## פסיקת המקלדת INT 16h

בפסיקה מספר 16h הינה פסיקת התוכנה של המקלדת. תוכניות שמעוניינות באינפורמציה מהמקלדת פונות אליה, בדרך כלל בעקיפין דרך מערכת ההפעלה והיא כאילו התוכנית הסטנדרטית לקרוא מידע מהמקלדת. זוהי פסיקת תוכנה של ה-BIOS ובעיקרו של דבר היא מהווה מעין קוד המכיר את מבנה הנתונים של ה-BIOS ומתפקדת התאם.

בשטח זיכרון מיוחד בהתחלה של הזיכרון של המחשב (מיד אחרי השטח של IV, כלומר מכתובת 1024 ואילך) ישנו שטח זיכרון של ה-BIOS שמכיל בין השאר שטח המשקף את מצב המקלדת, איזה מקשים נלחצו וטרם נקראו ע"י שום תוכנית, מצב הנורות (Caps Lock, Insert, Scroll, Num), מצב מקשי הסטטוס (ה-Shift, Alt, Ctrl) וכו'. למשל בבתים בכתובות מוחלטות 417h (1041) ו- 418h (1042) ישנם בתים המהווים משתני דגלים של מצב הנורות ומקשי הסטטוס. נוסף לכך יש שם, חוצץ לשמירת מידע על עד ל-20 לחיצות מקשים שלא נקראו עדיין (מעבר לכך התגובה ללחיצות נוספות יהיה צפצוף). השטח הזה מתוחזק ומעודכן בעיקר ע"י ה-ISR של פסיקה מספר 9, פסיקת החומרה של המקלדת. המידע שנשמר שם הוא על לחיצות על (כמעט) כל המקשים של המקלדת, כולל המקשים F1 - F12, Page Down, Esc, Ins, Del ... וכו'.

כאשר תוכנית מבקשת מ-INT 16h אינפורמציה על לחיצת מקש, האיפורמציה מגיעה בשתי צורות: קוד Ascii וקוד Scan. קוד ה-Ascii הוא בדרך כלל מה שאנחנו צריכים, למשל אם נלחץ על המקש המסומן A נקבל או את הקוד Ascii 'a' או 'A' בהתאם למצב נורת ה-Caps Lock ומקשי ה-Shift וכו'. קוד ה-Scan הוא מספר יחודי לכל מקש, בתחום 1 - 127. צריך לזכור שיש מקשים שאין להם קוד Ascii (כמו F1, Esc, Page Down) ויש קודי Ascii שיש להם יותר ממקש אחד (למשל הספרות העשרוניות, הסימנים +, -, *, /).

ל- INT 16h אופציות רבות, הנבחרות לפי הערך של האוגר AH ברגע הקריאה, השימושיות ביותר מבחינתינו הם:

INT 16h, AH = 0, קריאת מקש:

קריאה ל-INT 16h כאשר AH = 0 פירושו בקשה לקבלת מידע על לחיצת מקש של המשתמש. אם אין מידע מסוג זה בהמתנה, החזרה מהקריאה תתעקב עד שחיצה כזו תתבצע (כלומר התוכנית <u>תעצר</u> לזמן בלתי מוגבל). עם החזרה (בין עם אחרי המתנה ובין שלא) יהיו באוגר AX האינפורמציה המבוקשת: ב-AL יהיה קוד ה-Ascii של המקש שנלחץ ואילו ב-AH יהיה קוד Scan של המקש. אותה לחיצת מקש שהמידע עליה מוחזרת לקורא ל-INT 16h נחשבת ל"נקראה" קרי "מבוטלת" כלומר שקריאה נוספת ל-INT 16h אופציה AH = 0 לא תתיחס אליה ותקרא את האינפורמציה

על הלחיצה הבאה. זה נראה מובן מאליו, אבל זה לא נכון לאופציה הבאה.

,INT 16h
1 = AH, עיון במקש:

האופציה הזו דומה לאופציה 0 = AH, אבל במספר הבדלים מהותיים. ראשית, הקריאה תמיד חוזרת מיד לקורא ל-INT 16h, גם אם אין אינפורמציה על לחיצת מקש שטרם נקראה. התוכנית הקוראת יכולה להבחין אם היתה לחיצת מקש או לא לפי הערך של הדגל ZF, 1 = ZF אם לא היתה אינפורמציה של לחיצת מקש בהמתנה לקריאה, אם היתה 0 = ZF. במידה והיתה אינפורמציה ללחיצת מקש ממתינה, היא תוחזר ב-AX ממש כמו ב-INT 16h אופציה 0 = AH, אך בניגוד למקרה הקודם הקריאה ל-INT 16h אופציה 1 = AH אינה "מבטלת" את הלחיצה. הקריאה הבאה ל-INT 16h באופציה 0 = AH או 1 = AH יקראו את אותו מקש עצמו. כלומר קריאה ל-INT 16h אופציה 1 = AH היא קריאה "לא מחייבת" של המקש, רק מעין הצצה לבדוק מה יש שם, אם בכלל. למשל, אם נרצה לבדוק אם יש אינפורמציה על לחיצת מקש ממתינה ובמידה ויש לקרוא אותה אבל מבלי להמתין ללחיצה כזו במידה ואין, אנחנו נקרא קודם ל-INT 16h אופציה 1 = AH, נבדוק ש-ZF = 0 ובמידה וזה המצב, נקרא ל-INT 16h אופציה 0 = AH.

