MPX386.s
Inicialización y manejo de interrupciones.

Carlos J. Suárez Rodríguez
Eduardo Mas Carrillo

Introducción.-

El fichero mpx386.s es el encargado de realizar dos funciones importantes del sistema operativo. Por un lado es el encargado de inicializar el sistema, creando las tablas y la pila necesaria para el posterior procesamiento.

La otra función básica de este fichero es el manejo de las interrupciones, incluyendo como tales las interrupciones hardware, las software y las excepciones.

Inicialización del sistema.-

Una vez que el proceso de autoarranque ha cargado el sistema operativo en memoria, se pasa a ejecutar este fichero, comenzando la ejecución en la etiqueta MINIX (línea 6051).

Existe un salto de unos cuantos bytes que contiene flags del boot monitor.

Seguidamente se crea una pila para proveer el entorno apropiado para el código compilado por el compilador de C. Para ello se copian tablas usadas por el procesador para definir los segmentos de memoria y se inicializan varios registros del procesador. Todo este proceso ocurre entre las líneas 6063 y 6108.

A continuación el proceso continua llamando a la función **cstart**. Esta a su vez llama a otra rutina para inicializar la Tabla de Descriptores Globales (GDT), estructura de datos del procesador para la protección de memoria, y la Tabla de Descriptores de Interrupciones (IDT), usada para elegir el código que será ejecutado en cada posible tipo de interrupción.

Al retorno de esta función, las tablas que se han creado se hacen efectivas mediante las instrucciones ***lgdt*** y ***lidt***, y se fuerza su uso mediante la intrucción ***jmpf CS_SELECTOR: csinit*** (línea 6118).

Después de realizar algunas operaciones con los registros del procesador MINIX termina con un salto, NO una llamada, al punto de entrada del kernel **main** (6131)**.**

En este momento el código de inicialización del mpx386.s está completo. El resto del fichero contiene código para la recuperación de procesos, manejo de interrupciones y otras rutinas de soporte.

Manejo de interrupciones.-

El 80386 soporta tres tipos de fuentes de interrupción:

1) **Interrupción hardware**. Producida por dispositivos externos (reloj, disquetera, etc).

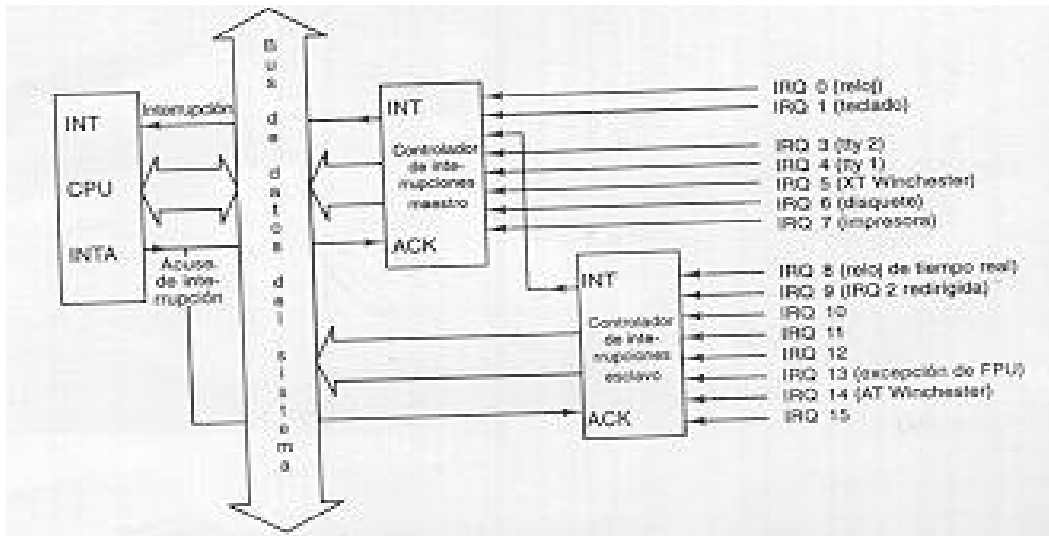
2) **Interrupción software**. Producida por la ejecución de ciertas instrucciones.

3) **Excepción**. Producida por el propio procesador al encontrar una determinada situación interna.

Cada tipo diferente de interrupción o excepción tiene un número identificativo en el rango de 0-255. Algunos números de interrupción y excepción están predefinidos por el procesador, otros están reservados por Intel y la mayoría están disponibles para el S.O. Es conveniente recordar que una interrupción detiene un proceso y que la finalización del servicio de la interrupción reinicia otro proceso, tal vez, distinto al que la causó.

