```
25 :
26 : static int I,
27 :     waitchr = 0,
28 :     vflag = False,
29 :     capbp,
30 :     capbc,
31 :     Ch,
32 :     Want_7_Bit = True,
33 :     ESC_Seq_State = 0;        /* escape sequence state variable     */
34 :
35 : int _cx ,
36 :      _cy,
37 :      _atr = 0x07,                      /* white on black           */
38 :      _pag = 0,
39 :      oldtop = 0,
40 :      oldbot = 0x184f;
41 :
42 : FILE * in_file = NULL;          /* start with keyboard input        */
43 : FILE * cap_file = NULL;
44 :
45 : #include "cterm.h"              /* external declarations, etc.      */
46 :
47 : int Wants_To_Abort ()           /* checks for interrupt of script   */
48 : { return broke ();
49 : }
50 : void
51 :
52 : main ( argc, argv ) int argc ; /* main routine                     */
53 :  char * argv [];
54 : { char * cp,
55 :      * addext ();
56 :   if ( argc > 1 )                 /* check for script filename      */
57 :     in_file = fopen ( addext ( argv [ 1 ], ".SCR" ), "r" );
58 :   if ( argc > 2 )                 /* check for capture filename     */
59 :     cap_file = fopen ( addext ( argv [ 2 ], ".CAP" ), "w" );
60 :   set_int ();                     /* install CH1 module             */
61 :   Set_Vid ();                     /* get video setup                */
62 :   cls ();                         /* clear the screen               */
63 :   cputs ( "Terminal Emulator" ); /* tell who's working              */
64 :   cputs ( "\r\n< ESC for local commands >\r\n\n" );
65 :   Want_7_Bit = True;
66 :   ESC_Seq_State = 0;
67 :   Init_Comm ();                   /* set up drivers, etc.           */
68 :   while ( 1 )                           /* main loop                */
69 :     { if (( Ch = kb_file ()) > 0 )      /* check local              */
70 :         { if ( Is_Function_Key ( Ch ))
71 :             { if ( docmd () < 0 )       /* command                  */
72 :                 break;
73 :             }
74 :           else
75 :             Send_Byte ( Ch & 0x7F );    /* else send it             */
```

Figure 6-8. Continued.

(more)

```
76 :          }
77 :        if (( Ch = Read_Modem ()) >= 0 )  /* check remote              */
78 :          { if ( Want_7_Bit )
79 :              Ch &= 0x7F;                 /* trim off high bit         */
80 :            switch ( ESC_Seq_State )      /* state machine             */
81 :              {
82 :              case 0 :                    /* no Esc sequence           */
83 :                switch ( Ch )
84 :                  {
85 :                  case ESC  :             /* Esc char received         */
86 :                    ESC_Seq_State = 1;
87 :                    break;
88 :
89 :                  default :
90 :                    if ( Ch == waitchr )  /* wait if required          */
91 :                      waitchr = 0;
92 :                    if ( Ch == 12 )       /* clear screen on FF        */
93 :                      cls ();
94 :                    else
95 :                      if ( Ch != 127 )    /* ignore rubouts            */
96 :                        { putchx ( (char) Ch );  /* handle all others */
97 :                          put_cap ( (char) Ch );
98 :                        }
99 :                  }
100 :               break;
101 :
102 :             case 1 : /* ESC -- process any escape sequences here    */
103 :               switch ( Ch )
104 :                 {
105 :                 case 'A' :              /* VT52 up                   */
106 :                   ;                     /* nothing but stubs here    */
107 :                   ESC_Seq_State = 0;
108 :                   break;
109 :
110 :                 case 'B' :              /* VT52 down                 */
111 :                   ;
112 :                   ESC_Seq_State = 0;
113 :                   break;
114 :
115 :                 case 'C' :              /* VT52 left                 */
116 :                   ;
117 :                   ESC_Seq_State = 0;
118 :                   break;
119 :
120 :                 case 'D' :              /* VT52 right                */
121 :                   ;
122 :                   ESC_Seq_State = 0;
123 :                   break;
124 :
125 :                 case 'E' :              /* VT52 Erase CRT            */
126 :                   cls ();               /* actually do this one      */
```

Figure 6-8. Continued. *(more)*

```
127 :                 ESC_Seq_State = 0;
128 :                 break;
129 :
130 :               case 'H' :              /* VT52 home cursor          */
131 :                 locate ( 0, 0 );
132 :                 ESC_Seq_State = 0;
133 :                 break;
134 :
135 :               case 'j' :              /* VT52 Erase to EOS         */
136 :                 deos ();
137 :                 ESC_Seq_State = 0;
138 :                 break;
139 :
140 :               case '[' :      /* ANSI.SYS - VT100 sequence         */
141 :                 ESC_Seq_State = 2;
142 :                 break;
143 :
144 :               default :
145 :                 putchx ( ESC );       /* pass thru all others      */
146 :                 putchx ( (char) Ch );
147 :                 ESC_Seq_State = 0;
148 :               }
149 :             break;
150 :
151 :           case 2 :                    /* ANSI 3.64 decoder         */
152 :             ESC_Seq_State = 0;        /* not implemented           */
153 :           }
154 :       }
155 :     if ( broke ())                    /* check CH1A handlers       */
156 :       { cputs ( "\r\n***BREAK***\r\n" );
157 :         break;
158 :       }
159 :   }                                   /* end of main loop          */
160 :   if ( cap_file )                     /* save any capture          */
161 :     cap_flush ();
162 :   Term_Comm ();                       /* restore when done         */
163 :   rst_int ();                         /* restore break handlers    */
164 :   exit ( 0 );                         /* be nice to MS-DOS         */
165 : }
166 :
167 : docmd ()                             /* local command shell       */
168 : { FILE * getfil ();
169 :   int wp;
170 :   wp = True;
171 :   if ( ! in_file || vflag )
172 :     cputs ( "\r\n\tCommand: " );      /* ask for command           */
173 :   else
174 :     wp = False;
175 :   Ch = toupper ( kbd_wait ());        /* get response              */
176 :   if ( wp )
177 :     putchx ( (char) Ch );
```

Figure 6-8. Continued.

(more)

```
178 :   switch  ( Ch )                          /* and act on it            */
179 :     {
180 :     case 'S' :
181 :       if ( wp )
182 :         cputs ( "low speed\r\n" );
183 :       Set_Baud ( 300 );
184 :       break;
185 :
186 :     case 'D' :
187 :       if ( wp )
188 :         cputs ( "elay (1-9 sec): " );
189 :       Ch = kbd_wait ();
190 :       if ( wp )
191 :         putchx ( (char) Ch );
192 :       Delay ( 1000 * ( Ch - '0' ));
193 :       if ( wp )
194 :         putchx ( '\n' );
195 :       break;
196 :
197 :     case 'E' :
198 :       if ( wp )
199 :         cputs ( "ven Parity\r\n" );
200 :       Set_Parity ( 2 );
201 :       break;
202 :
203 :     case 'F' :
204 :       if ( wp )
205 :         cputs ( "ast speed\r\n" );
206 :       Set_Baud ( 1200 );
207 :       break;
208 :
209 :     case 'H' :
210 :       if ( wp )
211 :         { cputs ( "\r\n\tVALID COMMANDS:\r\n" );
212 :           cputs ( "\tD = delay 0-9 seconds.\r\n" );
213 :           cputs ( "\tE = even parity.\r\n" );
214 :           cputs ( "\tF = (fast) 1200-baud.\r\n" );
215 :           cputs ( "\tN = no parity.\r\n" );
216 :           cputs ( "\tO = odd parity.\r\n" );
217 :           cputs ( "\tQ = quit, return to DOS.\r\n" );
218 :           cputs ( "\tR = reset modem.\r\n" );
219 :           cputs ( "\tS = (slow) 300-baud.\r\n" );
220 :           cputs ( "\tU = use script file.\r\n" );
221 :           cputs ( "\tV = verify file input.\r\n" );
222 :           cputs ( "\tW = wait for char." );
223 :         }
224 :       break;
225 :
226 :     case 'N' :
227 :       if ( wp )
```

Figure 6-8. Continued. *(more)*

```
228 :         cputs ( "o Parity\r\n" );
229 :       Set_Parity ( 1 );
230 :       break;
231 :
232 :     case 'O' :
233 :       if ( wp )
234 :         cputs ( "dd Parity\r\n" );
235 :       Set_Parity ( 3 );
236 :       break;
237 :
238 :     case 'R' :
239 :       if ( wp )
240 :         cputs ( "ESET Comm Port\r\n" );
241 :       Init_Comm ();
242 :       break;
243 :
244 :     case 'Q' :
245 :       if ( wp )
246 :         cputs ( " = QUIT Command\r\n" );
247 :       Ch = ( ~ 1 );
248 :       break;
249 :
250 :     case 'U' :
251 :       if ( in_file && ! vflag )
252 :         putchx ( 'U' );
253 :       cputs ( "se file: " );
254 :       getfil ();
255 :       cputs ( "File " );
256 :       cputs ( in_file ? "Open\r\n" : "Bad\r\n" );
257 :       waitchr = 0;
258 :       break;
259 :
260 :     case 'V' :
261 :       if ( wp )
262 :         { cputs ( "erify flag toggled " );
263 :           cputs ( vflag ? "OFF\r\n" : "ON\r\n" );
264 :         }
265 :       vflag = vflag ? False : True;
266 :       break;
267 :
268 :     case 'W' :
269 :       if ( wp )
270 :         cputs ( "ait for: <" );
271 :       waitchr = kbd_wait ();
272 :       if ( waitchr == ' ' )
273 :         waitchr = 0;
274 :       if ( wp )
275 :         { if ( waitchr )
276 :             putchx ( (char) waitchr );
277 :           else
278 :             cputs ( "no wait" );
```

Figure 6-8. Continued.

(more)

```
279 :           cputs ( ">\r\n" );
280 :         }
281 :       break;
282 :
283 :     default :
284 :       if ( wp )
285 :         { cputs ( "Don't know " );
286 :           putchx ( (char) Ch );
287 :           cputs ( "\r\nUse 'H' command for Help.\r\n" );
288 :         }
289 :       Ch = '?';
290 :     }
291 :   if ( wp )                              /* if window open....       */
292 :     { cputs ( "\r\n[any key]\r" );
293 :       while ( Read_Keyboard () == EOF ) /* wait for response        */
294 :         ;
295 :     }
296 :   return Ch ;
297 : }
298 :
299 : kbd_wait ()                              /* wait for input           */
300 : { int c ;
301 :   while (( c = kb_file ()) == ( - 1 ))
302 :     ;
303 :   return c & 255;
304 : }
305 :
306 : kb_file ()                               /* input from kb or file    */
307 : { int c ;
308 :   if ( in_file )                         /* USING SCRIPT             */
309 :     { c = Wants_To_Abort ();             /* use first as flag        */
310 :       if ( waitchr && ! c )
311 :         c = ( - 1 );                     /* then for char            */
312 :       else
313 :         if ( c || ( c = getc ( in_file )) == EOF || c == 26 )
314 :           { fclose ( in_file );
315 :             cputs ( "\r\nScript File Closed\r\n" );
316 :             in_file = NULL;
317 :             waitchr = 0;
318 :             c = ( - 1 );
319 :           }
320 :       else
321 :         if ( c == '\n' )                 /* ignore LFs in file       */
322 :           c = ( - 1 );
323 :       if ( c == '\\' )                   /* process Esc sequence     */
324 :         c = esc ();
325 :       if ( vflag && c != ( - 1 ))        /* verify file char         */
326 :         { putchx ( '{' );
327 :           putchx ( (char) c );
328 :           putchx ( '}' );
329 :         }
```

Figure 6-8. Continued. *(more)*

```
330 :     }
331 :   else                                  /* USING CONSOLE          */
332 :     c = Read_Keyboard ();               /* if not using file      */
333 :   return ( c );
334 : }
335 :
336 : esc ()                                  /* script translator      */
337 : { int c ;
338 :   c = getc ( in_file );                 /* control chars in file  */
339 :   switch ( toupper ( c ))
340 :     {
341 :     case 'E' :
342 :       c = ESC;
343 :       break;
344 :
345 :     case 'N' :
346 :       c = '\n';
347 :       break;
348 :
349 :     case 'R' :
350 :       c = '\r';
351 :       break;
352 :
353 :     case 'T' :
354 :       c = '\t';
355 :       break;
356 :
357 :     case '^' :
358 :       c = getc ( in_file ) & 31;
359 :       break;
360 :     }
361 :   return ( c );
362 : }
363 :
364 : FILE * getfil ()
365 : { char fnm [ 20 ];
366 :   getnam ( fnm, 15 );                   /* get the name           */
367 :   if ( ! ( strchr ( fnm, '.' )))
368 :     strcat ( fnm, ".SCR" );
369 :   return ( in_file = fopen ( fnm, "r" ));
370 : }
371 :
372 : void getnam ( b, s ) char * b;          /* take input to buffer   */
373 :  int s ;
374 : { while ( s -- > 0 )
375 :     { if (( * b = (char) kbd_wait ()) != '\r' )
376 :         putchx ( * b ++ );
377 :       else
378 :         break ;
379 :     }
380 :   putchx ( '\n' );
```

Figure 6-8. Continued.

(more)

```
381 :    * b = 0;
382 : }
383 :
384 : char * addext ( b,                          /* add default EXTension    */
385 :      e ) char * b,
386 :      * e;
387 : { static char bfr [ 20 ];
388 :   if ( strchr ( b, '.' ))
389 :     return ( b );
390 :   strcpy ( bfr, b );
391 :   strcat ( bfr, e );
392 :   return ( bfr );
393 : }
394 :
395 : void put_cap ( c ) char c ;
396 : { if ( cap_file && c != 13 )              /* strip out CRs            */
397 :     fputc ( c, cap_file );                /* use MS-DOS buffering     */
398 : }
399 :
400 : void cap_flush ()                         /* end Capture mode         */
401 : { if ( cap_file )
402 :     { fclose ( cap_file );
403 :       cap_file = NULL;
404 :       cputs ( "\r\nCapture file closed\r\n" );
405 :     }
406 : }
407 :
408 : /*      TIMER SUPPORT STUFF (IBMPC/MSDOS)       */
409 : static long timr;                         /* timeout register         */
410 :
411 : static union REGS rgv ;
412 :
413 : long getmr ()
414 : { long now ;                              /* msec since midnite       */
415 :   rgv.x.ax = 0x2c00;
416 :   intdos ( & rgv, & rgv );
417 :   now = rgv.h.ch;                         /* hours                    */
418 :   now *= 60L;                             /* to minutes               */
419 :   now += rgv.h.cl;                        /* plus min                 */
420 :   now *= 60L;                             /* to seconds               */
421 :   now += rgv.h.dh;                        /* plus sec                 */
422 :   now *= 100L;                            /* to 1/100                 */
423 :   now += rgv.h.dl;                        /* plus 1/100               */
424 :   return ( 10L * now );                   /* msec value               */
425 : }
426 :
427 : void Delay ( n ) int n ;                  /* sleep for n msec         */
428 : { long wakeup ;
429 :   wakeup = getmr () + ( long ) n;         /* wakeup time              */
430 :   while ( getmr () < wakeup )
431 :     ;                                     /* now sleep                */
```

Figure 6-8. Continued. *(more)*

```
}

void Start_Timer ( n ) int n ;          /* set timeout for n sec      */
{ timr = getmr () + ( long ) n * 1000L;
}

Timer_Expired ()         /* if timeout return 1 else return 0         */
{ return ( getmr () > timr );
}

Set_Vid ()
{ _i_v ();                                  /* initialize video       */
  return 0;
}

void locate ( row, col ) int row ,
     col;
{ _cy = row % 25;
  _cx = col % 80;
  _wrpos ( row, col );                      /* use ML from CH2.ASM    */
}

void deol ()
{ _deol ();                                 /* use ML from CH2.ASM    */
}

void deos ()
{ deol ();
  if ( _cy < 24 )                           /* if not last, clear     */
    { rgv.x.ax = 0x0600;
      rgv.x.bx = ( _atr << 8 );
      rgv.x.cx = ( _cy + 1 ) << 8;
      rgv.x.dx = 0x184F;
      int86 ( 0x10, & rgv, & rgv );
    }
  locate ( _cy, _cx );
}

void cls ()
{ _cls ();                                  /* use ML                 */
}

void cursor ( yn ) int yn ;
{ rgv.x.cx = yn ? 0x0607 : 0x2607;          /* ON/OFF                 */
  rgv.x.ax = 0x0100;
  int86 ( 0x10, & rgv, & rgv );
}

void revvid ( yn ) int yn ;
{ if ( yn )
    _atr = _color ( 8, 7 );                 /* black on white         */
```

Figure 6-8. Continued.

(more)

```
483 :   else
484 :     _atr = _color ( 15, 0 );            /* white on black          */
485 : }
486 :
487 : putchx ( c ) char c ;                    /* put char to CRT         */
488 : { if ( c == '\n' )
489 :     putch ( '\r' );
490 :   putch ( c );
491 :   return c ;
492 : }
493 :
494 : Read_Keyboard ()                      /* get keyboard character
495 :                                          returns -1 if none present  */
496 : { int c ;
497 :   if ( kbhit ())                        /* no char at all          */
498 :     return ( getch ());
499 :   return ( EOF );
500 : }
501 :
502 : /*      MODEM SUPPORT                   */
503 : static char mparm,
504 :      wrk [ 80 ];
505 :
506 : void Init_Comm ()                  /* initialize comm port stuff      */
507 : { static int ft = 0;                  /* firstime flag           */
508 :   if ( ft ++ == 0 )
509 :     i_m ();
510 :   Set_Parity ( 1 );                   /* 8,N,1                   */
511 :   Set_Baud ( 1200 );                  /* 1200 baud               */
512 : }
513 :
514 : #define B1200 0x80                    /* baudrate codes          */
515 : #define B300 0x40
516 :
517 : Set_Baud ( n ) int n ;                /* n is baud rate          */
518 : { if ( n == 300 )
519 :     mparm = ( mparm & 0x1F ) + B300;
520 :   else
521 :     if ( n == 1200 )
522 :       mparm = ( mparm & 0x1F ) + B1200;
523 :   else
524 :     return 0;                         /* invalid speed           */
525 :   sprintf ( wrk, "Baud rate = %d\r\n", n );
526 :   cputs ( wrk );
527 :   set_mdm ( mparm );
528 :   return n ;
529 : }
530 :
531 : #define PAREVN 0x18                   /* MCR bits for commands   */
532 : #define PARODD 0x10
533 : #define PAROFF 0x00
```

Figure 6-8. Continued. *(more)*

```
534 : #define STOP2 0x40
535 : #define WORD8 0x03
536 : #define WORD7 0x02
537 : #define WORD6 0x01
538 :
539 : Set_Parity ( n ) int n ;                  /* n is parity code          */
540 : { static int mmode;
541 :   if ( n == 1 )
542 :     mmode = ( WORD8 ¦ PAROFF );           /* off                       */
543 :   else
544 :     if ( n == 2 )
545 :       mmode = ( WORD7 ¦ PAREVN );         /* on and even               */
546 :   else
547 :     if ( n == 3 )
548 :       mmode = ( WORD7 ¦ PARODD );         /* on and odd                */
549 :   else
550 :     return 0;                             /* invalid code              */
551 :   mparm = ( mparm & 0xE0 ) + mmode;
552 :   sprintf ( wrk, "Parity is %s\r\n", ( n == 1 ? "OFF" :
553 :                                      ( n == 2 ? "EVEN" : "ODD" )));
554 :   cputs ( wrk );
555 :   set_mdm ( mparm );
556 :   return n ;
557 : }
558 :
559 : Write_Modem ( c ) char c ;                  /* return 1 if ok, else 0    */
560 : { wrtmdm ( c );
561 :   return ( 1 );                           /* never any error           */
562 : }
563 :
564 : Read_Modem ()
565 : { return ( rdmdm ());                     /* from int bfr              */
566 : }
567 :
568 : void Term_Comm ()                 /* uninstall comm port drivers       */
569 : { u_m ();
570 : }
571 :
572 : /* end of cterm.c */
```

Figure 6-8. Continued.

CTERM features file-capture capabilities, a simple yet effective script language, and a number of stub (that is, incompletely implemented) actions, such as emulation of the VT52 and VT100 series terminals, indicating various directions in which it can be developed.

The names of a script file and a capture file can be passed to CTERM in the command line. If no filename extensions are included, the default for the script file is .SCR and that for the capture file is .CAP. If extensions are given, they override the default values. The capture feature can be invoked only if a filename is supplied in the command line, but a script file can be called at any time via the Esc command sequence, and one script file can call for another with the same feature.

The functions included in CTERM.C are listed and summarized in Table 6-13.

Table 6-13. CTERM.C Functions.

Lines	Name	Description
1–5		Program documentation.
7–11		*Include* files.
12–20		Definitions.
22–43		Global data areas.
45		External prototype declaration.
47–49	*Wants_To_Abort()*	Checks for Ctrl-Break or Ctrl-C being pressed.
52–165	*main()*	Main program loop; includes modem engine and sequential state machine to decode remote commands.
167–297	*docmd()*	Gets, interprets, and performs local (console or script) command.
299–304	*kbd_wait()*	Waits for input from console or script file.
306–334	*kb_file()*	Gets keystroke from console or script; returns EOF if no character available.
336–362	*esc()*	Translates script escape sequence.
364–370	*getfil()*	Gets name of script file and opens the file.
372–382	*getnam()*	Gets string from console or script into designated buffer.
384–393	*addext()*	Checks buffer for extension; adds one if none given.
395–398	*put_cap()*	Writes character to capture file if capture in effect.
400–406	*cap_flush()*	Closes capture file and terminates capture mode if capture in effect.
408–411		Timer data locations.
413–425	*getmr()*	Returns time since midnight, in milliseconds.
427–432	*Delay()*	Sleeps *n* milliseconds.
434–436	*Start_Timer()*	Sets timer for *n* seconds.
438–440	*Timer_Expired()*	Checks timer versus clock.
442–445	*Set_Vid()*	Initializes video data.
447–452	*locate()*	Positions cursor on display.
454–456	*deol()*	Deletes to end of line.
458–468	*deos()*	Deletes to end of screen.
470–472	*cls()*	Clears screen.
474–478	*cursor()*	Turns cursor on or off.
480–485	*revvid()*	Toggles inverse/normal video display attributes.
487–492	*putchx()*	Writes char to display using *putch()* (Microsoft C library).

(more)

Table 6-13. *Continued.*

Lines	Name	Description
494–500	*Read_Keyboard()*	Gets keystroke from keyboard.
502–504		Modem data areas.
506–512	*Init_Comm()*	Installs ISR and so forth and initializes modem.
514–515		Baud-rate definitions.
517–529	*Set_Baud()*	Changes bps rate of UART.
531–537		Parity, WL definitions.
539–557	*Set_Parity()*	Establishes UART parity mode.
559–562	*Write_Modem()*	Sends character to UART.
564–566	*Read_Modem()*	Gets character from ISR's buffer.
568–570	*Term_Comm()*	Uninstalls ISR and so forth and restores original vectors.

For communication with the console, CTERM uses the special Microsoft C library functions defined by CONIO.H, augmented with the functions in the CH2.ASM handler. Much of the code may require editing if used with other compilers. CTERM also uses the function prototype file CTERM.H, listed in Figure 6-9, to optimize function calling within the program.

```
/* CTERM.H - function prototypes for CTERM.C */
int Wants_To_Abort(void);
void main(int ,char * *);
int docmd(void);
int kbd_wait(void);
int kb_file(void);
int esc(void);
FILE *getfil(void);
void getnam(char *,int );
char *addext(char *,char *);
void put_cap(char );
void cap_flush(void);
long getmr(void);
void Delay(int );
void Start_Timer(int );
int Timer_Expired(void);
int Set_Vid(void);
void locate(int ,int );
void deol(void);
void deos(void);
void cls(void);
void cursor(int );
void revvid(int );
int putchx(char );
```

Figure 6-9. CTERM.H. *(more)*

```
int Read_Keyboard(void);
void Init_Comm(void);
int Set_Baud(int );
int Set_Parity(int );
int Write_Modem(char );
int Read_Modem(void);
void Term_Comm(void);

/* CH1.ASM functions - modem interfacing */
void i_m(void);
void set_mdm(int);
void wrtmdm(int);
void Send_Byte(int);
int  rdmdm(void);
void u_m(void);

/* CH1A.ASM functions - exception handlers */
void set_int (void);
void rst_int (void);
int broke (void);

/* CH2.ASM functions - video interfacing */
void _i_v(void);
int  _wrpos(int, int);
void _deol(void);
void _cls(void);
int  _color(int, int);
```

Figure 6-9. Continued.

Program execution begins at the entry to *main()*, line 52. CTERM first checks (lines 56 through 59) whether any filenames were passed in the command line; if they were, CTERM opens the corresponding files. Next, the program installs the exception handler (line 60), initializes the video handler (line 61), clears the display (line 62), and announces its presence (lines 63 and 64). The serial driver is installed and initialized to 1200 bps and no parity (lines 65 through 67), and the program enters its main modem-engine loop (lines 68 through 159).

This loop is functionally the same as that used in ENGINE, but it has been extended to detect an Esc from the keyboard as signalling the start of a local command sequence (lines 70 through 73) and to include a state-machine technique (lines 80 through 153) to recognize incoming escape sequences, such as the VT52 or VT100 codes. To specify a local command from the keyboard, press the Escape (Esc) key, then the first letter of the local command desired. After the local command has been selected, press any key (such as Enter or the spacebar) to continue. To get a listing of all the commands available, press Esc-H.

The *kb_file()* routine of CTERM (called in the main loop at line 69) can get its input from either a script file or the keyboard. If a script file is open (lines 308 through 330), it is used until EOF is reached or until the operator presses Ctrl-C to stop script-file input. Otherwise,

input is taken from the keyboard (lines 331 and 332). If a script file is in use, its input is echoed to the display (lines 325 through 329) if the V command has been given.

To permit the Esc character itself to be placed in script files, the backslash (\) character serves as a secondary escape signal. When a backslash is detected (lines 323 and 324) in the input stream, the next character input is translated according to the following rules:

Character	Interpretation
E *or* e	Translates to Esc.
N *or* n	Translates to Linefeed.
R *or* r	Translates to Enter (CR).
T *or* t	Translates to Tab.
^	Causes the *next* character input to be converted into a control character.

Any other character, including another \, is not translated at all.

When the Esc character is detected from either the console or a script file, the *docmd()* function (lines 167 through 297) is called to prompt for and decode the next input character as a command and to perform appropriate actions. Valid command characters, and the actions they invoke, are as follows:

Command Character	Action
D	Delay 0–9 seconds, then proceed. Must be followed by a decimal digit that indicates how long to delay.
E	Set EVEN parity.
F	Set (fast) 1200 baud.
H	Display list of valid commands.
N	Set no parity.
O	Set ODD parity.
Q	Quit; return to MS-DOS command prompt.
R	Reset modem.
S	Set (slow) 300 baud.
U	Use script file (CTERM prompts for filename).
V	Verify file input. Echoes each script-file byte.
W	Wait for character; the next input character is the one that must be matched.

Any other character input after an Esc and the resulting Command prompt generates the message *Don't know X* (where *X* stands for the actual input character) followed by the prompt *Use 'H' command for Help.*

If input is taken from a script and the V flag is off, *docmd()* performs its task quietly, with no output to the screen. If input is received from the console, however, the command letter, followed by a descriptive phrase, is echoed to the screen. Input, detection, and execution of the local commands are accomplished much as in CDVUTL, by way of a large *switch()* statement (lines 178 through 290).

Although the listed commands are only a subset of the features available in CDVUTL for the device-driver program, they are more than adequate for creating useful scripts. The predecessor of CTERM (DT115.EXE), which included the CompuServe B-Protocol file-transfer capability but had no additional commands, has been in use since early 1986 to handle automatic uploading and downloading of files from the CompuServe Information Service by means of script files. In conjunction with an auto-dialing modem, DT115.EXE handles the entire transaction, from login through logout, without human intervention.

All the bits and pieces of CTERM are put together by assembling the three handlers with MASM, compiling CTERM with Microsoft C, and linking all four object modules into an executable file. Figure 6-10 shows the complete sequence and also the three ways of using the finished program.

Compiling:

```
C>MASM CH1;  <Enter>
C>MASM CH1A;  <Enter>
C>MASM CH2;  <Enter>
C>MSC CTERM;  <Enter>
```

Linking:

```
C>LINK CTERM+CH1+CH1A+CH2;  <Enter>
```

Use:
(no files)

```
C>CTERM  <Enter>
```

or
(script only)

```
C>CTERM scriptfile  <Enter>
```

or

```
C>CTERM scriptfile capturefile  <Enter>
```

Figure 6-10. Putting CTERM together and using it.

Jim Kyle
Chip Rabinowitz

Article 7
File and Record Management

The core of most application programs is the reading, processing, and writing of data stored on magnetic disks. This data is organized into files, which are identified by name; the files, in turn, can be organized by grouping them into directories. Operating systems provide application programs with services that allow them to manipulate these files and directories without regard to the hardware characteristics of the disk device. Thus, applications can concern themselves solely with the form and content of the data, leaving the details of the data's location on the disk and of its retrieval to the operating system.

The disk storage services provided by an operating system can be categorized into file functions and record functions. The file functions operate on entire files as named entities, whereas the record functions provide access to the data contained within files. (In some systems, an additional class of directory functions allows applications to deal with collections of files as well.) This article discusses the MS-DOS function calls that allow an application program to create, open, close, rename, and delete disk files; read data from and write data to disk files; and inspect or change the information (such as attributes and date and time stamps) associated with disk filenames in disk directories.
See also PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: MS-DOS Storage Devices; PROGRAMMING FOR MS-DOS: Disk Directories and Volume Labels.