הקוד יכול להראות כך:

```
MOV AH,1
INT 16h
JZ No_Infol
 MOV AH,0
 INT 16h
 JMP With_Infol
No_Infol:

  ....

With_Infol:
```

,INT 16h
2 = AH, סטטוס המקלדת. מחזיר ב-AL אחד משני בתי הדגלים של המקשים המיוחדים (נורות ה-Caps Lock, Num, Scroll, מצב מקשי ה-Shift, Alt, Ctrl). מידע נוסף ניתן לקרוא מכתובת מוחלטת 418h או ע"י קריאה ל-INT 16h אופציה 12h = AH.

,INT 16h

5 = AH, הדמית מקש. אפשר בתוכנית ליצור מצב שבו "כאילו" נלחץ מקש כלשהוא, כאשר האינפורמציה של לחיצת המקש אותו רוצים לדמות מועברת דרך CX: CL יהיה קוד ה-Ascii ו-CH קוד ה-Scan.

## Type 16 Interrupt Operations

The Type 16 (Keyboard I/O) interrupt calls KEYBOARD_IO, which starts at location F000:E82E. This routine lets you select from three different operations, based on a value in AH:

- If AH=0, KEYBOARD_IO reads the scan code of the next key in the buffer into AH and its character code into AL, then advances the buffer pointer. If the buffer is empty, KEYBOARD_IO waits for a key to be pressed before proceeding.
- If AH=1, KEYBOARD_IO returns the status of the keyboard buffer in the Zero Flag (ZF). If the buffer is empty, ZF is 1. If any key codes are available for reading, ZF is 0. If ZF is 0, the next available character is in AX, and the entry remains in the buffer.
- If AH=2, KEYBOARD_IO returns a keyboard status byte in AL. A description of this byte follows.

The KEYBOARD_IO routine affects only AX and the flags.

The upper half of Figure 6-6 shows the arrangement of the status byte returned by the AH=2 option. In this byte (KB_FLAG in the BIOS listing) the upper four bits tell you whether various keyboard modes are on (1) or off (0), and the lower four bits tell you whether the Alt, Ctrl, or shift key is being pressed.

The lower half of Figure 6-6 shows a companion byte to KB_FLAG that gives additional keyboard status information. The KEYBOARD_IO routine uses this second byte (KB_FLAG_1) internally but provides no way to read it into a register. However, KB_FLAG_1 follows KB_FLAG in the BIOS, so to find out how KB_FLAG_1 is configured, you read the contents of location 417BH (KB_FLAG is at 417AH).

The first KEYBOARD_IO option, AH=0, is convenient for setting up *interactive* operations with the computer. In such operations you essentially tell the computer to wait for an operator to type something at the keyboard before proceeding. Let's take a look at some possible applications.

### A Single-Key Read Operation

Suppose your program has displayed a message that instructs the operator to press either "Y" or "N" to signify Yes or No. A Y response makes the program jump to a set of instructions labeled YES, and an N makes it jump to instructions labeled NO. Any other key makes the program wait until it receives either Y or N. This sequence should do the job:

```
        STI                     ;Enable Interrupts
GET_KEY: MOV    AH,0            ;Read a key
        INT     16H
```