Interrupciones hardware.-

En el Intel 386 las interrupciones Hardware se controlan mediante dos chips, uno maestro y otro esclavo. El chip esclavo tiene su salida conectada a la entrada del maestro, que es el que emite la interrupción al procesador.



Las interrupciones hardware se producen por la petición de algún dispositivo externo. La CPU inhabilita todas las interrupciones cuando se recibe una interrupción de este tipo.

En el fuente existe un punto de entrada para cada interrupción. El código de cada entrada de **_hwint00** hasta **_hwint07** (líneas 6164 a 6193) son llamadas a **HWINT_MASTER** y las entradas de **_hwint08** hasta **_hwint15** (líneas 6222 a 6251) son llamadas a **HWINT_SLAVE**. Además, parece que en cada entrada se pasa un parámetro en la llamada, indicando el dispositivo que necesita el servicio. De hecho, en realidad no son llamadas sino macros, ya que así el código ensamblado queda muy compacto permitiendo una mayor brevedad de ejecución.

En la parte referente a **HWINT_MASTER** lo primero que se realiza es llamar a **save** (6144), subrutina que guarda todos los registros necesarios para reanudar un proceso interrumpido.

Cuando la CPU recibe una interrupción deshabilita todas las interrupciones. Es por esto que **HWINT_MASTER** desactiva las interrupciones del tipo recibido, para evitar recibir otra del mismo tipo, y activa el resto, permitiendo que se puedan anidar interrupciones de distinto tipo.

A continuación se llama a la parte del código que atiende al dispositivo que causó la interrupción utilizando el número de esta. Cuando retorna, se activa de nuevo la interrupción tratada.

Llegado a este punto **HWINT_MASTER** termina con la instrucción ***ret***. El retorno no tiene por que ser al proceso que se estaba ejecutando cuando se produjo la interrupción.

HWINT_SLAVE es muy similar a **HWINT_MASTER**, excepto en que se debe habilitar tanto el controlador maestro como el esclavo.

En las macros anteriores se hace uso de la función **save** cuya principal función es salvar el contexto del proceso que fue interrumpido. Para ello usa la variable ***_k_reenter*** que cuenta y determina el nivel de anidamiento de las interrupciones (línea 6261). Si existe un proceso ejecutándose cuando ocurre la interrupción, se almacena el estado de este y se selecciona la pila del kernel, colocando la dirección de ***_restart***. En otro caso, la pila del kernel está ya en uso y se coloca la dirección de ***_restart1***. Se sale con un salto (lineas 6277 ó 6282) y no con una instrucción de retorno para evitar conflictos con las pilas.

Interrupciones software.-

El siguiente procedimiento es **_s_call**. El control llega cuando se produce una interrupción software. Estas son tratadas como las hardware, excepto en que el índice a la IDT se coge de la propia instrucción y no del

controlador. En este procedimiento también se guardan los registros necesarios, se pasa a la pila del kernel y se habilitan el resto de las interrupciones. Lo siguiente que se hace es llamar a **_sys_call** que convierte la interrupción en un mensaje. La ejecución continua con el procedimiento **_restart**.

Restart.-

Esta función comprueba si hay interrupciones no atendidas que llegaron mientras se estaba tratando la actual, y en caso de que existan llama a **unhold** para que sean tratadas antes de reanudar un proceso. Si la interrupción que se está tratando llegó mientras no había ningún proceso en ejecución, la primera parte de **_restart** no es necesaria, ya que la pila del kernel ya está en uso, y la ejecución continúa por la etiqueta ***restart1***. En este punto ***k_reenter*** es decrementado para registrar que un posible nivel de interrupción ha sido liberado y el siguiente conjunto de instrucciones restaura al procesador al estado del último proceso ejecutado.

Excepciones.-

A parte de las interrupciones hardware y software, varias condiciones de error internas a la CPU pueden causar la iniciación de una excepción.

Hay 16 puntos de entrada para el manejo de excepciones (líneas 6350 a 6412). Se utiliza el mismo mecanismo para manejar interrupciones y excepciones. Los puntos de entrada a interrupciones y excepciones se hace a través de la tabla de descriptores de interrupción (IDT) y todas llaman a ***save***.

