

```
#
2 ! This file, mpx386.s, is included by mpx.s when Minix is compiled for
3 ! 32-bit Intel CPUs. The alternative mpx88.s is compiled for 16-bit CPUs.
4
5 ! This file is part of the lowest layer of the MINIX kernel. (The other part
6 ! is "proc.c".) The lowest layer does process switching and message handling.
7 ! Furthermore it contains the assembler startup code for Minix and the 32-bit
8 ! interrupt handlers. It cooperates with the code in "start.c" to set up a
9 ! good environment for main().
10
11 ! Every transition to the kernel goes through this file. Transitions to the
12 ! kernel may be nested. The initial entry may be with a system call (i.e.,
13 ! send or receive a message), an exception or a hardware interrupt; kernel
14 ! reentries may only be made by hardware interrupts. The count of reentries
15 ! is kept in "k_reenter". It is important for deciding whether to switch to
16 ! the kernel stack and for protecting the message passing code in "proc.c".
17
18 ! For the message passing trap, most of the machine state is saved in the
19 ! proc table. (Some of the registers need not be saved.) Then the stack is
20 ! switched to "k_stack", and interrupts are reenabled. Finally, the system
21 ! call handler (in C) is called. When it returns, interrupts are disabled
22 ! again and the code falls into the restart routine, to finish off held-up
23 ! interrupts and run the process or task whose pointer is in "proc_ptr".
24
25 ! Hardware interrupt handlers do the same, except (1) The entire state must
26 ! be saved. (2) There are too many handlers to do this inline, so the save
27 ! routine is called. A few cycles are saved by pushing the address of the
28 ! appropriate restart routine for a return later. (3) A stack switch is
29 ! avoided when the stack is already switched. (4) The (master) 8259 interrupt
30 ! controller is reenabled centrally in save(). (5) Each interrupt handler
31 ! masks its interrupt line using the 8259 before enabling (other unmasked)
32 ! interrupts, and unmask it after servicing the interrupt. This limits the
33 ! nest level to the number of lines and protects the handler from itself.
34
35 ! For communication with the boot monitor at startup time some constant
36 ! data are compiled into the beginning of the text segment. This facilitates
37 ! reading the data at the start of the boot process, since only the first
38 ! sector of the file needs to be read.
39
40 ! Some data storage is also allocated at the end of this file. This data
41 ! will be at the start of the data segment of the kernel and will be read
42 ! and modified by the boot monitor before the kernel starts.
43
44 ! sections
45
46 .sect .text
47 begtext:
48 .sect .rom
49 begrom:
50 .sect .data
51 begdata:
```

```
52 .sect .bss
53 begbss:
54
55 #include <minix/config.h>
56 #include <minix/const.h>
57 #include <minix/com.h>
58 #include <ibm/interrupt.h>
59 #include "const.h"
60 #include "protect.h"
61 #include "sconst.h"
62
63 /* Selected 386 tss offsets. */
64 #define TSS3_S_SP0    4
65
66 ! Exported functions
67 ! Note: in assembly language the .define statement applied to a function name
68 ! is loosely equivalent to a prototype in C code -- it makes it possible to
69 ! link to an entity declared in the assembly code but does not create
70 ! the entity.
71
72 .define _restart
73 .define save
74
75 .define _divide_error
76 .define _single_step_exception
77 .define _nmi
78 .define _breakpoint_exception
79 .define _overflow
80 .define _bounds_check
81 .define _inval_opcode
82 .define _copr_not_available
83 .define _double_fault
84 .define _copr_seg_overrun
85 .define _inval_tss
86 .define _segment_not_present
87 .define _stack_exception
88 .define _general_protection
89 .define _page_fault
90 .define _copr_error
91
92 .define _hwint00    ! handlers for hardware interrupts
93 .define _hwint01
94 .define _hwint02
95 .define _hwint03
96 .define _hwint04
97 .define _hwint05
98 .define _hwint06
99 .define _hwint07
100 .define _hwint08
101 .define _hwint09
102 .define _hwint10
```

```

103 .define _hwint11
104 .define _hwint12
105 .define _hwint13
106 .define _hwint14
107 .define _hwint15
108
109 .define _s_call
110 .define _p_s_call
111 .define _level0_call
112
113 ! Exported variables.
114 .define begbss
115 .define begdata
116
117 .sect .text
118 !*=====
119 !*          MINIX          *
120 !*=====
121 MINIX:          ! this is the entry point for the MINIX kernel
122   jmp  over_flags ! skip over the next few bytes
123   .data2 CLICK_SHIFT ! for the monitor: memory granularity
124 flags:
125   .data2 0x01FD ! boot monitor flags:
126           ! call in 386 mode, make bss, make stack,
127           ! load high, don't patch, will return,
128           ! uses generic INT, memory vector,
129           ! new boot code return
130   nop ! extra byte to sync up disassembler
131 over_flags:
132
133 ! Set up a C stack frame on the monitor stack. (The monitor sets cs and ds
134 ! right. The ss descriptor still references the monitor data segment.)
135   movzx esp, sp ! monitor stack is a 16 bit stack
136   push ebp
137   mov  ebp, esp
138   push esi
139   push edi
140   cmp  4(ebp), 0 ! monitor return vector is
141   jz  noret ! nonzero if return possible
142   inc  (_mon_return)
143 noret: mov  (_mon_sp), esp ! save stack pointer for later return
144
145 ! Copy the monitor global descriptor table to the address space of kernel and
146 ! switch over to it. Prot_init() can then update it with immediate effect.
147
148   sgdt  (_gdt+GDT_SELECTOR) ! get the monitor gdt
149   mov  esi, (_gdt+GDT_SELECTOR+2) ! absolute address of GDT
150   mov  ebx, _gdt ! address of kernel GDT
151   mov  ecx, 8*8 ! copying eight descriptors
152 copygdt:
153   eseg movb al, (esi)