Historical Perspective

Current versions of MS-DOS provide two overlapping sets of file and record management services to support application programs: the handle functions and the file control block (FCB) functions. Both sets are available through Interrupt 21H (Table 7-1). *See* SYSTEM CALLS: INTERRUPT 21H. The reasons for this surprising duplication are strictly historical.

The earliest versions of MS-DOS used FCBs for all file and record access because CP/M, which was the dominant operating system on 8-bit microcomputers, used FCBs. Microsoft chose to maintain compatibility with CP/M to aid programmers in converting the many existing CP/M application programs to the 16-bit MS-DOS environment; consequently, MS-DOS versions 1.x included a set of FCB functions that were a functional superset of those present in CP/M. As personal computers evolved, however, the FCB access method did not lend itself well to the demands of larger, faster disk drives.

Accordingly, MS-DOS version 2.0 introduced the handle functions to provide a file and record access method similar to that found in UNIX/XENIX. These functions are easier to use and more flexible than their FCB counterparts and fully support a hierarchical (tree-like) directory structure. The handle functions also allow character devices, such as the

console or printer, to be treated for some purposes as though they were files. MS-DOS version 3.0 introduced additional handle functions, enhanced some of the existing handle functions for use in network environments, and provided improved error reporting for all functions.

The handle functions, which offer far more capability and performance than the FCB functions, should be used for all new applications. Therefore, they are discussed first in this article.

Table 7-1. Interrupt 21H Function Calls for File and Record Management.

Operation	Handle Function	FCB Function
Create file.	3CH	16H
Create new file.	5BH	
Create temporary file.	5AH	
Open file.	3DH	0FH
Close file.	3EH	10H
Delete file.	41H	13H
Rename file.	56H	17H
Perform sequential read.	3FH	14H
Perform sequential write.	40H	15H
Perform random record read.	3FH	21H
Perform random record write.	40H	22H
Perform random block read.		27H
Perform random block write.		28H
Set disk transfer area address.		1AH
Get disk transfer area address.		2FH
Parse filename.		29H
Position read/write pointer.	42H	
Set random record number.		24H
Get file size.	42H	23H
Get/Set file attributes.	43H	
Get/Set date and time stamp.	57H	
Duplicate file handle.	45H	
Redirect file handle.	46H	

Using the Handle Functions

The initial link between an application program and the data stored on disk is the name of a disk file in the form

drive:path\filename.ext

where *drive* designates the disk on which the file resides, *path* specifies the directory on that disk in which the file is located, and *filename.ext* identifies the file itself. If *drive* and/or *path* is omitted, MS-DOS assumes the default disk drive and current directory. Examples of acceptable pathnames include

C:\PAYROLL\TAXES.DAT
LETTERS\MEMO.TXT
BUDGET.DAT

Pathnames can be hard-coded into a program as part of its data. More commonly, however, they are entered by the user at the keyboard, either as a command-line parameter or in response to a prompt from the program. If the pathname is provided as a command-line parameter, the application program must extract it from the other information in the command line. Therefore, to allow a program to distinguish between pathnames and other parameters when the two are combined in a command line, the other parameters, such as switches, usually begin with a slash (/) or dash (-) character.

All handle functions that use a pathname require the name to be in the form of an ASCIIZ string — that is, the name must be terminated by a null (zero) byte. If the pathname is hard-coded into a program, the null byte must be part of the ASCIIZ string. If the pathname is obtained from keyboard input or from a command-line parameter, the null byte must be appended by the program. *See* Opening an Existing File below.

To use a disk file, a program opens or creates the file by calling the appropriate MS-DOS function with the ASCIIZ pathname. MS-DOS checks the pathname for invalid characters and, if the open or create operation is successful, returns a 16-bit handle, or identification code, for the file. The program uses this handle for subsequent operations on the file, such as record reads and writes.

The total number of handles for simultaneously open files is limited in two ways. First, the per-process limit is 20 file handles. The process's first five handles are always assigned to the standard devices, which default to the CON, AUX, and PRN character devices:

Handle	Service	Default
0	Standard input	Keyboard (CON)
1	Standard output	Video display (CON)
2	Standard error	Video display (CON)
3	Standard auxiliary	First communications port (AUX)
4	Standard list	First parallel printer port (PRN)

Ordinarily, then, a process has only 15 handles left from its initial allotment of 20; however, when necessary, the 5 standard device handles can be redirected to other files and devices or closed and reused.

In addition to the per-process limit of 20 file handles, there is a system-wide limit. MS-DOS maintains an internal table that keeps track of all the files and devices opened with file handles for all currently active processes. The table contains such information as the current file pointer for read and write operations and the time and date of the last write to the file. The size of this table, which is set when MS-DOS is initially loaded into memory, determines the system-wide limit on how many files and devices can be open simultaneously. The default limit is 8 files and devices; thus, this system-wide limit usually overrides the per-process limit.

To increase the size of MS-DOS's internal handle table, the statement *FILES*=nnn can be included in the CONFIG.SYS file. (CONFIG.SYS settings take effect the next time the system is turned on or restarted.) The maximum value for FILES is 99 in MS-DOS versions 2.x and 255 in versions 3.x. *See* USER COMMANDS: CONFIG.SYS: FILES.

Error handling and the handle functions

When a handle-based file function succeeds, MS-DOS returns to the calling program with the carry flag clear. If a handle function fails, MS-DOS sets the carry flag and returns an error code in the AX register. The program should check the carry flag after each operation and take whatever action is appropriate when an error is encountered. Table 7-2 lists the most frequently encountered error codes for file and record I/O (exclusive of network operations).

Table 7-2. Frequently Encountered Error Diagnostics for File and Record Management.

Code	Error
02	File not found
03	Path not found
04	Too many open files (no handles left)
05	Access denied
06	Invalid handle
11	Invalid format
12	Invalid access code
13	Invalid data
15	Invalid disk drive letter
17	Not same device
18	No more files

The error codes used by MS-DOS in versions 3.0 and later are a superset of the MS-DOS version 2.0 error codes. *See* APPENDIX B: CRITICAL ERROR CODES; APPENDIX C: EXTENDED ERROR CODES. Most MS-DOS version 3 error diagnostics relate to network operations, which provide the program with a greater chance for error than does a single-user system.

Programs that are to run in a network environment need to anticipate network problems. For example, the server can go down while the program is using shared files.

Under MS-DOS versions 3.x, a program can also use Interrupt 21H Function 59H (Get Extended Error Information) to obtain more details about the cause of an error after a failed handle function. The information returned by Function 59H includes the type of device that caused the error and a recommended recovery action.

Warning: Many file and record I/O operations discussed in this article can result in or be affected by a hardware (critical) error. Such errors can be intercepted by the program if it contains a custom critical error exception handler (Interrupt 24H). *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Exception Handlers.

Creating a file

MS-DOS provides three Interrupt 21H handle functions for creating files:

Function	Name
3CH	Create File with Handle (versions 2.0 and later)
5AH	Create Temporary File (versions 3.0 and later)
5BH	Create New File (versions 3.0 and later)

Each function is called with the segment and offset of an ASCIIZ pathname in the DS:DX registers and the attribute to be assigned to the new file in the CX register. The possible attribute values are

Code	Attribute
00H	Normal file
01H	Read-only file
02H	Hidden file
04H	System file

Files with more than one attribute can be created by combining the values listed above. For example, to create a file that has both the read-only and system attributes, the value 05H is placed in the CX register.

If the file is successfully created, MS-DOS returns a file handle in AX that must be used for subsequent access to the new file and sets the file read/write pointer to the beginning of the file; if the file is not created, MS-DOS sets the carry flag (CF) and returns an error code in AX.

Function 3CH is the only file-creation function available under MS-DOS versions 2.x. It must be used with caution, however, because if a file with the specified name already exists, Function 3CH will open it and truncate it to zero length, eradicating the previous contents of the file. This complication can be avoided by testing for the previous existence of the file with an open operation before issuing the create call.

Under MS-DOS versions 3.0 and later, Function 5BH is the preferred function in most cases because it will fail if a file with the same name already exists. In networking environments, this function can be used to implement semaphores, allowing the synchronization of programs running in different network nodes.

Function 5AH is used to create a temporary work file that is guaranteed to have a unique name. This capability is important in networking environments, where several copies of the same program, running in different nodes, may be accessing the same logical disk volume on a server. The function is passed the address of a buffer that can contain a drive and/or path specifying the location for the created file. MS-DOS generates a name for the created file that is a sequence of alphanumeric characters derived from the current time and returns the entire ASCIIZ pathname to the program in the same buffer, along with the file's handle in AX. The program must save the filename so that it can delete the file later, if necessary; the file created with Function 5AH is not destroyed when the program exits.

Example: Create a file named MEMO.TXT in the \LETTERS directory on drive C using Function 3CH. Any existing file with the same name is truncated to zero length and opened.

```
fname   db      'C:\LETTERS\MEMO.TXT',0
fhandle dw      ?
        .
        .
        .
        mov     dx,seg fname    ; DS:DX = address of
        mov     ds,dx           ; pathname for file
        mov     dx,offset fname
        xor     cx,cx           ; CX = normal attribute
        mov     ah,3ch          ; Function 3CH = create
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if create failed
        mov     fhandle,ax      ; else save file handle
        .
        .
        .
```

Example: Create a temporary file using Function 5AH and place it in the \TEMP directory on drive C. MS-DOS appends the filename it generates to the original path in the buffer named *fname*. The resulting file specification can be used later to delete the file.

```
fname   db      'C:\TEMP\'      ; generated ASCIIZ filename
        db      13 dup (0)      ; is appended by MS-DOS

fhandle dw      ?
        .
        .
        .
```

(more)

```
mov     dx,seg fname    ; DS:DX = address of
mov     ds,dx           ; path for temporary file
mov     dx,offset fname
xor     cx,cx           ; CX = normal attribute
mov     ah,5ah          ; Function 5AH = create
                        ; temporary file
int     21h             ; transfer to MS-DOS
jc      error           ; jump if create failed
mov     fhandle,ax      ; else save file handle
.
.
.
```

Opening an existing file

Function 3DH (Open File with Handle) opens an existing normal, system, or hidden file in the current or specified directory. When calling Function 3DH, the program supplies a pointer to the ASCIIZ pathname in the DS:DX registers and a 1-byte access code in the AL register. This access code includes the read/write permissions, the file-sharing mode, and an inheritance flag. The bits of the access code are assigned as follows:

Bit(s)	Description
0–2	Read/write permissions (versions 2.0 and later)
3	Reserved
4–6	File-sharing mode (versions 3.0 and later)
7	Inheritance flag (versions 3.0 and later)

The read/write permissions field of the access code specifies how the file will be used and can take the following values:

Bits 0–2	Description
000	Read permission desired
001	Write permission desired
010	Read and write permission desired

For the open to succeed, the permissions field must be compatible with the file's attribute byte in the disk directory. For example, if the program attempts to open an existing file that has the read-only attribute when the permissions field of the access code byte is set to write or read/write, the open function will fail and an error code will be returned in AX.

The sharing-mode field of the access code byte is important in a networking environment. It determines whether other programs will also be allowed to open the file and, if so, what operations they will be allowed to perform. Following are the possible values of the file-sharing mode field:

Bits 4–6	Description
000	Compatibility mode. Other programs can open the file and perform read or write operations as long as no process specifies any sharing mode other than compatibility mode.
001	Deny all. Other programs cannot open the file.
010	Deny write. Other programs cannot open the file in compatibility mode or with write permission.
011	Deny read. Other programs cannot open the file in compatibility mode or with read permission.
100	Deny none. Other programs can open the file and perform both read and write operations but cannot open the file in compatibility mode.

When file-sharing support is active (that is, SHARE.EXE has previously been loaded), the result of any open operation depends on both the contents of the permissions and file-sharing fields of the access code byte and the permissions and file-sharing requested by other processes that have already successfully opened the file.

The inheritance bit of the access code byte controls whether a child process will inherit that file handle. If the inheritance bit is cleared, the child can use the inherited handle to access the file without performing its own open operation. Subsequent operations performed by the child process on inherited file handles also affect the file pointer associated with the parent's file handle. If the inheritance bit is set, the child process does not inherit the handle.

If the file is opened successfully, MS-DOS returns its handle in AX and sets the file read/write pointer to the beginning of the file; if the file is not opened, MS-DOS sets the carry flag and returns an error code in AX.

Example: Copy the first parameter from the program's command tail in the program segment prefix (PSP) into the array *fname* and append a null character to form an ASCIIZ filename. Attempt to open the file with compatibility sharing mode and read/write access. If the file does not already exist, create it and assign it a normal attribute.

```
cmdtail equ     80h             ; PSP offset of command tail
fname   db      64 dup (?)
fhandle dw      ?

        .
        .
        .
                                ; assume that DS already
                                ; contains segment of PSP
```

(more)

```
                                ; prepare to copy filename...
        mov     si,cmdtail      ; DS:SI = command tail
        mov     di,seg fname    ; ES:DI = buffer to receive
        mov     es,di           ; filename from command tail
        mov     di,offset fname
        cld                     ; safety first!

        lodsb                   ; check length of command tail
        or      al,al
        jz      error           ; jump, command tail empty

label1:                         ; scan off leading spaces
        lodsb                   ; get next character
        cmp     al,20h          ; is it a space?
        jz      label1          ; yes, skip it

label2:
        cmp     al,0dh          ; look for terminator
        jz      label3          ; quit if return found
        cmp     al,20h
        jz      label3          ; quit if space found
        stosb                   ; else copy this character
        lodsb                   ; get next character
        jmp     label2

label3:
        xor     al,al           ; store final NULL to
        stosb                   ; create ASCIIZ string

                                ; now open the file...
        mov     dx,seg fname    ; DS:DX = address of
        mov     ds,dx           ; pathname for file
        mov     dx,offset fname
        mov     ax,3d02h        ; Function 3DH = open r/w
        int     21h             ; transfer to MS-DOS
        jnc     label4          ; jump if file found

        cmp     ax,2            ; error 2 = file not found
        jnz     error           ; jump if other error
                                ; else make the file...
        xor     cx,cx           ; CX = normal attribute
        mov     ah,3ch          ; Function 3CH = create
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if create failed

label4:
        mov     fhandle,ax      ; save handle for file
        .
        .
        .
```

Closing a file

Function 3EH (Close File) closes a file created or opened with a file handle function. The program must place the handle of the file to be closed in BX. If a write operation was performed on the file, MS-DOS updates the date, time, and size in the file's directory entry.

Closing the file also flushes the internal MS-DOS buffers associated with the file to disk and causes the disk's file allocation table (FAT) to be updated if necessary.

Good programming practice dictates that a program close files as soon as it finishes using them. This practice is particularly important when the file size has been changed, to ensure that data will not be lost if the system crashes or is turned off unexpectedly by the user. A method of updating the FAT without closing the file is outlined below under Duplicating and Redirecting Handles.

Reading and writing with handles

Function 3FH (Read File or Device) enables a program to read data from a file or device that has been opened with a handle. Before calling Function 3FH, the program must set the DS:DX registers to point to the beginning of a data buffer large enough to hold the requested transfer, put the file handle in BX, and put the number of bytes to be read in CX. The length requested can be a maximum of 65535 bytes. The program requesting the read operation is responsible for providing the data buffer.

If the read operation succeeds, the data is read, beginning at the current position of the file read/write pointer, to the specified location in memory. MS-DOS then increments its internal read/write pointer for the file by the length of the data transferred and returns the length to the calling program in AX with the carry flag cleared. The only indication that the end of the file has been reached is that the length returned is less than the length requested. In contrast, when Function 3FH is used to read from a character device that is *not* in raw mode, the read will terminate at the requested length or at the receipt of a carriage return character, whichever comes first. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Character Device Input and Output. If the read operation fails, MS-DOS returns with the carry flag set and an error code in AX.

Function 40H (Write File or Device) writes from a buffer to a file (or device) using a handle previously obtained from an open or create operation. Before calling Function 40H, the program must set DS:DX to point to the beginning of the buffer containing the source data, put the file handle in BX, and put the number of bytes to write in CX. The number of bytes to write can be a maximum of 65535.

If the write operation is successful, MS-DOS puts the number of bytes written in AX and increments the read/write pointer by this value; if the write operation fails, MS-DOS sets the carry flag and returns an error code in AX.

Records smaller than one sector (512 bytes) are not written directly to disk. Instead, MS-DOS stores the record in an internal buffer and writes it to disk when the internal buffer is full, when the file is closed, or when a call to Interrupt 21H Function 0DH (Disk Reset) is issued.

Note: If the destination of the write operation is a disk file and the disk is full, the only indication to the calling program is that the length returned in AX is not the same as the length requested in CX. *Disk full* is not returned as an error with the carry flag set.

A special use of the Write function is to truncate or extend a file. If Function 40H is called with a record length of zero in CX, the file size will be adjusted to the current location of the file read/write pointer.

Example: Open the file MYFILE.DAT, create the file MYFILE.BAK, copy the contents of the .DAT file into the .BAK file using 512-byte reads and writes, and then close both files.

```
file1   db      'MYFILE.DAT',0
file2   db      'MYFILE.BAK',0

handle1 dw      ?               ; handle for MYFILE.DAT
handle2 dw      ?               ; handle for MYFILE.BAK

buff    db      512 dup (?)     ; buffer for file I/O
        .
        .
        .
                                ; open MYFILE.DAT...
        mov     dx,seg file1    ; DS:DX = address of filename
        mov     ds,dx
        mov     dx,offset file1
        mov     ax,3d00h        ; Function 3DH = open (read-only)
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if open failed
        mov     handle1,ax      ; save handle for file

                                ; create MYFILE.BAK...
        mov     dx,offset file2 ; DS:DX = address of filename
        mov     cx,0            ; CX = normal attribute
        mov     ah,3ch          ; Function 3CH = create
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if create failed
        mov     handle2,ax      ; save handle for file

loop:                           ; read MYFILE.DAT
        mov     dx,offset buff  ; DS:DX = buffer address
        mov     cx,512          ; CX = length to read
        mov     bx,handle1      ; BX = handle for MYFILE.DAT
        mov     ah,3fh          ; Function 3FH = read
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if read failed
        or      ax,ax           ; were any bytes read?
        jz      done            ; no, end of file reached

                                ; write MYFILE.BAK
        mov     dx,offset buff  ; DS:DX = buffer address
        mov     cx,ax           ; CX = length to write
        mov     bx,handle2      ; BX = handle for MYFILE.BAK
        mov     ah,40h          ; Function 40H = write
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if write failed
        cmp     ax,cx           ; was write complete?
        jne     error           ; jump if disk full
        jmp     loop            ; continue to end of file
```

(more)

```
done:                           ; now close files...
        mov     bx,handle1      ; handle for MYFILE.DAT
        mov     ah,3eh          ; Function 3EH = close file
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if close failed

        mov     bx,handle2      ; handle for MYFILE.BAK
        mov     ah,3eh          ; Function 3EH = close file
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if close failed

        .
        .
        .
```

Positioning the read/write pointer

Function 42H (Move File Pointer) sets the position of the read/write pointer associated with a given handle. The function is called with a signed 32-bit offset in the CX and DX registers (the most significant half in CX), the file handle in BX, and the positioning mode in AL:

Mode	Significance
00	Supplied offset is relative to beginning of file.
01	Supplied offset is relative to current position of read/write pointer.
02	Supplied offset is relative to end of file.

If Function 42H succeeds, MS-DOS returns the resulting absolute offset (in bytes) of the file pointer relative to the beginning of the file in the DX and AX registers, with the most significant half in DX; if the function fails, MS-DOS sets the carry flag and returns an error code in AX.

Thus, a program can obtain the size of a file by calling Function 42H with an offset of zero and a positioning mode of 2. The function returns a value in DX:AX that represents the offset of the end-of-file position relative to the beginning of the file.

Example: Assume that the file MYFILE.DAT was previously opened and its handle is saved in the variable *fhandle*. Position the file pointer 32768 bytes from the beginning of the file and then read 512 bytes of data starting at that file position.

```
fhandle dw      ?               ; handle from previous open
buff    db      512 dup (?)     ; buffer for data from file

        .
        .
        .
```

(more)

```
                                ; position the file pointer...
        mov     cx,0            ; CX = high part of file offset
        mov     dx,32768        ; DX = low part of file offset
        mov     bx,fhandle      ; BX = handle for file
        mov     al,0            ; AL = positioning mode
        mov     ah,42h          ; Function 42H = position
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if function call failed

                                ; now read 512 bytes from file
        mov     dx,offset buff  ; DS:DX = address of buffer
        mov     cx,512          ; CX = length of 512 bytes
        mov     bx,fhandle      ; BX = handle for file
        mov     ah,3fh          ; Function 3FH = read
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if read failed
        cmp     ax,512          ; was 512 bytes read?
        jne     error           ; jump if partial rec. or EOF
        .
        .
        .
```

Example: Assume that the file MYFILE.DAT was previously opened and its handle is saved in the variable *fhandle*. Find the size of the file in bytes by positioning the file pointer to zero bytes relative to the end of the file. The returned offset, which is relative to the beginning of the file, is the file's size.

```
fhandle dw      ?               ; handle from previous open

        .
        .
        .
                                ; position the file pointer
                                ; to the end of file...
        mov     cx,0            ; CX = high part of offset
        mov     dx,0            ; DX = low part of offset
        mov     bx,fhandle      ; BX = handle for file
        mov     al,2            ; AL = positioning mode
        mov     ah,42h          ; Function 42H = position
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if function call failed

                                ; if call succeeded, DX:AX
                                ; now contains the file size
        .
        .
        .
```

Other handle operations

MS-DOS provides other handle-oriented functions to rename (or move) a file, delete a file, read or change a file's attributes, read or change a file's date and time stamp, and duplicate or redirect a file handle. The first three of these are "file-handle-like" because they use an ASCIIZ string to specify the file; however, they do not return a file handle.

Renaming a file

Function 56H (Rename File) renames an existing file and/or moves the file from one location in the hierarchical file structure to another. The file to be renamed cannot be a hidden or system file or a subdirectory and must not be currently open by any process; attempting to rename an open file can corrupt the disk. MS-DOS renames a file by simply changing its directory entry; it moves a file by removing its current directory entry and creating a new entry in the target directory that refers to the same file. The location of the file's actual data on the disk is not changed.

Both the current and the new filenames must be ASCIIZ strings and can include a drive and path specification; wildcard characters (* and ?) are not permitted in the filenames. The program calls Function 56H with the address of the current pathname in the DS:DX registers and the address of the new pathname in ES:DI. If the path elements of the two strings are not the same and both paths are valid, the file "moves" from the source directory to the target directory. If the paths match but the filenames differ, MS-DOS simply modifies the directory entry to reflect the new filename.

If the function succeeds, MS-DOS returns to the calling program with the carry flag clear. The function fails if the new filename is already in the target directory; in that case, MS-DOS sets the carry flag and returns an error code in AX.

Example: Change the name of the file MYFILE.DAT to MYFILE.OLD. In the same operation, move the file from the \WORK directory to the \BACKUP directory.

```
file1   db      '\WORK\MYFILE.DAT',0
file2   db      '\BACKUP\MYFILE.OLD',0
        .
        .
        .
        mov     dx,seg file1    ; DS:DX = old filename
        mov     ds,dx
        mov     es,dx
        mov     dx,offset file1
        mov     di,offset file2 ; ES:DI = new filename
        mov     ah,56h          ; Function 56H = rename
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if rename failed
        .
        .
        .
```

Deleting a file

Function 41H (Delete File) effectively deletes a file from a disk. Before calling the function, a program must set the DS:DX registers to point to the ASCIIZ pathname of the file to be deleted. The supplied pathname cannot specify a subdirectory or a read-only file, and the file must not be currently open by any process.

If the function is successful, MS-DOS deletes the file by simply marking the first byte of its directory entry with a special character (0E5H), making the entry subsequently unrecognizable. MS-DOS then updates the disk's FAT so that the clusters that previously belonged to the file are "free" and returns to the program with the carry flag clear. If the delete function fails, MS-DOS sets the carry flag and returns an error code in AX.

The actual contents of the clusters assigned to the file are not changed by a delete operation, so for security reasons sensitive information should be overwritten with spaces or some other constant character before the file is deleted with Function 41H.

Example: Delete the file MYFILE.DAT, located in the \WORK directory on drive C.

```
fname   db      'C:\WORK\MYFILE.DAT',0
        .
        .
        .
        mov     dx,seg fname    ; DS:DX = address of filename
        mov     ds,dx
        mov     dx,offset fname
        mov     ah,41h          ; Function 41H = delete
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if delete failed
        .
        .
        .
```

Getting/setting file attributes

Function 43H (Get/Set File Attributes) obtains or modifies the attributes of an existing file. Before calling Function 43H, the program must set the DS:DX registers to point to the ASCIIZ pathname for the file. To read the attributes, the program must set AL to zero; to set the attributes, it must set AL to 1 and place an attribute code in CX. *See* Creating a File above.

If the function is successful, MS-DOS reads or sets the attribute byte in the file's directory entry and returns with the carry flag clear and the file's attribute in CX. If the function fails, MS-DOS sets the carry flag and returns an error code in AX.

Function 43H cannot be used to set the volume-label bit (bit 3) or the subdirectory bit (bit 4) of a file. It also should not be used on a file that is currently open by any process.

Example: Change the attributes of the file MYFILE.DAT in the \BACKUP directory on drive C to read-only. This prevents the file from being accidentally deleted from the disk.

```
fname   db      'C:\BACKUP\MYFILE.DAT',0
        .
        .
        .
        mov     dx,seg fname    ; DS:DX = address of filename
        mov     ds,dx
        mov     dx,offset fname
        mov     cx,1            ; CX = attribute (read-only)
        mov     al,1            ; AL = mode (0 = get, 1 = set)
```

(more)

```
mov     ah,43h          ; Function 43H = get/set attr
int     21h             ; transfer to MS-DOS
jc      error           ; jump if set attrib. failed
.
.
.
```

Getting/setting file date and time

Function 57H (Get/Set Date/Time of File) reads or sets the directory time and date stamp of an open file. To set the time and date to a particular value, the program must call Function 57H with the desired time in CX, the desired date in DX, the handle for the file (obtained from a previous open or create operation) in BX, and the value 1 in AL. To read the time and date, the function is called with AL containing 0 and the file handle in BX; the time is returned in the CX register and the date is returned in the DX register. As with other handle-oriented file functions, if the function succeeds, the carry flag is returned cleared; if the function fails, MS-DOS returns the carry flag set and an error code in AX.

The formats used for the file time and date are the same as those used in disk directory entries and FCBs. *See* Structure of the File Control Block below.

The main uses of Function 57H are to force the time and date entry for a file to be updated when the file has *not* been changed and to circumvent MS-DOS's modification of a file date and time when the file *has* been changed. In the latter case, a program can use this function with AL = 0 to obtain the file's previous date and time stamp, modify the file, and then restore the original file date and time by re-calling the function with AL = 1 before closing the file.

Duplicating and redirecting handles

Ordinarily, the disk FAT and directory are not updated until a file is closed, even when the file has been modified. Thus, until the file is closed, any new data added to the file can be lost if the system crashes or is turned off unexpectedly. The obvious defense against such loss is simply to close and reopen the file every time the file is changed. However, this is a relatively slow procedure and in a network environment can cause the program to lose control of the file to another process.

Use of a second file handle, created by using Function 45H (Duplicate File Handle) to duplicate the original handle of the file to be updated, can protect data added to a disk file before the file is closed. To use Function 45H, the program must put the handle to be duplicated in BX. If the operation is successful, MS-DOS clears the carry flag and returns the new handle in AX; if the operation fails, MS-DOS sets the carry flag and returns an error code in AX.

If the function succeeds, the duplicate handle can simply be closed in the usual manner with Function 3EH. This forces the desired update of the disk directory and FAT. The original handle remains open and the program can continue to use it for file read and write operations.

Note: While the second handle is open, moving the read/write pointer associated with either handle moves the pointer associated with the other.

Example: Assume that the file MYFILE.DAT was previously opened and the handle for that file has been saved in the variable *fhandle*. Duplicate the handle and then close the duplicate to ensure that any data recently written to the file is saved on the disk and that the directory entry for the file is updated accordingly.

```
fhandle dw      ?               ; handle from previous open

        .
        .
        .
                                ; duplicate the handle...
        mov     bx,fhandle      ; BX = handle for file
        mov     ah,45h          ; Function 45H = dup handle
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if function call failed

                                ; now close the new handle...
        mov     bx,ax           ; BX = duplicated handle
        mov     ah,3eh          ; Function 3EH = close
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if close failed
        mov     bx,fhandle      ; replace closed handle with active handle
        .
        .
        .
```

Function 45H is sometimes also used in conjunction with Function 46H (Force Duplicate File Handle). Function 46H forces a handle to be a duplicate for another open handle — in other words, to refer to the same file or device at the same file read/write pointer location. The handle is then said to be redirected.

The most common use of Function 46H is to change the meaning of the standard input and standard output handles before loading a child process with the EXEC function. In this manner, the input for the child program can be redirected to come from a file or its output can be redirected into a file, without any special knowledge on the part of the child program. In such cases, Function 45H is used to also create duplicates of the standard input and standard output handles before they are redirected, so that their original meanings can be restored after the child exits. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Writing MS-DOS Filters.

Using the FCB Functions

A file control block is a data structure, located in the application program's memory space, that contains relevant information about an open disk file: the disk drive, the filename and extension, a pointer to a position within the file, and so on. Each open file must have its own FCB. The information in an FCB is maintained cooperatively by both MS-DOS and the application program.

MS-DOS moves data to and from a disk file associated with an FCB by means of a data buffer called the disk transfer area (DTA). The current address of the DTA is under the control of the application program, although each program has a 128-byte default DTA at offset 80H in its program segment prefix (PSP). *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program.

