CHAPTER 13: PROGRAMMING TECHNIQUES

Introduction

In this chapter a sample assembler program will be explored which illustrates a number of the system calls and interface strategies which have been documented in earlier chapters. Assembler was used in order to stay as generic as possible. The process of adapting this program to a high level language such as C or Pascal should be relatively straightforward, as most current implementations include mechanisms to issue software interrupt calls and cast pointers. A version of this program done in C is also included on the disk.

Accessing System Data Structures

As MOS evolves, the method used to access the kernel's data structures has had to change. Prior to version 4.10, direct access to the SCB was achieved through a pointer obtained from Extended Services function 02H. Likewise, function 04H was used to obtain a pointer for direct TCB access.

Beginning with version 4.10, all access to MOS's data structures should be achieved through the use of Extended Services functions 26H, 27H, 28H, 29H and 2AH. This group of functions provides a means of access which will insulate the developer from any further changes in access methods.

This change has been made to accommodate the eventual relocation of MOS's kernel and kernel data structures into memory above the 1st megabyte. Once this migration is completed, the use of these new Extended Services functions will be the only way to access data within the SCB and TCB data structures.

The sample assembler program illustrated within this chapter contains a set of subroutines to simplify SCB and TCB access. These routines are GET_SCB, PUT_SCB, GET_TCB and PUT_TCB.

Many of the data structures used by MOS are covered in Chapter 9. The companion disk also contains include files which may be used with Assembler and C programs. The SCB and TCB structures are the ones which are likely to be of the most interest with regard to the development of applications.

The primary data structure is known as the SCB -- short for System Control Block. There is one copy of this structure per system, and it is located within a portion of the kernel. The SCB is used to hold information about the kernel which is global in nature rather than task-specific.

As each new task is spawned with the ADDTASK command or the ADDTASK API function, one copy of a structure known as a TCB (Task Control Block) is allocated from the SMP and initialized. This data structure is used to hold information specific to each task. It also contains the stack space for each task when the kernel switches to an internal stack.

It can be useful in certain circumstances to be able to access each of the TCB's within a system. The task table implementation discussed in Chapter 12 involves the need to verify that a task still exists. Another situation could be in the construction of a status display utility to show certain state information about each task.

Each TCB begins on a segment boundary and is linked into a simple linked list. The root pointer to this list is kept in the SCB variable SCBTCBPF (PF for Pointer First) and there will always be at least one TCB for the foreground task. In fact, the first TCB in this list, the one pointed to by the SCBTCBPF root pointer, will always be the one for the foreground task.

The "next" pointer in this linked list is held within the variable TCBTCBPN (PN for Pointer Next). When a 0 is found within this word pointer, the end of the list has been reached. Otherwise, this field contains a segment pointer to the next TCB in the list.

The PUSHF/FAR CALL Calling Method

Making calls to an interrupt vector using the pushf/far call coding approach is recommended as an efficiency measure. When running on an 80386-based machine where the $386.SYS memory management driver is being used, the use of virtual 8086 mode imposes an additional overhead of several hundred cycles on each software interrupt call. The protected mode supervisor (part of $386.SYS) must intercept all interrupts as part of supporting a virtual machine environment.

When you are working in assembler there is an alternate method of entering a software interrupt handler. This can be done by pushing a copy of the flags register onto the stack and then making a far call to a copy of the vector which has been placed in a local variable. The following macro can be used to simplify this optimized calling method. The initialization of the I21VECT pointer is illustrated in listing 13-1.

```
I21CALL MACRO
    PUSHF           ;; CALL INT21H SERVICES
    CLI
    CALL [I21VECT]  ;; USING THE LOCAL VECTOR
    ENDM
```

Doing a pushf and a far call is much faster than executing a software interrupt when running in a VM86 mode task (a task which provides a PC-DOS type environment). However, when coding applications for the 80386's 32-bit protected mode environment, it is essential that a software interrupt be issued. If the PUSHF/FAR CALL method were used, the protected mode supervisor would not get a chance to translate system calls between the 32-bit application level and the 16-bit kernel level.