```

```

154  movb (ebx), al
155  inc  esi
156  inc  ebx
157  loop copygdt
158  mov  eax, (_gdt+DS_SELECTOR+2)  ! base of kernel data
159  and  eax, 0x00FFFFFF           ! only 24 bits
160  add  eax, _gdt                 ! eax = vir2phys(gdt)
161  mov  (_gdt+GDT_SELECTOR+2), eax  ! set base of GDT
162  lgdt (_gdt+GDT_SELECTOR)       ! switch over to kernel GDT
163
164 ! Locate boot parameters, set up kernel segment registers and stack.
165  mov  ebx, 8(ebp)  ! boot parameters offset
166  mov  edx, 12(ebp) ! boot parameters length
167  mov  eax, 16(ebp) ! address of a.out headers
168  mov  (_aout), eax
169  mov  ax, ds      ! kernel data
170  mov  es, ax
171  mov  fs, ax
172  mov  gs, ax
173  mov  ss, ax
174  mov  esp, k_stktop ! set sp to point to the top of kernel stack
175
176 ! Call C startup code to set up a proper environment to run main().
177  push  edx
178  push  ebx
179  push  SS_SELECTOR
180  push  DS_SELECTOR
181  push  CS_SELECTOR
182  call _cstart  ! cstart(cs, ds, mds, parmoff, parmlen)
183  add  esp, 5*4
184
185 ! Reload gdt, idtr and the segment registers to global descriptor table set
186 ! up by prot_init().
187
188  lgdt (_gdt+GDT_SELECTOR)
189  lidt (_gdt+IDT_SELECTOR)
190
191  jmpf CS_SELECTOR:csinit
192 csinit:
193  o16 mov  ax, DS_SELECTOR
194  mov  ds, ax
195  mov  es, ax
196  mov  fs, ax
197  mov  gs, ax
198  mov  ss, ax
199  o16 mov  ax, TSS_SELECTOR  ! no other TSS is used
200  ltr  ax
201  push  0                    ! set flags to known good state
202  popf                       ! esp, clear nested task and int enable
203
204  jmp  _main                 ! main()

```

```

205
206
207 !*=====
208 !*          interrupt handlers          *
209 !*          interrupt handlers for 386 32-bit protected mode      *
210 !*=====
211
212 !*=====
213 !*          hwint00 - 07          *
214 !*=====
215 ! Note this is a macro, it just looks like a subroutine.
216 #define hwint_master(irq)  \
217     call save          /* save interrupted process state */;\
218     push  (_irq_handlers+4*irq) /* irq_handlers[irq] */;\
219     call _intr_handle   /* intr_handle(irq_handlers[irq]) */;\
220     pop   ecx          ;\
221     cmp  (_irq_actids+4*irq), 0 /* interrupt still active? */;\
222     jz   0f            ;\
223     inb INT_CTLMASK    /* get current mask */      ;\
224     orb  al, [1<<irq] /* mask irq */              ;\
225     outb INT_CTLMASK   /* disable the irq */;\
226 0:   movb al, END_OF_INT ;\
227     outb INT_CTL      /* reenable master 8259 */;\
228     ret              /* restart (another) process */
229
230 ! Each of these entry points is an expansion of the hwint_master macro
231     .align 16
232 _hwint00:      ! Interrupt routine for irq 0 (the clock).
233     hwint_master(0)
234
235     .align 16
236 _hwint01:      ! Interrupt routine for irq 1 (keyboard)
237     hwint_master(1)
238
239     .align 16
240 _hwint02:      ! Interrupt routine for irq 2 (cascade!)
241     hwint_master(2)
242
243     .align 16
244 _hwint03:      ! Interrupt routine for irq 3 (second serial)
245     hwint_master(3)
246
247     .align 16
248 _hwint04:      ! Interrupt routine for irq 4 (first serial)
249     hwint_master(4)
250
251     .align 16
252 _hwint05:      ! Interrupt routine for irq 5 (XT winchester)
253     hwint_master(5)
254
255     .align 16