Under early versions of MS-DOS, the only limit on the number of files that can be open simultaneously with FCBs is the amount of memory available to the application to hold the FCBs and their associated disk buffers. However, under MS-DOS versions 3.0 and later, when file-sharing support (SHARE.EXE) is loaded, MS-DOS places some restrictions on the use of FCBs to simplify the job of maintaining network connections for files. If the application attempts to open too many FCBs, MS-DOS simply closes the least recently used FCBs to keep the total number within a limit.

The CONFIG.SYS file directive FCBS allows the user to control the allowed maximum number of FCBs and to specify a certain number of FCBs to be protected against automatic closure by the system. The default values are a maximum of four files open simultaneously using FCBs and zero FCBs protected from automatic closure by the system. *See* USER COMMANDS: CONFIG.SYS: FCBS.

Because the FCB operations predate MS-DOS version 2.0 and because FCBs have a fixed structure with no room to contain a path, the FCB file and record services do not support the hierarchical directory structure. Many FCB operations can be performed only on files in the current directory of a disk. For this reason, the use of FCB file and record operations should be avoided in new programs.

Structure of the file control block

Each FCB is a 37-byte array allocated from its own memory space by the application program that will use it. The FCB contains all the information needed to identify a disk file and access the data within it: drive identifier, filename, extension, file size, record size, various file pointers, and date and time stamps. The FCB structure is shown in Table 7-3.

Table 7-3. Structure of a Normal File Control Block.

Maintained by	Offset (bytes)	Size (bytes)	Description
Program	00H	1	Drive identifier
Program	01H	8	Filename
Program	09H	3	File extension
MS-DOS	0CH	2	Current block number
Program	0EH	2	Record size (bytes)
MS-DOS	10H	4	File size (bytes)
MS-DOS	14H	2	Date stamp
MS-DOS	16H	2	Time stamp
MS-DOS	18H	8	Reserved
MS-DOS	20H	1	Current record number
Program	21H	4	Random record number

Drive identifier: Initialized by the application to designate the drive on which the file to be opened or created resides. 0 = default drive, 1 = drive A, 2 = drive B, and so on. If the application supplies a zero in this byte (to use the default drive), MS-DOS alters the byte during the open or create operation to reflect the actual drive used; that is, after an open or create operation, this drive will always contain a value of 1 or greater.

Filename: Standard eight-character filename; initialized by the application; must be left justified and padded with blanks if the name has fewer than eight characters. A device name (for example, PRN) can be used; note that there is no colon after a device name.

File extension: Three-character file extension; initialized by the application; must be left justified and padded with blanks if the extension has fewer than three characters.

Current block number: Initialized to zero by MS-DOS when the file is opened. The block number and the record number together make up the record pointer during sequential file access.

Record size: The size of a record (in bytes) as used by the program. MS-DOS sets this field to 128 when the file is opened or created; the program can modify the field afterward to any desired record size. If the record size is larger than 128 bytes, the default DTA in the PSP cannot be used because it will collide with the program's own code or data.

File size: The size of the file in bytes. MS-DOS initializes this field from the file's directory entry when the file is opened. The first 2 bytes of this 4-byte field are the least significant bytes of the file size.

Date stamp: The date of the last write operation on the file. MS-DOS initializes this field from the file's directory entry when the file is opened. This field uses the same format used by file handle Function 57H (Get/Set/Date/Time of File):

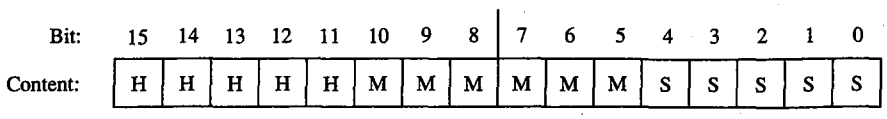
Date Format

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Content:	Y	Y	Y	Y	Y	Y	Y	M	M	M	M	D	D	D	D	D

Bits	Contents
0–4	Day of month (1–31)
5–8	Month (1–12)
9–15	Year (relative to 1980)

Time stamp: The time of the last write operation on the file. MS-DOS initializes this field from the file's directory entry when the file is opened. This field uses the same format used by file handle Function 57H (Get/Set/Date/Time of File):

Time Format



Bits	Contents
0–4	Number of 2-second increments (0–29)
5–10	Minutes (0–59)
11–15	Hours (0–23)

Current record number: Together with the block number, constitutes the record pointer used during sequential read and write operations. MS-DOS does not initialize this field when a file is opened. The record number is limited to the range 0 through 127; thus, there are 128 records per block. The beginning of a file is record 0 of block 0.

Random record pointer: A 4-byte field that identifies the record to be transferred by the random record functions 21H, 22H, 27H, and 28H. If the record size is 64 bytes or larger, only the first 3 bytes of this field are used. MS-DOS updates this field after random block reads and writes (Functions 27H and 28H) but not after random record reads and writes (Functions 21H and 22H).

An extended FCB, which is 7 bytes longer than a normal FCB, can be used to access files with special attributes such as hidden, system, and read-only. The extra 7 bytes of an extended FCB are simply prefixed to the normal FCB format (Table 7-4). The first byte of an extended FCB always contains 0FFH, which could never be a legal drive code and therefore serves as a signal to MS-DOS that the extended format is being used. The next 5 bytes are reserved and must be zero, and the last byte of the prefix specifies the attributes of the file being manipulated. The remainder of an extended FCB has exactly the same layout as a normal FCB. In general, an extended FCB can be used with any MS-DOS function call that accepts a normal FCB.

Table 7-4. Structure of an Extended File Control Block.

Maintained by	Offset (bytes)	Size (bytes)	Description
Program	00H	1	Extended FCB flag = 0FFH
MS-DOS	01H	5	Reserved
Program	06H	1	File attribute byte
Program	07H	1	Drive identifier
Program	08H	8	Filename

(more)

Table 7-4. *Continued.*

Maintained by	Offset (bytes)	Size (bytes)	Description
Program	10H	3	File extension
MS-DOS	13H	2	Current block number
Program	15H	2	Record size (bytes)
MS-DOS	17H	4	File size (bytes)
MS-DOS	1BH	2	Date stamp
MS-DOS	1DH	2	Time stamp
MS-DOS	1FH	8	Reserved
MS-DOS	27H	1	Current record number
Program	28H	4	Random record number

Extended FCB flag: When 0FFH is present in the first byte of an FCB, it is a signal to MS-DOS that an extended FCB (44 bytes) is being used instead of a normal FCB (37 bytes).

File attribute byte: Must be initialized by the application when an extended FCB is used to open or create a file. The bits of this field have the following significance:

Bit	Meaning
0	Read-only
1	Hidden
2	System
3	Volume label
4	Directory
5	Archive
6	Reserved
7	Reserved

FCB functions and the PSP

The PSP contains several items that are of interest when using the FCB file and record operations: two FCBs called the default FCBs, the default DTA, and the command tail for the program. The following table shows the size and location of these elements:

PSP Offset (bytes)	Size (bytes)	Description
5CH	16	Default FCB #1
6CH	20	Default FCB #2
80H	1	Length of command tail
81H	127	Command-tail text
80H	128	Default disk transfer area (DTA)

When MS-DOS loads a program into memory for execution, it copies the command tail into the PSP at offset 81H, places the length of the command tail in the byte at offset 80H, and parses the first two parameters in the command tail into the default FCBs at PSP offsets 5CH and 6CH. (The command tail consists of the command line used to invoke the program minus the program name itself and any redirection or piping characters and their associated filenames or device names.) MS-DOS then sets the initial DTA address for the program to PSP:0080H.

For several reasons, the default FCBs and the DTA are often moved to another location within the program's memory area. First, the default DTA allows processing of only very small records. In addition, the default FCBs overlap substantially, and the first byte of the default DTA and the last byte of the first FCB conflict. Finally, unless either the command tail or the DTA is moved beforehand, the first FCB-related file or record operation will destroy the command tail.

Function 1AH (Set DTA Address) is used to alter the DTA address. It is called with the segment and offset of the new buffer to be used as the DTA in DS:DX. The DTA address remains the same until another call to Function 1AH, regardless of other file and record management calls; it does not need to be reset before each read or write.

Note: A program can use Function 2FH (Get DTA Address) to obtain the current DTA address before changing it, so that the original address can be restored later.

Parsing the filename

Before a file can be opened or created with the FCB function calls, its drive, filename, and extension must be placed within the proper fields of the FCB. The filename can be coded into the program itself, or the program can obtain it from the command tail in the PSP or by prompting the user and reading it in with one of the several function calls for character device input.

MS-DOS automatically parses the first two parameters in the program's command tail into the default FCBs at PSP:005CH and PSP:006CH. It does not, however, attempt to differentiate between switches and filenames, so the pre-parsed FCBs are not necessarily useful to the application program. If the filenames were preceded by any switches, the program itself has to extract the filenames directly from the command tail. The program is then responsible for determining which parameters are switches and which are filenames, as well as where each parameter begins and ends.

After a filename has been located, Function 29H (Parse Filename) can be used to test it for invalid characters and separators and to insert its various components into the proper fields in an FCB. The filename must be a string in the standard form *drive:filename.ext*. Wildcard characters are permitted in the filename and/or extension; asterisk (*) wildcards are expanded to question mark (?) wildcards.

To call Function 29H, the DS:SI registers must point to the candidate filename, ES:DI must point to the 37-byte buffer that will become the FCB for the file, and AL must hold the parsing control code. *See* SYSTEM CALLS: INTERRUPT 21H: Function 29H.

If a drive code is not included in the filename, MS-DOS inserts the drive number of the current drive into the FCB. Parsing stops at the first terminator character encountered in the filename. Terminators include the following:

; , = + / " [ ] ¦ < > ¦ space tab

If a colon character (:) is not in the proper position to delimit the disk drive identifier or if a period (.) is not in the proper position to delimit the extension, the character will also be treated as a terminator. For example, the filename C:MEMO.TXT will be parsed correctly; however, ABC:DEF.DAY will be parsed as ABC.

If an invalid drive is specified in the filename, Function 29H returns 0FFH in AL; if the filename contains any wildcard characters, it returns 1. Otherwise, AL contains zero upon return, indicating a valid, unambiguous filename.

Note that this function simply parses the filename into the FCB. It does not initialize any other fields of the FCB (although it does zero the current block and record size fields), and it does not test whether the specified file actually exists.

Error handling and FCB functions

The FCB-related file and record functions do not return much in the way of error information when a function fails. Typically, an FCB function returns a zero in AL if the function succeeded and 0FFH if the function failed. Under MS-DOS versions 2.x, the program is left to its own devices to determine the cause of the error. Under MS-DOS versions 3.x, however, a failed FCB function call can be followed by a call to Interrupt 21H Function 59H (Get Extended Error Information). Function 59H will return the same descriptive codes for the error, including the error locus and a suggested recovery strategy, as would be returned for the counterpart handle-oriented file or record function.

Creating a file

Function 16H (Create File with FCB) creates a new file and opens it for subsequent read/write operations. The function is called with DS:DX pointing to a valid, unopened FCB. MS-DOS searches the current directory for the specifed filename. If the filename is found, MS-DOS sets the file length to zero and opens the file, effectively truncating it to a zero-length file; if the filename is not found, MS-DOS creates a new file and opens it. Other fields of the FCB are filled in by MS-DOS as described below under Opening a File.

If the create operation succeeds, MS-DOS returns zero in AL; if the operation fails, it returns 0FFH in AL. This function will not ordinarily fail unless the file is being created in the root directory and the directory is full.

Warning: To avoid loss of existing data, the FCB open function should be used to test for file existence before creating a file.

Opening a file

Function 0FH opens an existing file. DS:DX must point to a valid, unopened FCB containing the name of the file to be opened. If the specified file is found in the current directory, MS-DOS opens the file, fills in the FCB as shown in the list below, and returns with AL set to 00H; if the file is not found, MS-DOS returns with AL set to 0FFH, indicating an error.

When the file is opened, MS-DOS

- Sets the drive identifier (offset 00H) to the actual drive (01 = A, 02 = B, and so on).
- Sets the current block number (offset 0CH) to zero.
- Sets the file size (offset 10H) to the value found in the directory entry for the file.
- Sets the record size (offset 0EH) to 128.
- Sets the date and time stamp (offsets 14H and 16H) to the values found in the directory entry for the file.

The program may need to adjust the FCB — change the record size and the random record pointer, for example — before proceeding with record operations.

Example: Display a prompt and accept a filename from the user. Parse the filename into an FCB, checking for an illegal drive identifier or the presence of wildcards. If a valid, unambiguous filename has been entered, attempt to open the file. Create the file if it does not already exist.

```
kbuf    db      64,0,64 dup (0)
prompt  db      0dh,0ah,'Enter filename: $'
myfcb   db      37 dup (0)

        .
        .
        .
                                ; display the prompt...
        mov     dx,seg prompt   ; DS:DX = prompt address
        mov     ds,dx
        mov     es,dx
        mov     dx,offset prompt
        mov     ah,09h          ; Function 09H = print string
        int     21h             ; transfer to MS-DOS

                                ; now input filename...
        mov     dx,offset kbuf  ; DS:DX = buffer address
        mov     ah,0ah          ; Function 0AH = enter string
        int     21h             ; transfer to MS-DOS

                                ; parse filename into FCB...
        mov     si,offset kbuf+2 ; DS:SI = address of filename
        mov     di,offset myfcb ; ES:DI = address of fcb
        mov     ax,2900h        ; Function 29H = parse name
        int     21h             ; transfer to MS-DOS
        or      al,al           ; jump if bad drive or
        jnz     error           ; wildcard characters in name
```

(more)

```
                                ; try to open file...
        mov     dx,offset myfcb ; DS:DX = FCB address
        mov     ah,0fh          ; Function 0FH = open file
        int     21h             ; transfer to MS-DOS
        or      al,al           ; check status
        jz      proceed         ; jump if open successful

                                ; else create file...
        mov     dx,offset myfcb ; DS:DX = FCB address
        mov     ah,16h          ; Function 16H = create
        int     21h             ; transfer to MS-DOS
        or      al,al           ; did create succeed?
        jnz     error           ; jump if create failed

proceed:
        .                       ; file has been opened or
        .                       ; created, and FCB is valid
        .                       ; for read/write operations...
```

Closing a file

Function 10H (Close File with FCB) closes a file previously opened with an FCB. As usual, the function is called with DS:DX pointing to the FCB of the file to be closed. MS-DOS updates the directory, if necessary, to reflect any changes in the file's size and the date and time last written.

If the operation succeeds, MS-DOS returns 00H in AL; if the operation fails, MS-DOS returns 0FFH.

Reading and writing files with FCBs

MS-DOS offers a choice of three FCB access methods for data within files: sequential, random record, and random block.

Sequential operations step through the file one record at a time. MS-DOS increments the current record and current block numbers after each file access so that they point to the beginning of the next record. This method is particularly useful for copying or listing files.

Random record access allows the program to read or write a record from any location in the file, without sequentially reading all records up to that point in the file. The program must set the random record number field of the FCB appropriately before the read or write is requested. This method is useful in database applications, in which a program must manipulate fixed-length records.

Random block operations combine the features of sequential and random record access methods. The program can set the record number to point to any record within a file, and MS-DOS updates the record number after a read or write operation. Thus, sequential operations can easily be initiated at any file location. Random block operations with a record length of 1 byte simulate file-handle access methods.

All three methods require that the FCB for the file be open, that DS:DX point to the FCB, that the DTA be large enough for the specified record size, and that the DTA address be previously set with Function 1AH if the default DTA in the program's PSP is not being used.

MS-DOS reports the success or failure of any FCB-related read operation (sequential, random record, or random block) with one of four return codes in register AL:

Code	Meaning
00H	Successful read
01H	End of file reached; no data read into DTA
02H	Segment wrap (DTA too close to end of segment); no data read into DTA
03H	End of file reached; partial record read into DTA

MS-DOS reports the success or failure of an FCB-related write operation as one of three return codes in register AL:

Code	Meaning
00H	Successful write
01H	Disk full; partial or no write
02H	Segment wrap (DTA too close to end of segment); write failed

For FCB write operations, records smaller than one sector (512 bytes) are not written directly to disk. Instead, MS-DOS stores the record in an internal buffer and writes the data to disk only when the internal buffer is full, when the file is closed, or when a call to Interrupt 21H Function 0DH (Disk Reset) is issued.

Sequential access: reading

Function 14H (Sequential Read) reads records sequentially from the file to the current DTA address, which must point to an area at least as large as the record size specified in the file's FCB. After each read operation, MS-DOS updates the FCB block and record numbers (offsets 0CH and 20H) to point to the next record.

Sequential access: writing

Function 15H (Sequential Write) writes records sequentially from memory into the file. The length written is specified by the record size field (offset 0EH) in the FCB; the memory address of the record to be written is determined by the current DTA address. After each sequential write operation, MS-DOS updates the FCB block and record numbers (offsets 0CH and 20H) to point to the next record.

Random record access: reading

Function 21H (Random Read) reads a specific record from a file. Before requesting the read operation, the program specifies the record to be transferred by setting the record size and random record number fields of the FCB (offsets 0EH and 21H). The current DTA address must also have been previously set with Function 1AH to point to a buffer of adequate size if the default DTA is not large enough.

After the read, MS-DOS sets the current block and current record number fields (offsets 0CH and 20H) to point to the same record. Thus, the program is set up to change to sequential reads or writes. However, if the program wants to continue with random record access, it must continue to update the random record field of the FCB before each random record read or write operation.

Random record access: writing

Function 22H (Random Write) writes a specific record from memory to a file. Before issuing the function call, the program must ensure that the record size and random record pointer fields at FCB offsets 0EH and 21H are set appropriately and that the current DTA address points to the buffer containing the data to be written.

After the write, MS-DOS sets the current block and current record number fields (offsets 0CH and 20H) to point to the same record. Thus, the program is set up to change to sequential reads or writes. If the program wants to continue with random record access, it must continue to update the random record field of the FCB before each random record read or write operation.

Random block access: reading

Function 27H (Random Block Read) reads a block of consecutive records. Before issuing the read request, the program must specify the file location of the first record by setting the record size and random record number fields of the FCB (offsets 0EH and 21H) and must put the number of records to be read in CX. The DTA address must have already been set with Function 1AH to point to a buffer large enough to contain the group of records to be read if the default DTA was not large enough. The program can then issue the Function 27H call with DS:DX pointing to the FCB for the file.

After the random block read operation, MS-DOS resets the FCB random record pointer (offset 21H) and the current block and current record number fields (offsets 0CH and 20H) to point to the beginning of the next record not read and returns the number of records actually read in CX.

If the record size is set to 1 byte, Function 27H reads the number of bytes specified in CX, beginning with the byte position specified in the random record pointer. This simulates (to some extent) the handle type of read operation (Function 3FH).

Random block access: writing

Function 28H (Random Block Write) writes a block of consecutive records from memory to disk. The program specifies the file location of the first record to be written by setting the record size and random record pointer fields in the FCB (offsets 0EH and 21H). If the default DTA is not being used, the program must also ensure that the current DTA address is set appropriately by a previous call to Function 1AH. When Function 28H is called, DS:DX must point to the FCB for the file and CX must contain the number of records to be written.

After the random block write operation, MS-DOS resets the FCB random record pointer (offset 21H) and the current block and current record number fields (offsets 0CH and 20H) to point to the beginning of the next block of data and returns the number of records actually written in CX.

If the record size is set to 1 byte, Function 28H writes the number of bytes specified in CX, beginning with the byte position specified in the random record pointer. This simulates (to some extent) the handle type of write operation (Function 40H).

Calling Function 28H with a record count of zero in register CX causes the file length to be extended or truncated to the current value in the FCB random record pointer field (offset 21H) multiplied by the contents of the record size field (offset 0EH).

Example: Open the file MYFILE.DAT and create the file MYFILE.BAK on the current disk drive, copy the contents of the .DAT file into the .BAK file using 512-byte reads and writes, and then close both files.

```
fcb1    db      0               ; drive = default
        db      'MYFILE  '      ; 8 character filename
        db      'DAT'           ; 3 character extension
        db      25 dup (0)      ; remainder of fcb1
fcb2    db      0               ; drive = default
        db      'MYFILE  '      ; 8 character filename
        db      'BAK'           ; 3 character extension
        db      25 dup (0)      ; remainder of fcb2
buff    db      512 dup (?)     ; buffer for file I/O
        .
        .
        .
                                ; open MYFILE.DAT...
        mov     dx,seg fcb1     ; DS:DX = address of FCB
        mov     ds,dx
        mov     dx,offset fcb1
        mov     ah,0fh          ; Function 0FH = open
        int     21h             ; transfer to MS-DOS
        or      al,al           ; did open succeed?
        jnz     error           ; jump if open failed
                                ; create MYFILE.BAK...
        mov     dx,offset fcb2  ; DS:DX = address of FCB
        mov     ah,16h          ; Function 16H = create
        int     21h             ; transfer to MS-DOS
        or      al,al           ; did create succeed?
        jnz     error           ; jump if create failed
                                ; set record length to 512
        mov     word ptr fcb1+0eh,512
        mov     word ptr fcb2+0eh,512
                                ; set DTA to our buffer...
        mov     dx,offset buff  ; DS:DX = buffer address
        mov     ah,1ah          ; Function 1AH = set DTA
        int     21h             ; transfer to MS-DOS
loop:                           ; read MYFILE.DAT
        mov     dx,offset fcb1  ; DS:DX = FCB address
        mov     ah,14h          ; Function 14H = seq. read
        int     21h             ; transfer to MS-DOS
        or      al,al           ; was read successful?
        jnz     done            ; no, quit
                                ; write MYFILE.BAK...
```

(more)

```
        mov     dx,offset fcb2  ; DS:DX = FCB address
        mov     ah,15h          ; Function 15H = seq. write
        int     21h             ; transfer to MS-DOS
        or      al,al           ; was write successful?
        jnz     error           ; jump if write failed
        jmp     loop            ; continue to end of file
done:                           ; now close files...
        mov     dx,offset fcb1  ; DS:DX = FCB for MYFILE.DAT
        mov     ah,10h          ; Function 10H = close file
        int     21h             ; transfer to MS-DOS
        or      al,al           ; did close succeed?
        jnz     error           ; jump if close failed
        mov     dx,offset fcb2  ; DS:DX = FCB for MYFILE.BAK
        mov     ah,10h          ; Function 10H = close file
        int     21h             ; transfer to MS-DOS
        or      al,al           ; did close succeed?
        jnz     error           ; jump if close failed
        .
        .
        .
```

Other FCB file operations

As it does with file handles, MS-DOS provides FCB-oriented functions to rename or delete a file. Unlike the other FCB functions and their handle counterparts, these two functions accept wildcard characters. An additional FCB function allows the size or existence of a file to be determined without actually opening the file.

Renaming a file

Function 17H (Rename File) renames a file (or files) in the current directory. The file to be renamed cannot have the hidden or system attribute. Before calling Function 17H, the program must create a special FCB that contains the drive code at offset 00H, the old filename at offset 01H, and the new filename at offset 11H. Both the current and the new filenames can contain the ? wildcard character.

When the function call is made, DS:DX must point to the special FCB structure. MS-DOS searches the current directory for the old filename. If it finds the old filename, MS-DOS then searches for the new filename and, if it finds no matching filename, changes the directory entry for the old filename to reflect the new filename. If the old filename field of the special FCB contains any wildcard characters, MS-DOS renames every matching file. Duplicate filenames are not permitted; the process will fail at the first duplicate name.

If the operation is successful, MS-DOS returns zero in AL; if the operation fails, it returns 0FFH. The error condition may indicate either that no files were renamed or that at least one file was renamed but the operation was then terminated because of a duplicate filename.

Example: Rename all the files with the extension .ASM in the current directory of the default disk drive to have the extension .COD.

```
renfcb  db      0               ; default drive
        db      '????????'      ; wildcard filename
        db      'ASM'           ; old extension
        db      5 dup (0)       ; reserved area
        db      '????????'      ; wildcard filename
        db      'COD'           ; new extension
        db      15 dup (0)      ; remainder of FCB

        .
        .
        .
        mov     dx,seg renfcb   ; DS:DX = address of
        mov     ds,dx           ; "special" FCB
        mov     dx,offset renfcb
        mov     ah,17h          ; Function 17H = rename
        int     21h             ; transfer to MS-DOS
        or      al,al           ; did function succeed?
        jnz     error           ; jump if rename failed
        .
        .
        .
```

Deleting a file

Function 13H (Delete File) deletes a file from the current directory. The file should not be currently open by any process. If the file to be deleted has special attributes, such as read-only, the program must use an extended FCB to remove the file. Directories cannot be deleted with this function, even with an extended FCB.

Function 13H is called with DS:DX pointing to an unopened, valid FCB containing the name of the file to be deleted. The filename can contain the ? wildcard character; if it does, MS-DOS deletes all files matching the specified name. If at least one file matches the FCB and is deleted, MS-DOS returns 00H in AL; if no matching filename is found, it returns 0FFH.

Note: This function, if it succeeds, does not return any information about which and how many files were deleted. When multiple files must be deleted, closer control can be exercised by using the Find File functions (Functions 11H and 12H) to inspect candidate filenames. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Disk Directories and Volume Labels. The files can then be deleted individually.

Example: Delete all the files in the current directory of the current disk drive that have the extension .BAK and whose filenames have *A* as the first character.

```
delfcb  db      0               ; default drive
        db      'A???????'      ; wildcard filename
        db      'BAK'           ; extension
        db      25 dup (0)      ; remainder of FCB
```

(more)

```
.
.
.
mov     dx,seg delfcb   ; DS:DX = FCB address
mov     ds,dx
mov     dx,offset delfcb
mov     ah,13h          ; Function 13H = delete
int     21h             ; transfer to MS-DOS
or      al,al           ; did function succeed?
jnz     error           ; jump if delete failed
.
.
.
```

Finding file size and testing for existence

Function 23H (Get File Size) is used primarily to find the size of a disk file without opening it, but it may also be used instead of Function 11H (Find First File) to simply test for the existence of a file. Before calling Function 23H, the program must parse the filename into an unopened FCB, initialize the record size field of the FCB (offset 0EH), and set the DS:DX registers to point to the FCB.

When Function 23H returns, AL contains 00H if the file was found in the current directory of the specified drive and 0FFH if the file was not found.

If the file was found, the random record field at FCB offset 21H contains the number of records (rounded upward) in the target file, in terms of the value in the record size field (offset 0EH) of the FCB. If the record size is at least 64 bytes, only the first 3 bytes of the random record field are used; if the record size is less than 64 bytes, all 4 bytes are used. To obtain the size of the file in bytes, the program must set the record size field to 1 before the call. This method is not any faster than simply opening the file, but it does avoid the overhead of closing the file afterward (which is necessary in a networking environment).

Summary

MS-DOS supports two distinct but overlapping sets of file and record management services. The handle-oriented functions operate in terms of null-terminated (ASCIIZ) filenames and 16-bit file identifiers, called handles, that are returned by MS-DOS after a file is opened or created. The filenames can include a full path specifying the file's location in the hierarchical directory structure. The information associated with a file handle, such as the current read/write pointer for the file, the date and time of the last write to the file, and the file's read/write permissions, sharing mode, and attributes, is maintained in a table internal to MS-DOS.

In contrast, the FCB-oriented functions use a 37-byte structure called a file control block, located in the application program's memory space, to specify the name and location of the file. After a file is opened or created, the FCB is used by both MS-DOS and the application to hold other information about the file, such as the current read/write file pointer, while that file is in use. Because FCBs predate the hierarchical directory structure that was introduced in MS-DOS version 2.0 and do not have room to hold the path for a file, the FCB functions cannot be used to access files that are not in the current directory of the specified drive.

In addition to their lack of support for pathnames, the FCB functions have much poorer error reporting capabilities than handle functions and are nearly useless in networking environments because they do not support file sharing and locking. Consequently, it is strongly recommended that the handle-related file and record functions be used exclusively in all new applications.

Robert Byers
Code by Ray Duncan

Article 8
Disk Directories and Volume Labels

MS-DOS, being a disk operating system, provides facilities for cataloging disk files. The data structure used by MS-DOS for this purpose is the directory, a linear list of names in which each name is associated with a physical location on the disk. Directories are accessed and updated implicitly whenever files are manipulated, but both directories and their contents can also be manipulated explicitly using several of the MS-DOS Interrupt 21H service functions.