This is another good reason to use a macro such as I21CALL for system calls. Should you need to port code you've developed for a 16-bit mode up to 32-bit protected mode, change the contents of the I21CALL macro as follows:

```
I21 CALL MACRO
    INT     21H
    ENDM
```

Program Initialization

You may wish to design an application program, system utility, or device driver which takes advantage of specific features of the PC-MOS operating system. Shown in listing 13-1 is a sample assembler language program which can provide a basis for such a development.

As shown, this program is intended to produce a .COM type of program module. It illustrates how a program can verify that it is running under PC-MOS, how to obtain system call vectors and pointers, and how to use some of MOS's API functions. Not all situations will require the pointers and API calls illustrated here -- and some will require many more. This example is provided as a base to expand on.

The first subroutine called, the ismos routine, serves to protect a user from inadvertently using a PC-MOS specific program in any other environment. MOS will report two different version numbers through function 30H of INT 21H. If the four general purpose registers, AX, BX, CX and DX all have the same value when this call is made then the PC-MOS version number is returned in AX. When all four registers are not equal then the PC-DOS equivalent version number is reported.

In order to insure that this comparison of version numbers will remain a reliable indicator, MOS's version number will always be kept different than the DOS equivalent.

In order to facilitate the pushf/far call calling method, the GET_VECTS procedure initializes a set of local doubleword pointer variables. Those for INT 14H, INT 16H and INT 21H are obtained by calling function 35H of INT 21H. Initializing a pointer to MOS's Extended Services could also be done by reading the INT 38H or INT D4H vector with function 35H but the method shown is recommended for reliability.

A call is issued to function 34H of INT 21H and the value returned in ES:BX is used to fetch the desired pointer. This method insures that the right vector will be obtained no matter what any applications might do with the 38H and D4H vectors.

Under DOS, issuing a call to function 34H of INT 21H will return with ES:BX pointing to an internal flag within the kernel. This flag is known as the INDOS flag and is often used by TSR programs and applications with certain types of interrupt handlers to determine when it is safe to issue INT 21H system calls. MOS supports this same system call by returning with ES:BX pointing to it own version of this flag, the INMOS flag. In addition, this flag happens to be located within MOS's SCB at a fixed offset from the field which contains a far pointer to the Extended Services entry point.

Determining the Data Access Method

Next, the PC-MOS version number is used to determine which method must be used for access the kernel's data structures. For programs running under a version of MOS prior to 4.10, the "old" data access method must be used. Extended Services function 02H is used to obtain a far pointer to the base of the SCB data structure and function 04H is used to obtain the segment of the TCB data structure.

Access to fields within the SCB is achieved by adding an offset to the far pointer which function 02H returns in ES:BX. Since TCB structures always start exactly on a segment boundary, the offset of the data you're interested in is simply used directly with the segment.

For programs running under version 4.10 or higher, access to the kernel's data structures should be achieved through the use of Extended Services functions 26H, 27H, 28H, 29H and 2AH.

Adding a Task

The next thing that happens in listing 13-1 is the addition of a background task. Through MOS's system calls, it is possible for an application to spawn a background task, communicate with it through files, the pipe driver, MOS's NETBIOS emulator or a custom device driver and then remove the task when its function is complete. Another alternative would be for the application to custom write the startup batch file for the new task.

A background task is distinguished from a workstation task by the field within the addtask data structure which contains the entry point of the DDT driver. A zero in this doubleword field means to add a background task. The port number and baud rate fields should also be 0 in this case.

If the pointer to the optional startup batch file is used, only the batch file's name must be given. This batch file must exist within the root directory of the current drive -- and no drive, path, or extension elements can be included within the string passed to the addtask API function. If no startup batch file is to be used, a doubleword pointer to a binary zero must be set into the addparm.tbatch field.

The DISP_STATS procedure is provided to illustrate access to the return data which the addtask function will place in the data structure. Routines to handle binary to ascii conversion and display formatting have not been included for the sake of brevity.

Establishing Raw Mode

When a workstation task is to be added, a terminal device driver (DDT) must be specified along with whatever port number and baud rate information is relevant to the driver being used. Note that in the case of workstations such as VNA and SunRiver, there is no such thing as a baud rate. In addition, certain manipulations must be done on the port number as is described below.