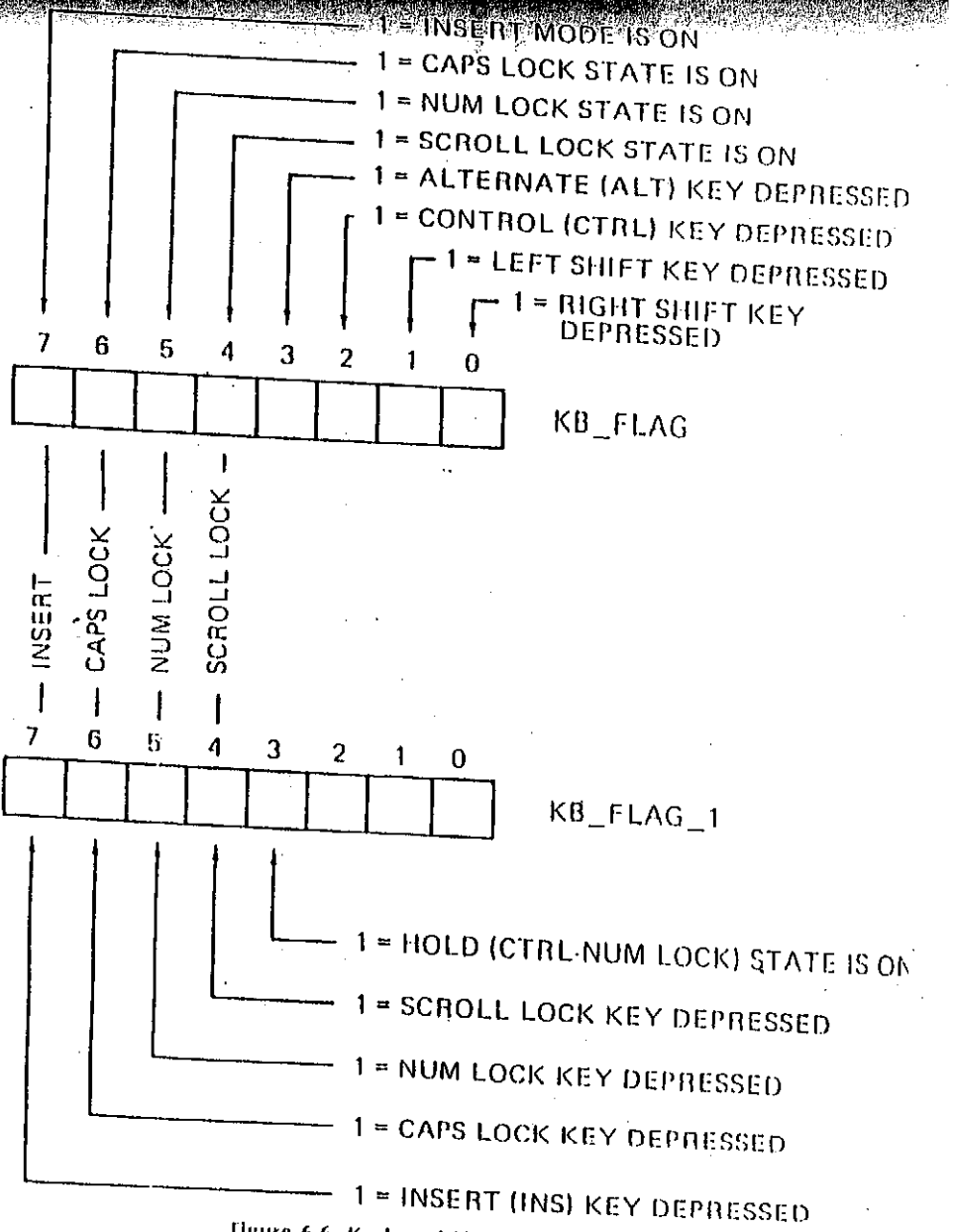


Figure 6-6. Keyboard I/O shift status bytes.

## תוכניות דוגמא dem_key1.c, dem_key5.c

התוכניות הללו תפקידן להמחיש שימוש ב-INT 16h. dem_key1.c ממחישה שימוש באופציה AH = 0 קריאת מקש מהמקלדת עם המתנה אם יש צורך בכך. dem_key5.c ממחישה שימוש באופציה AH = 2 בדיקת סטטוס המקלדת.

### התוכנית dem_key1.c

קטע ה-inline assembler

```
MOV AH,0
INT 16h
MOV scan_temp,AH
MOV inp_char,AL
```

הינו קריאת מקש (עם המתנה) והצבת ה-Scan code לתוך משתנה ה-C scan_temp והצבת ה-Ascii code לתוך משתנה ה-C inp_char. משם תוכנית ה-C מדפיסה אותם בצורה נוחה למשתמש.

התוכנית הראשית מממשת לולאה החוזרת על עצמה עד לרגע שהמשתמש לוחץ על המקש Esc שהתוכנית מזהה לפי Scan code = 1. בדוגמת הריצה הספציפית הזו נלחצו

```
'9',
'Shift-9',
'r',
'Shift-r',
F1,
F10,
Shift-F1,
Shift-F10
```

שימו לב שללחיצות עצמם <u>אין</u> השתקפות בפלט (echo) - זה אף פעם לא אוטומטי ומיושם בתוכנה ברובד זה או אחר.
מהפלט אפשר לראות שללחיצות '9' ו-'Shift-9' אותו Scan code (10) וכנ"ל ללחיצות 'r' ו-'Shift-r' (19) אך הם נבדלים ב-Ascii code ('9' לעומת ')','r' לעומת 'R'). מי שיריץ את התוכנית יראה שמספרי ה-Scan code ניתנו למקשים בעיקרו של דבר לפי מיקומם במקלדת (ביחוד לאותם מקלדות של ה-PC מראשית שנות השמונים).

למקשים שאין להם Ascii code בדרך כלל מוחזר ב-AL אפס, ונסיון להדפיס אפס מתבטא ברווח. ללחיצת Esc יש משום מה Ascii code = 27 שמשום מה מוחק תו מהפלט, בגלל זה נמחק גרש בשורה האחרונה בפלט.

למקש F1 יש את Scan code = 59 ול-F10 יש את Scan code = 68 למעשה כל המקשים F1 - F10 יש Scan code = 59 - 68 בהתאמה. לחיצת המקשים הללו ביחד עם Shift גורמת לתוספת של 25 ל-Scan code אולי בגלל שאין למקשים הללו Ascii code ולא ניתן להבדיל ביניהם באמצעותו.

```
/* dem_key1.c - demostrate use of int 16h, AH = 0  */

#include <stdio.h>

void main(void)
 {

   unsigned int scan_code;
   char scan_temp, inp_char;

   printf("\nPress ESC to exit program\n");

   do {
        printf("Press any key (almost)\n:");

        asm  {
               MOV AH,0            ; /* BIOS read char from buffer option */
               INT 16h             ; /* BIOS read char from buffer        */
               MOV scan_temp,AH    ; /* Transfer scan code to program     */
               MOV inp_char,AL     ; /* Transfer char to program          */
             }

        scan_code = (unsigned int) scan_temp;

        if (scan_code == 1)
            printf("You pressed ESC, \n");

        printf("You pressed key assigned"
              " scan code = %d, char_value= '%c'\n",
                                  scan_code,  inp_char);


    } while(!(scan_code == 1));


 } /* main */
```