MS-DOS versions 1.x support only one directory on each disk. Versions 2.0 and later, however, support multiple directories linked in a two-way, hierarchical tree structure (Figure 8-1), and the complete specification of the name of a file or directory thus must describe the location in the directory hierarchy in which the name appears. This specification, or path, is created by concatenating a disk drive specifier (for example, A: or C:), the

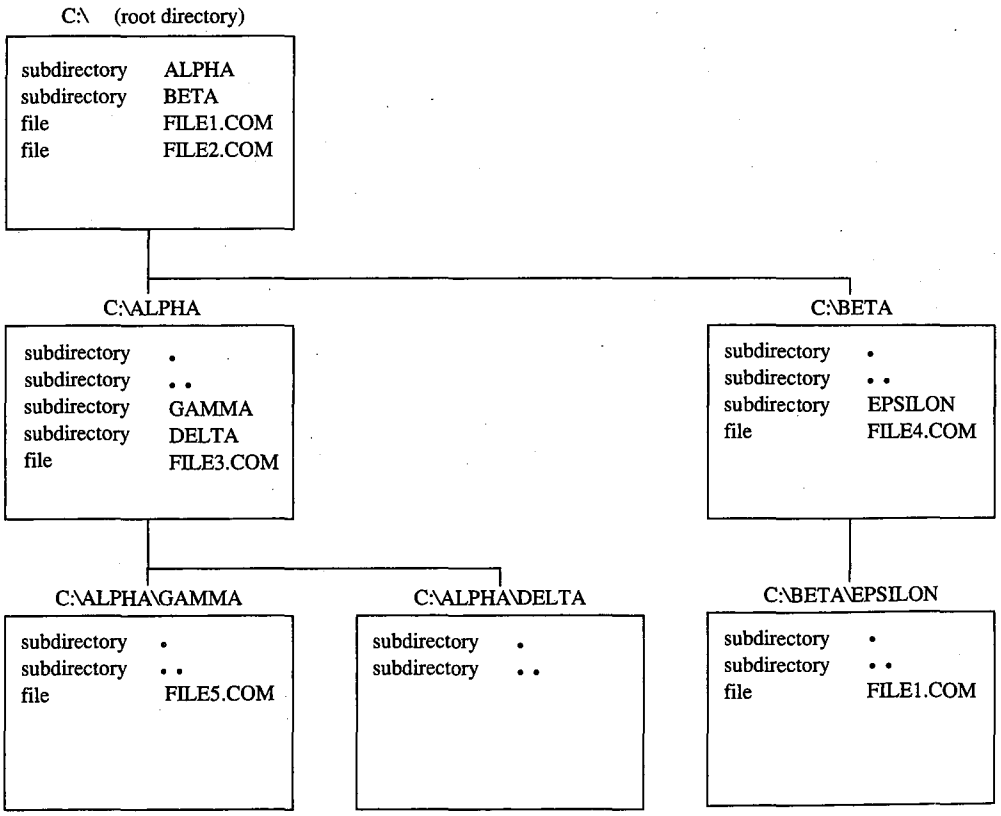


Figure 8-1. Typical hierarchical directory structure (MS-DOS versions 2.0 and later).

names of the directories in hierarchical order starting with the root directory, and finally the name of the file or directory. For example, in Figure 8-1, the complete pathname for FILE5.COM is C:\ALPHA\GAMMA\FILE5.COM. The two instances of FILE1.COM, in the root directory and in the directory EPSILON, are distinguished by their pathnames: C:\FILE1.COM in the first instance and C:\BETA\EPSILON\FILE1.COM in the second.

Note: If no drive is specified, the current drive is assumed. Also, if the first name in the specification is not preceded by a backslash, the specification is assumed to be relative to the current directory. For example, if the current directory is C:\BETA\EPSILON, the specification \FILE1.COM indicates the file FILE1.COM in the root directory and the specification FILE1.COM indicates the file FILE1.COM in the directory C:\BETA\EPSILON. *See* Figure 8-1.

Although the casual user of MS-DOS need not be concerned with how this hierarchical directory structure is implemented, MS-DOS programmers should be familiar with the internal structure of directories and with the Interrupt 21H functions available for manipulating directory contents and maintaining the links between directories. This article provides that information.

Logical Structure of MS-DOS Directories

An MS-DOS directory consists of a list of 32-byte directory entries, each of which contains a name and descriptive information. In MS-DOS versions 1.x, each name must be a filename; in versions 2.0 and later, volume labels and directory names can also appear in directory entries.

Directory searches

Directory entries are not sorted, nor are they maintained as a linked list. Thus, when MS-DOS searches a directory for a name, the search must proceed linearly from the first name in the directory. In MS-DOS versions 1.x, a directory search continues until the specified name is found or until every entry in the directory has been examined. In versions 2.0 and later, the search continues until the specified name is found or until a null directory entry (that is, one whose first byte is zero) is encountered. This null entry indicates the logical end of the directory.

Adding and deleting directory entries

MS-DOS deletes a directory entry by marking it with 0E5H in the first byte rather than by erasing it or excising it from the directory. New names are added to the directory by reusing the first deleted entry in the list. If no deleted entries are available, MS-DOS appends the new entry to the list.

The current directory

When more than one directory exists on a disk, MS-DOS keeps track of a default search directory known as the current directory. The current directory is the directory used for all implicit directory searches, such as those occasioned by a request to open a file, if no alternative path is specified. At startup, MS-DOS makes the root directory the current directory, but any other directory can be designated later, either interactively by using the CHDIR command or from within an application by using Interrupt 21H Function 3BH (Change Current Directory).

Directory Format

The root directory is created by the MS-DOS FORMAT program. *See* USER COMMANDS: FORMAT. The FORMAT program places the root directory immediately after the disk's file allocation tables (FATs). FORMAT also determines the size of the root directory. The size depends on the capacity of the storage medium: FORMAT places larger root directories on high-capacity fixed disks and smaller root directories on floppy disks. In contrast, the size of subdirectories is limited only by the storage capacity of the disk because disk space for subdirectories is allocated dynamically, as it is for any MS-DOS file. The size and physical location of the root directory can be derived from data in the BIOS parameter block (BPB) in the disk boot sector. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: MS-DOS Storage Devices.

Because space for the root directory is allocated only when the disk is formatted, the root directory cannot be deleted or moved. Subdirectories, whose disk space is allocated dynamically, can be added or deleted as needed.

Directory entry format

Each 32-byte directory entry consists of seven fields, including a name, an attribute byte, date and time stamps, and information that describes the file's size and physical location on the disk (Figure 8-2). The fields are formatted as described in the following paragraphs.

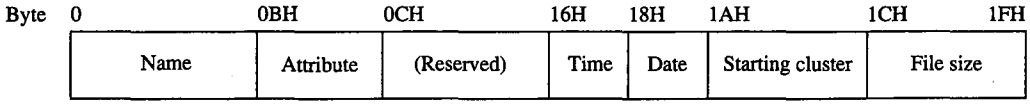


Figure 8-2. Format of a directory entry.

The name field (bytes 0–0AH) contains an 11-byte name unless the first byte of the field indicates that the directory entry is deleted or null. The name can be an 11-byte filename (8-byte name followed by a 3-byte extension), an 11-byte subdirectory name (8-byte name

followed by a 3-byte extension), or an 11-byte volume label. Names less than 8 bytes and extensions less than 3 bytes are padded to the right with blanks so that the extension always appears in bytes 08-0AH of the name field. The first byte of the name field can contain certain reserved values that affect the way MS-DOS processes the directory entry:

Value	Meaning
0	Null directory entry (logical end of directory in MS-DOS versions 2.0 and later)
5	First character of name to be displayed as the character represented by 0E5H (MS-DOS version 3.2)
0E5H	Deleted directory entry

When MS-DOS creates a subdirectory, it always includes two aliases as the first two entries in the newly created directory. The name . (an ASCII period) is an alias for the name of the current directory; the name .. (two ASCII periods) is an alias for the directory's parent directory — that is, the directory in which the entry containing the name of the current directory is found.

The attribute field (byte 0BH) is an 8-bit field that describes the way MS-DOS processes the directory entry (Figure 8-3). Each bit in the attribute field designates a particular attribute of that directory entry; more than one of the bits can be set at a time.

Bit	7	6	5	4	3	2	1	0
	(Reserved)	(Reserved)	Archive	Sub-directory	Volume label	System file	Hidden file	Read-only file

Figure 8-3. Format of the attribute field in a directory entry.

The read-only bit (bit 0) is set to 1 to mark a file read-only. Interrupt 21H Function 3DH (Open File with Handle) will fail if it is used in an attempt to open this file for writing. The hidden bit (bit 1) is set to 1 to indicate that the entry is to be skipped in normal directory searches — that is, in directory searches that do not specifically request that hidden entries be included in the search. The system bit (bit 2) is set to 1 to indicate that the entry refers to a file used by the operating system. Like the hidden bit, the system bit excludes a directory entry from normal directory searches. The volume label bit (bit 3) is set to 1 to indicate that the directory entry represents a volume label. The subdirectory bit (bit 4) is set to 1 when the directory entry contains the name and location of another directory. This bit is always set for the directory entries that correspond to the current directory (.) and the parent directory (..). The archive bit (bit 5) is set to 1 by MS-DOS functions that close a file that has been written to. Simply opening and closing a file is not sufficient to update the archive bit in the file's directory entry.

The time and date fields (bytes 16–17H and 18–19H) are initialized by MS-DOS when the directory entry is created. These fields are updated whenever a file is written to. The formats of these fields are shown in Figures 8-4 and 8-5.

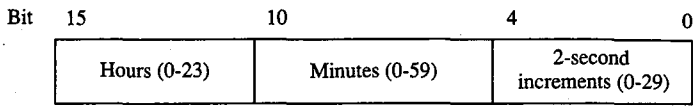


Figure 8-4. Format of the time field in a directory entry.

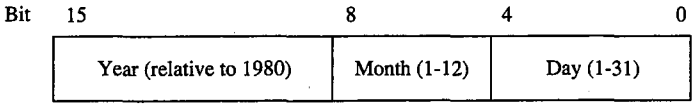


Figure 8-5. Format of the date field in a directory entry.

The starting cluster field (bytes 1A–1BH) indicates the disk location of the first cluster assigned to the file. This cluster number can be used as an entry point to the file allocation table (FAT) for the disk. (Cluster numbers can be converted to logical sector numbers with the aid of the information in the disk's BPB.)

For the . entry (the alias for the directory that contains the entry), the starting cluster field contains the starting cluster number of the directory itself. For the .. entry (the alias for the parent directory), the value in the starting cluster field refers to the parent directory unless the parent directory is the root directory, in which case the starting cluster number is zero.

The file size field (bytes 1C–1FH) is a 32-bit integer that indicates the file size in bytes.

Volume Labels

The generic term *volume* refers to a unit of auxiliary storage such as a floppy disk, a fixed disk, or a reel of magnetic tape. In computer environments where many different volumes might be used, the operating system can uniquely identify each volume by initializing it with a volume label.

Volume labels are implemented in MS-DOS versions 2.0 and later as a specific type of directory entry specified by setting bit 3 in the attribute field to 1. In a volume label directory entry, the name field contains an 11-byte string specifying a name for the disk volume. A volume label can appear only in the root directory of a disk, and only one volume label can be present on any given disk.

In MS-DOS versions 2.0 and later, the FORMAT command can be used with the /V switch to initialize a disk with a volume label. In versions 3.0 and later, the LABEL command can be used to create, update, or delete a volume label. Several commands can display a disk's volume label, including VOL, DIR, LABEL, TREE, and CHKDSK. *See* USER COMMANDS.

In MS-DOS versions 2.x, volume labels are simply a convenience for the user; no MS-DOS routine uses a volume label for any other purpose. In MS-DOS versions 3.x, however, the SHARE command examines a disk's volume label when it attempts to verify whether a disk volume has been inadvertently replaced in the midst of a file read or write operation. Removable disk volumes should therefore be assigned unique volume names if they are to contain shared files.

Functional Support for MS-DOS Directories

Several Interrupt 21H service routines can be useful to programmers who need to manipulate directories and their contents (Table 8-1). The routines can be broadly grouped into two categories: those that use a modified file control block (FCB) to pass filenames to and from the Interrupt 21H service routines (Functions 11H, 12H, 17H, and 23H) and those that use hierarchical path specifications (Functions 39H, 3AH, 3BH, 43H, 47H, 4EH, 4FH, 56H, and 57H). *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management; SYSTEM CALLS: INTERRUPT 21H.

The functions that use an FCB require that the calling program reserve enough memory for an extended FCB before the Interrupt 21H function is called. The calling program initializes the filename and extension fields of the FCB and passes the address of the FCB to the MS-DOS service routine in DS:DX. The functions that use pathnames expect all pathnames to be in ASCIIZ format—that is, the last character of the name must be followed by a zero byte.

Names in pathnames passed to Interrupt 21H functions can be separated by either a backslash (\) or a forward slash (/). (The forward slash is the separator character used in pathnames in UNIX/XENIX systems.) For example, the pathnames C:/MSP/SOURCE/ROSE.PAS and C:\MSP\SOURCE\ROSE.PAS are equivalent when passed to an Interrupt 21H function. The forward slash can thus be used in a pathname in a program that must run on both MS-DOS and UNIX/XENIX. However, the MS-DOS comand processor (COMMAND.COM) recognizes only the backslash as a pathname separator character, so forward slashes cannot be used as separators in the command line.

Table 8-1. MS-DOS Functions for Accessing Directories.

Function	Call With	Returns	Comment
Find First File	AH = 11H DS:DX = pointer to unopened FCB INT 21H	AL = 0 (directory entry found) or 0FFH (not found) DTA updated (if directory entry found)	If default not satisfactory, DTA must be set before using this function.
Find Next File	AH = 12H DS:DX = pointer to unopened FCB INT 21H	AL = 0 (directory entry found) or 0FFH (not found) DTA updated (if directory entry found)	Use the same FCB for Function 11H and Function 12H.

(more)

Table 8-1. *Continued.*

Function	Call With	Returns	Comment
Rename File	AH = 17H DS:DX = pointer to modified FCB INT 21H	AL = 0 (file renamed) or 0FFH (no directory entry or duplicate filename)	
Get File Size	AH = 23H DS:DX = pointer to unopened FCB INT 21H	AL = 0 (directory entry found) or 0FFH (not found) FCB updated with number of records in file	
Create Directory	AH = 39H DS:DX = pointer to ASCIIZ pathname INT 21H	Carry flag set (if error) AX = error code (if error)	
Remove Directory	AH = 3AH DS:DX = pointer to ASCIIZ pathname INT 21H	Carry flag set (if error) AX = error code (if error)	
Change Current Directory	AH = 3BH DS:DX = pointer to ASCIIZ pathname INT 21H	Carry flag set (if error) AX = error code (if error)	
Get/Set File Attributes	AH = 43H AL = 0 (get attributes) 1 (set attributes) CX = attributes (if AL = 1) DS:DX = pointer to ASCIIZ pathname INT 21H	Carry flag set (if error) AX = error code (if error) CX = attribute field from directory entry (if called with AL = 0)	Cannot be used to modify the volume label or subdirectory bits.
Get Current Directory	AH = 47H DS:SI = pointer to 64-byte buffer DL = drive number INT 21H	Carry flag set (if error) AX = error code (if error) Buffer updated with pathname of current directory	
Find First File	AH = 4EH DS:DX = pointer to ASCIIZ pathname CX = file attributes to match INT 21H	Carry flag set (if error) AX = error code (if error) DTA updated	If default not satisfactory, DTA must be set before using this function.
Find Next File	AH = 4FH INT 21H	Carry flag set (if error) AX = error code (if error) DTA updated	

(more)

Table 8-1. *Continued.*

Function	Call With	Returns	Comment
Rename File	AH = 56H DS:DX = pointer to ASCIIZ pathname ES:DI = pointer to new ASCIIZ pathname INT 21H	Carry flag set (if error) AX = error code (if error)	
Get/Set Date/Time of File	AH = 57H AL = 0 (get date/time) 1 (set date/time) BX = handle CX = time (if AL = 1) DX = date (if AL = 1) INT 21H	Carry flag set (if error) AX = error code (if error) CX = time (if AL = 0) DX = date (if AL = 0)	

Searching a directory

Two pairs of Interrupt 21H functions are available for directory searches. Functions 11H and 12H use FCBs to transfer filenames to MS-DOS; these functions are available in all versions of MS-DOS, but they cannot be used with pathnames. Functions 4EH and 4FH support pathnames, but these functions are unavailable in MS-DOS versions 1.x. All four functions require the address of the disk transfer area (DTA) to be initialized appropriately before the function is invoked. When Function 12H or 4FH is used, the current DTA must be the same as the DTA for the preceding call to Function 11H or 4EH.

The Interrupt 21H directory search functions are designed to be used in pairs. The Find First File functions return the first matching directory entry in the current directory (Function 11H) or in the specified directory (Function 4EH). The Find Next File functions (Functions 12H and 4FH) can be called repeatedly after a successful call to the corresponding Find First File function. Each call to one of the Find Next File functions returns the next directory entry that matches the name originally specified to the Find First File function. A directory search can thus be summarized as follows:

```
call "find first file" function

while ( matching directory entry returned )
    call "find next file" function
```

Wildcard characters

This search strategy is used because name specifications can include the wildcard characters ?, which matches any single character, and * (*see* below). When one or more wildcard characters appear in the name specified to one of the Find First File functions, only the nonwildcard characters in the name participate in the directory search. Thus, for example, the specification FOO? matches the filenames FOO1, FOO2, and so on; the specification FOO?????.??? matches FOO4.COM, FOOBAR.EXE, and FOONEW.BAK, as well as FOO1 and FOO2; the specification ????????.TXT matches all files whose extension is .TXT; the specification ????????.??? matches all files in the directory.

Function 4EH also recognizes the wildcard character *, which matches any remaining characters in a filename or extension. MS-DOS expands the * wildcard character internally to question marks. Thus, for example, the specification FOO * is the same as FOO?????; the specification FOO *.* is the same as FOO?????.???; and, of course, the specification *.* is the same as ????????.???.

Examining a directory entry

All four Interrupt 21H directory search functions return the name, attribute, file size, time, and date fields for each directory entry found during a directory search. The current DTA is used to return this data, although the format is different for the two pairs of functions: Functions 11H and 12H return a copy of the 32-byte directory entry—including the cluster number—in the DTA; Functions 4EH and 4FH return a 43-byte data structure that does not include the starting cluster number. *See* SYSTEM CALLS: INTERRUPT 21H: Function 4EH.

The attribute field of a directory entry can be examined using Function 43H (Get/Set File Attributes). Also, Function 57H (Get/Set Date/Time of File) can be used to examine a file's time or date. However, unlike the other functions discussed here, Function 57H is intended only for files that are being actively used within an application—that is, Function 57H can be called to examine the file's time or date stamp only after the file has been opened or created using an Interrupt 21H function that returns a handle (Function 3CH, 3DH, 5AH, or 5BH).

Modifying a directory entry

Four Interrupt 21H functions can modify the contents of a directory entry. Function 17H (Rename File) can be used to change the name field in any directory entry, including hidden or system files, subdirectories, and the volume label. Related Function 56H (Rename File) also changes the name field of a filename but cannot rename a volume label or a hidden or system file. However, it can be used to move a directory entry from one directory to another. (This capability is restricted to filenames only; subdirectory entries cannot be moved with Function 56H.)

Functions 43H (Get/Set File Attributes) and 57H (Get/Set Date/Time of File) can be used to modify specific fields in a directory entry. Function 43H can mark a directory entry as a hidden or system file, although it cannot modify the volume label or subdirectory bits. Function 57H, as noted above, can be used only with a previously opened file; it provides a way to read or update a file's time and date stamps without writing to the file itself.

Creating and deleting directories

Function 39H (Create Directory) exists only to create directories—that is, directory entries with the subdirectory bit set to 1. (Interrupt 21H functions that create files, such as Function 3CH, cannot assign the subdirectory attribute to a directory entry.) The converse function, 3AH (Remove Directory), deletes a subdirectory entry from a directory. (The subdirectory must be completely empty.) Again, Interrupt 21H functions that delete files from directories, such as Function 41H, cannot be used to delete subdirectories.

Specifying the current directory

A call to Interrupt 21H Function 47H (Get Current Directory) returns the pathname of the current directory in use by MS-DOS to a user-supplied buffer. The converse operation, in which a new current directory can be specified to MS-DOS, is performed by Function 3BH (Change Current Directory).

Programming examples: Searching for files

The subroutines in Figure 8-6 below illustrate Functions 4EH and 4FH, which use path specifications passed as ASCIIZ strings to search for files. Figure 8-7 applies these assembly-language subroutines in a simple C program that lists the attributes associated with each entry in the current directory. Note how the directory search is performed in the WHILE loop in Figure 8-7 by using a global wildcard file specification (*.*) and by repeatedly executing *FindNextFile()* until no further matching filenames are found. (*See* Programming Example: Updating a Volume Label for examples of the FCB-related search functions, 11H and 21H.)

```
                TITLE   'DIRS.ASM'

;
; Subroutines for DIRDUMP.C
;


ARG1            EQU     [bp + 4]        ; stack frame addressing for C arguments
ARG2            EQU     [bp + 6]


_TEXT           SEGMENT byte public 'CODE'
                ASSUME  cs:_TEXT

;--------------------------------------------------------------------------
;
; void SetDTA( DTA );
;         char *DTA;
;
;--------------------------------------------------------------------------

                PUBLIC  _SetDTA
_SetDTA         PROC    near

                push    bp
                mov     bp,sp

                mov     dx,ARG1         ; DS:DX -> DTA
                mov     ah,1Ah          ; AH = INT 21H function number
                int     21h             ; pass DTA to MS-DOS
```

Figure 8-6. Subroutines illustrating Interrupt 21H Functions 4EH and 4FH. *(more)*

```
                pop     bp
                ret

_SetDTA         ENDP

;-------------------------------------------------------------------------
;
; int GetCurrentDir( *path );           /* returns error code */
;         char *path;                   /* pointer to buffer to contain path */
;
;-------------------------------------------------------------------------

                PUBLIC  _GetCurrentDir
_GetCurrentDir  PROC    near

                push    bp
                mov     bp,sp
                push    si

                mov     si,ARG1         ; DS:SI -> buffer
                xor     dl,dl           ; DL = 0 (default drive number)
                mov     ah,47h          ; AH = INT 21H function number
                int     21h             ; call MS-DOS; AX = error code
                jc      L01             ; jump if error

                xor     ax,ax           ; no error, return AX = 0

L01:            pop     si
                pop     bp
                ret

_GetCurrentDir  ENDP

;-------------------------------------------------------------------------
;
; int FindFirstFile( path, attribute ); /* returns error code */
;         char *path;
;         int  attribute;
;
;-------------------------------------------------------------------------

                PUBLIC  _FindFirstFile
_FindFirstFile  PROC    near

                push    bp
                mov     bp,sp

                mov     dx,ARG1         ; DS:DX -> path
                mov     cx,ARG2         ; CX = attribute
                mov     ah,4Eh          ; AH = INT 21H function number
                int     21h             ; call MS-DOS; AX = error code
                jc      L02             ; jump if error
```

Figure 8-6. Continued. *(more)*

```
                xor     ax,ax           ; no error, return AX = 0

L02:            pop     bp
                ret

_FindFirstFile  ENDP

;--------------------------------------------------------------------------
;
; int FindNextFile();                   /* returns error code */
;
;--------------------------------------------------------------------------

                PUBLIC  _FindNextFile
_FindNextFile   PROC    near

                push    bp
                mov     bp,sp

                mov     ah,4Fh          ; AH = INT 21H function number
                int     21h             ; call MS-DOS; AX = error code
                jc      L03             ; jump if error

                xor     ax,ax           ; if no error, set AX = 0


L03:            pop     bp
                ret

_FindNextFile   ENDP

_TEXT           ENDS


_DATA           SEGMENT word public 'DATA'

CurrentDir      DB      64 dup(?)
DTA             DB      64 dup(?)

_DATA           ENDS

                END
```

Figure 8-6. Continued.

```
/* DIRDUMP.C */

#define AllAttributes   0x3F            /* bits set for all attributes */

main()
{
        static  char CurrentDir[64];
        int     ErrorCode;
        int     FileCount = 0;

        struct
        {
          char    reserved[21];
          char    attrib;
          int     time;
          int     date;
          long    size;
          char    name[13];
        }         DTA;

/* display current directory name */

        ErrorCode = GetCurrentDir( CurrentDir );
        if( ErrorCode )
        {
          printf( "\nError %d:  GetCurrentDir", ErrorCode );
          exit( 1 );
        }

        printf( "\nCurrent directory is \\%s", CurrentDir );


/* display files and attributes */

        SetDTA( &DTA );                 /* pass DTA to MS-DOS */

        ErrorCode = FindFirstFile( "*.*", AllAttributes );

        while( !ErrorCode )
        {
          printf( "\n%12s -- ", DTA.name );
          ShowAttributes( DTA.attrib );
          ++FileCount;

          ErrorCode = FindNextFile( );
        }

/* display file count and exit */

        printf( "\nCurrent directory contains %d files\n", FileCount );
        return( 0 );
}
```

Figure 8-7. The complete DIRDUMP.C program.

(more)

```
ShowAttributes( a )
int     a;
{
        int     i;
        int     mask = 1;

        static char *AttribName[] =
        {
          "read-only ",
          "hidden ",
          "system ",
          "volume ",
          "subdirectory ",
          "archive "
        };


        for( i=0; i<6; i++ )            /* test each attribute bit */
        {
          if( a & mask )
            printf( AttribName[i] );    /* display a message if bit is set */
          mask = mask << 1;
        }
}
```

Figure 8-7. Continued.

Programming example: Updating a volume label

To create, modify, or delete a volume-label directory entry, the Interrupt 21H functions that work with FCBs should be used. Figure 8-8 contains four subroutines that show how to search for, rename, create, or delete a volume label in MS-DOS versions 2.0 and later.

```
                TITLE   'VOLS.ASM'

;-----------------------------------------------------------------------------
;
; C-callable routines for manipulating MS-DOS volume labels.
; Note: These routines modify the current DTA address.
;
;-----------------------------------------------------------------------------

ARG1            EQU     [bp + 4]        ; stack frame addressing

DGROUP          GROUP   _DATA

_TEXT           SEGMENT byte public 'CODE'
                ASSUME  cs:_TEXT,ds:DGROUP
```

Figure 8-8. Subroutines for manipulating volume labels. *(more)*

```
;-------------------------------------------------------------------------
;
; char *GetVolLabel();          /* returns pointer to volume label name */
;
;-------------------------------------------------------------------------

                PUBLIC  _GetVolLabel
_GetVolLabel    PROC    near

                push    bp
                mov     bp,sp
                push    si
                push    di

                call    SetDTA          ; pass DTA address to MS-DOS
                mov     dx,offset DGROUP:ExtendedFCB
                mov     ah,11h          ; AH = INT 21H function number
                int     21h             ; Search for First Entry
                test    al,al
                jnz     L01
                                        ; label found so make a copy
                mov     si,offset DGROUP:DTA + 8
                mov     di,offset DGROUP:VolLabel
                call    CopyName
                mov     ax,offset DGROUP:VolLabel ; return the copy's address
                jmp     short L02

L01:            xor     ax,ax           ; no label, return 0 (null pointer)

L02:            pop     di
                pop     si
                pop     bp
                ret

_GetVolLabel    ENDP

;-------------------------------------------------------------------------
;
; int RenameVolLabel( label );          /* returns error code */
;         char *label;                  /* pointer to new volume label name */
;
;-------------------------------------------------------------------------

                PUBLIC  _RenameVolLabel
_RenameVolLabel PROC    near

                push    bp
                mov     bp,sp
                push    si
                push    di
```

Figure 8-8. Continued. *(more)*

```
                mov     si,offset DGROUP:VolLabel  ; DS:SI -> old volume name
                mov     di,offset DGROUP:Name1
                call    CopyName        ; copy old name to FCB

                mov     si,ARG1
                mov     di,offset DGROUP:Name2
                call    CopyName        ; copy new name into FCB

                mov     dx,offset DGROUP:ExtendedFCB    ; DS:DX -> FCB
                mov     ah,17h          ; AH = INT 21H function number
                int     21h             ; rename
                xor     ah,ah           ; AX = 00H (success) or 0FFH (failure)

                pop     di              ; restore registers and return
                pop     si
                pop     bp
                ret

_RenameVolLabel ENDP

;---------------------------------------------------------------------------
;
; int NewVolLabel( label );             /* returns error code */
;         char *label;                  /* pointer to new volume label name */
;
;---------------------------------------------------------------------------

                PUBLIC  _NewVolLabel
_NewVolLabel    PROC    near

                push    bp
                mov     bp,sp
                push    si
                push    di

                mov     si,ARG1
                mov     di,offset DGROUP:Name1
                call    CopyName        ; copy new name to FCB

                mov     dx,offset DGROUP:ExtendedFCB
                mov     ah,16h          ; AH = INT 21H function number
                int     21h             ; create directory entry
                xor     ah,ah           ; AX = 00H (success) or 0FFH (failure)

                pop     di              ; restore registers and return
                pop     si
                pop     bp
                ret

_NewVolLabel    ENDP
```

Figure 8-8. Continued. *(more)*

```
;--------------------------------------------------------------------------
;
; int DeleteVolLabel();                 /* returns error code */
;
;--------------------------------------------------------------------------

                PUBLIC  _DeleteVolLabel
_DeleteVolLabel PROC    near

                push    bp
                mov     bp,sp
                push    si
                push    di

                mov     si,offset DGROUP:VolLabel
                mov     di,offset DGROUP:Name1
                call    CopyName        ; copy current volume name to FCB

                mov     dx,offset DGROUP:ExtendedFCB
                mov     ah,13h          ; AH = INT 21H function number
                int     21h             ; delete directory entry
                xor     ah,ah           ; AX = 00H (success) or 0FFH (failure)

                pop     di              ; restore registers and return
                pop     si
                pop     bp
                ret

_DeleteVolLabel ENDP

;--------------------------------------------------------------------------
;
; miscellaneous subroutines
;
;--------------------------------------------------------------------------

SetDTA          PROC    near

                push    ax              ; preserve registers used
                push    dx

                mov     dx,offset DGROUP:DTA    ; DS:DX -> DTA
                mov     ah,1Ah          ; AH = INT 21H function number
                int     21h             ; set DTA

                pop     dx              ; restore registers and return
                pop     ax
                ret

SetDTA          ENDP
```

Figure 8-8. Continued. *(more)*

```
CopyName        PROC    near            ; Caller:  SI -> ASCIIZ source
                                        ;          DI -> destination
                push    ds
                pop     es              ; ES = DGROUP
                mov     cx,11           ; length of name field

L11:            lodsb                   ; copy new name into FCB ..
                test    al,al
                jz      L12             ; .. until null character is reached
                stosb
                loop    L11

L12:            mov     al,' '          ; pad new name with blanks
                rep     stosb
                ret

CopyName        ENDP

_TEXT           ENDS


_DATA           SEGMENT word public 'DATA'

VolLabel        DB      11 dup(0),0

ExtendedFCB     DB      0FFh            ; must be 0FFH for extended FCB
                DB      5 dup(0)        ; (reserved)
                DB      1000b           ; attribute byte (bit 3 = 1)
                DB      0               ; default drive ID
Name1           DB      11 dup('?')     ; global wildcard name
                DB      5 dup(0)        ; (unused)
Name2           DB      11 dup(0)       ; second name (for renaming entry)
                DB      9 dup(0)        ; (unused)

DTA             DB      64 dup(0)

_DATA           ENDS

                END
```

Figure 8-8. Continued.