The data which must be supplied to the addtask API function about the DDT is in the form of a far pointer to the DDT's entry point. Obtaining this doubleword pointer requires that the driver be opened and that a read of 4 bytes be done from it.

Since the driver appears to the operating system as a character type of driver it will be opened in what is known as "cooked mode" by default. However, since the data being read is an address, it is possible that one or more of the four bytes will appear as a character which the cooking logic acts on. If raw mode were not established by making the IOCTL calls, the four byte read could be truncated if one of the bytes happened to be a 1AH. It would appear to the cooking logic as if a Ctrl-Z code had been encountered.

When opening a DDT type driver, the device name is formed by prepending two dollar signs to the file name. For example, to open the PCTERM.SYS driver, use the device name '$$PCTERM'.

The CHK_PORTNUM Procedure

This procedure is provided to handle certain peculiarities which come into play when adding a VNA or SunRiver workstation type of task. Tests must be done to determine if a co-resident type of system configuration is being used. For the case of a serial terminal workstation task, this procedure verifies that the port number is within range and that a serial driver has been installed. Note also that for the serial terminal case, the port number supplied to the addtask API call is zero-based. Thus, if you would use:

```
ADDTASK 300,,,,PCTERM,1,19200
```

on the command line (where port number 1 is specified), you would need to specify port number 0 in the addtask data structure.

The Wait for Event Call

The last section of the sample program shows how a program can explicitly give up its share of the CPU time. This would be done when the program needs to periodically perform some function but does not want to be a CPU "hog" and reduce the overall system throughput. Note that function 7 of Extended Services has many more options than those made use of here.

If you want to set up a time-out for a certain interval and do not want to have the time- out end when a key is pressed, you should still set bit 1 of AL. If you literally tell MOS to suspend your task until a certain time interval passes, MOS will do just that -- and you will not be able to PAMswitch out of that task. The third example use of function 7 shows how to set up a simple time delay without defeating PAMswitching.

