Cathode Ray Tube (CRT) under program control. If, for example, a program generates output to the screen, it might be desirable to automatically save this output and INT 5 is useful for this purpose. INT 8 is the Timer Interrupt and returns a time count in locations 0040:006C and 0040:006E where these are the low and high count, respectively. This 32-bit time count can be used to time events in the program. INT 9 activates when keys are pressed or released and simply stops the processing. This interrupt is more of a system interrupt. The INT 16H keyboard I/O interrupt will be discussed in detail rather than focusing on INT 9. Both address the keyboard operation; however, INT 16H has considerable flexibility over INT 9. Interrupts 0AH to 0FH tend to be either unused at present or system oriented and will not be discussed further. INT 10H, however, is the video I/O routine already mentioned. This interrupt has many options as discussed in the program listing in the *IBM Technical Reference manual*. Essentially all screen I/O under program control can be implemented using this interrupt. The following discussion addresses this interrupt in some depth.

5.1.2 Interrupt 10H: Video I/O

The type 10H interrupt has 16 basic options controlled by the AH register at the time the interrupt is called. Table 5.3 describes these options and the registers that must be set to control output. The mode option (AH = 0) determines the screen resolution and what graphics are available. To handle the increased memory requirements for the graphics modes, additional video memory must be provided and is usually supplied with the graphics card purchased. The following discussion illustrates the use of this video interrupt and how to achieve screen graphics.

All standard screen modes involve the use of a raster scan which simply means the electron beam moves vertically downward after scanning across for each line from left to right. The resolution of a dot in one of the graphics modes is determined by the number of lines vertically down the screen and by the number of columns horizontally across the screen. In the 640 × 200 graphics mode, for example, the height of the screen can be thought of as 200 dots high. Similarly, the screen width would be 640

תוכניות דוגמא c_intrld.c, c_intr2d, c_intr3d.c

התוכניות הללו הן תוכניות C, להמחשה של אספקטים של נושא הפסיקות.

התוכנית c_intrld.c

התוכנית c_intrld.c היא תוכנית Turbo C טהורה, להמחשה ראשונית של נושא הפסיקות. ההדגשה היא "תוכנית Turbo C" ולא סתם "תוכנית C" משום שהתוכנית הזו תתקמפל רק ע"י קומפילר של C עבור DOS (ביחוד Turbo C ו-Microsoft C) משום שהיא מישמת רעיון שבכל מערכות ההפעלה של המחשב האישי שבאו אחרי DOS אינה נתמכת יותר: שתוכנית אפליקציה משתלטת על פסיקה. תחת DOS הדבר היה אפשרי והרבה תוכניות אפליקציה עשו זאת, ותוכניות האמולציה (הדמיה) של DOS בדרך כלל משתדלות לתמוך בהדמיה כזו או אחרת של הטכניקה הזו. בגלל זה ניתן להריץ את התוכניות הללו בחלונות ה-command-וה-cmd של WINDOWS אם כי ישנם תוכניות דומות (למשל כזו המסתמכת על מקש ה-Print-Screen) שיעבדו אך ורק תחת DOS האמיתי.

כפי שנאמר, תפקידה של c_intrld.c לשמש המחשה ראשונית של מוסג הפסיקה. שימו לב למבנה הבא של התוכנית:

```
volatile int Ctrl_Break_Flag;
```

```
....
```

```
void main()
```

```
{
```

```
....
```

```
Ctrl_Break_Flag = 0;
```

```
while(Ctrl_Break_Flag == 0)
```

```
;
```

```
....
```

```
} /* main */
```

על משמעות מילת המפתח volatile נרחיב בהמשך. בשלב זה נתעלם ממנו. על פי הכללים של שפת C, Ctrl_Break_Flag הינו משתנה גלובלי שלם. במהלך הריצה של התוכנית הראשית, אנחנו מציבים לתוכו אפס ומשם נכנסים לתוך לולאה while שאינה עושה דבר למעט לבדוק אם Ctrl_Break_Flag עדיין שווה לאפס ואם כן לחזור על הלולאה. לכאורה לולאת ה-while היא בחזקת לולאה אינסופית ומכאן סתם, שכן היא תלויה בערכו של משתנה שאותו אין היא משנה תוך כדי ביצוע הלולאה. ואכן הרצת התוכנית תתקע את המחשב (במקרה של DOS) או את חלון האמולציה עד שהמשתמש ילחץ על מקש ה-Ctrl-Break.