Richard Wilton

Article 9
Memory Management

Personal computers that are MS-DOS compatible can be outfitted with as many as three kinds of random-access memory (RAM): conventional memory, expanded memory, and extended memory.

All MS-DOS machines have at least some conventional memory, but the presence of expanded or extended memory depends on the installed hardware options and the model of microprocessor on which the computer is based. Each storage class has its own capabilities, characteristics, and limitations. Each also has its own management techniques, which are the subject of this chapter.

Conventional Memory

Conventional memory is the term for the up to 1 MB of memory that is directly addressable by an Intel 8086/8088 microprocessor or by an 80286 or 80386 microprocessor running in real mode (8086-emulation mode). Physical addresses for references to conventional memory are generated by a 16-bit segment register, which acts as a base register and holds a paragraph address, combined with a 16-bit offset contained in an index register or in the instruction being executed.

On IBM PCs and compatibles, MS-DOS and the programs that run under its control occupy the bottom 640 KB or less of the conventional memory space. The memory space above the 640 KB mark is partitioned among ROM (read-only memory) chips on the system board that contain various primitive device handlers and test programs and among RAM and ROM chips on expansion boards that are used for input and output buffers and for additional device-dependent routines.

The bottom 640 KB of memory administered by MS-DOS is divided into three zones (Figure 9-1):

- The interrupt vector table
- The operating system area
- The transient program area

The interrupt vector table occupies the lowest 1024 bytes of memory (locations 00000–003FFH); its address and length are hard-wired into the processor and cannot be changed. Each doubleword position in the table is called an interrupt vector and contains the segment and offset of an interrupt handler routine for the associated hardware or software interrupt number. Interrupt handler routines are usually built into the operating system,

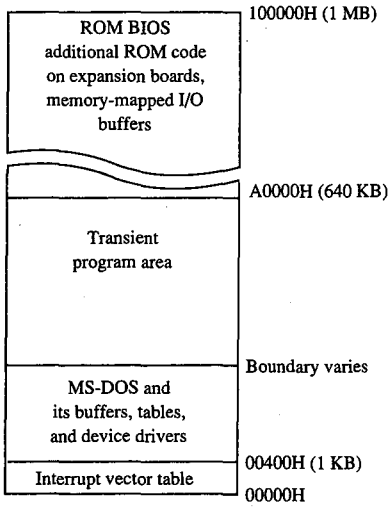


Figure 9-1. A diagram showing conventional memory in an IBM PC-compatible MS-DOS system. The bottom 1024 bytes of memory are used for the interrupt vector table. The memory above the vector table, up to the 640 KB boundary, is available for use by MS-DOS and the programs that run under its control. The top 384 KB are used for the ROM BIOS, other device-control and diagnostic routines, and memory-mapped input and output.

but in special cases application programs can contain handler routines of their own. Vectors for interrupt numbers that are not used for software linkages or by some hardware device are usually initialized by the operating system to point to a simple interrupt return (IRET) instruction or to a routine that displays an error message.

The operating-system area begins immediately above the interrupt vector table and holds the operating system proper, its tables and buffers, any additional installable device drivers specified in the CONFIG.SYS file, and the resident portion of the COMMAND.COM command interpreter. The amount of memory occupied by the operating-system area varies with the version of MS-DOS being used, the number of disk buffers, and the number and size of installed device drivers.

The transient program area (TPA) is the remainder of RAM above the operating-system area, extending to the 640 KB limit or to the end of installed RAM (whichever is smaller). External MS-DOS commands (such as CHKDSK) and other programs are loaded into the TPA for execution. The transient portion of COMMAND.COM also runs in this area.

The TPA is organized into a structure called the memory arena, which is divided into portions called *arena entries* (or memory blocks). These entries are allocated in paragraph (16-byte) multiples and can be as small as one paragraph or as large as the entire TPA. Each arena entry is preceded by a control structure called an arena entry header, which contains information indicating the size and status of the arena entry.

MS-DOS inspects the arena entry headers whenever a function requesting a memory-block allocation, modification, or release is issued; when a program is loaded and executed with the EXEC function (Interrupt 21H Function 4BH); or when a program is terminated. If any of the arena entry headers appear to be damaged, MS-DOS returns an error to the calling process. If that process is COMMAND.COM, COMMAND.COM then displays the message *Memory allocation error* and halts the system.

MS-DOS support for conventional memory management

The MS-DOS kernel supports three memory-management functions, invoked with Interrupt 21H, that operate on the TPA:

- Function 48H (Allocate Memory Block)
- Function 49H (Free Memory Block)
- Function 4AH (Resize Memory Block)

These three functions (Table 9-1) can be called by application programs, by the command processor, and by MS-DOS itself to dynamically allocate, resize, and release arena entries as they are needed. *See* SYSTEM CALLS: INTERRUPT 21H: Functions 48H; 49H; 4AH.

Table 9-1. MS-DOS Memory-Management Functions.

Function Name	Call With	Returns
Allocate Memory Block	AH = 48H BX = paragraphs needed	AX = segment of allocated block If failed: BX = size of largest available block in paragraphs
Free Memory Block	AH = 49H ES = segment of block to release	nothing
Resize (Allocated) Memory Block	AH = 4AH BX = new size of block in paragraphs ES = segment of block to resize	If failed: BX = maximum size for block in paragraphs
Get/Set Allocation Strategy*	AH = 58H AL = 00H (get strategy) 01H (set strategy) If setting: BX = strategy: 00H = first fit 01H = best fit 02H = last fit	If getting: AX = strategy code

*MS-DOS versions 3.x only.

When the MS-DOS kernel receives a memory-allocation request, it inspects the chain of arena entry headers to find a free arena entry that can satisfy the request. The memory manager can use any of three allocation strategies:

- First fit—the arena entry at the lowest address that is large enough to satisfy the request
- Best fit—the smallest available arena entry that satisfies the request, regardless of its position
- Last fit—the arena entry at the highest address that is large enough to satisfy the request

If the arena entry selected is larger than the size needed to fulfill the request, the arena entry is divided and the program is given an arena entry exactly the size it requires. A new arena entry header is then created for the remaining portion of the original arena entry; it is marked "unowned" and can be used to satisfy subsequent allocation calls.

Research on allocation strategies has demonstrated that the first-fit approach is most efficient, and this is the default strategy used by MS-DOS. However, in MS-DOS versions 3.0 and later, an application program can select a different strategy for the memory manager with Interrupt 21H Function 58H (Get/Set Allocation Strategy). *See* SYSTEM CALLS: INTERRUPT 21H: Function 58H.

Using the memory-management functions

When a program begins executing, it already owns two arena entries allocated on its behalf by the MS-DOS EXEC function (Interrupt 21H Function 4BH). The first entry holds the program's environment and is just large enough to contain this information; the second entry (called the program block in this article) contains the program's PSP, code, data, and stack.

The amount of memory MS-DOS allocates to the program block for a newly loaded transient program depends on its type (.COM or .EXE). Under typical conditions, a .COM program is allocated all of the first arena entry that is large enough to hold the contents of its file, plus 256 bytes for the PSP and at least 2 bytes for the stack. Because the TPA is seldom fragmented into more than one arena entry before a program is loaded, a .COM program usually ends up owning all the memory in the system that does not belong to the operating system itself—memory divided between a relatively small environment and a comparatively immense program block.

The amount of memory allocated to a .EXE program, on the other hand, is controlled by two fields called MINALLOC and MAXALLOC in the .EXE program file header. The MINALLOC field tells the MS-DOS loader how many paragraphs of memory, in addition to the memory required to hold the initialized code and the data present in the file, *must* be available for the program to execute at all. The MAXALLOC field contains the maximum number of excess paragraphs, *if available,* to allocate to the program.

The default value placed in MAXALLOC by the Microsoft Object Linker is FFFFH paragraphs, corresponding to 1 MB. Consequently, a .EXE program is typically allocated all of available memory when it is loaded, as is a .COM file. Although it is possible to set the MAXALLOC field to other, smaller values with the linker's /CPARMAXALLOC switch or with the EXEMOD utility supplied with Microsoft language compilers, few programmers bother to do so.

In short, when a program begins executing, it usually owns all of available memory—frequently much more memory than it needs. If the program wants to be well behaved in its use of memory and, possibly, load child programs as well, it should immediately release any extra memory. In assembly-language programs, the extra memory is released by calling Interrupt 21H Function 4AH (Resize Memory Block) with the segment of the program's PSP in the ES register and the number of paragraphs of memory to retain for the program's use in the BX register. (*See* Figures 9-2 and 9-3.) In most high-level languages, such as Microsoft C, excess memory is released by the run-time library's startup module.

```
        .
        .
        .
_TEXT   segment para public 'CODE'

        org     100h

        assume  cs:_TEXT,ds:_TEXT,es:_TEXT,ss:_TEXT

main    proc    near            ; entry point from MS-DOS
                                ; CS = DS = ES = SS = PSP

                                ; first move our stack
        mov     sp,offset stk   ; to a safe place...

                                ; now release extra memory...
        mov     bx,offset stk   ; calculate paragraphs to keep
        mov     cl,4            ; (divide offset of end of
        shr     bx,cl           ; program by 16 and round up)
        inc     bx
        mov     ah,4ah          ; Fxn 4AH = resize mem block
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if resize failed
        .
        .                       ; otherwise go on with work...
        .
main    endp

        .
        .
        .
```

(more)

Figure 9-2. An example of a .COM program releasing excess memory after it receives control from MS-DOS. Interrupt 21H Function 4AH is called with the segment address of the program's PSP in register ES and the number of paragraphs of memory to retain in register BX.

```
        dw      64 dup (?)
stk     equ     $               ; base of new stack area

_TEXT   ends

        end     main            ; defines program entry point
```

Figure 9-2. Continued.

```
_TEXT   segment word public 'CODE'      ; executable code segment

        assume  cs:_TEXT,ds:_DATA,ss:STACK

main    proc    far             ; entry point from MS-DOS
                                ; CS = _TEXT segment,
                                ; DS = ES = PSP

        mov     ax,_DATA        ; set DS = our data segment
        mov     ds,ax

                                ; give back extra memory...
        mov     ax,es           ; let AX = segment of PSP base
        mov     bx,ss           ; and BX = segment of stack base
        sub     bx,ax           ; reserve seg stack - seg psp
        add     bx,stksize/16   ; plus paragraphs of stack
        inc     bx              ; round up
        mov     ah,4ah          ; Fxn 4AH = resize memory block
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if resize failed

        .
        .
        .

main    endp

_TEXT   ends

_DATA   segment word public 'DATA'      ; static & variable data

        .
        .
        .

_DATA   ends
```

(more)

Figure 9-3. An example of a .EXE program releasing excess memory after it receives control from MS-DOS. This particular code sequence depends on the segment order shown. When a .EXE program is linked from many different object modules, other techniques may be needed to determine the amount of memory occupied by the program at run time.

```
STACK   segment para stack 'STACK'

        db      stksize dup (?)

STACK   ends

        end     main            ; defines program entry point
```

Figure 9-3. Continued.

Later, if the transient program needs additional memory for a buffer, table, or other work area, it can call Interrupt 21H Function 48H (Allocate Memory Block) with the desired number of paragraphs. If a sufficiently large block of memory is available, MS-DOS creates a new arena entry of the requested size and returns a pointer to its base in the form of a segment address in the AX register. If an arena entry of the requested size cannot be created, MS-DOS returns an error code in the AX register and the size in paragraphs of the largest available block of memory in the BX register. The application program can inspect this value to determine whether it can continue in a degraded fashion with a smaller amount of memory.

When a program finishes using an allocated arena entry, it should promptly call Interrupt 21H Function 49H to release it. This allows MS-DOS to collect small blocks of freed memory into contiguous arena entries and reduces the chance that future allocation requests by the same program will fail because of memory fragmentation. In any case, all arena entries owned by a program are released when the program terminates with Interrupt 20H or with Interrupt 21H Function 00H or 4CH.

A program skeleton demonstrating the use of dynamic memory allocation services is shown in Figure 9-4.

```
        .
        .
        .
mov     bx,800h         ; 800H paragraphs = 32 KB
mov     ah,48h          ; Fxn 48H = allocate block
int     21h             ; transfer to MS-DOS
jc      error           ; jump if allocation failed
mov     bufseg,ax       ; save segment of block

                        ; open working file...
mov     dx,offset file1 ; DS:DX = filename address
mov     ax,3d00h        ; Fxn 3DH = open, read only
int     21h             ; transfer to MS-DOS
jc      error           ; jump if open failed
mov     handle1,ax      ; save handle for work file
```

(more)

Figure 9-4. A skeleton example of dynamic memory allocation. The program requests a 32 KB memory block, uses it to copy its working file to a backup file, and then releases the memory block. Note the use of ASSUME directives to force the assembler to generate proper segment overrides on references to variables containing file handles.

```
                                ; create backup file...
        mov     dx,offset file2 ; DS:DX = filename address
        mov     cx,0            ; CX = attribute (normal)
        mov     ah,3ch          ; Fxn 3CH = create file
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if create failed
        mov     handle2,ax      ; save handle for backup file

        push    ds              ; set ES = our data segment
        pop     es
        mov     ds,bufseg       ; set DS:DX = allocated block
        xor     dx,dx

        assume  ds:NOTHING,es:_DATA     ; tell assembler

next:                           ; read working file...
        mov     bx,handle1      ; handle for work file
        mov     cx,8000h        ; try to read 32 KB
        mov     ah,3fh          ; Fxn 3FH = read
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if read failed
        or      ax,ax           ; was end of file reached?
        jz      done            ; yes, exit this loop

                                ; now write backup file...
        mov     cx,ax           ; set write length = read length
        mov     bx,handle2      ; handle for backup file
        mov     ah,40h          ; Fxn 40H = write
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if write failed
        cmp     ax,cx           ; was write complete?
        jne     error           ; no, disk must be full
        jmp     next            ; transfer another record

done:   push    es              ; restore DS = data segment
        pop     ds

        assume  ds:_DATA,es:NOTHING     ; tell assembler

                                ; release allocated block...
        mov     es,bufseg       ; segment base of block
        mov     ah,49h          ; Fxn 49H = release block
        int     21h             ; transfer to MS-DOS
        jc      error           ; (should never fail)

                                ; now close backup file...
        mov     bx,handle2      ; handle for backup file
        mov     ah,3eh          ; Fxn 3EH = close
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if close failed
```

Figure 9-4. Continued. *(more)*

```
        .
        .
        .

file1   db      'MYFILE.DAT',0  ; name of working file
file2   db      'MYFILE.BAK',0  ; name of backup file

handle1 dw      ?               ; handle for working file
handle2 dw      ?               ; handle for backup file
bufseg  dw      ?               ; segment of allocated block
```

Figure 9-4. Continued.

Expanded Memory

The original Expanded Memory Specification (EMS) version 3.0 was developed as a joint effort of Lotus Development Corporation and Intel Corporation and was announced at the Spring COMDEX in 1985. The EMS was designed to provide a uniform means for applications running on 8086/8088-based personal computers, or on 80286/80386-based computers in real mode, to circumvent the 1 MB limit on conventional memory, thus providing such programs with much larger amounts of fast random-access storage. The EMS version 3.2, modified from 3.0 to add support for multitasking operating systems, was released shortly afterward as a joint effort of Lotus, Intel, and Microsoft.

The EMS is a functional definition of a bank-switched memory subsystem; it consists of user-installable boards that plug into the IBM PC's expansion bus and a resident driver program called the Expanded Memory Manager (EMM) that is provided by the board manufacturer. As much as 8 MB of expanded memory can be installed in a single machine. Expanded memory is made available to application software in 16 KB pages, which are mapped by the EMM into a contiguous 64 KB area called the page frame somewhere above the conventional memory area used by MS-DOS (0–640 KB). An application program can thus access as many as four 16 KB expanded memory pages simultaneously. The location of the page frame is user configurable so that it will not conflict with other hardware options (Figure 9-5).

The Expanded Memory Manager

The Expanded Memory Manager provides a hardware-independent interface between application programs and the expanded memory board(s). The EMM is supplied by the board manufacturer in the form of an installable character-device driver and is linked into MS-DOS by a DEVICE directive added to the CONFIG.SYS file on the system startup disk.

Internally, the EMM is divided into two distinct components that can be referred to as the driver and the manager. The driver portion mimics some of the actions of a genuine installable device driver, in that it includes Initialization and Output Status subfunctions and a valid device header. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Installable Device Drivers.

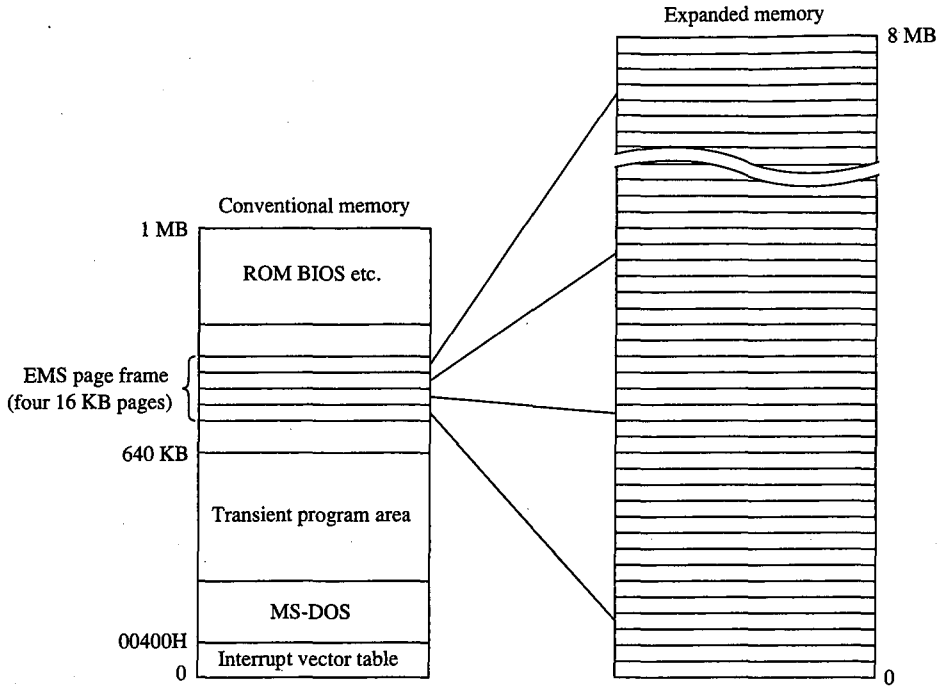


Figure 9-5. A sketch of the relationship of expanded memory to conventional memory; 16 KB pages of expanded memory are mapped into a 64 KB area, called the page frame, above the 640 KB boundary. The location of the page frame can be configured by the user to eliminate conflicts with ROMs or I/O buffers on expansion boards.

The second, and major, element of the EMM is the true interface between application software and the expanded memory hardware. Several classes of services provide

- Status of the expanded memory subsystem
- Allocation of expanded memory pages
- Mapping of logical pages into physical memory
- Deallocation of expanded memory pages
- Support for multitasking operating systems
- Diagnostic routines

Application programs communicate with the EMM directly by means of a software interrupt (Interrupt 67H). The MS-DOS kernel does not take part in expanded memory manipulations and does not use expanded memory for its own purposes.

Checking for expanded memory

Before it attempts to use expanded memory for storage, an application program must establish that the EMM is present and functional, and then it must use the manager portion of the EMM to check the status of the memory boards themselves. There are two methods a program can use to test for the existence of the EMM.

The first method is to issue an Open File or Device request (Interrupt 21H Function 3DH) using the guaranteed device name of the EMM driver: EMMXXXX0. If the open operation succeeds, one of two conditions is indicated — either the driver is present or a file with the same name exists in the current directory of the default disk drive. To rule out the latter possibility, the application can issue IOCTL Get Device Information (Interrupt 21H Function 44H Subfunction 00H) and Check Output Status (Interrupt 21H Function 44H Subfunction 07H) requests to determine whether the handle returned by the open operation is associated with a file or with a device. In either case, the handle that was obtained from the open function should then be closed (Interrupt 21H Function 3EH) so that it can be reused for another file or device.

The second method of testing for the driver is to use the address that is found in the vector for Interrupt 67H to inspect the device header of the presumed EMM. (The contents of the vector can be obtained conveniently with Interrupt 21H Function 35H.) If the EMM is present, the name field at offset 0AH of the device header contains the string *EMMXXXX0*. This method is nearly foolproof, and it avoids the relatively high overhead of an MS-DOS open function. However, it is somewhat less well behaved because it involves inspection of memory that does not belong to the application.

The two methods of testing for the existence of the EMM are illustrated in Figures 9-6 and 9-7.

```
.
.
.
                        ; attempt to "open" EMM...
mov     dx,seg emm_name ; DS:DX = address of name
mov     ds,dx           ; of EMM
mov     dx,offset emm_name
mov     ax,3d00h        ; Fxn 3DH, Mode = 00H
                        ; = open, read-only
int     21h             ; transfer to MS-DOS
jc      error           ; jump if open failed

                        ; open succeeded, make sure
                        ; it was not a file...
```

(more)

Figure 9-6. Testing for the presence of the Expanded Memory Manager with the MS-DOS Open File or Device (Interrupt 21H Function 3DH) and IOCTL (Interrupt 21H Function 44H) functions.

```
        mov     bx,ax           ; BX = handle from open
        mov     ax,4400h        ; Fxn 44H Subfxn 00H
                                ; = IOCTL Get Device Information
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if IOCTL call failed
        and     dx,80h          ; Bit 7 = 1 if character device
        jz      error           ; jump if it was a file

                                ; EMM is present, make sure
                                ; it is available...
                                ; (BX still contains handle)
        mov     ax,4407h        ; Fxn 44H Subfxn 07H
                                ; = IOCTL Get Output Status
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if IOCTL call failed
        or      al,al           ; test device status
        jz      error           ; if AL = 0 EMM is not available

                                ; now close handle ...
                                ; (BX still contains handle)
        mov     ah,3eh          ; Fxn 3EH = Close
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if close failed
        .
        .
        .

emm_name db     'EMMXXXX0',0    ; guaranteed device name for EMM
```

Figure 9-6. Continued.

```
emm_int equ     67h             ; EMM software interrupt

        .
        .
        .
                                ; first fetch contents of
                                ; EMM interrupt vector...
        mov     al,emm_int      ; AL = EMM int number
        mov     ah,35h          ; Fxn 35H = get vector
        int     21h             ; transfer to MS-DOS
                                ; now ES:BX = handler address

                                ; assume ES:0000 points
                                ; to base of the EMM...
```

(more)

Figure 9-7. Testing for the presence of the Expanded Memory Manager by inspecting the name field in the device driver header.

```
        mov     di,10           ; ES:DI = address of name
                                ; field in device header
        mov     si,seg emm_name ; DS:SI = address of
        mov     ds,si           ; expected EMM driver name
        mov     si,offset emm_name
        mov     cx,8            ; length of name field
        cld
        repz cmpsb              ; compare names...
        jnz     error           ; jump if driver absent
        .
        .
        .


emm_name db     'EMMXXXX0'      ; guaranteed device name for EMM
```

Figure 9-7. Continued.

Using expanded memory

After establishing that the EMM is present, the application program can bypass MS-DOS and communicate with the EMM directly by means of software Interrupt 67H. The calling sequence is as follows:

```
mov     ah,function     ; AH selects EMM function

.                       ; Load other registers with
.                       ; values specific to the
.                       ; requested service

int     67h             ; Transfer to EMM
```

In general, the ES:DI registers are used to pass the address of a buffer or an array, and the DX register is used to hold an expanded memory "handle." Some EMM functions also use other registers (chiefly AL and BX) to pass such information as logical and physical page numbers. Table 9-2 summarizes the services available from the EMM.

Upon return from an EMM function call, the AH register contains zero if the function was successful; otherwise, AH contains an error code with the most significant bit set (Table 9-3). Other values are typically returned in the AL and BX registers or in a user-specified buffer.

Table 9-2. Summary of the Software Interface to Application Programs Provided by the EMM.*

Function Name	Action	Call With	Returns	Comments
Get Manager Status	Test whether the expanded memory software and hardware are functional.	AH = 40H	AH = status	This call is used after the program has established, with one of the techniques presented in Figures 9-6 and 9-7, that the EMM is present.
Get Page Frame Segment	Obtain the segment address of the EMM page frame.	AH = 41H	AH = status BX = segment of page frame, if AH = 00H	The page frame is divided into four 16 KB pages that are used to map logical expanded memory pages into the physical memory space of the 8086/8088 processor.
Get Expanded Memory Pages	Obtain the number of logical expanded memory pages present in the system and the number of pages that are not already allocated.	AH = 42H	AH = status BX = unallocated EMM pages, if AH = 00H DX = total EMM pages in system	The application need not have already acquired an EMM handle to use this function.
Allocate Expanded Memory	Obtain an EMM handle and allocate logical pages to be controlled by that handle.	AH = 43H BX = logical pages to allocate	AH = status DX = handle, if AH = 00H	This function is equivalent to a file-open function for the EMM. The handle returned is analogous to a file handle and owns a certain number of EMM pages. The handle must be used with every subsequent request to map memory and must be released by a close operation when the application is finished. This function can fail because either the available EMM handles or the EMM pages have been exhausted. Function 42H can be called by the application to determine the actual number of pages available.
Map Memory	Map one of the logical pages of expanded memory assigned to a handle onto one of the four physical pages within the EMM's page frame.	AH = 44H AL = physical page (0–3) BX = logical page (0... *n*-1) DX = EMM handle	AH = status	The logical page number must be in the range 0–*n*-1, where *n* is the number of logical pages previously allocated to the EMM handle with Function 43H. To access the memory after it has been mapped to a physical page, the application also needs the segment of the EMM's page frame, which can be obtained with Function 41H.

Release Handle and Memory	Deallocate the logical pages of expanded memory currently assigned to a handle and then release the handle itself for reuse.	AH = 45H DX = EMM handle	AH = status	This function is the equivalent of a close operation on a file. It notifies the EMM that the application will not be making further use of the data it may have stored within expanded memory pages.
Get EMM Version	Return the version number of the EMM software.	AH = 46H	AH = status AL = EMM version, if AH = 00H	The returned value is the version of the EMM with which the driver complies. The version number is encoded as BCD, with the integer part in the upper 4 bits and the fractional part in the lower 4 bits.
Save Mapping Context	Save the contents of the expanded memory page-mapping registers on the expanded memory boards, associating those contents with a specific EMM handle.	AH = 47H DX = EMM handle	AH = status	This function is designed for use by interrupt handlers and resident drivers or utilities that must access expanded memory. The handle supplied to the function is the handle that was assigned to the interrupt handler during its initialization sequence, not to the program that was interrupted.
Restore Mapping Context	Restore the contents of all expanded memory hardware page-mapping registers to the values associated with the given handle.	AH = 48H DX = EMM handle	AH = status	Use of this function must be balanced by a previous call to EMM Function 47H. It allows an interrupt handler or a resident driver that used expanded memory to restore the mapping context to its state at the point of interruption.
Get Number of EMM Handles	Return the number of active EMM handles.	AH = 4BH	AH = status BX = number of EMM handles, if AH = 00H	If the number of handles returned is zero, none of the expanded memory is in use. The number of active EMM handles never exceeds 255. A single program can make several allocation requests and therefore own several EMM handles.
Get Pages Owned by Handle	Return the number of logical expanded memory pages allocated to a specific handle.	AH = 4CH DX = EMM handle	AH = status BX = logical pages, if AH = 00H	The number of pages returned if the function is successful is always in the range 1–512. An EMM handle never has zero pages of memory allocated to it.

*EMM Functions 49H and 4AH (not listed) were defined in EMS version 3.0 and are "reserved" in later EMS versions.

(more)

Table 9-2. *Continued.*

Function Name	Action	Call With	Returns	Comments
Get Pages for All Handles	Return an array that contains all the active handles and the number of logical expanded memory pages associated with each handle.	AH = 4DH DI = offset of array to receive information ES = array segment	AH = status BX = number of active EMM handles If AH = 00H, array is filled in as described in comments column	The array is filled in with doubleword entries. The first word of each entry contains a handle; the second word contains the number of pages associated with that handle. The value returned in BX gives the number of valid doubleword entries in the array. Because 255 is the maximum number of EMM handles, the array need not be larger than 1020 bytes.
Get/Set Page Map	Save or set the contents of the EMM page-mapping registers on the expanded memory boards.	AH = 4EH AL = subfunction number DS:SI = array holding mapping information (Subfunctions 01H, 02H) ES:DI = array to receive information (Subfunctions 00H, 02H)	AH = status AL = bytes in page-mapping array (Subfunction 03H) Array pointed to by ES:DI receives mapping information for Subfunctions 00H and 02H	Subfunctions: 00H = get mapping registers into array 01H = set mapping registers from array 02H = get and set mapping registers in one operation 03H = return needed size of page-mapping array This function was added in EMM version 3.2 and is designed to support multitasking. It should not ordinarily be used by application programs. The content of the array is hardware and EMM software dependent. In addition to the contents of the page-mapping registers, it may contain other information that is necessary to restore the expanded memory subsystem to its previous state.

Table 9-3. The Expanded Memory Manager (EMM) Error Codes.

Error Code	Significance
00H	Function was successful.
80H	Internal error in the EMM software. Possible causes include an error in the driver itself or damage to its memory image.
81H	Malfunction in the expanded memory hardware.
82H	EMM is busy.
83H	Invalid expanded memory handle.
84H	Function requested by the application is not supported by the EMM.
85H	No more expanded memory handles available.
86H	Error in save or restore of mapping context.
87H	Allocation request specified more logical pages than are available in the system; no pages were allocated.
88H	Allocation request specified more logical pages than are currently available in the system (the request does not exceed the physical pages that exist, but some are already allocated to other handles); no pages were allocated.
89H	Zero pages cannot be allocated.
8AH	Logical page requested for mapping is outside the range of pages assigned to the handle.
8BH	Illegal physical page number in mapping request (not in the range 0–3).
8CH	Save area for mapping contexts is full.
8DH	Save of mapping context failed because save area already contains a context associated with the requested handle.
8EH	Restore of mapping context failed because save area does not contain a context for the requested handle.
8FH	Subfunction parameter not defined.