Program Listing 1:

```
; (c) Copyright 1989, 1990 The Software Link, Incorporated

; tref.asm - sample program for PC-MOS Tech Ref

;=== Include files ===================

scb     segment at 0
        include mosscbdf.inc                    ; for access to SCB fields
scb     ends

tcb     segment at 0
        include mostcb.inc              ; for access to TCB fields
tcb     ends

;=== Macros ==========================

i21call macro
        pushf               ;; call int21h services
        cli                 ;; using the local vector
        call [i21vect]
        endm

i14call macro
        pushf               ;; call int14h services
        cli                 ;; using the local vector
        call [i14vect]
        endm

i16call macro
        pushf               ;; call int16h services
        cli                 ;; using the local vector
        call [i16vect]
        endm
```

```
extcall   macro
          pushf                ;; call MOS's extended services
          cli                  ;; using the local vector
          call [extvect]
          endm

;=== Structures ======================

; the following structure outlines the data fields which must
; be provided for the addtask API function call

add_data struc
tsize     dw 0                 ; task size
tid       dw 0                 ; task id
tclass    db ' '               ; task class
tbatch    dd 0                 ; task startup batchfile
tdriver   dd 0                 ; task terminal driver
tport     dw 0                 ; task port
tbaud     dd 0                 ; task baud rate
tmemtot   dd 0                 ; total ext mem       (RETURN)
tmemalc   dd 0                 ; ext mem allocated (RETURN)
tsmpal    dw 0                 ; task smp allocate  (RETURN)
tsmpsiz   dw 0                 ; task smp size      (RETURN)
tpercent  dw 0                 ; task percent heap (RETURN)
tres      db 3 dup (0)         ; reserved
add_data ends

;=== Main code segment ===============

code      segment para public 'code'
          assume  cs:code, ds:code, es:code
          org 0100H

start     proc      near              ; this is the entry point for
          jmp       begin             ; a .COM type program module
```

```
;=== Variables ======================

i21vect         dd      0       ; int21 func handler ptr
i14vect         dd      0       ; int14 func handler ptr
i16vect         dd      0       ; int16 func handler ptr
extvect         dd      0       ; extended svcs handler ptr
access          db      0       ; scb/tcb access method flag
scbptr          dw      0,0     ; scb far ptr or scb selector
tempb1          db      0       ; for access to scb/tcb
tempw1  dw      0       ; for access to scb/tcb
addparm add_data <>     ; structured variable for addtask

;=== Strings ========================

strtbat  db     0               ; pointer to startup batch file
ddtname  db     '$$pcterm',0    ; pointer to ddt
ermsg1   db     13,10,'PC-MOS must be the operating system',13,10,'$'
ermsgx   db     13,10,'An error occured',13,10,'$'

;=== Subroutines ====================

;=GET_SCB ===================================================
; entry parms:  bx = offset of scb data item
;               es:di -> destination for scb data
;               cx = number of bytes to transfer
; exit parms:   none
;
; NOTES: this procedure requires the access flag and the
; scbptr variables to be initialized
;= ==========================================================
get_scb:
        push    ax
        cmp     [access],0
        jne     get_snew
        push    cx
        push    si
        push    di
        push    ds
        lds     si,dword ptr [scbptr]
        add     si,bx                       ; use the old access method
        cld
        rep     movsb
        pop     ds
        pop     di
        pop     si
        pop     cx
        jmp     short get_scont

get_snew:
        mov     dx,[scbptr]
        mov     ah,28h                      ; use the new method
        extcall

get_scont:
        pop     ax
        ret
```

```
;= PUT_SCB ======================================================
; entry parms:      bx = offset of scb data item
;                   ds:si -> source of the data
;                   cx = number of bytes to transfer
; exit parms:       none
;
; NOTES: this procedure requires the access flag and the
; scbptr variables to be initialized
;================================================================
put_scb:
        push    ax
        cmp     [access],0
        jne     put_snew
        push    cx
        push    si
        push    di
        push    es
        les     di,dword ptr [scbptr]
        add     di,bx                   ; use the old access method
        cld
        rep     movsb
        pop     es
        pop     di
        pop     si
        pop     cx
        jmp     short put_scont

put_snew:
        mov     ah,29h                  ; use the new method
        extcall

put_scont:
        pop     ax
        ret
```

```
;= GET_TCB ==========================================================
; entry parms:      bx = offset of tcb data item
;                   dx = segment/selector of the tcb
;                   es:di -> destination for tcb data
;                   cx = number of bytes to transfer
; exit parms:       none
;
; NOTES: this procedure requires the access flag be initialized
;====================================================================
get_tcb:
        push    ax
        cmp     [access],0
        jne     get_tnew
        push    cx
        push    si
        push    di
        push    ds
        mov     ds,dx
        mov     si,bx                   ; use the old access method