---

```
E:\>dem_key1

Press ESC to exit program
Press any key (almost)
:You pressed key assigned scan code = 10, char_value= '9'
Press any key (almost)
:You pressed key assigned scan code = 10, char_value= '('
Press any key (almost)
:You pressed key assigned scan code = 19, char_value= 'r'
Press any key (almost)
:You pressed key assigned scan code = 19, char_value= 'R'
Press any key (almost)
:You pressed key assigned scan code = 59, char_value= ' '
Press any key (almost)
:You pressed key assigned scan code = 84, char_value= ' '
Press any key (almost)
:You pressed key assigned scan code = 68, char_value= ' '
Press any key (almost)
:You pressed key assigned scan code = 93, char_value= ' '
Press any key (almost)
:You pressed ESC,
You pressed key assigned scan code = 1, char_value= '

E:\>
```

## התוכנית dem_key5.c

התוכנית dem_key5.c משתמשת ברוטינת ה-BIOS INT 16h, AH = 2 לבדיקת סטטוס המקלדת. הרוטינה הזו מחזירה את בית הדגלים של המקלדת KB_FLAG ב-AL. התוכנית גם קוראת 2 הבתים בכתובות אבסולוטיות 417h (1041) ו-418h (1042) כלומר קצת מעבר ל-IV. כפי שאנחנו רואים בית 417h הינו בעצם KB_FLAG ואילו בית 418h הוא בעצם KB_FLAG_1. התוכנית מדפיסה את הסיביות באמצעות טכניקה בשפת C המאפשר גישה לסיביות של בתי זכרון הנקרא bit fields. שימו לב שתוכנית יכולה באמצעות KB_FLAG ו-KB_FLAG_1 לדעת אם המקשים Shift, Ctrl, Alt לחוצים (אפילו להבדיל בין 2 ה-Shift-ים), לדעת אם הנוריות Caps lock, Num lock, Scroll Lock דלוקות או לא ואפילו לדעת אם המקשים הללו לחוצים כרגע להבדיל ממצב הנוריות (הנוריות יכולות להיות כבויות גם אם המקש לחוץ). ניתן גם להבחין אם המקלדת נעולה ע"י מפתח (Hold) דבר שנתמך בחלק מה-PC-ים.

```
/* dem_key5.c - demostrate use of int 16h, AH = 2, including aux byte. */

#include <stdio.h>

typedef struct status_byte
{
  unsigned int right_shift_key : 1;
  unsigned int left_shift_key : 1;
  unsigned int ctrl_key : 1;
  unsigned int alt_key : 1;
  unsigned int scroll_lock_on : 1;
  unsigned int num_lock_on : 1;
  unsigned int caps_lock_on : 1;
  unsigned int insert_mode_on : 1;
}  STATUS_BYTE;

typedef struct aux_byte
{
  unsigned int unused : 3;
  unsigned int hold_on : 1; /* ctrl-num lock */
  unsigned int scroll_lock_depressed : 1;
  unsigned int num_lock_depressed : 1;
  unsigned int caps_lock_depressed : 1;
  unsigned int insert_key_depressed : 1;
}  AUX_BYTE;


void main(void)
 {

   STATUS_BYTE sbyte, sbyte1;
   char *p, *p1;
   AUX_BYTE  abyte;

   asm  {
          MOV AH,2            ; /* BIOS read keyboard status option */
          INT 16h             ; /* BIOS read keyboard status        */
          MOV BYTE PTR sbyte,AL    ; /* Transfer char to program   */
          PUSH ES
          MOV  AX,0
          MOV  ES,AX            /* ES = 0  */
          MOV  AL,ES:[418h]     /* read phisical locations 417h, 418h */
          MOV  BYTE PTR abyte,AL
          MOV  AL,ES:[417h]
          MOV  BYTE PTR sbyte1,AL
          POP ES
        }

   p = (char *) &sbyte;
   p1 = (char *) &sbyte1;

   printf("\n sbyte = %d, sbyte1 = %d\n", (int) *p, (int) *p1);
```

```
    printf("\nKeyboard status:\n\n");
    printf("Right shift: %d,\nLeft shift: %d,\nCtrl key: %d,\nAlt key: %d,\n",
             (unsigned int) sbyte.right_shift_key,
               (unsigned int) sbyte.left_shift_key,
                 (unsigned int) sbyte.ctrl_key,
                   (unsigned int) sbyte.alt_key);

    printf("Scroll lock on: %d,\nNum lock on: %d,"
       "\nCaps Lock on: %d,\nInsert mode: %d",
             (unsigned int) sbyte.scroll_lock_on,
               (unsigned int) sbyte.num_lock_on,
                 (unsigned int) sbyte.caps_lock_on,
                   (unsigned int) sbyte.insert_mode_on);
    putchar('\n');

    printf("\nAuxilary byte:\n\n");
    printf("Hold on: %d,\nScroll lock dep: %d,\nNum lock dep: %d,"
              "\nCaps Lock dep: %d,\nInsert key dep: %d",
             (unsigned int) abyte.hold_on,
             (unsigned int) abyte.scroll_lock_depressed,
               (unsigned int) abyte.num_lock_depressed,
                 (unsigned int) abyte.caps_lock_depressed,
                   (unsigned int) abyte.insert_key_depressed);
    putchar('\n');

  } /* main */
```

---

```
E:\>DEM_KEY5

 sbyte = 97, sbyte1 = 97

Keyboard status:

Right shift: 1,
Left shift: 0,
Ctrl key: 0,
Alt key: 0,
Scroll lock on: 0,
Num lock on: 1,
Caps Lock on: 1,
Insert mode: 0

Auxilary byte:

Hold on: 0,
Scroll lock dep: 1,
Num lock dep: 1,
Caps Lock dep: 0,
Insert key dep: 0

E:\>
```