An application program that uses expanded memory should regard that memory as a system resource, such as a file or a device, and use only the documented EMM services to allocate, access, and release expanded memory pages. Here is the general strategy that can be used by such a program:

1. Establish the presence of the EMM by one of the two methods demonstrated in Figures 9-6 and 9-7.
2. After the driver is known to be present, check its operational status with EMM Function 40H.
3. Check the version number of the EMM with EMM Function 46H to ensure that all services the application will request are available.
4. Obtain the segment of the page frame used by the EMM with EMM Function 41H.
5. Allocate the desired number of expanded memory pages with EMM Function 43H. If the allocation is successful, the EMM returns a handle in DX that is used by the application to refer to the expanded memory pages it owns. This step is exactly analogous

to opening a file and using the handle obtained from the open function for subsequent read/write operations on the file.

6. If the requested number of pages is not available, query the EMM for the actual number of pages available (EMM Function 42H) and determine whether the program can continue.
7. After successfully allocating the number of expanded memory pages needed, use EMM Function 44H to map logical pages in and out of the physical page frame, to store and retrieve data in expanded memory.
8. When finished using the expanded memory pages, release them by calling EMM Function 45H. Otherwise, the pages will not be available for use by other programs until the system is restarted.

A program skeleton that illustrates this general approach to the use of expanded memory is shown in Figure 9-8.

```
.
.
.
mov     ah,40h          ; test EMM status
int     67h
or      ah,ah
jnz     error           ; jump if bad status from EMM

mov     ah,46h          ; check EMM version
int     67h
or      ah,ah
jnz     error           ; jump if couldn't get version
cmp     al,30h          ; make sure at least ver. 3.0
jb      error           ; jump if wrong EMM version

mov     ah,41h          ; get page frame segment
int     67h
or      ah,ah
jnz     error           ; jump if failed to get frame
mov     page_frame,bx   ; save segment of page frame

mov     ah,42h          ; get no. of available pages
int     67h
or      ah,ah
jnz     error           ; jump if get pages error
mov     total_pages,dx  ; save total EMM pages
mov     avail_pages,bx  ; save available EMM pages
or      bx,bx
jz      error           ; abort if no pages available

mov     ah,43h          ; try to allocate EMM pages
```

(more)

Figure 9-8. A program skeleton for the use of expanded memory. This code assumes that the presence of the Expanded Memory Manager has already been verified with one of the techniques shown in Figures 9-6 and 9-7.

```
mov     bx,needed_pages
int     67h             ; if allocation is successful
or      ah,ah
jnz     error           ; jump if allocation failed

mov     emm_handle,dx   ; save handle for allocated pages

.
.                       ; now we are ready for other
.                       ; processing using EMM pages
.
                        ; map in EMM memory page...
mov     bx,log_page     ; BX <- EMM logical page number
mov     al,phys_page    ; AL <- EMM physical page (0-3)
mov     dx,emm_handle   ; EMM handle for our pages
mov     ah,44h          ; Fxn 44H = map EMM page
int     67h
or      ah,ah
jnz     error           ; jump if mapping error

.
.
.
                        ; program ready to terminate,
                        ; give up allocated EMM pages...
mov     dx,emm_handle   ; handle for our pages
mov     ah,45h          ; EMM Fxn 45H = release pages
int     67h
or      ah,ah
jnz     error           ; jump if release failed
.
.
.
```

Figure 9-8. Continued.

An interrupt handler or resident driver that uses the EMM follows the same general procedure outlined in steps 1 through 8, with a few minor variations. It may need to acquire an EMM handle and allocate pages before the operating system is fully functional; in particular, the MS-DOS services Open File or Device (Interrupt 21H Function 3DH), IOCTL (Interrupt 21H Function 44H), and Get Interrupt Vector (Interrupt 21H Function 35H) cannot be assumed to be available. Thus, such a handler or driver must use a modified version of the "get interrupt vector" technique to test for the existence of the EMM, fetching the contents of the Interrupt 67H vector directly instead of using MS-DOS Interrupt 21H Function 35H.

A device driver or interrupt handler typically owns its expanded memory pages on a permanent basis (until the system is restarted) and never deallocates them. Such a program must also take care to save (EMM Function 47H) and restore (EMM Function 48H) the EMM's page-mapping context (the EMM pages mapped into the page frame at the time the device driver or interrupt handler takes control of the system) so that use of the expanded memory by a foreground program will not be disturbed.

The EMM relies heavily on the good behavior of application software to avoid the corruption of expanded memory. If several applications that use expanded memory are running under a multitasking manager, such as Microsoft Windows, and one or more of those applications does not abide strictly by the EMM's conventions, the data stored in expanded memory can be corrupted.

Extended Memory

Extended memory is that storage at addresses above 1 MB (100000H) that can be accessed by an 80286 or 80386 microprocessor running in protected mode. IBM PC/AT-compatible machines can (theoretically) have as much as 15 MB of extended memory installed, in addition to the usual 1 MB of conventional memory address space. Unlike expanded memory, extended memory is linearly addressable: The address of each memory cell is fixed, so no special manager program is required.

Protected-mode operating systems, such as Microsoft XENIX and MS OS/2, can use extended memory for execution of programs. MS-DOS, on the other hand, runs in real mode on an 80286 or 80386, and programs running under its control cannot ordinarily execute from extended memory or even address that memory for storage of data.

To provide some access to extended memory for real-mode programs, IBM PC/AT-compatible machines contain two routines in their ROM BIOS (Tables 9-4 and 9-5) that allow the amount of extended memory present to be determined (Interrupt 15H Function 88H) and that transfer blocks of data between conventional memory and extended

Table 9-4. IBM PC/AT ROM BIOS Interrupt 15H Functions for Access to Extended Memory.

Interrupt 15H Function	Call With	Returns
Move Extended Memory Block	AH = 87H* CX = length (words) ES:SI = address of block move descriptor table	Carry flag = 0 if successful 1 if error AH = status: 00H no error 01H RAM parity error 02H exception interrupt error 03H gate address line 20 failed
Obtain Size of Extended Memory	AH = 88H	AX = kilobytes of memory installed above 1 MB

*Table 9-5 shows the descriptor table format used by Function 87H.

memory (Interrupt 15H Function 87H). These routines can be used by electronic disks (RAMdisks) and by other programs that wish to use extended memory for fast storage and retrieval of information that would otherwise have to be written to a slower physical disk drive.

Table 9-5. Block Move Descriptor Table Format for IBM PC/AT ROM BIOS Interrupt 15H Function 87H (Move Extended Memory Block).

Bytes	Contents
00–0FH	Zero
10–11H	Segment length in bytes (2*CX–1 or greater)
12–14H	24-bit source address
15H	Access rights byte (93H)
16–17H	Zero
18–19H	Segment length in bytes (2*CX–1 or greater)
1A–1CH	24-bit destination address
1DH	Access rights byte (93H)
1E–1FH	Zero
20–2FH	Zero

Note: This data structure actually constitutes a global descriptor table (GDT) to be used by the CPU while it is running in protected mode; the zero bytes at offsets 0–0FH and 20–2FH are filled in by the ROM BIOS code before the mode transition. The supplied 24-bit address is a linear address in the range 000000–FFFFFFH (not a segment and offset), with the least significant byte first and the most significant byte last.

Programmers should use these ROM BIOS routines with caution. Data stored in extended memory is volatile; it is lost if the machine is turned off. The transfer of data to or from extended memory involves a switch from real mode to protected mode and back again. This is a relatively slow process on 80286-based machines; in some cases it is only marginally faster than actually reading the data from a fixed disk. In addition, programs that use the ROM BIOS extended memory functions are not compatible with the MS-DOS 3.x Compatibility Box of MS OS/2, nor are they reliable if used for communications or networking.

Finally, a major deficit in these ROM BIOS functions is that they do not make any attempt to arbitrate between two or more programs or device drivers that are using extended memory for temporary storage. For example, if an application program and an installed RAMdisk driver attempt to put data in the same area of extended memory, no error is returned to either program, but the data belonging to one or both may be destroyed.

Figure 9-9 demonstrates the use of the ROM BIOS routines to transfer a block of data from extended memory to conventional memory.

```
                                ; block move descriptor table
bmdt    db      8 dup (0)       ; dummy descriptor
        db      8 dup (0)       ; GDT descriptor
        db      8 dup (0)       ; source segment descriptor
        db      8 dup (0)       ; destination segment descriptor
        db      8 dup (0)       ; BIOS CS segment descriptor
        db      8 dup (0)       ; BIOS SS segment descriptor

buff    db      80h dup (0)     ; buffer to receive data

        .
        .
        .
        mov     dx,10h          ; DX:AX = source extended memory
        mov     ax,0            ; address 100000H (1 MB)
        mov     bx,seg buff     ; DS:BX = destination conventional
        mov     ds,bx           ; memory address
        mov     bx,offset buff
        mov     cx,80h          ; CX = length to move (bytes)
        mov     si,seg bmdt     ; ES:SI = block move descriptor table
        mov     es,si
        mov     si,offset bmdt
        call    getblk          ; get block from extended memory
        or      ah,ah           ; test status
        jnz     error           ; jump if block move failed
        .
        .
        .

getblk  proc    near            ; transfer block from extended
                                ; memory to real memory
                                ; call with
                                ; DX:AX = extended memory address
                                ; DS:BX = destination buffer
                                ;    CX = length (bytes)
                                ; ES:SI = block move descriptor table
                                ; returns
                                ;    AH = 0 if transfer OK
        mov     es:[si+10h],cx  ; store length in descriptors
        mov     es:[si+18h],cx
                                ; store access rights bytes
        mov     byte ptr es:[si+15h],93h
        mov     byte ptr es:[si+1dh],93h
```

(more)

Figure 9-9. Demonstration of a block move from extended memory to conventional memory using the ROM BIOS routine. The procedure getblk *accepts a source address in extended memory, a destination address in conventional memory, a length in bytes, and the segment and offset of a block move descriptor table. The extended-memory address is a linear 32-bit address, of which only the lower 24 bits are significant; the conventional-memory address is a segment and offset. The* getblk *routine converts the destination segment and offset to a linear address, builds the appropriate fields in the block move descriptor table, invokes the ROM BIOS routine to perform the transfer, and returns the status in the AH register.*

```
                        ; source (extended memory) address
mov     es:[si+12h],ax
mov     es:[si+14h],dl
                        ; destination (conv memory) address
mov     ax,ds           ; segment * 16
mov     dx,16
mul     dx
add     ax,bx           ; + offset -> linear address
adc     dx,0
mov     es:[si+1ah],ax
mov     es:[si+1ch],dl

shr     cx,1            ; convert length to words
mov     ah,87h          ; Fxn 87H = block move
int     15h             ; transfer to ROM BIOS

ret                     ; back to caller
```

Figure 9-9. Continued.

Summary

Personal computers that run MS-DOS can support as many as three different types of fast, random-access memory (RAM). Each type has specific characteristics and requires different techniques for its management.

Conventional memory is the term used for the 1 MB of linear address space that can be accessed by an 8086 or 8088 microprocessor or by an 80286 or 80386 microprocessor running in real mode. MS-DOS and the programs that execute under its control run in this address space. MS-DOS provides application programs with services to dynamically allocate and release blocks of conventional memory.

As much as 8 MB of expanded memory can be installed in a PC and used for electronic disks, disk caching, and storage of application program data. The memory is made available in 16 KB pages and is administered by a driver program called the Expanded Memory Manager, which provides allocation, mapping, deallocation, and multitasking support.

Extended memory refers to the memory at addresses above 1 MB that can be accessed by an 80286-based or 80386-based microprocessor running in protected mode; it is not available in PCs based on the 8086 or 8088 microprocessors. As much as 15 MB of extended memory can be installed; however, the ROM BIOS services to access the memory are primitive and slow, and no manager is provided to arbitrate between multiple programs that attempt to use the same extended memory addresses for storage.

Ray Duncan

Article 10
The MS-DOS EXEC Function

The MS-DOS system loader, which brings .COM or .EXE files from disk into memory and executes them, can be invoked by any program with the MS-DOS EXEC function (Interrupt 21H Function 4BH). The default MS-DOS command interpreter, COMMAND.COM, uses the EXEC function to load and run its external commands, such as CHKDSK, as well as other application programs. Many popular commercial programs, such as databases and word processors, use EXEC to load and run subsidiary programs (spelling checkers, for example) or to load and run a second copy of COMMAND.COM. This allows a user to run subsidiary programs or enter MS-DOS commands without losing his or her current working context.

When EXEC is used by one program (called the parent) to load and run another (called the child), the parent can pass certain information to the child in the form of a set of strings called the environment, a command line, and two file control blocks. The child program also inherits the parent program's handles for the MS-DOS standard devices and for any other files or character devices the parent has opened (unless the open operation was performed with the "noninheritance" option). Any operations performed by the child on inherited handles, such as seeks or file I/O, also affect the file pointers associated with the parent's handles. A child program can, in turn, load another program, and the cycle can be repeated until the system's memory area is exhausted.

Because MS-DOS is not a multitasking operating system, a child program has complete control of the system until it has finished its work; the parent program is suspended. This type of processing is sometimes called synchronous execution. When the child terminates, the parent regains control and can use another system function call (Interrupt 21H Function 4DH) to obtain the child's return code and determine whether the program terminated normally, because of a critical hardware error, or because the user entered a Control-C.

In addition to loading a child program, EXEC can also be used to load subprograms and overlays for application programs written in assembly language or in a high-level language that does not include an overlay manager in its run-time library. Such overlays typically cannot be run as self-contained programs; most require "helper" routines or data in the application's root segment.

The EXEC function is available only with MS-DOS versions 2.0 and later. With MS-DOS versions 1.x, a parent program can use Interrupt 21H Function 26H to create a program segment prefix for a child but must carry out the loading, relocation, and execution of the child's code and data itself, without any assistance from the operating system.

How EXEC Works

When the EXEC function receives a request to execute a program, it first attempts to locate and open the specified program file. If the file cannot be found, EXEC fails immediately and returns an error code to the caller.

If the file exists, EXEC opens the file, determines its size, and inspects the first block of the file. If the first 2 bytes of the block are the ASCII characters *MZ*, the file is assumed to contain a .EXE load module, and the sizes of the program's code, data, and stack segments are obtained from the .EXE file header. Otherwise, the entire file is assumed to be an absolute load image (a .COM program). The actual filename extension (.COM or .EXE) is ignored in this determination.

At this point, the amount of memory needed to load the program is known, so EXEC attempts to allocate two blocks of memory: one to hold the new program's environment and one to contain the program's code, data, and stack segments. Assuming that enough memory is available to hold the program itself, the amount actually allocated to the program varies with its type. Programs of the .COM type are usually given all the free memory in the system (unless the memory area has previously become fragmented), whereas the amount assigned to a .EXE program is controlled by two fields in the file header, MINALLOC and MAXALLOC, that are set by the Microsoft Object Linker (LINK). *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program; PROGRAMMING TOOLS: The Microsoft Object Linker; PROGRAMMING UTILITIES: LINK.

EXEC then copies the environment from the parent into the memory allocated for child's environment, builds a program segment prefix (PSP) at the base of the child's program memory block, and copies into the child's PSP the command tail and the two default file control blocks passed by the parent. The previous contents of the terminate (Interrupt 22H), Control-C (Interrupt 23H), and critical error (Interrupt 24H) vectors are saved in the new PSP, and the terminate vector is updated so that control will return to the parent program when the child terminates or is aborted.

The actual code and data portions of the child program are then read from the disk file into the program memory block above the newly constructed PSP. If the child is a .EXE program, a relocation table in the file header is used to fix up segment references within the program to reflect its actual load address.

Finally, the EXEC function sets up the CPU registers and stack according to the program type and transfers control to the program. The entry point for a .COM file is always offset 100H within the program memory block (the first byte following the PSP). The entry point for a .EXE file is specified in the file header and can be anywhere within the program. *See also* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program.

When EXEC is used to load and execute an overlay rather than a child program, its operation is much simpler than described above. For an overlay, EXEC does not attempt to allocate memory or build a PSP or environment. It simply loads the contents of the file at the

address specified by the calling program and performs any necessary relocations (if the overlay file has a .EXE header), using a segment value that is also supplied by the caller. EXEC then returns to the program that invoked it, rather than transferring control to the code in the newly loaded file. The requesting program is responsible for calling the overlay at the appropriate location.

Using EXEC to Load a Program

When one program loads and executes another, it must follow these steps:

1. Ensure that enough free memory is available to hold the code, data, and stack of the child program.
2. Set up the information to be passed to EXEC and the child program.
3. Call the MS-DOS EXEC function to run the child program.
4. Recover and examine the child program's termination and return codes.

Making memory available

MS-DOS typically allocates all available memory to a .COM or .EXE program when it is loaded. (The infrequent exceptions to this rule occur when the transient program area is fragmented by the presence of resident data or programs or when a .EXE program is loaded that was linked with the /CPARMAXALLOC switch or modified with EXEMOD.) Therefore, before a program can load another program, it must free any memory it does not need for its own code, data, and stack.

The extra memory is released with a call to the MS-DOS Resize Memory Block function (Interrupt 21H Function 4AH). In this case, the segment address of the parent's PSP is passed in the ES register, and the BX register holds the number of paragraphs of memory the program must retain for its own use. If the prospective parent is a .COM program, it must be certain to move its stack to a safe area if it is reducing its memory allocation to less than 64 KB.

Preparing parameters for EXEC

When used to load and execute a program, the EXEC function must be supplied with two principal parameters:

- The address of the child program's pathname
- The address of a parameter block

The parameter block, in turn, contains the addresses of information to be passed to the child program.

The program name

The pathname for the child program must be an unambiguous, null-terminated (ASCIIZ) file specification (no wildcard characters). If a path is not included, the current directory is searched for the program; if a drive specifier is not present, the default drive is used.

The parameter block

The parameter block contains the addresses of four data items (Figure 10-1):

- The environment block
- The command tail
- The two default file control blocks (FCBs)

The position reserved in the parameter block for the pointer to an environment is only 2 bytes and contains a segment address, because an environment is always paragraph aligned (its address is always evenly divisible by 16); a value of 0000H indicates the parent program's environment should be inherited unchanged. The remaining three addresses are all doubleword addresses in the standard Intel format, with an offset value in the lower word and a segment value in the upper word.

To Call

AH	= 4BH	
AL	= 00H	load and execute child process
	03H	load overlay
DS:DX	= segment:offset of ASCIIZ pathname for an executable program file	
ES:BX	= segment:offset of parameter block	

Returns

If function is successful:
Carry flag is clear.
Other registers are preserved if MS-DOS version 3.0 or later, destroyed if MS-DOS versions 2.x.

If function is not successful:
Carry flag is set.

AX = error code

Parameter Block Format

Offset	Contents
If AL = 00H (load and execute program):	
00H	Segment pointer of the environment to be passed
02H	Offset of command-line tail for the new PSP
04H	Segment of command-line tail for the new PSP
06H	Offset of first file control block, to be copied into new PSP at offset 5CH
08H	Segment of first file control block
0AH	Offset of second file control block, to be copied into new PSP at offset 6CH
0CH	Segment of second file control block
If AL = 03H (load overlay):	
00H	Segment address where overlay is to be loaded
02H	Relocation factor to apply to loaded image

Figure 10-1. Synopsis of calling conventions for the MS-DOS EXEC function (Interrupt 21H Function 4BH), which can be used to load and execute child processes or overlays.

The environment

An environment always begins on a paragraph boundary and is composed of a series of null-terminated (ASCIIZ) strings of the form:

name=variable

The end of the entire set of strings is indicated by an additional null byte.

If the environment pointer in the parameter block supplied to an EXEC call contains zero, the child simply acquires a copy of the parent's environment. The parent can, however, provide a segment pointer to a different or expanded set of strings. In either case, under MS-DOS versions 3.0 and later, EXEC appends the child program's fully qualified pathname to its environment block. The maximum size of an environment is 32 KB, so very large amounts of information can be passed between programs by this mechanism.

The original, or master, environment for the system is owned by the command processor that is loaded when the system is turned on or restarted (usually COMMAND.COM). Strings are placed in the system's master environment by COMMAND.COM as a result of PATH, SHELL, PROMPT, and SET commands, with default values always present for the first two. For example, if an MS-DOS version 3.2 system is started from drive C and a PATH command is not present in the AUTOEXEC.BAT file nor a SHELL command in the CONFIG.SYS file, the master environment will contain the two strings:

PATH=
COMSPEC=C:\COMMAND.COM

These specifications are used by COMMAND.COM to search for executable "external" commands and to find its own executable file on the disk so that it can reload its transient portion when necessary. When the PROMPT string is present (as a result of a previous PROMPT or SET PROMPT command), COMMAND.COM uses it to tailor the prompt displayed to the user.

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F 0123456789ABCDEF
0000 43 4F 4D 53 50 45 43 3D 43 3A 5C 43 4F 4D 4D 41 COMSPEC=C:\COMMA
0010 4E 44 2E 43 4F 4D 00 50 52 4F 4D 50 54 3D 24 70 ND.COM.PROMPT=$p
0020 24 5F 24 64 20 20 20 24 74 24 68 24 68 24 68 24 $_$d   $t$h$h$h$
0030 68 24 68 24 68 20 24 71 24 71 24 67 00 50 41 54 h$h$h $q$q$g.PAT
0040 48 3D 43 3A 5C 53 59 53 54 45 4D 3B 43 3A 5C 41 H=C:\SYSTEM;C:\A
0050 53 4D 3B 43 3A 5C 57 53 3B 43 3A 5C 45 54 48 45 SM;C:\WS;C:\ETHE
0060 52 4E 45 54 3B 43 3A 5C 46 4F 52 54 48 5C 50 43 RNET;C:\FORTH\PC
0070 33 31 3B 00 00 01 00 43 3A 5C 46 4F 52 54 48 5C 31;....C:\FORTH\
0080 50 43 33 31 5C 46 4F 52 54 48 2E 43 4F 4D 00    PC31\FORTH.COM.
```

Figure 10-2. Dump of a typical environment under MS-DOS version 3.2. This particular example contains the default COMSPEC parameter and two relatively complex PATH and PROMPT control strings that were set up by entries in the user's AUTOEXEC file. Note the two null bytes at offset 73H, which indicate the end of the environment. These bytes are followed by the pathname of the program that owns the environment.

Other strings in the environment are used only for informational purposes by transient programs and do not affect the operation of the operating system proper. For example, the Microsoft C Compiler and the Microsoft Object Linker look in the environment for INCLUDE, LIB, and TMP strings that specify the location of *include* files, library files, and temporary working files. Figure 10-2 contains a hex dump of a typical environment block.

The command tail

The command tail to be passed to the child program takes the form of a byte indicating the length of the remainder of the command tail, followed by a string of ASCII characters terminated with an ASCII carriage return (0DH); the carriage return is not included in the length byte. The command tail can include switches, filenames, and other parameters that can be inspected by the child program and used to influence its operation. It is copied into the child program's PSP at offset 80H.

When COMMAND.COM uses EXEC to run a program, it passes a command tail that includes everything the user typed in the command line except the name of the program and any redirection parameters. I/O redirection is processed within COMMAND.COM itself and is manifest in the behavior of the standard device handles that are inherited by the child program. Any other program that uses EXEC to run a child program must try to perform any necessary redirection on its own and must supply an appropriate command tail so that the child program will behave as though it had been loaded by COMMAND.COM.

The default file control blocks

The two default FCBs pointed to by the EXEC parameter block are copied into the child program's PSP at offsets 5CH and 6CH. *See also* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management.

Few of the currently popular application programs use FCBs for file and record I/O because FCBs do not support the hierarchical directory structure. But some programs do inspect the default FCBs as a quick way to isolate the first two switches or other parameters from the command tail. Therefore, to make its own identity transparent to the child program, the parent should emulate the action of COMMAND.COM by parsing the first two parameters of the command tail into the default FCBs. This can be conveniently accomplished with the MS-DOS function Parse Filename (Interrupt 21H Function 29H).

If the child program does not require one or both of the default FCBs, the corresponding address in the parameter block can be initialized to point to two dummy FCBs in the application's memory space. These dummy FCBs should consist of 1 zero byte followed by 11 bytes containing ASCII blank characters (20H).

Running the child program

After the parent program has constructed the necessary parameters, it can invoke the EXEC function by issuing Interrupt 21H with the registers set as follows:

AH	= 4BH
AL	= 00H (EXEC subfunction to load and execute program)
DS:DX	= segment:offset of program pathname
ES:BX	= segment:offset of parameter block

Upon return from the software interrupt, the parent must test the carry flag to determine whether the child program did, in fact, run. If the carry flag is clear, the child program was successfully loaded and given control. If the carry flag is set, the EXEC function failed, and the error code returned in AX can be examined to determine why. The usual reasons are

- The specified file could not be found.
- The file was found, but not enough memory was free to load it.

Other causes are uncommon and can be symptoms of more severe problems in the system as a whole (such as damage to disk files or to the memory image of MS-DOS). With MS-DOS versions 3.0 and later, additional details about the cause of an EXEC failure can be obtained by subsequently calling Interrupt 21H Function 59H (Get Extended Error Information).

In general, supplying either an invalid address for an EXEC parameter block or invalid addresses within the parameter block itself does *not* cause a failure of the EXEC function, but may result in the child program behaving in unexpected ways.

Special considerations

With MS-DOS versions 2.x, the previous contents of all the parent registers except for CS:IP can be destroyed after an EXEC call, including the stack pointer in SS:SP. Consequently, before issuing the EXEC call, the parent must push onto the stack the contents of any registers that it needs to preserve, and then it must save the stack segment and offset in a location that is addressable with the CS segment register. Upon return, the stack segment and offset can be loaded into SS:SP with code segment overrides, and then the other registers can be restored by popping them off the stack. With MS-DOS versions 3.0 and later, registers are preserved across an EXEC call in the usual fashion.

Note: The code segments of Windows applications that use this technique should be given the IMPURE attribute.

In addition, a bug in MS-DOS version 2.0 and in PC-DOS versions 2.0 and 2.1 causes an arbitrary doubleword in the parent's stack segment to be destroyed during an EXEC call. When the parent is a .COM program and SS = PSP, the damaged location falls within the PSP and does no harm; however, in the case of a .EXE parent where DS = SS, the affected location may overlap the data segment and cause aberrant behavior or even a crash after the return from EXEC. This bug was fixed in MS-DOS versions 2.11 and later and in PC-DOS versions 3.0 and later.

Examining the child program's return codes

If the EXEC function succeeds, the parent program can call Interrupt 21H Function 4DH (Get Return Code of Child Process) to learn whether the child executed normally to completion and passed back a return code or was terminated by the operating system because of an external event. Function 4DH returns

AH = termination type:

00H	Child terminated normally (that is, exited via Interrupt 20H or Interrupt 21H Function 00H or Function 4CH).
01H	Child was terminated by user's entry of a Ctrl-C.
02H	Child was terminated by critical error handler (either the user responded with *A* to the *Abort, Retry, Ignore* prompt from the system's default Interrupt 24H handler, or a custom Interrupt 24H handler returned to MS-DOS with action code = 02H in register AL).
03H	Child terminated normally and stayed resident (that is, exited via Interrupt 21H Function 31H or Interrupt 27H).

AL = return code:

Value passed by the child program in register AL when it terminated with Interrupt 21H Function 4CH or 31H.

00H if the child terminated using Interrupt 20H, Interrupt 27H, or Interrupt 21H Function 00H.

These values are only guaranteed to be returned once by Function 4DH. Thus, a subsequent call to Function 4DH, without an intervening EXEC call, does not necessarily return any useful information. Additionally, if Function 4DH is called without a preceding successful EXEC call, the returned values are meaningless.

Using COMMAND.COM with EXEC

An application program can "shell" to MS-DOS — that is, provide the user with an MS-DOS prompt without terminating — by using EXEC to load and execute a secondary copy of COMMAND.COM with an empty command tail. The application can obtain the location of the COMMAND.COM disk file by inspecting its own environment for the COMSPEC string. The user returns to the application from the secondary command processor by typing *exit* at the COMMAND.COM prompt.

Batch-file interpretation is carried out by COMMAND.COM, and a batch (.BAT) file cannot be called using the EXEC function directly. Similarly, the sequential search for .COM, .EXE, and .BAT files in all the locations specified in the environment's PATH variable is a function of COMMAND.COM, rather than of EXEC. To execute a batch file or search the system path for a program, an application program can use EXEC to load and execute a secondary copy of COMMAND.COM to use as an intermediary. The application finds the location of COMMAND.COM as described in the preceding paragraph, but it passes a command tail in the form:

/C *program parameter1 parameter2* ...

where *program* is the .EXE, .COM, or .BAT file to be executed. When *program* terminates, the secondary copy of COMMAND.COM exits and returns control to the parent.

A parent and child example

The source programs PARENT.ASM in Figure 10-3 and CHILD.ASM in Figure 10-4 illustrate how one program uses EXEC to load another.