Algunas de estas excepciones se ignoran, otras causan errores graves, y otras más causan el envío de señales a procesos. Si agrega un código de error a la pila, salta a ***errexception*** (línea 6431), sino salta a ***exception*** (línea 6420). Al final siempre llaman a la rutina, escrita en C, ***_exception.***

Código fuente del mpx386.s.-

```
!*===========================================================================*
!*                              MINIX                                      *
!*===========================================================================*
MINIX:                          ! this is the entry point for the MINIX kernel
        jmp    over_flags       ! skip over the next few bytes
        .data2 CLICK_SHIFT      ! for the monitor: memory granularity
flags:
        .data2 0x002D           ! boot monitor flags:
                                !      call in 386 mode, make stack,
                                !      load high, will return
        nop                     ! extra byte to sync up disassembler
over_flags:

! Set up a C stack frame on the monitor stack.  (The monitor sets cs and ds
! right.  The ss descriptor still references the monitor data segment.)
        movzx  esp, sp          ! monitor stack is a 16 bit stack
        push   ebp
        mov    ebp, esp
        push   esi
        push   edi
        cmp    4(ebp), 0        ! nonzero if return possible
        jz     noret
        inc    (_mon_return)
noret:  mov    (_mon_sp), esp ! save stack pointer for later return

! Copy the monitor global descriptor table to the address space of kernel and
! switch over to it.  Prot_init() can then update it with immediate effect.

        sgdt   (_gdt+GDT_SELECTOR)           ! get the monitor gdtr
        mov    esi, (_gdt+GDT_SELECTOR+2)    ! absolute address of GDT
        mov    ebx, _gdt                     ! address of kernel GDT
        mov    ecx, 8*8                      ! copying eight descriptors
copygdt:
 eseg   movb   al, (esi)
        movb   (ebx), al
        inc    esi
        inc    ebx
        loop   copygdt
        mov    eax, (_gdt+DS_SELECTOR+2)     ! base of kernel data
        and    eax, 0x00FFFFFF                      ! only 24 bits
        add    eax, _gdt                     ! eax = vir2phys(gdt)
        mov    (_gdt+GDT_SELECTOR+2), eax    ! set base of GDT
        lgdt   (_gdt+GDT_SELECTOR)           ! switch over to kernel GDT

! Locate boot parameters, set up kernel segment registers and stack.
        mov    ebx, 8(ebp)      ! boot parameters offset
        mov    edx, 12(ebp)     ! boot parameters length
        mov    ax, ds           ! kernel data
        mov    es, ax
        mov    fs, ax
        mov    gs, ax
        mov    ss, ax
        mov    esp, k_stktop  ! set sp to point to the top of kernel stack

! Call C startup code to set up a proper environment to run main().
        push   edx
        push   ebx
        push   SS_SELECTOR
        push   MON_CS_SELECTOR
        push   DS_SELECTOR
        push   CS_SELECTOR
        call   _cstart          ! cstart(cs, ds, mcs, mds, parmoff, parmlen)
        add    esp, 6*4

! Reload gdtr, idtr and the segment registers to global descriptor table set
! up by prot_init().

        lgdt   (_gdt+GDT_SELECTOR)
        lidt   (_gdt+IDT_SELECTOR)

        jmpf   CS_SELECTOR:csinit
csinit:
    o16 mov    ax, DS_SELECTOR
```

```
        mov     ds, ax
        mov     es, ax
        mov     fs, ax
        mov     gs, ax
        mov     ss, ax
    o16 mov     ax, TSS_SELECTOR      ! no other TSS is used
        ltr     ax
        push    0                     ! set flags to known good state
        popf                          ! esp, clear nested task and int enable

        jmp     _main                 ! main()


!*===========================================================================*
!*                          interrupt handlers                               *
!*              interrupt handlers for 386 32-bit protected mode             *
!*===========================================================================*

!*===========================================================================*
!*                              hwint00 - 07                                 *
!*===========================================================================*
! Note this is a macro, it looks like a subroutine.
#define hwint_master(irq)     \
        call    save                  /* save interrupted process state */;\
        inb     INT_CTLMASK                                               ;\
        orb     al, [1<<irq]                                              ;\
        outb    INT_CTLMASK           /* disable the irq              */;\
        movb    al, ENABLE                                                ;\
        outb    INT_CTL               /* reenable master 8259                 */;\
        sti                           /* enable interrupts            */;\
        push    irq                   /* irq                          */;\
        call    (_irq_table + 4*irq)  /* eax = (*irq_table[irq])(irq)   */;\
        pop     ecx                                                       ;\
        cli                           /* disable interrupts           */;\
        test    eax, eax              /* need to reenable irq?        */;\
        jz      0f                                                        ;\
        inb     INT_CTLMASK                                               ;\