        cld
        rep     movsb
        pop     ds
        pop     di
        pop     si
        pop     cx
        jmp     short get_tcont

get_tnew:
        mov     ah,28h                  ; use the new method
        extcall

get_tcont:
        pop     ax
        ret
```

```
;= PUT_TCB ===============================================================
; entry parms:      bx = offset of tcb data item
;                   dx = segment/selector of the tcb
;                   ds:si -> source of the data
;                   cx = number of bytes to transfer
; exit parms:       none
;
; NOTES: this procedure requires the access flag be initialized
;=========================================================================
put_tcb:
          push      ax
          cmp       [access],0
          jne       put_tnew
          push      cx
          push      si
          push      di
          push      es
          mov       es,dx
          mov       di,bx                       ; use the old access method
          cld
          rep       movsb
          pop       es
          pop       di
          pop       si
          pop       cx
          jmp       short put_tcont

put_tnew:
          mov       ah,29h                      ; use the new method
          extcall

put_tcont:
          pop       ax
          ret
```

```
;= ISMOS ==========================================================
; entry parms:      none
; exit parms:       nz flag if the OS is MOS
;                   zr flag if not
;
; NOTES: Verify that this program is running under the PC-MOS
; operating system before MOS specific system calls are made.
;==================================================================
ismos:
        push    ax
        push    bx
        push    cx
        push    dx
        mov     ax,3000h
        mov     bx,ax                   ; set ax == bx == cx == dx
        mov     cx,ax                   ; to read the MOS version #
        mov     dx,ax
        int     21h
        push    ax
        mov     ax,3099h                ; now insure ax is different
        int     21h                     ; to read the DOS version #
        pop     bx
        cmp     bx,ax                   ; if bx != ax then MOS
        pop     dx                      ; if bx == ax then not
        pop     cx
        pop     bx
        pop     ax
        ret
```

```
;= GET_VECTS ==================================================================
; entry parms:      none
; exit parms:       the global variables  i21vect, i14vect, i16vect
;                            and extvect are initialized
;==============================================================================
get_vects:
          push     ax
          push     bx
          push     es
          mov      ax,3521h
          int      21h                              ; initialize i21vect
          mov      word ptr [i21vect],bx
          mov      word ptr [i21vect+2],es
          mov      ax,3514h
          i21call                                   ; initialize i14vect
          mov      word ptr [i14vect],bx
          mov      word ptr [i14vect+2],es
          mov      ax,3516h
          i21call                                   ; initialize i16vect
          mov      word ptr [i16vect],bx
          mov      word ptr [i16vect+2],es
          mov      ah,34h
          i21call
          les      bx,es:[bx-18h]                   ; initialize extvect by
          mov      word ptr [extvect],bx            ; indirect access to
          mov      word ptr [extvect+2],es          ; MOS's SCB data structure.
          pop      es
          assume   es:code
          pop      bx
          pop      ax
          ret
```

```
;= GET_ACCESS ================================================
; entry parms:      none
; exit parms:       the global variables access and scbptr
;                   are initialized.
;=============================================================
get_access:
          push      ax
          push      bx
          push      cx
          push      dx
          push      es
          mov       ax,3000h
          mov       bx,ax                        ; set ax == bx == cx == dx
          mov       cx,ax                        ; to read the MOS version #
          mov       dx,ax
          i21call
          cmp       al,4
          jb        get_aold                     ; if running MOS version
          ja        get_anew                     ; 4.10 or newer, use the
          cmp       ah,0ah                       ; new data access method
          jb        get_aold
get_anew:
          mov       [access],1
          mov       ah,26h
          extcall
          mov       [scbptr],dx
          jmp       short get_acont
get_aold:
          mov       ah,2
          extcall
          mov       [scbptr],bx
          mov       [scbptr+2],es
get_acont:
          pop       es
          pop       dx
          pop       cx
          pop       bx
          pop       ax
          ret
```

```
;= READ_DRIVER ==================================================
; entry parms:    ds:dx -> pointer to device driver's name
;                         (in asciiz string format)
; exit parms:     nc if no error
;                         es:bx = pointer read from driver
;                 cy if error
;                         ax = error code
;                         es:bx indeterminate
;
; NOTES: this procedure opens a character device driver in raw mode
; and does a read of 4 bytes to get the driver's alternate entry point.
;================================================================
read_driver:
        push    bp
        sub     sp,4            ; establish a temporary