מה שצריך להיות ברור הוא שסיום לולאת ה-while כרוך בהפרת העקרון שבו כל פקודת מכונה שמבצע המחשב נקבע ע"י פקודת המכונה שלפניה. אנחנו רואים לפי הריצה של התוכנית שזה אכן מה שקורה. הדרך שאנחנו עושים זאת היא על ידי השתלטות על פסיקת מספר 27 (1Bh בהקסה), פסיקה שמתרחשת בכל רגע שנלחץ המקש Ctrl-Break. בפקודה

```
setvect(27, Ctrl_Break_Handler);
```

קבענו את הרטינה המיוחדת Ctrl_Break_Handler בתור רטינת הטיפול בפסיקה של פסיקה 27. מרגע זה ואילך הרטינה הזו היא שתבצע עם כל לחיצה של המקש Ctrl-Break ולא זו הרגילה (הגורמת בתנאים מסוימים חזרה ל-DOS). מאחר ואחד הפקודות של הרטינה הזו היא

```
Ctrl_Break_Flag = 1;
```

בפעם הראשונה שהרטינה תבצע הערך של המשתנה הגלובלי Ctrl_Break_Flag ישתנה ועם החזרה מפסיקה התוכנית אכן תשתחרר מלולאת ה-while. setvect היא רטינה פנימית של Turbo C המאפשרת השתלטות על פסיקה נתונה, תוך ציון הרטינה החלופית לפסיקה.

מהדפסות הנראות בפלט

```
C: Ctrl-Break has been pressed.
```

```
C: Terminating ...
```

צריך להיות ברור שלחיצת Ctrl-Break אכן גרמה הסתעפות לרוטינה
Ctrl_Break_Handler ושעם סיומו חזר לתוכנית הראשית ומשם לסיום.

נקודות נוספות שצריך לשם לב עליהם:

- שינוי טיפול בפסיקה היא פעולה שאינה משתחזרת או מתבטלת עם סיום
התוכנית. לכן בכדי שהמערכת תמשיך לתפקד עם סיום התוכנית הראשית, יש
לשמר את הטיפול המקורי בפסיקה ולשחזר אותה לפני סיום. לשם כך קיימת
הקריאה ל-getvect בפקודה

```
Int27Save = getvect(27);
```

getvect היא רוטינה פנימית של Turbo C המחזיר את כתובת הרוטינה
הנוכחית שמטפלת בפסיקה שמספרה היא מקבלת כפרמטר. היא כמוכן אינה משנה
אותו. Int27Save הינו משתנה 32 ביט בזכרון שעוד נתיחס איך מגדירים
אותו, אבל בסופו של חשבון הוא פשוט 32 ביט המאחסנים את התוכן הכניסה.
התוכנית הראשית פשוט משמרת את כתובת המטפל הנוכחי לפני הקריאה ל-
setvect ולפני הסיום משחזרת אותו ע"י הפקודה

```
setvect(27, Int27Save);
```

- בשביל לכתוב רוטינה שתטפל בפסיקה, היא חייבת להיות מוגדרת
כפונקציה void interrupt עם רשימת פרמטרים void. לפיכך ההגדרה של
Ctrl_Break_Handler הוא

```
void interrupt Ctrl_Break_Handler(void);
```

המילה interrupt היא מילה שמורה של Turbo C. משמעות ההגדרה הזו
שהרוטינה משמרת את ערכי כל האוגרים ומסימת את ריצתה ע"י פקודת המכונה
.IRET

- ההגדרה של Int27Save

```
void interrupt (*Int27Save)(void);
```

הינו הכרזה ש- Int27Save הוא פוינטר לפונציה מסוג void interrupt.
מלבד הקצאת שטח מתאים (32 ביט) Turbo C יסכים לקמפל קריאה ל-setvect
ו-getvect רק למשתנים מהסוג הזה.

- המילה volatile המופיעה בהכרזה

```
volatile int Ctrl_Break_Flag;
```

מנחה את הקומפילר שהמשתנה הזה עשוי לקבל ערכים באופן לא צפוי ולכן
יש לגשת לזכרון כל אימת שצריך לעמוד על ערכו. הדבר נחוץ כי אחרת
הקומפילר עשוי לעשות אופטימיזציה נוסף קריאת הערך פעם אחת לתוך אוגר
ולסמוך על כך שכיוון שהתוכנית לא שינתה את המשתנה, אפשר לסמוך על
הערך שנמצא באוגר ולחסוך בכך גישה לזכרון. המילה volatile היא מילה
שמורה של שפת C הסטנדרטית והיא רלוונטית לא רק כאן.