## תוכניות דוגמא testslp1.c, mysleep1.asm

התוכנית המשולבת הזו ממחישה איך ב-DOS תוכנית אפליקציה יכולה להתשמש במנגנון הפסיקות לצרכיה ובאופן עקרוני איך מערכות הפעלה עושות זאת היום. העיקר בתוכניות הדוגמא הללו הוא מימוש mysleep, מעין מימוש פרטי של הרוטינה sleep של שפת C. כמו ב-sleep הסטנדרטי mysleep מקבלת פרמטר שלם אותו היא מפרשת כזמן הרדמה בשניות ועוצרת את התוכנית הקוראת לה לזמן הזה. זהו שירות שימושי למדי המאפשרת לתוכנית לממש אלמנטים זמניים (למשל תצוגה זמנית על המסך).

המימוש של השרות הזה כאן הוא באמצעות השתלטות על פסיקה 8 (פסיקת הזמן) ומתוך הנחה שהפסיקה מתרחשת 18.2065 פעמים בשניה.

מה שהמימוש של mysleep עושה למעשה הוא קביעת רוטינה פנימית שלו Timer בתור הרוטינה מטפלת בפסיקה 8 (תוך שימור כתובת רוטינת הטיפול המקורית), התמרת זמן ההרדמה משניות לפסיקות שעון ע"י הכפלה ב-182065 וחילוק ללא שארית ב-10000. את התוצאה mysleep שומר ב-AX. זהו דיוק מספיק בכדי לממש עיכוב בדיוק גבוה למספר רב למדי של שניות.

mysleep מאתחל מונה בשם counter באפס. הרוטינה Timer מקדם את המונה הזה פעם אחת לכל פסיקת שעון, ו-mysleep ממש לולאה המשווה את זמן העיכוב בפסיקות שעון לערך של המונה בלולאה שהוא יוצא ממנה רק כאשר המונה עובר את זמן ההרדמה. למעשה העיכוב בזמן מתבצע בפועל כאן.

פקודות הלולאה הם

```
MOV Counter,0
Delay:
CMP AX,Counter
JA Delay
```

מיד אחרי יציאה מהלולאה הזו mysleep משחזר את כניסה 8 ב-IV וחוזר לקוד הקורא.

נקודה טכנית היא העובדה שפסיקה 8 היא מאותם פסיקות שיש להודיע לחומרה שהיא טופלה ואחת הדרכים לדאוג לכך היא לקרוא רוטינת הטיפול בפסיקה 8 המקורית. לפיכך הקוד של Timer הוא

```
Timer PROC FAR
;
  PUSHF
  CALL DWORD PTR Timer_Offs
  ;
  INC Counter
  ;
  IRET
;
Timer ENDP
```

כאן אני מנצל את העובדה ש-Timer_Seg נמצא מיד אחרי Timer_Offs.
השימוש בסגמנט הקוד לאכסון מידע הוא נוח במימוש פסיקות, כי על הערך של CS תמיד אפשר לסמוך.

שימו לב שכתובת אבסולוטית 32 ( = 4*8) עד 35 היא הכתובת כל כניסה מספר 8 ב-IV. הפקודות

```
MOV AX,ES:[32]
MOV Timer_Offs
MOV AX,ES:[34]
MOV Timer_Seg,AX
```

הם פקודות השימור של הכניסה המקורית ואילו הפקודות

```
CLI
MOV WORD PTR ES:[32],OFFSET Timer
MOV ES:[34],CS
STI
```

הם קביעת הרוטינה Timer כרוטינת הטיפול בפסיקה. כאן אני מנצל את העובדה שאני יודע ש-Timer ו-mysleep נמצאים באותו תאור סגמנט, דבר שלא היה בהכרח נכון אם הם היו בקבצים נפרדים.

```
/* testslp1.c - test mysleep */

#include <stdio.h>

extern void mysleep( int secs );

void main()
{
  int n=20;

  printf("Before mysleep(%d):\n", n);
  mysleep(n);
  printf("After mysleep(%d):\n", n);

}  /* main */
```

---

```
C:\temp>tcc -ml -r- testslp1.c mysleep1.asm
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
testslp1.c:
mysleep1.asm:
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland
International

Assembling file:   mysleep1.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  390k

Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

Available memory 4107872

C:\temp>testslp1
Before mysleep(20):
After mysleep(20):

C:\temp>
```

---

```
; mysleep1.asm - demonstrate Interrupt service routine
;
     .MODEL LARGE
     .STACK  100h
     .DATA
C182       DD 182065
C100       DD 10000
      ;
     .CODE
     .386
Timer_Offs DW 0
Timer_Seg DW  0
Counter   DW  0
;
; My Ctrl-Break ISR
;
Timer PROC FAR
;
     PUSHF
     CALL DWORD PTR Timer_Offs
     ;
     INC  Counter
Return:
     IRET    ;
;
Timer ENDP
```

```
;
;
_mysleep  PROC  FAR
      PUBLIC _mysleep
      PUSH BP
      MOV  BP,SP
      PUSH ES
;
      MOV  AX,0
      MOV  ES,AX
         ;
         ;
;
      MOV  AX,ES:[32]          ; Preserve original ISR pointers
      MOV  Timer_Offs,AX
      MOV  AX,ES:[34]
      MOV  Timer_Seg,AX
      ;
      ; Change Timer pointers
      CLI
      MOV  WORD PTR ES:[32],OFFSET Timer
      MOV  ES:[34],CS
      STI
      MOV  AX,DS
      MOV  ES,AX
                  ;
;
;
      XOR  EAX,EAX
      MOV  AX,[BP+6]    ; Retrieve secs parameter
          ; Compute AX = secs * 18.2065 to convert to ticks
      MUL  C182     ;
      DIV  C100

      MOV  Counter,0            ; Init counter
Delay1:
      ;
      CMP  AX,Counter   ; wait n secs
      JA   Delay1
;
;  Return to caller
      ; Restore old Timer
      ;
                  ;
      MOV  AX,0
      MOV  ES,AX
      MOV  AX,Timer_Offs    ;
      CLI
      MOV  ES:[32],AX
      MOV  AX,Timer_Seg    ;
      MOV  ES:[34],AX
      STI
                  ;
      POP ES
      POP BP
      RET     ; return to caller
      ENDP _mysleep
      END
```