        mov     bp,sp           ; double word variable at [bp]
        push    cx
        push    dx
        push    ds
        mov     ax,3d02h        ; open the driver
        i21call
        jc      rdverr
        mov     bx,ax
        mov     ax,4400h
        i21call
        jc      rdverr
        xor     dh,dh           ; use ioctl to
        or      dl,00100000b    ; establish raw mode
        mov     ax,4401h
        i21call
        jc      rdverr
        mov     ah,3fh          ; read 4 bytes
        mov     cx,4
        mov     dx,bp           ; make ds:dx point to the
        push    ss              ; temporary stack variable
        pop     ds
        i21call
        jc      rdverr
        mov     ah,3eh              ; close the handle
        i21call
        jc      rdverr
        mov     bx,[bp]             ; return the pointer
        mov     es,[bp+2]           ; in es:bx
        pop     ds
        pop     dx
        pop     cx
        add     sp,4                ; cancel the temp var
        pop     bp
        ret
rdverr:
        pop     ds
        pop     dx
        pop     cx
        add     sp,4
        pop     bp                  ; in case of an error,
        stc                         ; must reset the CY flag
        ret                         ; after the add sp,4

;= CHK_PORTNUM ==================================================
; entry parms:      the filled in addparm structureq
;                   ds:dx pointing to the ddt name string
; exit parms:       nc if no error
;                           port number in addpart structure adjusted
;                           as required.
;                   cy if error
;                           ax = error code
;                           ax = 87 for invalid parameter
;                           ax = 0ffffh for serial driver not installed
;
; NOTES: this procedure performs adjustments to the port number as
; are required for special cases.
;================================================================
comment ¦

        if adding VNA or SunRiver (buad rate == 0)
        if master console is SunRiver
        if adding a SunRiver task  then increment port number
        return(0)
        if master console is VNA then increment port number
        return error if port number out of range, else just return
        else                    (serial terminal case)
        return error code if port number out of range
        check for serial driver
        return error code if not found
¦

chk_portnum:
        push    bx
        push    dx
        push    si
        push    es
        mov     ax,word ptr [addparm.tbaud]             ; if baud != 0 then
        or      ax,word ptr [addparm.tbaud+2]           ; must be a serial
        jnz     chkp15                                  ; terminal workstation
        mov     bx,offset scbtcbpf
        mov     di,cs                           ; no baud rate specified,
        mov     es,di                           ; must be VNA or SunRiver
        mov     di,offset tempw1                ; point to tcb of
        mov     cx,2                            ; master console (mc).
        call    get_scb
        mov     dx,[tempw1]                     ; get the tcb segment selector
        mov     bx,offset tcbvram               ; for the foreground task
        mov     di,offset tempb1
        mov     cx,1                            ; fetch the tcbvram byte from
        call    get_tcb                         ; the foreground tcb
        test    [tempb1],4                      ; is mc SunRiver?
        jz      chkp05                          ; no
        mov     bx,dx
        cmp     word ptr [bx+2],'RS'        ; and adding SunRiver?
        jne     chkp30
        inc     [addparm.tport]             ; yes, increment port
        jmp     chkp30
chkp05:                                     ; must be adding VNA
        test    [tempb1],8                  ; is mc VNA?
        jz      chkp10
        inc     [addparm.tport]             ; yes, increment port
chkp10:
        cmp     [addparm.tport],16          ; too large?
        jb      chkp30
        mov     ax,87                       ; invalid parameter
        jmp     chkp25
chkp15:
        cmp     [addparm.tport],24          ; validate port number for
        jb      chkp20                      ; serial port case
        mov     ax,87
        jmp     chkp25
chkp20:
        mov     ah,6
        mov     dx,[addparm.tport]
        i14call                             ; make sure a serial
        test    ah,80h                      ; driver is installed
        jnz     chkp30
        mov     ax,-1
chkp25:
        stc
        jmp     chkp35
chkp30:
        clc
chkp35:
        pop     es
        assume  es:code
        pop     si
        pop     dx
        pop     bx
        ret
```

```
;= DISP_STATS ==================================================
; entry parms:      the filled in addparm structure
;                   es points to the new task's tcb
; exit parms:       none
;
; NOTES: this procedure displays the memory statistics after a call
; is made to the addtask api function.
;===============================================================
disp_stats:
        mov     dx,es
        mov     bx,offset tcbid
        mov     cx,2
        mov     di,cs
        mov     es,di
        mov     di,offset tempw1
        call    get_tcb
        mov     ax,[tempw1]             ; get the new task's id number

        ; your display code goes here

        mov     ax,word ptr [addparm.tmemtot]
        mov     dx,word ptr [addparm.tmemtot+2]
        push    ax
        or      ax,dx                   ; if tmemtot == 0 then