- פקודת הקימפל של התוכנית

```
tcc -ml -r- c_intrld.c
```

מנחה את הקומפילר לקמפל את התוכנית

- א. במודל Large ע"י ההנחיה -ml
- ב. ללא שימוש במשתני אוגר ע"י ההנחיה -r-.

עבור תוכניות המשתלטות על פסיקות בצורה הזו הדבר הכרחי ממספר
סיבות, שלא ניכנס להם כאן.

c_intr2d.c, c_intr3d.c התוכניות

התוכניות הללו הן גירסאות של c_intrld.c רק שהרטינות getvect ו-
setvect מוחלפים במימושים באסמבלי. הקוד באסמבלי ממומש כאן בשיטה של
inline assembly כלומר באופציה של כתיבת קוד אסמבלי בתוך תוכנית C.
הדבר נעשה משיקולים של נוחות של תצוגה, השימוש בטכניקה הזו היא לא
דווקא קשור לפסיקות. בשיטה הזו ממשים קוד באסמבלי בלוקים מהסוג


```
asm {
    קוד אסמבלי
}
```

או בשורות קוד מהסוג

פקודת אסמבלי אחת asm

באסמבלי מהסוג הזה יש מגבלות (למשל אין תמיכה בפקודות הסתעפות מותנות). ההערות לפי המוסכמות של שפת C.

התוכנית c_intr2d.c

- בתוכנית c_intr2d.c הפעולות getvect ו-setvect נעשות ע"י גישה ישירה לזכרון לפי כתובת, כאשר התוכנית מציבה 0 לתוך ה-ES על מנת שתצביע על השטח שבו נמצאת טבלת ה-IV ובהיסטים קבועים יחסית אליו. למשל הפקודה

```
MOV AX,ES:[27*4]
```

הינו קריאה של שדה ההיסט של הכניסה מספר 27 ב-IV. הכפל הזה נעשה ע"י האסמבלר בזמן אסמבלי כלומר רק לקבועים ניתן לבצע פעולות אריתמטיות (לא ניתן לכתוב 5*BX בתוך הסוגריים המרובעים, למשל). באופן שקול ניתן היה לכתוב

```
MOV AX,ES:[108]
```

- המימושים של setvect כרוכים ביותר מפקודת מכונה אחת ולכן מקדים אותם הפקודה CLI ולאחריהם פקודת המכונה STI. הדבר נעשה על מנת למנוע תקלה כתוצאה מטיפול בבדיקת הפסיקה הזו לאחר שהחל השינוי של הכניסה ב-IV ולפני ההשלמה שלו. כמובן שהסכוי שתקלה כזו תתרחש היא אפסית, אך תמיד חייבים להתיחס לזה. למשל במימוש של

```
setvect(27,Ctrl_Break_Handler)
```

```
CLI
MOV WORD PTR ES:[27*4], OFFSET Ctrl_Break_Handler
MOV WORD PTR ES:[27*4+2], SEG Ctrl_Break_Handler
STI
```

אילו לא ביצענו את ה-CLI ואם יתרחש פסיקה מספר 27 בדיוק לאחר ה-MOV הראשון ולפני הביצוע ה-MOV השני ה-CPU יסתעף לכתובת של מספר סגמנט לפי רוטינת הטיפול המקורי ואילו ההיסט לפי רוטינת הטיפול החדש. בקיצור לא פה ולא שם. העקרון הכללי הוא פשוט למנוע טיפול בפסיקות בזמן שמבנה הנתונים של ה-IV התחיל לעבור שינויים והשינויים הללו עוד לא הושלמו. מובן שאם היינו שוכחים לכתוב את CLI ו-STI סביר להניח שהשגיאה הזו לא מורגשת או באה לידי ביטוי. אך באופן כללי, חייבים להקפיד על עקרונות כאלו.

- האופרטור SEG נותנת אפקט דומה לאופרטור OFFSET רק שכאן זה קבוע של מספר סגמנט. הבדל אחר הוא שלא כל כך קל לממש אותו שכן מספר הסגמנט נקבע בזמן ריצה. יש לזה פתרון שלא נדון בו כאן.