```

```

256 _hwint06:      ! Interrupt routine for irq 6 (floppy)
257   hwint_master(6)
258
259   .align 16
260 _hwint07:      ! Interrupt routine for irq 7 (printer)
261   hwint_master(7)
262
263 !*=====
264 !*           hwint08 - 15           *
265 !*=====
266 ! Note this is a macro, it just looks like a subroutine.
267 #define hwint_slave(irq)  \
268   call save           /* save interrupted process state */;\
269   push  (_irq_handlers+4*irq) /* irq_handlers[irq]           */;\
270   call _intr_handle   /* intr_handle(irq_handlers[irq]) */;\
271   pop   ecx           ;\
272   cmp  (_irq_actids+4*irq), 0 /* interrupt still active? */;\
273   jz   0f             ;\
274   inb  INT2_CTLMASK   ;\
275   orb  al, [1<<[irq-8]] ;\
276   outb INT2_CTLMASK   /* disable the irq           */;\
277 0:   movb al, END_OF_INT ;\
278   outb INT_CTL       /* reenable master 8259           */;\
279   outb INT2_CTL      /* reenable slave 8259           */;\
280   ret                /* restart (another) process */
281
282 ! Each of these entry points is an expansion of the hwint_slave macro
283   .align 16
284 _hwint08:      ! Interrupt routine for irq 8 (realtime clock)
285   hwint_slave(8)
286
287   .align 16
288 _hwint09:      ! Interrupt routine for irq 9 (irq 2 redirected)
289   hwint_slave(9)
290
291   .align 16
292 _hwint10:      ! Interrupt routine for irq 10
293   hwint_slave(10)
294
295   .align 16
296 _hwint11:      ! Interrupt routine for irq 11
297   hwint_slave(11)
298
299   .align 16
300 _hwint12:      ! Interrupt routine for irq 12
301   hwint_slave(12)
302
303   .align 16
304 _hwint13:      ! Interrupt routine for irq 13 (FPU exception)
305   hwint_slave(13)
306

```

```

307     .align 16
308 _hwint14:      ! Interrupt routine for irq 14 (AT winchester)
309     hwint_slave(14)
310
311     .align 16
312 _hwint15:      ! Interrupt routine for irq 15
313     hwint_slave(15)
314
315 !*=====
316 !*          save          *
317 !*=====
318 ! Save for protected mode.
319 ! This is much simpler than for 8086 mode, because the stack already points
320 ! into the process table, or has already been switched to the kernel stack.
321
322     .align 16
323 save:
324     cld          ! set direction flag to a known value
325     pushad       ! save "general" registers
326     o16 push ds   ! save ds
327     o16 push es   ! save es
328     o16 push fs   ! save fs
329     o16 push gs   ! save gs
330     mov  dx,ss    ! ss is kernel data segment
331     mov  ds,dx    ! load rest of kernel segments
332     mov  es,dx    ! kernel does not use fs, gs
333     mov  eax,esp  ! prepare to return
334     incb (_k_reenter) ! from -1 if not reentering
335     jnz  set_restart1 ! stack is already kernel stack
336     mov  esp,k_stktop
337     push _restart  ! build return address for int handler
338     xor  ebp,ebp  ! for stacktrace
339     jmp  RETADR-P_STACKBASE(eax)
340
341     .align 4
342 set_restart1:
343     push restart1
344     jmp  RETADR-P_STACKBASE(eax)
345
346 !*=====
347 !*          _s_call      *
348 !*=====
349     .align 16
350 _s_call:
351 _p_s_call:
352     cld          ! set direction flag to a known value
353     sub  esp,6*4  ! skip RETADR, eax, ecx, edx, ebx, est
354     push ebp      ! stack already points into proc table
355     push esi
356     push edi
357     o16 push ds