        pop     ax                      ; there is no paging
        jz      no_mem_manag            ; capable memory management

; dx:ax now contains the total number of 4K
; blocks of extended memory


        ; your display code goes here

        mov     ax,word ptr [addparm.tmemalc]
        mov     dx,word ptr [addparm.tmemalc+2]

; dx:ax now contains the number of 4K
; blocks of extended memory currently allocated
```

```
        ; your display code goes here

no_mem_manag:
        mov     ax,[addparm.tsmpal]             ; smp paragraphs allocated

        ; your display code goes here

        ; repeat for tsmpsiz and tpercent

        ret

;========================================================================
;======================== MAIN PROCEDURE =======================
;========================================================================
        assume  cs:code, ds:code, es:code
begin:
        call    ismos                           ; right OS?
        jne     yes_mos
        mov     dx,offset [ermsg1]  ; no - report
        jmp     error2

; At this point, it has been verified that the
; operating system is PC-MOS so it is safe to
; use MOS specific function calls and interface to
; MOS's data structures

yes_mos:
        call    get_vects                       ; setup call macros
        call    get_access                      ; determine access method

; At this point, pointers and status
; variables have been initialized.
; add a background task

        mov     [addparm.tsize],32          ; 32K
        mov     [addparm.tid],0             ; let MOS pick the id
        mov     [addparm.tclass],' '        ; no security
        mov     word ptr [addparm.tbatch],offset strtbat
        mov     word ptr [addparm.tbatch+2],cs
        mov     word ptr [addparm.tdriver],0
        mov     word ptr [addparm.tdriver+2],0
        mov     [addparm.tport],0
        mov     word ptr [addparm.tbaud],0
        mov     word ptr [addparm.tbaud+2],0
        mov     si,offset [addparm]
        mov     ah,22h
        extcall
        jnc     addbg_ok
        jmp     error1
addbg_ok:
        call    disp_stats

; add a workstation task

        mov     [addparm.tsize],32          ; 32K
        mov     [addparm.tid],0             ; let MOS pick the id
        mov     [addparm.tclass],' '        ; no security
        mov     word ptr [addparm.tbatch],offset strtbat
        mov     word ptr [addparm.tbatch+2],cs
        mov     dx,offset ddtname
        call    read_driver                 ; make es:bx point to driver
        jnc     $+5
        jmp     error1
        mov     word ptr [addparm.tdriver],bx
        mov     word ptr [addparm.tdriver+2],es
        mov     [addparm.tport],0           ; use zero based port numbering
        mov     word ptr [addparm.tbaud],19200
        mov     word ptr [addparm.tbaud+2],0
        mov     dx,offset ddtname
        call    chk_portnum
        jc      error1
        mov     si,offset [addparm]
        mov     ah,22h
        extcall
        jnc     addwks_ok
        jmp     error1
addwks_ok:
        call    disp_stats

; This section illusrates the use of the suspend call
; to wait for a certain number of timer ticks (in bx) or until
; a key is pressed. NOTE that this call will return when any key
; is pressed which produces a scan code. This includes the shift,
; cntrl and alt keys.

again1:
        mov     ah,7                    ; function 7 - suspend
        mov     al,00000011b            ; wait for key or time out
        mov     bx,10                   ; ticks to wait
        extcall
        jc      error1
        test    al,1                    ; was a key pressed?
        jnz     got_key1

; insert code here to manage periodic events

        jmp     again1
got_key1:

; insert code here to handle keystrokes
; This section illusrates the use of the suspend call
; to wait for a certain number of timer ticks (in bx) or until
; a key is pressed. This time, INT 16H, function 1 is used to
; prevent a response to the shift keys.

again2:
        mov     ah,7                    ; function 7 - suspend
        mov     al,00000011b            ; wait for key or time out
        mov     bx,10                   ; ticks to wait
        extcall
        jc      error1
        test    al,2                    ; was a key pressed?
        jnz     timeout2
        mov     ah,1                    ; if yes, filter out scan codes
        i16call
        jz      again2
        jmp     got_key2

timeout2:

; insert code here to manage periodic events

        jmp     again2
got_key2:

; insert code here to handle keystrokes


; This section illusrates the use of the suspend call
; to wait for a certain number of timer ticks (in bx) only.
; In order to allow pamswitch keystrokes to be recognized,
; the wait for key bit must also be used.
```

```
again3:
          mov     ah,7                          ; function 7 - suspend
          mov     al,00000011b                  ; wait for key or time out
          mov     bx,10                         ; ticks to wait
          extcall
          jc      error1
          test    al,1                          ; was a key pressed?
          jnz     again3                        ; yes, re-suspend

;=== Terminate handler ========

          mov     al,0                ; for errorlevel == 0
          jmp     terminate
error1:

; add your own error decoding and messages here

          mov     dx,offset [ermsgx]

error2:
          mov     ah,9
          int     21h
          mov     al,1                ; errorlevel 1
terminate:
          mov     ah,4ch
          int     21h

start     endp
code      ends
          end     start
```