- המשתנה Int27Save מוגדר כאן כ-long int על מנת להדגיש את העובדה שבכל התוכניות Int27Save הוא בסך הכל משתנה 32 ביט.

התוכנית c_intr3d.c

זוהי תוכנית כמעט זהה ל-c_intr2d.c רק ש-setvect ו-getvect ממומשים ע"י רוטיות DOS:

getvect ממומש ע"י

INT 21h, AH = 35h

"החזר ב-ES:BX את תוכן הכניסה ב-IV שמספרה מועבר אליך ב-AL".

setvect ממומש ע"י

INT 21h, AH = 25h

"הצב לכניסה ב-IV שמספרה מועבר אליך ב-AL את התוכן המועבר אליך ב-DS:DX".

```
/* c_intrld.c - Change Ctrl-Break interrupt handler.  
Pure C version.  */
```

```
#include <stdio.h>  
#include <dos.h>
```

```
volatile int Ctrl_Break_Flag;
```

```
void interrupt (*Int27Save) (void); /* Pointer to function */
```

```
void interrupt Ctrl_Break_Handler(void)  
{  
    Ctrl_Break_Flag = 1;  
    printf("C: Ctrl-Break has been pressed.\n");  
} /* Ctrl_Break_Handler */
```

```
void main(void)  
{  
    Int27Save = getvect(27); /* Preserve old pointer */  
    setvect(27, Ctrl_Break_Handler); /* Set entry to new handler */  
    printf("C: Press Ctrl-Break to continue.\n");  
  
    Ctrl_Break_Flag = 0;  
  
    while (Ctrl_Break_Flag == 0)  
        ; /* Do nothing */  
  
    printf("C: Terminating ...\n");  
  
    setvect(27, Int27Save); /* Restore old pointer */  
  
} /* main */
```

```
E:\users\eytan\asm>tcc -ml -r- c_intrld.c  
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International  
c_intrld.c:  
Turbo Link Version 5.0 Copyright (c) 1992 Borland International
```

```
Available memory 4112144
```

```
E:\>c_intrld.exe  
C: Press Ctrl-Break to continue.  
C: Ctrl-Break has been pressed.  
C: Terminating ...
```

```
E:\>
```

```

/* c_intr2d.c - Change Ctrl-Break interrupt handler.
   C + ASM version.   */

#include <stdio.h>
#include <dos.h>

volatile int Ctrl_Break_Flag;
long int Int27Save; /* 32bit used to hold old ISR address */

void interrupt Ctrl_Break_Handler(void)
{
    Ctrl_Break_Flag = 1;
    printf("C+InASM 1: Ctrl-Break has been pressed.\n");
}

void main(void)
{
    /* Int27Save = getvect(27);    Preserve old pointer */
    asm {
        PUSH ES          /* Preserve registers */
        PUSH AX
        /* Preserve pointer of old interrupt handler */
        MOV AX,0          /* Set ES to point to Interrupt vector segment */
        MOV ES,AX
        MOV AX,ES:[27*4]   /* Read offset entry of Ctrl-Break ... */
        MOV WORD PTR Int27Save,AX /* ... interrupt handler */
        MOV AX,ES:[27*4+2] /* Read segment entry of Ctrl-Break ... */
        MOV WORD PTR Int27Save+2,AX /* ... interrupt handler */

    /* setvect(27,Ctrl_Break_Handler);    Set new interrupt handler */
    CLI          /* Disable interrupts while IV is being changed */
    /* Set offset of new interrupt handler */
    MOV WORD PTR ES:[27*4],OFFSET Ctrl_Break_Handler
    /* Set segment number of new interrupt handler */
    MOV WORD PTR ES:[27*4+2],SEG Ctrl_Break_Handler
    STI          /* Re-enable interrupts (modification finished) */
    POP AX       /* Restore registers */
    POP ES
    }

    printf("C+InASM 1: Press Ctrl-Break to continue.\n");

    Ctrl_Break_Flag = 0;

    while (!Ctrl_Break_Flag)
        ; /* Do nothing */

    /* setvect(27,Int27Save);    Restore old pointer */
    asm {
        PUSH ES          /* Preserve registers */
        PUSH AX
        MOV AX,0          /* Set ES to point to Interrupt vector segment */
        MOV ES,AX
        /* Restore old interrupt handler */
        CLI          /* Disable interrupts while IV is being changed */
        MOV AX,WORD PTR Int27Save /* Restore the offset ... */
        MOV ES:[27*4],AX /* ... of old interrupt handler */
        MOV AX,WORD PTR Int27Save+2 /* Restore the segment number ... */
        MOV ES:[27*4+2],AX /* ... of old interrupt handler */
        STI          /* Re-enable interrupts (modification finished) */
        POP AX       /* Restore registers */
        POP ES
    }
} /* main */