```
        name    parent
        title   'PARENT --- demonstrate EXEC call'
;
; PARENT.EXE --- demonstration of EXEC to run process
;
; Uses MS-DOS EXEC (Int 21H Function 4BH Subfunction 00H)
; to load and execute a child process named CHILD.EXE,
; then displays CHILD's return code.
;
; Ray Duncan, June 1987
;

stdin   equ     0                       ; standard input
stdout  equ     1                       ; standard output
stderr  equ     2                       ; standard error

stksize equ     128                     ; size of stack

cr      equ     0dh                     ; ASCII carriage return
lf      equ     0ah                     ; ASCII linefeed


DGROUP  group   _DATA,_ENVIR,_STACK


_TEXT   segment byte public 'CODE'      ; executable code segment

        assume  cs:_TEXT,ds:_DATA,ss:_STACK


stk_seg dw      ?                       ; original SS contents
stk_ptr dw      ?                       ; original SP contents


main    proc    far                     ; entry point from MS-DOS

        mov     ax,_DATA                ; set DS = our data segment
        mov     ds,ax

                                        ; now give back extra memory
                                        ; so child has somewhere to run...
```

Figure 10-3. PARENT.ASM, source code for PARENT.EXE. *(more)*

```
mov     ax,es                   ; let AX = segment of PSP base
mov     bx,ss                   ; and BX = segment of stack base
sub     bx,ax                   ; reserve seg stack - seg psp
add     bx,stksize/16           ; plus paragraphs of stack
mov     ah,4ah                  ; fxn 4AH = modify memory block
int     21h
jc      main1
                                ; display parent message ...
mov     dx,offset DGROUP:msg1   ; DS:DX = address of message
mov     cx,msg1_len             ; CX = length of message
call    pmsg

push    ds                      ; save parent's data segment
mov     stk_seg,ss              ; save parent's stack pointer
mov     stk_ptr,sp

                                ; now EXEC the child process...
mov     ax,ds                   ; set ES = DS
mov     es,ax
mov     dx,offset DGROUP:cname  ; DS:DX = child pathname
mov     bx,offset DGROUP:pars   ; ES:BX = parameter block
mov     ax,4b00h                ; function 4BH subfunction 00H
int     21h                     ; transfer to MS-DOS

cli                             ; (for bug in some early 8088s)
mov     ss,stk_seg              ; restore parent's stack pointer
mov     sp,stk_ptr
sti                             ; (for bug in some early 8088s)
pop     ds                      ; restore DS = our data segment

jc      main2                   ; jump if EXEC failed

                                ; otherwise EXEC succeeded,
                                ; convert and display child's
                                ; termination and return codes...
mov     ah,4dh                  ; fxn 4DH = get return code
int     21h                     ; transfer to MS-DOS
xchg    al,ah                   ; convert termination code
mov     bx,offset DGROUP:msg4a
call    b2hex
mov     al,ah                   ; get back return code
mov     bx,offset DGROUP:msg4b  ; and convert it
call    b2hex
mov     dx,offset DGROUP:msg4   ; DS:DX = address of message
mov     cx,msg4_len             ; CX = length of message
call    pmsg                    ; display it

mov     ax,4c00h                ; no error, terminate program
int     21h                     ; with return code = 0
```

Figure 10-3. Continued. *(more)*

```
main1:  mov     bx,offset DGROUP:msg2a  ; convert error code
        call    b2hex
        mov     dx,offset DGROUP:msg2   ; display message 'Memory
        mov     cx,msg2_len             ; resize failed...'
        call    pmsg
        jmp     main3

main2:  mov     bx,offset DGROUP:msg3a  ; convert error code
        call    b2hex
        mov     dx,offset DGROUP:msg3   ; display message 'EXEC
        mov     cx,msg3_len             ; call failed...'
        call    pmsg

main3:  mov     ax,4c01h                ; error, terminate program
        int     21h                     ; with return code = 1

main    endp                            ; end of main procedure


b2hex   proc    near                    ; convert byte to hex ASCII
                                        ; call with AL = binary value
                                        ;           BX = addr to store string
        push    ax
        shr     al,1
        shr     al,1
        shr     al,1
        shr     al,1
        call    ascii                   ; become first ASCII character
        mov     [bx],al                 ; store it
        pop     ax
        and     al,0fh                  ; isolate lower 4 bits, which
        call    ascii                   ; become the second ASCII character
        mov     [bx+1],al               ; store it
        ret
b2hex   endp


ascii   proc    near                    ; convert value 00-0FH in AL
        add     al,'0'                  ; into a "hex ASCII" character
        cmp     al,'9'
        jle     ascii2                  ; jump if in range 00-09H,
        add     al,'A'-'9'-1            ; offset it to range 0A-0FH,

ascii2: ret                             ; return ASCII char. in AL
ascii   endp


pmsg    proc    near                    ; displays message on standard output
                                        ; call with DS:DX = address,
                                        ;             CX = length
```

Figure 10-3. Continued.

(more)

```
        mov     bx,stdout               ; BX = standard output handle
        mov     ah,40h                  ; function 40H = write file/device
        int     21h                     ; transfer to MS-DOS
        ret                             ; back to caller

pmsg    endp

_TEXT   ends


_DATA   segment para public 'DATA'      ; static & variable data segment

cname   db      'CHILD.EXE',0           ; pathname of child process

pars    dw      _ENVIR                  ; segment of environment block
        dd      tail                    ; long address, command tail
        dd      fcb1                    ; long address, default FCB #1
        dd      fcb2                    ; long address, default FCB #2

tail    db      fcb1-tail-2             ; command tail for child
        db      'dummy command tail',cr

fcb1    db      0                       ; copied into default FCB #1 in
        db      11 dup (' ')            ; child's program segment prefix
        db      25 dup (0)

fcb2    db      0                       ; copied into default FCB #2 in
        db      11 dup (' ')            ; child's program segment prefix
        db      25 dup (0)

msg1    db      cr,lf,'Parent executing!',cr,lf
msg1_len equ    $-msg1

msg2    db      cr,lf,'Memory resize failed, error code='
msg2a   db      'xxh.',cr,lf
msg2_len equ    $-msg2

msg3    db      cr,lf,'EXEC call failed, error code='
msg3a   db      'xxh.',cr,lf
msg3_len equ    $-msg3

msg4    db      cr,lf,'Parent regained control!'
        db      cr,lf,'Child termination type='
msg4a   db      'xxh, return code='
msg4b   db      'xxh.',cr,lf
msg4_len equ    $-msg4

_DATA   ends


_ENVIR  segment para public 'DATA'      ; example environment block
                                        ; to be passed to child
```

Figure 10-3. Continued. *(more)*

```
        db      'PATH=',0               ; basic PATH, PROMPT,
        db      'PROMPT=$p$_$n$g',0     ; and COMSPEC strings
        db      'COMSPEC=C:\COMMAND.COM',0
        db      0                       ; extra null terminates block

_ENVIR  ends


_STACK  segment para stack 'STACK'

        db      stksize dup (?)

_STACK  ends


        end     main                    ; defines program entry point
```

Figure 10-3. Continued.

```
         name      child
         title     'CHILD process'
;
; CHILD.EXE --- a simple process loaded by PARENT.EXE
; to demonstrate the MS-DOS EXEC call, Subfunction 00H.
;
; Ray Duncan, June 1987
;

stdin   equ     0                       ; standard input
stdout  equ     1                       ; standard output
stderr  equ     2                       ; standard error

cr      equ     0dh                     ; ASCII carriage return
lf      equ     0ah                     ; ASCII linefeed


DGROUP  group   _DATA,STACK


_TEXT   segment byte public 'CODE'      ; executable code segment

        assume  cs:_TEXT,ds:_DATA,ss:STACK

main    proc    far                     ; entry point from MS-DOS

        mov     ax,_DATA                ; set DS = our data segment
        mov     ds,ax

                                        ; display child message ...
```

Figure 10-4. CHILD.ASM, source code for CHILD.EXE. *(more)*

```
        mov     dx,offset msg           ; DS:DX = address of message
        mov     cx,msg_len              ; CX = length of message
        mov     bx,stdout               ; BX = standard output handle
        mov     ah,40h                  ; AH = fxn 40H, write file/device
        int     21h                     ; transfer to MS-DOS
        jc      main2                   ; jump if any error

        mov     ax,4c00h                ; no error, terminate child
        int     21h                     ; with return code = 0

main2:  mov     ax,4c01h                ; error, terminate child
        int     21h                     ; with return code = 1

main    endp                            ; end of main procedure

_TEXT   ends


_DATA   segment para public 'DATA'      ; static & variable data segment

msg     db      cr,lf,'Child executing!',cr,lf
msg_len equ     $-msg

_DATA   ends


STACK   segment para stack 'STACK'

        dw      64 dup (?)

STACK   ends


        end     main                    ; defines program entry point
```

Figure 10-4. Continued.

PARENT.ASM can be assembled and linked into the executable program PARENT.EXE with the following commands:

```
C>MASM PARENT;  <Enter>
C>LINK PARENT;  <Enter>
```

Similarly, CHILD.ASM can be assembled and linked into the file CHILD.EXE as follows:

```
C>MASM CHILD;  <Enter>
C>LINK CHILD;  <Enter>
```

When PARENT.EXE is executed with the command

```
C>PARENT  <Enter>
```

PARENT reduces the size of its main memory block with a call to Interrupt 21H Function 4AH, to maximize the amount of free memory in the system, and then calls the EXEC function to load and execute CHILD.EXE.

CHILD.EXE runs exactly as though it had been loaded directly by COMMAND.COM. CHILD resets the DS segment register to point to its own data segment, uses Interrupt 21H Function 40H to display a message on standard output, and then terminates using Interrupt 21H Function 4CH, passing a return code of zero.

When PARENT.EXE regains control, it first checks the carry flag to determine whether the EXEC call succeeded. If the EXEC call failed, PARENT displays an error message and terminates with Interrupt 21H Function 4CH, itself passing a nonzero return code to COMMAND.COM to indicate an error.

Otherwise, PARENT uses Interrupt 21H Function 4DH to obtain CHILD.EXE's termination type and return code, which it converts to ASCII and displays. PARENT then terminates using Interrupt 21H Function 4CH and passes a return code of zero to COMMAND.COM to indicate success. COMMAND.COM in turn receives control and displays a new user prompt.

Using EXEC to Load Overlays

Loading overlays with the EXEC function is much less complex than using EXEC to run another program. The main program, called the root segment, must carry out the following steps to load and execute an overlay:

1. Make a memory block available to receive the overlay.
2. Set up the overlay parameter block to be passed to the EXEC function.
3. Call the EXEC function to load the overlay.
4. Execute the code within the overlay by transferring to it with a far call.

The overlay itself can be constructed as either a memory image (.COM) or a relocatable (.EXE) file and need not be the same type as the root program. In either case, the overlay should be designed so that the entry point (or a pointer to the entry point) is at the beginning of the module after it is loaded. This allows the root and overlay modules to be maintained separately and avoids a need for the root to have "magical" knowledge of addresses within the overlay.

To prevent users from inadvertently running an overlay directly from the command line, overlay files should be assigned an extension other than .COM or .EXE. The most convenient method relates overlays to their root segment by assigning them the same filename but an extension such as .OVL or .OV1, .OV2, and so on.

Making memory available

If EXEC is to load a child program successfully, the parent must release memory. In contrast, EXEC loads an overlay into memory that *belongs* to the calling program. If the

root segment is a .COM program and has not explicitly released extra memory, the root segment program need only ensure that the system contains enough memory to load the overlay and that the overlay load address does not conflict with its own code, data, or stack areas.

If the root segment program was loaded from a .EXE file, no straightforward way exists for it to determine unequivocally how much memory it already owns. The simplest course is for the program to release all extra memory, as discussed earlier in the section on loading a child program, and then use the MS-DOS memory allocation function (Interrupt 21H Function 48H) to obtain a new block of memory that is large enough to hold the overlay.

Preparing overlay parameters

When it is used to load an overlay, the EXEC function requires two major parameters:

- The address of the pathname for the overlay file
- The address of an overlay parameter block

As for a child program, the pathname for the overlay file must be an unambiguous ASCIIZ file specification (again, no wildcard characters), and it must include an explicit extension. As before, if a path and/or drive are not included in the pathname, the current directory and default drive are used.

The overlay parameter block contains the segment address at which the overlay should be loaded and a fixup value to be applied to any relocatable items within the overlay file. If the overlay file is in .EXE format, the fixup value is typically the same as the load address; if the overlay is in memory-image (.COM) format, the fixup value should be zero. The EXEC function does not attempt to validate the load address or the fixup value or to ensure that the load address actually belongs to the calling program.

Loading and executing the overlay

After the root segment program has prepared the filename of the overlay file and the overlay parameter block, it can invoke the EXEC function to load the overlay by issuing an Interrupt 21H with the registers set as follows:

AH	= 4BH
AL	= 03H (EXEC subfunction to load overlay)
DS:DX	= segment:offset of overlay file pathname
ES:BX	= segment:offset of overlay parameter block

Upon return from Interrupt 21H, the root segment must test the carry flag to determine whether the overlay was loaded. If the carry flag is clear, the overlay file was located and brought into memory at the requested address. The overlay can then be entered by a far call and should exit back to the root segment with a far return.

If the carry flag is set, the overlay file was not found or some other (probably severe) system problem was encountered, and the AX register contains an error code. With MS-DOS

versions 3.0 and later, Interrupt 21H Function 59H can be used to get more information about the EXEC failure. An invalid load address supplied in the overlay parameter block does not (usually) cause the EXEC function itself to fail but may result in the disconcerting message *Memory Allocation Error, System Halted* when the root program terminates.

An overlay example

The source programs ROOT.ASM in Figure 10-5 and OVERLAY.ASM in Figure 10-6 demonstrate the use of EXEC to load a program overlay. The program ROOT.EXE is executable from the MS-DOS prompt; it represents the root segment of an application. OVERLAY is constructed as a .EXE file (although it is named OVERLAY.OVL because it cannot be run alone) and represents a subprogram that can be loaded by the root segment when and if it is needed.

```
        name    root
        title   'ROOT --- demonstrate EXEC overlay'
;
; ROOT.EXE --- demonstration of EXEC for overlays
;
; Uses MS-DOS EXEC (Int 21H Function 4BH Subfunction 03H)
; to load an overlay named OVERLAY.OVL, calls a routine
; within the OVERLAY, then recovers control and terminates.
;
; Ray Duncan, June 1987
;

stdin   equ     0                       ; standard input
stdout  equ     1                       ; standard output
stderr  equ     2                       ; standard error

stksize equ     128                     ; size of stack

cr      equ     0dh                     ; ASCII carriage return
lf      equ     0ah                     ; ASCII linefeed


DGROUP  group   _DATA,_STACK


_TEXT   segment byte public 'CODE'      ; executable code segment

        assume  cs:_TEXT,ds:_DATA,ss:_STACK


stk_seg dw      ?                       ; original SS contents
stk_ptr dw      ?                       ; original SP contents
```

Figure 10-5. ROOT.ASM, source code for ROOT.EXE. *(more)*

```
main    proc    far                     ; entry point from MS-DOS

        mov     ax,_DATA                ; set DS = our data segment
        mov     ds,ax

                                        ; now give back extra memory
        mov     ax,es                   ; AX = segment of PSP base
        mov     bx,ss                   ; BX = segment of stack base
        sub     bx,ax                   ; reserve seg stack - seg psp
        add     bx,stksize/16           ; plus paragraphs of stack
        mov     ah,4ah                  ; fxn 4AH = modify memory block
        int     21h                     ; transfer to MS-DOS
        jc      main1                   ; jump if resize failed

                                        ; display message 'Root
                                        ; segment executing...'
        mov     dx,offset DGROUP:msg1   ; DS:DX = address of message
        mov     cx,msg1_len             ; CX = length of message
        call    pmsg

                                        ; allocate memory for overlay
        mov     bx,1000h                ; get 64 KB (4096 paragraphs)
        mov     ah,48h                  ; fxn 48H, allocate mem block
        int     21h                     ; transfer to MS-DOS
        jc      main2                   ; jump if allocation failed

        mov     pars,ax                 ; set load address for overlay
        mov     pars+2,ax               ; set relocation segment for overlay
        mov     word ptr entry+2,ax     ; set segment of entry point

        push    ds                      ; save root's data segment
        mov     stk_seg,ss              ; save root's stack pointer
        mov     stk_ptr,sp

                                        ; now use EXEC to load overlay
        mov     ax,ds                   ; set ES = DS
        mov     es,ax
        mov     dx,offset DGROUP:oname  ; DS:DX = overlay pathname
        mov     bx,offset DGROUP:pars   ; ES:BX = parameter block
        mov     ax,4b03h                ; function 4BH, subfunction 03H
        int     21h                     ; transfer to MS-DOS

        cli                             ; (for bug in some early 8088s)
        mov     ss,stk_seg              ; restore root's stack pointer
        mov     sp,stk_ptr
        sti                             ; (for bug in some early 8088s)
        pop     ds                      ; restore DS = our data segment

        jc      main3                   ; jump if EXEC failed

                                        ; otherwise EXEC succeeded...
```

Figure 10-5. Continued. *(more)*

```
        push    ds                      ; save our data segment
        call    dword ptr entry         ; now call the overlay
        pop     ds                      ; restore our data segment

                                        ; display message that root
                                        ; segment regained control...
        mov     dx,offset DGROUP:msg5   ; DS:DX = address of message
        mov     cx,msg5_len             ; CX = length of message
        call    pmsg                    ; display it

        mov     ax,4c00h                ; no error, terminate program
        int     21h                     ; with return code = 0

main1:  mov     bx,offset DGROUP:msg2a  ; convert error code
        call    b2hex
        mov     dx,offset DGROUP:msg2   ; display message 'Memory
        mov     cx,msg2_len             ; resize failed...'
        call    pmsg
        jmp     main4

main2:  mov     bx,offset DGROUP:msg3a  ; convert error code
        call    b2hex
        mov     dx,offset DGROUP:msg3   ; display message 'Memory
        mov     cx,msg3_len             ; allocation failed...'
        call    pmsg
        jmp     main4

main3:  mov     bx,offset DGROUP:msg4a  ; convert error code
        call    b2hex
        mov     dx,offset DGROUP:msg4   ; display message 'EXEC
        mov     cx,msg4_len             ; call failed...'
        call    pmsg

main4:  mov     ax,4c01h                ; error, terminate program
        int     21h                     ; with return code = 1

main    endp                            ; end of main procedure


b2hex   proc    near                    ; convert byte to hex ASCII
                                        ; call with AL = binary value
                                        ; BX = addr to store string
        push    ax
        shr     al,1
        shr     al,1
        shr     al,1
        shr     al,1
        call    ascii                   ; become first ASCII character
        mov     [bx],al                 ; store it
        pop     ax
```

Figure 10-5. Continued.

(more)

```
        and     al,0fh                  ; isolate lower 4 bits, which
        call    ascii                   ; become the second ASCII character
        mov     [bx+1],al               ; store it
        ret
b2hex   endp


ascii   proc    near                    ; convert value 00-0FH in AL
        add     al,'0'                  ; into a "hex ASCII" character
        cmp     al,'9'
        jle     ascii2                  ; jump if in range 00-09H,
        add     al,'A'-'9'-1            ; offset it to range 0A-0FH,
ascii2: ret                             ; return ASCII char. in AL.
ascii   endp


pmsg    proc    near                    ; displays message on standard output
                                        ; call with DS:DX = address,
                                        ;             CX = length

        mov     bx,stdout               ; BX = standard output handle
        mov     ah,40h                  ; function 40H = write file/device
        int     21h                     ; transfer to MS-DOS
        ret                             ; back to caller

pmsg    endp

_TEXT   ends


_DATA   segment para public 'DATA'      ; static & variable data segment

oname   db      'OVERLAY.OVL',0         ; pathname of overlay file

pars    dw      0                       ; load address (segment) for file
        dw      0                       ; relocation (segment) for file

entry   dd      0                       ; entry point for overlay

msg1    db      cr,lf,'Root segment executing!',cr,lf
msg1_len equ    $-msg1

msg2    db      cr,lf,'Memory resize failed, error code='
msg2a   db      'xxh.',cr,lf
msg2_len equ    $-msg2

msg3    db      cr,lf,'Memory allocation failed, error code='
msg3a   db      'xxh.',cr,lf
msg3_len equ    $-msg3
```

Figure 10-5. Continued. *(more)*

```
msg4     db      cr,lf,'EXEC call failed, error code='
msg4a    db      'xxh.',cr,lf
msg4_len equ     $-msg4

msg5     db      cr,lf,'Root segment regained control!',cr,lf
msg5_len equ     $-msg5

_DATA    ends


_STACK   segment para stack 'STACK'

         db      stksize dup (?)

_STACK   ends


         end     main                    ; defines program entry point
```

Figure 10-5. Continued.

```
         name      overlay
         title     'OVERLAY segment'
;
; OVERLAY.OVL --- a simple overlay segment
; loaded by ROOT.EXE to demonstrate use of
; the MS-DOS EXEC call Subfunction 03H.
;
; The overlay does not contain a STACK segment
; because it uses the ROOT segment's stack.
;
; Ray Duncan, June 1987
;

stdin    equ     0                       ; standard input
stdout   equ     1                       ; standard output
stderr   equ     2                       ; standard error

cr       equ     0dh                     ; ASCII carriage return
lf       equ     0ah                     ; ASCII linefeed


_TEXT    segment byte public 'CODE'       ; executable code segment

         assume  cs:_TEXT,ds:_DATA
ovlay    proc    far                     ; entry point from root segment

         mov     ax,_DATA                ; set DS = local data segment
         mov     ds,ax
```

Figure 10-6. OVERLAY.ASM, source code for OVERLAY.OVL. *(more)*

```
                                        ; display overlay message ...
        mov     dx,offset msg           ; DS:DX = address of message
        mov     cx,msg_len              ; CX = length of message
        mov     bx,stdout               ; BX = standard output handle
        mov     ah,40h                  ; AH = fxn 40H, write file/device
        int     21h                     ; transfer to MS-DOS

        ret                             ; return to root segment

ovlay   endp                            ; end of ovlay procedure

_TEXT   ends


_DATA   segment para public 'DATA'      ; static & variable data segment

msg     db      cr,lf,'Overlay executing!',cr,lf
msg_len equ     $-msg

_DATA   ends

        end
```

Figure 10-6. Continued.

ROOT.ASM can be assembled and linked into the executable program ROOT.EXE with the following commands:

```
C>MASM ROOT;  <Enter>
C>LINK ROOT;  <Enter>
```

OVERLAY.ASM can be assembled and linked into the file OVERLAY.OVL by typing

```
C>MASM OVERLAY;  <Enter>
C>LINK OVERLAY,OVERLAY.OVL;  <Enter>
```

The Microsoft Object Linker will display the message

```
Warning: no stack segment
```

but this message can be ignored.

When ROOT.EXE is executed with the command