*Chapter 11*

# ROM BIOS Keyboard Services

Although the ROM BIOS services for the keyboard are not as numerous or as complicated as those for the display screen (Chapter 9) and for diskette drives (Chapter 10), the ROM BIOS keyboard services are important enough to warrant their own chapter. All other ROM BIOS services are gathered together in Chapter 12.

## Accessing the Keyboard Services

The keyboard services are invoked with interrupt 16H (decimal 22). As with all other ROM BIOS services, the keyboard services are selected according to the value in register AH. Figure 11-1 lists the ROM BIOS keyboard services.

| *Service* | *Description* |
|---|---|
| 00H | Read Next Keyboard Character. |
| 01H | Report Whether Character Ready. |
| 02H | Get Shift Status. |
| 03H | Set Typematic Rate and Delay. |
| 05H | Keyboard Write. |
| 10H | Extended Keyboard Read. |
| 11H | Get Extended Keystroke Status. |
| 12H | Get Extended Shift Status. |

**Figure 11-1.** *The ROM BIOS keyboard services.*

### Service 00H (decimal 0): Read Next Keyboard Character

Service 00H (decimal 0) reports the next keyboard input character. If a character is ready in the ROM BIOS keyboard buffer, it is reported immediately. If not, the service waits until one is ready. As described on page 134, each keyboard character is reported as a pair of bytes, which we call the main and auxiliary bytes. The main byte, returned in AL, is either 0 for special characters (such as the function keys) or else an ASCII code for ordinary ASCII characters. The auxiliary byte, returned in AH, is either the character ID for special characters or the standard PC-keyboard scan code that identifies which key was pressed.

If no character is waiting in the keyboard buffer when service 00H is called, the service waits—essentially freezing the program that called it—until a character does appear. The service we'll discuss next allows a program to test for keyboard input without the risk of suspending program execution.

Contrary to what some versions of the *IBM PC Technical Reference Manual* suggest, services 00H and 01H apply to both ordinary ASCII characters and special characters, such as function keys.

## Service 01H (decimal 1): Report Whether Character Ready

Service 01H (decimal 1) reports whether a keyboard input character is ready. This is a sneak-preview or look-ahead operation: Even though the character is reported, it remains in the keyboard input buffer of the ROM BIOS until it is removed by service 00H. The zero flag (ZF) is used as the signal: 1 indicates no input is ready; 0 indicates a character is ready. Take care not to be confused by the apparent reversal of the flag values — 1 means no and 0 means yes, in this instance. When a character is ready (ZF = 0), it is reported in AL and AH, just as it is with service 00H.

This service is particularly useful for two commonly performed program operations. One is *test-and-go*, where a program checks for keyboard action but needs to continue running if there is none. Usually, this is done to allow an ongoing process to be interrupted by a keystroke. The other common operation is clearing the keyboard buffer. Programs can generally allow users to type ahead, entering commands in advance; however, in some operations (for example, at safety-check points, such as "OK to end?") this practice can be unwise. In these circumstances, programs need to be able to flush the keyboard buffer, clearing it of any input. The keyboard buffer is flushed by using services 00H and 01H, as this program outline demonstrates:

```
call service 01H to test whether a character is available in the
keyboard buffer
WHILE (ZF = 0)
        BEGIN
        call service 00H to remove character from keyboard buffer
        call service 01H to test for another character
        END
```

Contrary to what some technical reference manuals suggest, services 00H and 01H apply to both ordinary ASCII characters and special characters, such as function keys.

## Service 02H (decimal 2): Get Shift Status

Service 02H (decimal 2) reports the shift status in register AL. The shift status is taken bit by bit from the first keyboard status byte, which is kept at

| Bit 7 6 5 4 3 2 1 0 | Meaning |
|---|---|
| X. . . . . | Insert state: 1 = active |
| . X. . . . . | CapsLock: 1 = active |
| . . X. . . . . | NumLock: 1 = active |
| . . . X. . . . | ScrollLock: 1 = active |
| . . . . X. . . | 1 = Alt pressed |
| . . . . . X. . | 1 = Ctrl pressed |
| . . . . . X. | 1 = Left Shift pressed |
| . . . . . X | 1 = Right Shift pressed |

**Figure 11-2.** *The keyboard status bits returned to register AL using keyboard service 02H.*

memory location 0040:0017H. Figure 11-2 describes the settings of each bit. (See page 137 for information about the other keyboard status byte at 0040:0018H.)