```

```

E:\>c_intr2d.exe
C+InASM 1: Press Ctrl-Break to continue.
C+InASM 1: Ctrl-Break has been pressed.

E:\>

```

```

/* c_intr3d.c - Change Ctrl-Break interrupt handler.
   C + ASM version, use DOS INT 21h AH=35h, 25h.   */

#include <stdio.h>
#include <dos.h>

int Ctrl_Break_Flag;
long int Int27Save; /* 32bit used to hold old ISR address */

void interrupt Ctrl_Break_Handler(void)
{
    Ctrl_Break_Flag = 1;
    printf("C+InASM 2: Ctrl-Break has been pressed.\n");
}

void main(void)
{
    /* Int27Save = getvect(27);    Preserve old pointer */
    asm {
        PUSH DS          /* Preserve registers */
        PUSH ES
        PUSH AX
        PUSH BX

        MOV AH,35h        /* Set DOS interrupt handler */
        MOV AL,27          /* Set interrupt vector entry number */
        INT 21h           /* Invoke DOS */
        /* Preserve offset of old interrupt handler */
        MOV WORD PTR Int27Save,BX
        MOV AX,ES          /* Preserve segment number of old ... */
        MOV WORD PTR Int27Save+2,AX /* ... interrupt handler */

        /* setvect(27, Ctrl_Break_Handler); Set new interrupt handler */
        /* Set offset of new interrupt handler */
        MOV DX,OFFSET Ctrl_Break_Handler
        MOV AX,SEG Ctrl_Break_Handler /* Set segment number of new ... */
        MOV DS,AX             /* ... interrupt handler */
        MOV AH,25h           /* Set DOS interrupt handler */
        MOV AL,27            /* Set interrupt vector entry number */
        INT 21h              /* Invoke DOS */

        POP BX               /* Restore registers */
        POP AX
        POP ES
        POP DS
    }

    printf("C+InASM 2: Press Ctrl-Break to continue.\n");

    Ctrl_Break_Flag = 0;
    while (!Ctrl_Break_Flag)
        ; /* Do nothing */

    /* setvect(27,Int27Save);    Restore old pointer */
    asm {
        PUSH DS          /* Preserve registers */
        PUSH AX

        /* Restore number old interrupt handler */
        MOV DX,WORD PTR Int27Save
        MOV AX,WORD PTR Int27Save+2 /* Restore the number ... */
        MOV DS,AX             /* ... of old interrupt handler */
        MOV AH,25h           /* Set DOS interrupt handler */
        MOV AL,27            /* Set interrupt vector entry number */
        INT 21h              /* Invoke DOS */
        POP AX               /* Restore registers */
        POP DS
    }
} /* main */

```

```

E:\>c_intr3d.exe
C+InASM 2: Press Ctrl-Break to continue.
C+InASM 2: Ctrl-Break has been pressed.

E:\>

```

תוכניות דוגמא mainint3.c, ctrlbrhl.asm

התוכנית mainint3.c כמעט זהה ל-c_intrld.c, למעט העובדה שהרוטינה Ctrl_Break_Handler אינה ממומשת שם אלא מוגדרת כ-extern ומצפה שהיא תהיה ממומשת בפועל בקובץ אחר. בפועל היא ממומשת פרוצדורה בשפת אסמבלי טהורה בקובץ ctrlbrhl.asm. לפיכך מה שאנחנו רואים כאן להבדיל מ-c_intrld.c היא השורה

```
extern void interrupt Ctrl_Break_Handler(void);
```