        andb    al, ~[1<<irq]                                             ;\
        outb    INT_CTLMASK           /* enable the irq               */;\
0:      ret                           /* restart (another) process      */

! Each of these entry points is an expansion of the hwint_master macro
        .align  16
_hwint00:               ! Interrupt routine for irq 0 (the clock).
        hwint_master(0)

        .align  16
_hwint01:               ! Interrupt routine for irq 1 (keyboard)
        hwint_master(1)

        .align  16
_hwint02:               ! Interrupt routine for irq 2 (cascade!)
        hwint_master(2)

        .align  16
_hwint03:               ! Interrupt routine for irq 3 (second serial)
        hwint_master(3)

        .align  16
_hwint04:               ! Interrupt routine for irq 4 (first serial)
        hwint_master(4)

        .align  16
_hwint05:               ! Interrupt routine for irq 5 (XT winchester)
        hwint_master(5)

        .align  16
_hwint06:               ! Interrupt routine for irq 6 (floppy)
        hwint_master(6)

        .align  16
_hwint07:               ! Interrupt routine for irq 7 (printer)
        hwint_master(7)

!*===========================================================================*
!*                              hwint08 - 15                                 *
!*===========================================================================*
```

```
! Note this is a macro, it looks like a subroutine.
#define hwint_slave(irq)      \
        call    save                 /* save interrupted process state */;\
        inb     INT2_CTLMASK                                            ;\
        orb     al, [1<<[irq-8]]                                        ;\
        outb    INT2_CTLMASK         /* disable the irq              */;\
        movb    al, ENABLE                                              ;\
        outb    INT_CTL              /* reenable master 8259                */;\
        jmp     .+2                  /* delay                        */;\
        outb    INT2_CTL             /* reenable slave 8259          */;\
        sti                          /* enable interrupts            */;\
        push    irq                  /* irq                          */;\
        call    (_irq_table + 4*irq) /* eax = (*irq_table[irq])(irq)   */;\
        pop     ecx                                                     ;\
        cli                          /* disable interrupts           */;\
        test    eax, eax             /* need to reenable irq?        */;\
        jz      0f                                                      ;\
        inb     INT2_CTLMASK                                            ;\
        andb    al, ~[1<<[irq-8]]                                       ;\
        outb    INT2_CTLMASK         /* enable the irq               */;\
0:      ret                          /* restart (another) process      */

! Each of these entry points is an expansion of the hwint_slave macro
        .align  16
_hwint08:               ! Interrupt routine for irq 8 (realtime clock)
        hwint_slave(8)

        .align  16
_hwint09:               ! Interrupt routine for irq 9 (irq 2 redirected)
        hwint_slave(9)

        .align  16
_hwint10:               ! Interrupt routine for irq 10
        hwint_slave(10)

        .align  16
_hwint11:               ! Interrupt routine for irq 11
        hwint_slave(11)

        .align  16
_hwint12:               ! Interrupt routine for irq 12
        hwint_slave(12)

        .align  16
_hwint13:               ! Interrupt routine for irq 13 (FPU exception)
        hwint_slave(13)

        .align  16
_hwint14:               ! Interrupt routine for irq 14 (AT winchester)
        hwint_slave(14)

        .align  16
_hwint15:               ! Interrupt routine for irq 15
        hwint_slave(15)

!*===========================================================================*
!*                              save                                         *
!*===========================================================================*
! Save for protected mode.
! This is much simpler than for 8086 mode, because the stack already points
! into the process table, or has already been switched to the kernel stack.

        .align  16
save:
        cld                     ! set direction flag to a known value
        pushad                  ! save "general" registers
    o16 push    ds              ! save ds
    o16 push    es              ! save es
    o16 push    fs              ! save fs
    o16 push    gs              ! save gs
        mov     dx, ss          ! ss is kernel data segment
        mov     ds, dx          ! load rest of kernel segments
        mov     es, dx          ! kernel does not use fs, gs
        mov     eax, esp        ! prepare to return
        incb    (_k_reenter)    ! from -1 if not reentering
        jnz     set_restart1    ! stack is already kernel stack
        mov     esp, k_stktop
```

```
        push    _restart        ! build return address for int handler
        xor     ebp, ebp        ! for stacktrace
        jmp     RETADR-P_STACKBASE(eax)

        .align  4
set_restart1:
        push    restart1
        jmp     RETADR-P_STACKBASE(eax)

!*===========================================================================*
!*                              _s_call                                      *
!*===========================================================================*
        .align  16
_s_call:
_p_s_call:
        cld                     ! set direction flag to a known value
        sub     esp, 6*4        ! skip RETADR, eax, ecx, edx, ebx, est
        push    ebp             ! stack already points into proc table
        push    esi
        push    edi
    o16 push    ds
    o16 push    es
    o16 push    fs
    o16 push    gs
        mov     dx, ss
        mov     ds, dx
        mov     es, dx
        incb    (_k_reenter)
        mov     esi, esp        ! assumes P_STACKBASE == 0
        mov     esp, k_stktop
        xor     ebp, ebp        ! for stacktrace
                                ! end of inline save
        sti                     ! allow SWITCHER to be interrupted
                                ! now set up parameters for sys_call()
        push    ebx             ! pointer to user message
        push    eax             ! src/dest
        push    ecx             ! SEND/RECEIVE/BOTH
        call    _sys_call       ! sys_call(function, src_dest, m_ptr)
                                ! caller is now explicitly in proc_ptr
        mov     AXREG(esi), eax        ! sys_call MUST PRESERVE si
        cli                     ! disable interrupts

! Fall into code to restart proc/task running.

!*===========================================================================*
!*                              restart                                      *
!*===========================================================================*
_restart:

! Flush any held-up interrupts.
! This reenables interrupts, so the current interrupt handler may reenter.
! This does not matter, because the current handler is about to exit and no
! other handlers can reenter since flushing is only done when k_reenter == 0.

        cmp     (_held_head), 0        ! do fast test to usually avoid function call
        jz      over_call_unhold
        call    _unhold         ! this is rare so overhead acceptable
over_call_unhold:
        mov     esp, (_proc_ptr)       ! will assume P_STACKBASE == 0
        lldt    P_LDT_SEL(esp)         ! enable segment descriptors for task
        lea     eax, P_STACKTOP(esp)   ! arrange for next interrupt
        mov     (_tss+TSS3_S_SP0), eax! to save state in process table
restart1:
        decb    (_k_reenter)
    o16 pop     gs
    o16 pop     fs
    o16 pop     es
    o16 pop     ds
        popad
        add     esp, 4          ! skip return adr
        iretd                   ! continue process

!*===========================================================================*
!*                              exception handlers                           *
!*===========================================================================*
_divide_error:
        push    DIVIDE_VECTOR
```

```
        jmp     exception

_single_step_exception:
        push    DEBUG_VECTOR
        jmp     exception

_nmi:
        push    NMI_VECTOR
        jmp     exception

_breakpoint_exception:
        push    BREAKPOINT_VECTOR
        jmp     exception

_overflow:
        push    OVERFLOW_VECTOR
        jmp     exception

_bounds_check:
        push    BOUNDS_VECTOR
        jmp     exception

_inval_opcode:
        push    INVAL_OP_VECTOR
        jmp     exception

_copr_not_available:
        push    COPROC_NOT_VECTOR
        jmp     exception

_double_fault:
        push    DOUBLE_FAULT_VECTOR
        jmp     errexception

_copr_seg_overrun:
        push    COPROC_SEG_VECTOR
        jmp     exception

_inval_tss:
        push    INVAL_TSS_VECTOR
        jmp     errexception

_segment_not_present:
        push    SEG_NOT_VECTOR
        jmp     errexception

_stack_exception:
        push    STACK_FAULT_VECTOR
        jmp     errexception

_general_protection:
        push    PROTECTION_VECTOR
        jmp     errexception

_page_fault:
        push    PAGE_FAULT_VECTOR
        jmp     errexception

_copr_error:
        push    COPROC_ERR_VECTOR
        jmp     exception

!*===========================================================================*
!*                              exception                                    *
!*===========================================================================*
! This is called for all exceptions which do not push an error code.

        .align  16
exception:
 sseg   mov     (trap_errno), 0                 ! clear trap_errno
 sseg   pop     (ex_number)
        jmp     exception1

!*===========================================================================*
!*                              errexception                                 *
!*===========================================================================*
! This is called for all exceptions which push an error code.
```

```
        .align  16
errexception:
 sseg   pop     (ex_number)
 sseg   pop     (trap_errno)
exception1:                             ! Common for all exceptions.
        push    eax                     ! eax is scratch register
        mov     eax, 0+4(esp)           ! old eip
 sseg   mov     (old_eip), eax
        movzx   eax, 4+4(esp)           ! old cs
 sseg   mov     (old_cs), eax
        mov     eax, 8+4(esp)           ! old eflags
 sseg   mov     (old_eflags), eax
        pop     eax
        call    save
        push    (old_eflags)
        push    (old_cs)
        push    (old_eip)
        push    (trap_errno)
        push    (ex_number)
        call    _exception              ! (ex_number, trap_errno, old_eip,
                                        !       old_cs, old_eflags)
        add     esp, 5*4
        cli
        ret
```