Generally, service 02H and the status bit information are not particularly useful. If you plan to do some fancy keyboard programming, however, they can come in handy. You'll frequently see them used in programs that do unconventional things, such as differentiating between the left and right Shift keys.

## Service 03H (decimal 3): Set Typematic Rate and Delay

Service 03H (decimal 3) was introduced with the PCjr, but has been supported in both the PC/AT (in ROM BIOS versions dated 11/15/85 and later) and in all PS/2s. It lets you adjust the rate at which the keyboard's typematic function operates; that is, the rate at which a keystroke repeats automatically while you hold down a key. This service also lets you to adjust the *typematic delay* (the amount of time you can hold down a key before the typematic repeat function takes effect).

To use this service, call interrupt 16H with AH = 03H, and AL = 05H. BL must contain a value between 00H and 1FH (decimal 31) that indicates the desired typematic rate (Figure 11-3). The value in BH specifies the typematic delay (Figure 11-4). The default typematic rate for the PC/AT is 10 characters per second; for PS/2s it is 10.9 characters per second. The default delay for both the PC/AT and PS/2s is 500 ms.

| | | |
|---|---|---|
| 00H = 30.0 | 0BH = 10.9 | 16H = 4.3 |
| 01H = 26.7 | 0CH = 10.0 | 17H = 4.0 |
| 02H = 24.0 | 0DH = 9.2 | 18H = 3.7 |
| 03H = 21.8 | 0EH = 8.6 | 19H = 3.3 |
| 04H = 20.0 | 0FH = 8.0 | 1AH = 3.0 |
| 05H = 18.5 | 10H = 7.5 | 1BH = 2.7 |
| 06H = 17.1 | 11H = 6.7 | 1CH = 2.5 |
| 07H = 16.0 | 12H = 6.0 | 1DH = 2.3 |
| 08H = 15.0 | 13H = 5.5 | 1EH = 2.1 |
| 09H = 13.3 | 14H = 5.0 | 1FH = 2.0 |
| 0AH = 12.0 | 15H = 4.6 | 20H through FFH - Reserved |

**Figure 11-3.** *Values for register BL in keyboard service 03H. The rates shown are in characters per second.*

00H = 250
01H = 500
02H = 750
03H = 1000
04H through FFH - Reserved

**Figure 11-4.** *Values for register BH in keyboard service 03H. The delay values shown are in milliseconds.*

## Service 05H (decimal 5): Keyboard Write

Service 05H (decimal 5) is handy because it lets you store keystroke data in the keyboard buffer as if a key were pressed. You must supply an ASCII code in register CL and a keyboard scan code in CH. The ROM BIOS places these codes into the keyboard buffer following any keystroke data that may already be present there.

Service 05H lets a program process input as if it were typed at the keyboard. For example, if you call service 05H with the following data, the result is the same as if the keys R-U-N-Enter were pressed:

```
CH = 13H, CL = 52H, call service 05H (the R key)
CH = 16H, CL = 55H, call service 05H (the U key)
CH = 31H, CL = 4EH, call service 05H (the N key)
CH = 1CH, CL = 0DH, call service 05H (the Enter key)
```

| | | |
|---|---|---|
| 00H = 30.0 | 0BH = 10.9 | 16H = 4.3 |
| 01H = 26.7 | 0CH = 10.0 | 17H = 4.0 |
| 02H = 24.0 | 0DH = 9.2 | 18H = 3.7 |
| 03H = 21.8 | 0EH = 8.6 | 19H = 3.3 |
| 04H = 20.0 | 0FH = 8.0 | 1AH = 3.0 |
| 05H = 18.5 | 10H = 7.5 | 1BH = 2.7 |
| 06H = 17.1 | 11H = 6.7 | 1CH = 2.5 |
| 07H = 16.0 | 12H = 6.0 | 1DH = 2.3 |
| 08H = 15.0 | 13H = 5.5 | 1EH = 2.1 |
| 09H = 13.3 | 14H = 5.0 | 1FH = 2.0 |
| 0AH = 12.0 | 15H = 4.6 | 20H through FFH - Reserved |

**Figure 11-3.** *Values for register BL in keyboard service 03H. The rates shown are in characters per second.*

00H = 250

01H = 500

02H = 750

03H = 1000

04H through FFH - Reserved

**Figure 11-4.** *Values for register BH in keyboard service 03H. The delay values shown are in milliseconds.*

## Service 05H (decimal 5): Keyboard Write

Service 05H (decimal 5) is handy because it lets you store keystroke data in the keyboard buffer as if a key were pressed. You must supply an ASCII code in register CL and a keyboard scan code in CH. The ROM BIOS places these codes into the keyboard buffer following any keystroke data that may already be present there.

Service 05H lets a program process input as if it were typed at the keyboard. For example, if you call service 05H with the following data, the result is the same as if the keys R-U-N-Enter were pressed:

```
CH = 13H, CL = 52H, call service 05H (the R key)
CH = 16H, CL = 55H, call service 05H (the U key)
CH = 31H, CL = 4EH, call service 05H (the N key)
CH = 1CH, CL = 0DH, call service 05H (the Enter key)
```

If your program did this when it detected that the F2 function key was pressed, the result would be the same as if the word RUN followed by the Enter key had been typed. (If you use BASIC, this should sound familiar.)

Beware: The keyboard buffer can hold only 15 character codes, so you can call service 05H a maximum of 15 consecutive times before the buffer overflows and the function fails.