```

```

358 o16 push es
359 o16 push fs
360 o16 push gs
361 mov dx, ss
362 mov ds, dx
363 mov es, dx
364 incb (_k_reenter)
365 mov esi, esp ! assumes P_STACKBASE == 0
366 mov esp, k_stktop
367 xor ebp, ebp ! for stacktrace
368 ! end of inline save
369 ! now set up parameters for sys_call()
370 push ebx ! pointer to user message
371 push eax ! src/dest
372 push ecx ! SEND/RECEIVE/BOTH
373 call _sys_call ! sys_call(function, src_dest, m_ptr)
374 ! caller is now explicitly in proc_ptr
375 mov AXREG(esi), eax ! sys_call MUST PRESERVE si
376
377 ! Fall into code to restart proc/task running.
378
379 !*=====*
380 !* restart *
381 !*=====*
382 _restart:
383
384 ! Restart the current process or the next process if it is set.
385
386 cmp (_next_ptr), 0 ! see if another process is scheduled
387 jz 0f
388 mov eax, (_next_ptr)
389 mov (_proc_ptr), eax ! schedule new process
390 mov (_next_ptr), 0
391 0: mov esp, (_proc_ptr) ! will assume P_STACKBASE == 0
392 lldt P_LDT_SEL(esp) ! enable process' segment descriptors
393 lea eax, P_STACKTOP(esp) ! arrange for next interrupt
394 mov (_tss+TSS3_S_SP0), eax ! to save state in process table
395 restart1:
396 decb (_k_reenter)
397 o16 pop gs
398 o16 pop fs
399 o16 pop es
400 o16 pop ds
401 popad
402 add esp, 4 ! skip return adr
403 iretd ! continue process
404
405 !*=====*
406 !* exception handlers *
407 !*=====*
408 _divide_error:

```

```
409    push  DIVIDE_VECTOR
410    jmp   exception
411
412 _single_step_exception:
413    push  DEBUG_VECTOR
414    jmp   exception
415
416 _nmi:
417    push  NMI_VECTOR
418    jmp   exception
419
420 _breakpoint_exception:
421    push  BREAKPOINT_VECTOR
422    jmp   exception
423
424 _overflow:
425    push  OVERFLOW_VECTOR
426    jmp   exception
427
428 _bounds_check:
429    push  BOUNDS_VECTOR
430    jmp   exception
431
432 _inval_opcode:
433    push  INVALID_OP_VECTOR
434    jmp   exception
435
436 _copr_not_available:
437    push  COPROC_NOT_VECTOR
438    jmp   exception
439
440 _double_fault:
441    push  DOUBLE_FAULT_VECTOR
442    jmp   errexception
443
444 _copr_seg_overrun:
445    push  COPROC_SEG_VECTOR
446    jmp   exception
447
448 _inval_tss:
449    push  INVALID_TSS_VECTOR
450    jmp   errexception
451
452 _segment_not_present:
453    push  SEG_NOT_VECTOR
454    jmp   errexception
455
456 _stack_exception:
457    push  STACK_FAULT_VECTOR
458    jmp   errexception
459
```

```

460 _general_protection:
461     push  PROTECTION_VECTOR
462     jmp   errexception
463
464 _page_fault:
465     push  PAGE_FAULT_VECTOR
466     jmp   errexception
467
468 _copr_error:
469     push  COPROC_ERR_VECTOR
470     jmp   exception
471
472 !*=====
473 !*           exception                *
474 !*=====
475 ! This is called for all exceptions which do not push an error code.
476
477     .align 16
478 exception:
479 sseg mov  (trap_errno), 0    ! clear trap_errno
480 sseg pop  (ex_number)
481     jmp   exception1
482
483 !*=====
484 !*           errexception            *
485 !*=====
486 ! This is called for all exceptions which push an error code.
487
488     .align 16
489 errexception:
490 sseg pop  (ex_number)
491 sseg pop  (trap_errno)
492 exception1:    ! Common for all exceptions.
493     push  eax            ! eax is scratch register
494     mov   eax, 0+4(esp)  ! old eip
495 sseg mov  (old_eip), eax
496     movzx eax, 4+4(esp)  ! old cs
497 sseg mov  (old_cs), eax
498     mov   eax, 8+4(esp) ! old eflags
499 sseg mov  (old_eflags), eax
500     pop   eax
501     call  save
502     push (old_eflags)
503     push (old_cs)
504     push (old_eip)
505     push (trap_errno)
506     push (ex_number)
507     call _exception    ! (ex_number, trap_errno, old_eip,
508                       !   old_cs, old_eflags)
509     add  esp, 5*4
510     ret

```

```
511
512 !*=====
513 !*          level0_call          *
514 !*=====
515 _level0_call:
516     call save
517     jmp  (_level0_func)
518
519 !*=====
520 !*          data          *
521 !*=====
522
523 .sect .rom    ! Before the string table please
524     .data2 0x526F    ! this must be the first data entry (magic #)
525
526 .sect .bss
527 k_stack:
528     .space K_STACK_BYTES ! kernel stack
529 k_stktop:      ! top of kernel stack
530     .comm ex_number, 4
531     .comm trap_errno, 4
532     .comm old_eip, 4
533     .comm old_cs, 4
534     .comm old_eflags, 4
```