שורת הקימפול של התוכנית המשלובת היא

```
tcc -ml -r- mainint3.c ctrlbrhl.asm
```

והתוכן של הקובץ ctrlbrhl.asm אשר לזה האחרון כדאי לשים לב לעובדה שהחזרה של הרוטינה ממומשת ע"י פקודת המכונה IRET, הרוטינה משמשת/משחזרת את כל האגרים בפקודות

```
PUSH AX
```

```
PUSH BX
```

```
...
```

```
PUSH BP
```

ומשחזרת אותם בפקודות

```
POP BP
```

```
POP DI
```

```
...
```

```
POP AX
```

מכיוון שהתוכנית מתקמפלת במודל LARGE, הקריאה ל-printf מחיבת דחיפה למחסנית של פוינטר מלא (גם מספר סגמנט וגם היסט) הממומשים בפקודות

```
MOV AX,SEG printf_str
```

```
....  
PUSH AX  
PUSH OFFSET printf_str
```

בנוסף ישנה פקודה

```
MOV DS,AX
```

בכדי להבטיח שה-printf של מודל LARGE יהיה לו את הערך שהוא מצפה לו. זאת מאחר שמדובר ברוטינת טיפול בפסיקה איננו יכולים לסמוך על ערכי העוגרים, שכן איננו יודעים בודאות מהכן הגיע ה-CPU לכאן.

```

/* mainint3.c - Main for pure assembler interrupt handler */

#include <stdio.h>
#include <dos.h>

volatile int Ctrl_Break_Flag;
void interrupt (*Int27Save) (void); /* Pointer to function */

extern void interrupt Ctrl_Break_Handler(void);

void main(void)
{
    Int27Save = getvect(27);          /* Preserve old pointer */
    setvect(27, Ctrl_Break_Handler);
    printf("C: Press Ctrl-Break to continue.\n");

    Ctrl_Break_Flag = 0;

    while (Ctrl_Break_Flag == 0)
        ; /* Do nothing */

    printf("Terminating ...\n");

    setvect(27, Int27Save);          /* Restore old pointer */
} /* main */

```

```

E:\>tcc -ml -r- mainint3.c ctrlbrh1.asm
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
mainint3.c:
ctrlbrh1.asm:
Turbo Assembler Version 3.1 Copyright (c) 1988, 1992 Borland International

```

```

Assembling file:  ctrlbrh1.ASM
Error messages:   None
Warning messages: None
Passes:           1
Remaining memory: 418k

```

```

Turbo Link Version 5.0 Copyright (c) 1992 Borland International

```

```

    Available memory 4112112

```

```

E:\>mainint3.exe
C: Press Ctrl-Break to continue.
ASM: Ctrl-Break has been pressed
Terminating ...

```

```

E:\USERS\EYTAN\ASM>

```



```

; ctrlbrh1.asm - pure assembler Ctrl-Break handler.
;

.MODEL LARGE
PUBLIC _Ctrl_Break_Handler
EXTRN _Ctrl_Break_Flag:WORD
EXTRN _printf:FAR

;
; void interrupt Ctrl_Break_Handler(void)
;

.DATA
printf_str DB 'ASM: Ctrl-Break has been pressed',10,0
.CODE
_Ctrl_Break_Handler PROC FAR
    PUSH    AX                ; Preseve all registers ...
    PUSH    BX                ; ... other than FLAGS and SP.
    PUSH    CX
    PUSH    DX
    PUSH    ES
    PUSH    DS
    PUSH    SI
    PUSH    DI
    PUSH    BP
    MOV     AX,SEG _Ctrl_Break_Flag    ; Set ES to segment ...
    MOV     ES,AX                    ; ... of Ctrl_Break_Flag
;
; {
;     Ctrl_Break_Flag = 1;
;
    MOV     ES:[_Ctrl_Break_Flag],1    ; Set global flag to 1
;
;     printf("ASM: Ctrl-Break has been pressed\n");
;
    MOV     AX,SEG printf_str          ; Set DS to segment ...
    MOV     DS,AX                      ; ... of printf_str
    PUSH    DS                          ; Store complete address
    PUSH    OFFSET printf_str           ; ... of printf_str
    CALL    _printf
    ADD     SP,4                        ; Free parameter space
;
; }
;

    POP     BP                ; Restore all registers ...
    POP     DI                ; ... other than FLAGS and SP.
    POP     SI
    POP     DS
    POP     ES
    POP     DX
    POP     CX
    POP     BX
    POP     AX
    IRET                        ; Return from interrupt
_Ctrl_Break_Handler ENDP
END

```