## Service 10H (decimal 16): Extended Keyboard Read

Service 10H (decimal 16) performs the same function as service 00H, but lets you take full advantage of the 101/102-key keyboard: It returns ASCII character codes and keyboard scan codes for keys that don't exist on the older 84-key keyboard. For example, the extra F11 and F12 keys found on the 101/102-key keyboard are ignored by service 00H but can be read using service 10H.

Another example: On the 101/102-key keyboard, an extra Enter key appears to the right of the numeric keypad. When this key is pressed, service 00H returns the same character code (0DH) and scan code (1CH) as it does for the standard Enter key. Service 10H lets you differentiate between the two Enter keys because it returns a different scan code (E0H) for the keypad Enter key.

## Service 11H (decimal 17): Get Extended Keystroke Status

Service 11H (decimal 17) is analogous to service 01H, but it, too, lets you use the 101/102-key keyboard to full advantage. The scan codes returned in register AH by this service distinguish between different keys on the 101/102-key keyboard.

## Service 12H (decimal 18): Get Extended Shift Status

Like services 10H and 11H, service 12H (decimal 18) provides additional support for the 101/102-key keyboard. Service 12H expands the function of service 02H to provide information on the extra shift keys provided on the 101/102-key keyboard. This service returns the same value in register AL as service 02H (Figure 11-2), but it also returns an additional byte of flags in register AH (Figure 11-5).

This extra byte indicates the status of each individual Ctrl and Alt key. It also indicates whether the Sys Req, Caps Lock, Num Lock, or Scroll Lock keys are currently pressed. This information lets you detect when a user presses any combination of these keys at the same time.

| *Bit* 7 6 5 4 3 2 1 0 | *Meaning* |
|---|---|
| X . . . . . . . | Sys Req pressed |
| . X . . . . . . | Caps Lock pressed |
| . . X . . . . . | Num Lock pressed |
| . . . X . . . . | Scroll Lock pressed |
| . . . . X . . . | Right Alt pressed |
| . . . . . X . . | Right Ctrl pressed |
| . . . . . X . | Left Alt pressed |
| . . . . . . . X | Left Ctrl pressed |

**Figure 11-5.** *Extended keyboard status bits returned in register AH by keyboard service 12H.*

## Comments and Example

If you are in a position to choose between the keyboard services of your programming language or the ROM BIOS keyboard services, you could safely and wisely use either one. Although in some cases there are arguments against using the ROM BIOS services directly, as with the diskette services, those arguments do not apply as strongly to the keyboard services. However, as always, you should fully examine the potential of the DOS services before resorting to the ROM BIOS services; you may find all you need there, and the DOS services are more long-lived in the ever-changing environments of personal computers.

Most programming languages depend on the DOS services for their keyboard operations, a factor that has some distinct advantages. One advantage is that the DOS services allow the use of the standard DOS editing operations on string input (input that is not acted on until the Enter key is pressed). Provided that you do not need input control of your own, it can save you a great deal of programming effort (and user education) to let DOS handle the string input, either directly through the DOS services or indirectly through your language's services. But if you need full control of keyboard input, you'll probably end up using the ROM BIOS routines in the long run. Either way, the choice is yours.

Another advantage to using the DOS keyboard services is that the DOS services can redirect keyboard input so that characters are read from a file instead of the keyboard. If you rely on the ROM BIOS keyboard services, you can't redirect keyboard input. (Chapters 16 and 17 contain information on input/output redirection.)

For our assembly-language example of the use of keyboard services, we'll get a little fancier than we have in previous examples and show you a complete buffer flusher. This routine will perform the action outlined under keyboard service 01H, the report-whether-character-ready service.

```
_TEXT           SEGMENT byte public 'CODE'
                ASSUME  cs:_TEXT

                PUBLIC  _kbclear
_kbclear        PROC    near

                push    bp
                mov     bp,sp

L01:            mov     ah,1            ; test whether buffer is empty
                int     16h
                jz      L02             ; if so, exit

                mov     ah,0
                int     16h             ; otherwise, discard data
                jmp     L01             ; .. and loop

L02:            pop     bp
                ret

_kbclear        ENDP

_TEXT           ENDS
```

The routine works by using interrupt 16H, service 01H to check whether the keyboard buffer is empty. If no characters exist in the buffer, service 01H sets the zero flag, and executing the instruction JZ L02 causes the routine to exit by branching to the instruction labeled L02. If the buffer still contains characters, however, service 01H clears the zero flag, and the JZ L02 instruction doesn't jump. In this case the routine continues to the instructions that call service 00H to read a character from the buffer. Then the process repeats because the instruction JMP L01 transfers control back to label L01. Sooner or later, of course, the repeated calls to service 00H empty the buffer, service 01H sets the zero flag, and the routine terminates.

Among the new things this buffer-flusher routine illustrates is the use of labels and branching. When we discussed the generalities of assembly-language interface routines in Chapter 8, we mentioned that an ASSUME CS statement is necessary in some circumstances, and you see one in action here.

The ASSUME directive in this example tells the assembler that the labels in the code segment (that is, labels that would normally be addressed using the CS register) do indeed lie in the segment whose name is _TEXT. This may seem obvious, since no other segments appear in this routine.

Nevertheless, it is possible to write assembly-language routines in which labels in one segment are addressed relative to some other segment; in such a case, the ASSUME directive would not necessarily reference the segment within which the labels appear. In later chapters you'll see examples of this technique, but here the only segment to worry about is the _TEXT segment, and the ASSUME directive makes this fact explicit.