```
C>ROOT  <Enter>
```

it first shrinks its main memory block with a call to Interrupt 21H Function 4AH and then allocates a separate block for the overlay with Interrupt 21H Function 48H. Next, ROOT calls the EXEC function to load the file OVERLAY.OVL into the newly allocated memory block. If the EXEC function fails, ROOT displays an error message and terminates with Interrupt 21H Function 4CH, passing a nonzero return code to COMMAND.COM to indicate an error. If the EXEC function succeeds, ROOT saves the contents of its DS segment register and then enters the overlay through an indirect far call.

The overlay resets the DS segment register to point to its own data segment, displays a message using Interrupt 21H Function 40H, and then returns. Note that the main procedure of the overlay is declared with the far attribute to force the assembler to generate the opcode for a far return.

When ROOT regains control, it restores the DS segment register to point to its own data segment again and displays an additional message, also using Interrupt 21H Function 40H, to indicate that the overlay executed successfully. ROOT then terminates using Interrupt 21H Function 4CH, passing a return code of zero to indicate success, and control returns to COMMAND.COM.

Ray Duncan

Part C
Customizing MS-DOS

Article 11
Terminate-and-Stay-Resident Utilities

The MS-DOS Terminate and Stay Resident system calls (Interrupt 21H Function 31H and Interrupt 27H) allow the programmer to install executable code or program data in a reserved block of RAM, where it resides while other programs execute. Global data, interrupt handlers, and entire applications can be made RAM-resident in this way. Programs that use the MS-DOS terminate-and-stay-resident capability are commonly known as TSR programs or TSRs.

This article describes how to install a TSR in RAM, how to communicate with the resident program, and how the resident program can interact with MS-DOS. The discussion proceeds from a general description of the MS-DOS functions useful to TSR programmers to specific details about certain MS-DOS structural elements necessary to proper functioning of a TSR utility and concludes with two programming examples.

Note: Microsoft cannot guarantee that the information in this article will be valid for future versions of MS-DOS.

Structure of a Terminate-and-Stay-Resident Utility

The executable code and data in TSRs can be separated into RAM-resident and transient portions (Figure 11-1). The RAM-resident portion of a TSR contains executable code and data for an application that performs some useful function on demand. The transient portion installs the TSR; that is, it initializes data and interrupt handlers contained in the RAM-resident portion of the program and executes an MS-DOS Terminate and Stay Resident function call that leaves the RAM-resident portion in memory and frees the memory used by the transient portion. The code in the transient portion of a TSR runs when the .EXE or .COM file containing the program is executed; the code in the RAM-resident portion runs only when it is explicitly invoked by a foreground program or by execution of a hardware or software interrupt.

TSRs can be broadly classified as passive or active, depending on the method by which control is transferred to the RAM-resident program. A passive TSR executes only when another program explicitly transfers control to it, either through a software interrupt or by means of a long JMP or CALL. The calling program is not interrupted by the TSR, so the status of MS-DOS, the system BIOS, and the hardware is well defined when the TSR program starts to execute.

In contrast, an active TSR is invoked by the occurrence of some event external to the currently running (foreground) program, such as a sequence of user keystrokes or a predefined hardware interrupt. Therefore, when it is invoked, an active TSR almost always

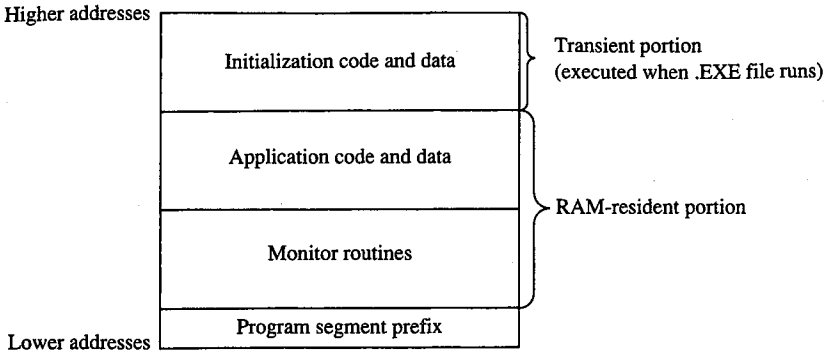


Figure 11-1. Organization of a TSR program in memory.

interrupts some other program and suspends its execution. To avoid disrupting the interrupted program, an active TSR must monitor the status of MS-DOS, the ROM BIOS, and the hardware and take control of the system only when it is safe to do so.

Passive TSRs are generally simpler in their construction than active TSRs because a passive TSR runs in the context of the calling program; that is, when the TSR executes, it assumes that it can use the calling program's program segment prefix (PSP), open files, current directory, and so on. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program. It is the calling program's responsibility to ensure that the hardware and MS-DOS are in a stable state before it transfers control to a passive TSR.

An active TSR, on the other hand, is invoked asynchronously; that is, the status of the hardware, MS-DOS, and the executing foreground program is indeterminate when the event that invokes the TSR occurs. Therefore, active TSRs require more complex code. The RAM-resident portion of an active TSR must contain modules that monitor the operating system to determine when control can safely be transferred to the application portion of the TSR. The monitor routines typically test the status of keyboard input, ROM BIOS interrupt processing, hardware interrupt processing, and MS-DOS function processing. The TSR activates the application (the part of the RAM-resident portion that performs the TSR's main task) only when it detects the appropriate keyboard input and determines that the application will not interfere with interrupt and MS-DOS function processing.

Keyboard input

An active TSR usually contains a RAM-resident module that examines keyboard input for a predetermined keystroke sequence called a "hot-key" sequence. A user executes the RAM-resident application by entering this hot-key sequence at the keyboard.

The technique used in the TSR to monitor keyboard input depends on the keyboard hardware implementation. On computers in the IBM PC and PS/2 families, the keyboard coprocessor generates an Interrupt 09H for each keypress. Therefore, a TSR can monitor user keystrokes by installing an interrupt handler (interrupt service routine, or ISR) for Interrupt 09H. This handler can thus detect a specified hot-key sequence.

ROM BIOS interrupt processing

The ROM BIOS routines in IBM PCs and PS/2s are not reentrant. An active TSR that calls the ROM BIOS must ensure that its code does not attempt to execute a ROM BIOS function that was already being executed by the foreground process when the TSR program took control of the system.

The IBM ROM BIOS routines are invoked through software interrupts, so an active TSR can monitor the status of the ROM BIOS by replacing the default interrupt handlers with custom interrupt handlers that intercept the appropriate BIOS interrupts. Each of these interrupt handlers can maintain a status flag, which it increments before transferring control to the corresponding ROM BIOS routine and decrements when the ROM BIOS routine has finished executing. Thus, the TSR monitor routines can test these flags to determine when non-reentrant BIOS routines are executing.

Hardware interrupt processing

The monitor routines of an active TSR, which may themselves be executed as the result of a hardware interrupt, should not activate the application portion of the TSR if any other hardware interrupt is being processed. On IBM PCs, for example, hardware interrupts are processed in a prioritized sequence determined by an Intel 8259A Programmable Interrupt Controller. The 8259A does not allow a hardware interrupt to execute if a previous interrupt with the same or higher priority is being serviced. All hardware interrupt handlers include code that signals the 8259A when interrupt processing is completed. (The programming interface to the 8259A is described in IBM's *Technical Reference* manuals and in Intel's technical literature.)

If a TSR were to interrupt the execution of another hardware interrupt handler before the handler signaled the 8259A that it had completed its interrupt servicing, subsequent hardware interrupts could be inhibited indefinitely. Inhibition of high-priority hardware interrupts such as the timer tick (Interrupt 08H) or keyboard interrupt (Interrupt 09H) could cause a system crash. For this reason, an active TSR must monitor the status of all hardware interrupt processing by interrogating the 8259A to ensure that control is transferred to the RAM-resident application only when no other hardware interrupts are being serviced.

MS-DOS function processing

Unlike the IBM ROM BIOS routines, MS-DOS is reentrant to a limited extent. That is, there are certain times when MS-DOS's servicing of an Interrupt 21H function call invoked by a foreground process can be suspended so that the RAM-resident application can make an Interrupt 21H function call of its own. For this reason, an active TSR must monitor operating system activity to determine when it is safe for the TSR application to make its calls to MS-DOS.

MS-DOS Support for Terminate-and-Stay-Resident Programs

Several MS-DOS system calls are useful for supporting terminate-and-stay-resident utilities. These are listed in Table 11-1. *See* SYSTEM CALLS.

Table 11-1. MS-DOS Functions Useful in TSR Programs.

Function Name	Call With	Returns	Comment
Terminate and Stay Resident	AH = 31H AL = return code DX = size of resident program (in 16-byte paragraphs) INT 21H	Nothing	Preferred over Interrupt 27H with MS-DOS versions 2.x and later
Terminate and Stay Resident	CS = PSP DX = size of resident program (bytes) INT 27H	Nothing	Provided for compatibility with MS-DOS versions 1.x
Set Interrupt Vector	AH = 25H AL = interrupt number DS:DX = address of interrupt handler INT 21H	Nothing	
Get Interrupt Vector	AH = 35H AL = interrupt number INT 21H	ES:BX = address of interrupt handler	
Set PSP Address	AH = 50H BX = PSP segment INT 21H	Nothing	
Get PSP Address	AH = 51H INT 21H	BX = PSP segment	
Set Extended Error Information	AX = 5D0AH DS:DX = address of 11-word data structure: word 0: register AX as returned by Function 59H word 1: register BX word 2: register CX word 3: register DX word 4: register SI word 5: register DI word 6: register DS word 7: register ES words 8–0AH: reserved; should be 0 INT 21H	Nothing	MS-DOS versions 3.1 and later

(more)

Table 11-1. *Continued.*

Function Name	Call With	Returns	Comment
Get Extended Error Information	AH = 59H BX = 0 INT 21H	AX = extended error code BH = error class BL = suggested action CH = error locus	
Set Disk Transfer Area Address	AH = 1AH DS:DX = address of DTA INT 21H	Nothing	
Get Disk Transfer Area Address	AH = 2FH INT 21H	ES:BX = address of current DTA	
Get InDOS Flag Address	AH = 34H INT 21H	ES:BX = address of InDOS flag	

Terminate-and-stay-resident functions

MS-DOS provides two mechanisms for terminating the execution of a program while leaving a portion of it resident in RAM. The preferred method is to execute Interrupt 21H Function 31H.

Interrupt 21H Function 31H

When this Interrupt 21H function is called, the value in DX specifies the amount of RAM (in paragraphs) that is to remain allocated after the program terminates, starting at the program segment prefix (PSP). The function is similar to Function 4CH (Terminate Process with Return Code) in that it passes a return code in AL, but it differs in that open files are not automatically closed by Function 31H.

Interrupt 27H

When Interrupt 27H is executed, the value passed in DX specifies the number of bytes of memory required for the RAM-resident program. MS-DOS converts the value passed in DX from bytes to paragraphs, sets AL to zero, and jumps to the same code that would be executed for Interrupt 21H Function 31H. Interrupt 27H is less flexible than Interrupt 21H Function 31H because it limits the size of the program that can remain resident in RAM to 64 KB, it requires that CS point to the base of the PSP, and it does not pass a return code. Later versions of MS-DOS support Interrupt 27H primarily for compatibility with versions 1.x.

TSR RAM management

In addition to the RAM explicitly allocated to the TSR by means of the value in DX, the RAM allocated to the TSR's environment remains resident when the installation portion of the TSR program terminates. (The paragraph address of the environment is found at

offset 2CH in the TSR's PSP.) Moreover, if the installation portion of a TSR program has used Interrupt 21H Function 48H (Allocate Memory Block) to allocate additional RAM, this memory also remains allocated when the program terminates. If the RAM-resident program does not need this additional RAM, the installation portion of the TSR program should free it explicitly by using Interrupt 21H Function 49H (Free Memory Block) before executing Interrupt 21H Function 31H.

Set and Get Interrupt Vector functions

Two Interrupt 21H function calls are available to inspect or update the contents of a specified 8086-family interrupt vector. Function 25H (Set Interrupt Vector) updates the vector of the interrupt number specified in the AL register with the segment and offset values specified in DS:DX. Function 35H (Get Interrupt Vector) performs the inverse operation: It copies the current vector of the interrupt number specified in AL into the ES:BX register pair.

Although it is possible to manipulate interrupt vectors directly, the use of Interrupt 21H Functions 25H and 35H is generally more convenient and allows for upward compatibility with future versions of MS-DOS.

Set and Get PSP Address functions

MS-DOS uses a program's PSP to keep track of certain data unique to the program, including command-line parameters and the segment address of the program's environment. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program. To access this information, MS-DOS maintains an internal variable that always contains the location of the PSP associated with the foreground process. When a RAM-resident application is activated, it should use Interrupt 21H Functions 50H (Set Program Segment Prefix Address) and 51H (Get Program Segment Prefix Address) to preserve the current contents of this variable and to update the variable with the location of its own PSP. Function 50H (Set Program Segment Prefix Address) updates an internal MS-DOS variable that locates the PSP currently in use by the foreground process. Function 51H (Get Program Segment Prefix Address) returns the contents of the internal MS-DOS variable to the caller.

Set and Get Extended Error Information functions

In MS-DOS versions 3.1 and later, the RAM-resident program should preserve the foreground process's extended error information so that, if the RAM-resident application encounters an MS-DOS error, the extended error information pertaining to the foreground process will still be available and can be restored. Interrupt 21H Functions 59H and 5D0AH provide a mechanism for the RAM-resident program to save and restore this information during execution of a TSR application.

Function 59H (Get Extended Error Information), which became available in version 3.0, returns detailed information on the most recently detected MS-DOS error. The inverse operation is performed by Function 5D0AH (Set Extended Error Information), which can be used only in MS-DOS versions 3.1 and later. This function copies extended error information to MS-DOS from a data structure defined in the calling program.

Set and Get Disk Transfer Area Address functions

Several MS-DOS data transfer functions, notably Interrupt 21H Functions 21H, 22H, 27H, and 28H (the Random Read and Write functions) and Interrupt 21H Functions 14H and 15H (the Sequential Read and Write functions), require a program to specify a disk transfer area (DTA). By default, a program's DTA is located at offset 80H in its program segment prefix. If a RAM-resident application calls an MS-DOS function that uses a DTA, the TSR should save the DTA address belonging to the interrupted program by using Interrupt 21H Function 2FH (Get Disk Transfer Area Address), supply its own DTA address to MS-DOS using Interrupt 21H Function 1AH (Set Disk Transfer Area Address), and then, before terminating, restore the interrupted program's DTA.

The MS-DOS idle interrupt (Interrupt 28H)

Several of the first 12 MS-DOS functions (01H through 0CH) must wait for the occurrence of an expected event such as a user keypress. These functions contain an "idle loop" in which looping continues until the event occurs. To provide a mechanism for other system activity to take place while the idle loop is executing, these MS-DOS functions execute an Interrupt 28H from within the loop.

The default MS-DOS handler for Interrupt 28H is only an IRET instruction. By supplying its own handler for Interrupt 28H, a TSR can perform some useful action at times when MS-DOS is otherwise idle. Specifically, a custom Interrupt 28H handler can be used to examine the current status of the system to determine whether or not it is safe to activate the RAM-resident application.

Determining MS-DOS Status

A TSR can infer the current status of MS-DOS from knowledge of its internal use of stacks and from a pair of internal status flags. This status information is essential to the proper execution of an active TSR because a RAM-resident application can make calls to MS-DOS only when those calls will not disrupt an earlier call made by the foreground process.

MS-DOS internal stacks

MS-DOS versions 2.0 and later may use any of three internal stacks: the I/O stack (*IOStack*), the disk stack (*DiskStack*), and the auxiliary stack (*AuxStack*). In general, *IOStack* is used for Interrupt 21H Functions 01H through 0CH and *DiskStack* is used for the remaining Interrupt 21H functions; *AuxStack* is normally used only when MS-DOS has detected a critical error and subsequently executed an Interrupt 24H. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Exception Handlers. Specifically, MS-DOS's internal stack use depends on which MS-DOS function is being executed and on the value of the critical error flag.

The critical error flag

The critical error flag (*ErrorMode*) is a 1-byte flag that MS-DOS uses to indicate whether or not a critical error has occurred. During normal, errorless execution, the value of the

critical error flag is zero. Whenever MS-DOS detects a critical error, it sets this flag to a nonzero value before it executes Interrupt 24H. If an Interrupt 24H handler subsequently invokes an MS-DOS function by using Interrupt 21H, the nonzero value of the critical error flag tells MS-DOS to use its auxiliary stack for Interrupt 21H Functions 01H through 0CH instead of using the I/O stack as it normally would.

In other words, when control is transferred to MS-DOS through Interrupt 21H, the function number and the critical error flag together determine MS-DOS stack use for the function. Figure 11-2 outlines the internal logic used on entry to an MS-DOS function to select which stack is to be used during processing of the function. As stated above, for Functions 01H through 0CH, MS-DOS uses *IOStack* if the critical error flag is zero and *AuxStack* if the flag is nonzero. For function numbers greater than 0CH, MS-DOS usually uses *DiskStack*, but Functions 50H, 51H, and 59H are important exceptions. Functions 50H and 51H use either *IOStack* (in versions 2.x) or the stack supplied by the calling program (in versions 3.x). In version 3.0, Function 59H uses either *IOStack* or *AuxStack*, depending on the value of the critical error flag, but in versions 3.1 and later, Function 59H always uses *AuxStack*.

MS-DOS versions 2.x

```
if   (FunctionNumber >= 01H and FunctionNumber <= 0CH)
     or
     FunctionNumber = 50H
     or
     FunctionNumber = 51H

then if   ErrorMode = 0
     then use IOStack
     else use AuxStack

else ErrorMode = 0
     use DiskStack
```

MS-DOS version 3.0

```
if   FunctionNumber = 50H
     or
     FunctionNumber = 51H
     or
     FunctionNumber = 62H

then use caller's stack

else if   (FunctionNumber >= 01H and FunctionNumber <= 0CH)
          or
          Function Number = 59H

     then if   ErrorMode = 0
          then use IOStack
          else use AuxStack

     else ErrorMode = 0
          use DiskStack
```

Figure 11-2. Strategy for use of MS-DOS internal stacks. *(more)*

MS-DOS versions 3.1 and later

```
if   FunctionNumber = 33H
     or
     FunctionNumber = 50H
     or
     FunctionNumber = 51H
     or
     FunctionNumber = 62H

then use caller's stack

else if     (FunctionNumber >= 01H and FunctionNumber <= 0CH)

     then if     ErrorMode = 0
          then use IOStack
          else use AuxStack

     else if FunctionNumber = 59H
          then use AuxStack
          else ErrorMode = 0
               use DiskStack
```

Figure 11-2. Continued.

This scheme makes Functions 01H through 0CH reentrant in a limited sense, in that a substitute critical error (Interrupt 24H) handler invoked while the critical error flag is nonzero can still use these Interrupt 21H functions. In this situation, because the flag is nonzero, *AuxStack* is used for Functions 01H through 0CH instead of *IOStack*. Thus, if *IOStack* is in use when the critical error is detected, its contents are preserved during the handler's subsequent calls to these functions.

The stack-selection logic differs slightly between MS-DOS versions 2 and 3. In versions 3.x, a few functions — notably 50H and 51H — avoid using any of the MS-DOS stacks. These functions perform uncomplicated tasks that make minimal demands for stack space, so the calling program's stack is assumed to be adequate for them.

The InDOS flag

InDOS is a 1-byte flag that is incremented each time an Interrupt 21H function is invoked and decremented when the function terminates. The flag's value remains nonzero as long as code within MS-DOS is being executed. The value of InDOS does not indicate which internal stack MS-DOS is using.

Whenever MS-DOS detects a critical error, it zeros InDOS before it executes Interrupt 24H. This action is taken to accommodate substitute Interrupt 24H handlers that do not return control to MS-DOS. If InDOS were not zeroed before such a handler gained control, its value would never be decremented and would therefore be incorrect during subsequent calls to MS-DOS.

The address of the 1-byte InDOS flag can be obtained from MS-DOS by using Interrupt 21H Function 34H (Return Address of InDOS Flag). In versions 3.1 and later, the 1-byte critical error flag is located in the byte preceding InDOS, so, in effect, the address of both

flags can be found using Function 34H. Unfortunately, there is no easy way to find the critical error flag in other versions. The recommended technique is to scan the MS-DOS segment, which is returned in the ES register by Function 34H, for one of the following sequences of instructions:

```
test    ss:[CriticalErrorFlag],0FFH      ;(versions 3.1 and later)
jne     NearLabel
push    ss:[NearWord]
int     28H
```

or

```
cmp     ss:[CriticalErrorFlag],00        ;(versions earlier than 3.1)
jne     NearLabel
int     28H
```

When the TEST or CMP instruction has been identified, the offset of the critical error flag can be obtained from the instruction's operand field.

The Multiplex Interrupt

The MS-DOS multiplex interrupt (Interrupt 2FH) provides a general mechanism for a program to verify the presence of a TSR and communicate with it. A program communicates with a TSR by placing an identification value in AH and a function number in AL and issuing an Interrupt 2FH. The TSR's Interrupt 2FH handler compares the value in AH to its own predetermined ID value. If they match, the TSR's handler keeps control and performs the function specified in the AL register. If they do not match, the TSR's handler relinquishes control to the previously installed Interrupt 2FH handler. (Multiplex ID values 00H through 7FH are reserved for use by MS-DOS; therefore, user multiplex numbers should be in the range 80H through 0FFH.)

The handler in the following example recognizes only one function, corresponding to AL = 00H. In this case, the handler returns the value 0FFH in AL, signifying that the handler is indeed resident in RAM. Thus, a program can detect the presence of the handler by executing Interrupt 2FH with the handler's ID value in AH and 00H in AL.

```
mov     ah,MultiplexID
mov     al,00H
int     2FH
cmp     al,0FFH
je      AlreadyInstalled
```

To ensure that the identification byte is unique, its value should be determined at the time the TSR is installed. One way to do this is to pass the value to the TSR program as a command-line parameter when the TSR program is installed. Another approach is to place the identification value in an environment variable. In this way, the value can be found in the environment of both the TSR and any other program that calls Interrupt 2FH to verify the TSR's presence.

In practice, the multiplex interrupt can also be used to pass information to and from a RAM-resident program in the CPU registers, thus providing a mechanism for a program to share control or status information with a TSR.

TSR Programming Examples

One effective way to become familiar with TSRs is to examine functional programs. Therefore, the subsequent pages present two examples: a simple passive TSR and a more complex active TSR.

HELLO.ASM

The "bare-bones" TSR in Figure 11-3 is a passive TSR. The RAM-resident application, which simply displays the message *Hello, World*, is invoked by executing a software interrupt. This example illustrates the fundamental interactions among a RAM-resident program, MS-DOS, and programs that execute after the installation of the RAM-resident utility.

```
;
; Name:          hello
;
; Description:   This RAM-resident (terminate-and-stay-resident) utility
;                displays the message "Hello, World" in response to a
;                software interrupt.
;
; Comments:      Assemble and link to create HELLO.EXE.
;
;                Execute HELLO.EXE to make resident.
;
;                Execute  INT 64h  to display the message.
;


TSRInt          EQU     64h
STDOUT          EQU     1

RESIDENT_TEXT   SEGMENT byte public 'CODE'
                ASSUME  cs:RESIDENT_TEXT,ds:RESIDENT_DATA

TSRAction       PROC    far

                sti                     ; enable interrupts

                push    ds              ; preserve registers
                push    ax
                push    bx
                push    cx
                push    dx
```

Figure 11-3. HELLO.ASM, a passive TSR. *(more)*

```
                mov     dx,seg RESIDENT_DATA
                mov     ds,dx
                mov     dx,offset Message       ; DS:DX -> message
                mov     cx,16                   ; CX = length
                mov     bx,STDOUT               ; BX = file handle
                mov     ah,40h                  ; AH = INT 21H function 40H
                                                ; (Write File)
                int     21h                     ; display the message

                pop     dx                      ; restore registers and exit
                pop     cx
                pop     bx
                pop     ax
                pop     ds
                iret

TSRAction       ENDP

RESIDENT_TEXT   ENDS


RESIDENT_DATA   SEGMENT word public 'DATA'

Message         DB      0Dh,0Ah,'Hello, World',0Dh,0Ah

RESIDENT_DATA   ENDS


TRANSIENT_TEXT  SEGMENT para public 'TCODE'
                ASSUME  cs:TRANSIENT_TEXT,ss:TRANSIENT_STACK

HelloTSR PROC   far                     ; At entry:     CS:IP -> SnapTSR
                                        ;               SS:SP -> stack
                                        ;               DS,ES -> PSP
; Install this TSR's interrupt handler

                mov     ax,seg RESIDENT_TEXT
                mov     ds,ax
                mov     dx,offset RESIDENT_TEXT:TSRAction
                mov     al,TSRInt
                mov     ah,25h
                int     21h

; Terminate and stay resident

                mov     dx,cs           ; DX = paragraph address of start of
                                        ; transient portion (end of resident
                                        ; portion)
                mov     ax,es           ; ES = PSP segment
                sub     dx,ax           ; DX = size of resident portion
```

Figure 11-3. Continued. *(more)*

```
                mov     ax,3100h        ; AH = INT 21H function number (TSR)
                                        ; AL = 00H (return code)
                int     21h

HelloTSR        ENDP

TRANSIENT_TEXT  ENDS


TRANSIENT_STACK SEGMENT word stack 'TSTACK'

                DB      80h dup(?)

TRANSIENT_STACK ENDS


                END     HelloTSR
```

Figure 11-3. Continued.

The transient portion of the program (in the segments *TRANSIENT_TEXT* and *TRANSIENT_STACK*) runs only when the file HELLO.EXE is executed. This installation code updates an interrupt vector to point to the resident application (the procedure *TSRAction*) and then calls Interrupt 21H Function 31H to terminate execution, leaving the segments *RESIDENT_TEXT* and *RESIDENT_DATA* in RAM.

The order in which the code and data segments appear in the listing is important. It ensures that when the program is executed as a .EXE file, the resident code and data are placed in memory at lower addresses than the transient code and data. Thus, when Interrupt 21H Function 31H is called, the memory occupied by the transient portion of the program is freed without disrupting the code and data in the resident portion.

The RAM containing the resident portion of the utility is left intact by MS-DOS during subsequent execution of other programs. Thus, after the TSR has been installed, any program that issues the software interrupt recognized by the TSR (in this example, Interrupt 64H) will transfer control to the routine *TSRAction*, which uses Interrupt 21H Function 40H to display a simple message on standard output.

Part of the reason this example is so short is that it performs no error checking. A truly reliable version of the program would check the version of MS-DOS in use, verify that the program was not already installed in memory, and chain to any previously installed interrupt handlers that use the same interrupt vector. (The next program, SNAP.ASM, illustrates these techniques.) However, the primary reason the program is small is that it makes the basic assumption that MS-DOS, the ROM BIOS, and the hardware interrupts are all stable at the time the resident utility is executed.

SNAP.ASM

The preceding assumption is a reliable one in the case of the passive TSR in Figure 11-3, which executes only when it is explicitly invoked by a software interrupt. However, the situation is much more complicated in the case of the active TSR in Figure 11-4. This

program is relatively long because it makes no assumptions about the stability of the operating environment. Instead, it monitors the status of MS-DOS, the ROM BIOS, and the hardware interrupts to decide when the RAM-resident application can safely execute.

```
;
; Name:         snap
;
; Description:  This RAM-resident (terminate-and-stay-resident) utility
;               produces a video "snapshot" by copying the contents of the
;               video regeneration buffer to a disk file.  It may be used
;               in 80-column alphanumeric video modes on IBM PCs and PS/2s.
;
; Comments:     Assemble and link to create SNAP.EXE.
;
;               Execute SNAP.EXE to make resident.
;
;               Press Alt-Enter to dump current contents of video buffer
;               to a disk file.
;

MultiplexID     EQU     0CAh            ; unique INT 2FH ID value

TSRStackSize    EQU     100h            ; resident stack size in bytes

KB_FLAG         EQU     17h             ; offset of shift-key status flag in
                                        ; ROM BIOS keyboard data area

KBIns           EQU     80h             ; bit masks for KB_FLAG
KBCaps          EQU     40h
KBNum           EQU     20h
KBScroll        EQU     10h
KBAlt           EQU     8
KBCtl           EQU     4
KBLeft          EQU     2
KBRight         EQU     1

SCEnter         EQU     1Ch

CR              EQU     0Dh
LF              EQU     0Ah
TRUE            EQU     -1
FALSE           EQU     0

                PAGE
;-----------------------------------------------------------------------------
;
; RAM-resident routines
;
;-----------------------------------------------------------------------------

RESIDENT_GROUP  GROUP   RESIDENT_TEXT,RESIDENT_DATA,RESIDENT_STACK
```

Figure 11-4. SNAP.ASM, a video snapshot TSR. *(more)*

```
RESIDENT_TEXT   SEGMENT byte public 'CODE'
                ASSUME  cs:RESIDENT_GROUP,ds:RESIDENT_GROUP

;--------------------------------------------------------------------------
; System verification routines
;--------------------------------------------------------------------------

VerifyDOSState  PROC    near            ; Returns:    carry flag set if MS-DOS
                                        ;             is busy
                push    ds              ; preserve these registers
                push    bx
                push    ax

                lds     bx,cs:ErrorModeAddr
                mov     ah,[bx]         ; AH = ErrorMode flag

                lds     bx,cs:InDOSAddr
                mov     al,[bx]         ; AL = InDOS flag

                xor     bx,bx           ; BH = 00H, BL = 00H
                cmp     bl,cs:InISR28   ; carry flag set if INT 28H handler
                                        ; is running
                rcl     bl,01h          ; BL = 01H if INT 28H handler is running

                cmp     bx,ax           ; carry flag zero if AH = 00H
                                        ; and AL <= BL
                pop     ax              ; restore registers
                pop     bx
                pop     ds
                ret

VerifyDOSState  ENDP


VerifyIntState  PROC    near            ; Returns:    carry flag set if hardware
                                        ;             or ROM BIOS unstable

                push    ax              ; preserve AX

; Verify hardware interrupt status by interrogating Intel 8259A Programmable
;  Interrupt Controller

                mov     ax,00001011b    ; AH = 0
                                        ; AL = 0CW3 for Intel 8259A (RR = 1,
                                        ; RIS = 1)
                out     20h,al          ; request 8259A's in-service register
                jmp     short L10       ; wait a few cycles
L10:            in      al,20h          ; AL = hardware interrupts currently
                                        ; being serviced (bit = 1 if in-service)
```

Figure 11-4. Continued. *(more)*

```
                cmp     ah,al
                jc      L11             ; exit if any hardware interrupts still
                                        ; being serviced

; Verify status of ROM BIOS interrupt handlers

                xor     al,al           ; AL = 00H

                cmp     al,cs:InISR5
                jc      L11             ; exit if currently in INT 05H handler

                cmp     al,cs:InISR9
                jc      L11             ; exit if currently in INT 09H handler

                cmp     al,cs:InISR10
                jc      L11             ; exit if currently in INT 10H handler

                cmp     al,cs:InISR13   ; set carry flag if currently in
                                        ; INT 13H handler
L11:            pop     ax              ; restore AX and return
                ret

VerifyIntState  ENDP


VerifyTSRState  PROC    near            ; Returns: carry flag set if TSR
                                        ;          inactive
                rol     cs:HotFlag,1    ; carry flag set if (HotFlag = TRUE)
                cmc                     ; carry flag set if (HotFlag = FALSE)
                jc      L20             ; exit if no hot key

                ror     cs:ActiveTSR,1  ; carry flag set if (ActiveTSR = TRUE)
                jc      L20             ; exit if already active

                call    VerifyDOSState
                jc      L20             ; exit if MS-DOS unstable

                call    VerifyIntState  ; set carry flag if hardware or BIOS
                                        ; unstable
L20:            ret

VerifyTSRState  ENDP

                PAGE
;-----------------------------------------------------------------------------
; System monitor routines
;-----------------------------------------------------------------------------

ISR5            PROC    far             ; INT 05H handler
                                        ; (ROM BIOS print screen)
                inc     cs:InISR5       ; increment status flag
```

Figure 11-4. Continued. *(more)*

```
                pushf
                cli
                call    cs:PrevISR5     ; chain to previous INT 05H handler

                dec     cs:InISR5       ; decrement status flag
                iret

ISR5            ENDP


ISR8            PROC    far             ; INT 08H handler (timer tick, IRQ0)

                pushf
                cli
                call    cs:PrevISR8     ; chain to previous handler

                cmp     cs:InISR8,0
                jne     L31             ; exit if already in this handler

                inc     cs:InISR8       ; increment status flag

                sti                     ; interrupts are ok
                call    VerifyTSRState
                jc      L30             ; jump if TSR is inactive

                mov     byte ptr cs:ActiveTSR,TRUE
                call    TSRapp
                mov     byte ptr cs:ActiveTSR,FALSE

L30:            dec     cs:InISR8

L31:            iret

ISR8            ENDP


ISR9            PROC    far             ; INT 09H handler
                                        ; (keyboard interrupt IRQ1)
                push    ds              ; preserve these registers
                push    ax
                push    bx

                push    cs
                pop     ds              ; DS -> RESIDENT_GROUP

                in      al,60h          ; AL = current scan code

                pushf                   ; simulate an INT
                cli
                call    ds:PrevISR9     ; let previous handler execute
```

Figure 11-4. Continued. *(more)*

```
                mov     ah,ds:InISR9    ; if already in this handler ..
                or      ah,ds:HotFlag   ; .. or currently processing hot key ..
                jnz     L43             ; .. jump to exit

                inc     ds:InISR9       ; increment status flag
                sti                     ; now interrupts are ok

; Check scan code sequence

                cmp     ds:HotSeqLen,0
                je      L40             ; jump if no hot sequence to match

                mov     bx,ds:HotIndex
                cmp     al,[bx+HotSequence]     ; test scan code sequence
                jne     L41             ; jump if no match

                inc     bx
                cmp     bx,ds:HotSeqLen
                jb      L42             ; jump if not last scan code to match

; Check shift-key state

L40:            push    ds
                mov     ax,40h
                mov     ds,ax           ; DS -> ROM BIOS data area
                mov     al,ds:[KB_FLAG] ; AH = ROM BIOS shift-key flags
                pop     ds

                and     al,ds:HotKBMask ; AL = flags AND "don't care" mask
                cmp     al,ds:HotKBFlag
                jne     L42             ; jump if shift state does not match

; Set flag when hot key is found

                mov     byte ptr ds:HotFlag,TRUE

L41:            xor     bx,bx           ; reinitialize index

L42:            mov     ds:HotIndex,bx  ; update index into sequence
                dec     ds:InISR9       ; decrement status flag

L43:            pop     bx              ; restore registers and exit
                pop     ax
                pop     ds
                iret

ISR9            ENDP
```

Figure 11-4. Continued. *(more)*

```
ISR10           PROC    far             ; INT 10H handler (ROM BIOS video I/O)

                inc     cs:InISR10      ; increment status flag

                pushf
                cli
                call    cs:PrevISR10    ; chain to previous INT 10H handler

                dec     cs:InISR10      ; decrement status flag
                iret

ISR10           ENDP


ISR13           PROC    far             ; INT 13H handler
                                        ; (ROM BIOS fixed disk I/O)
                inc     cs:InISR13      ; increment status flag

                pushf
                cli
                call    cs:PrevISR13    ; chain to previous INT 13H handler

                pushf                   ; preserve returned flags
                dec     cs:InISR13      ; decrement status flag
                popf                    ; restore flags register

                sti                     ; enable interrupts
                ret     2               ; simulate IRET without popping flags

ISR13           ENDP


ISR1B           PROC    far             ; INT 1BH trap (ROM BIOS Ctrl-Break)

                mov     byte ptr cs:Trap1B,TRUE
                iret

ISR1B           ENDP


ISR23           PROC    far             ; INT 23H trap (MS-DOS Ctrl-C)

                mov     byte ptr cs:Trap23,TRUE
                iret

ISR23           ENDP


ISR24           PROC    far             ; INT 24H trap (MS-DOS critical error)

                mov     byte ptr cs:Trap24,TRUE
```

Figure 11-4. Continued.

(more)

```
                xor     al,al           ; AL = 00H (MS-DOS 2.x):
                cmp     cs:MajorVersion,2 ; ignore the error
                je      L50

                mov     al,3            ; AL = 03H (MS-DOS 3.x):
                                        ; fail the MS-DOS call in which
                                        ; the critical error occurred

L50:            iret

ISR24           ENDP


ISR28           PROC    far             ; INT 28H handler
                                        ; (MS-DOS idle interrupt)
                pushf
                cli
                call    cs:PrevISR28    ; chain to previous INT 28H handler

                cmp     cs:InISR28,0
                jne     L61             ; exit if already inside this handler

                inc     cs:InISR28      ; increment status flag

                call    VerifyTSRState
                jc      L60             ; jump if TSR is inactive

                mov     byte ptr cs:ActiveTSR,TRUE
                call    TSRapp
                mov     byte ptr cs:ActiveTSR,FALSE

L60:            dec     cs:InISR28      ; decrement status flag

L61:            iret

ISR28           ENDP


ISR2F           PROC    far             ; INT 2FH handler
                                        ; (MS-DOS multiplex interrupt)
                                        ; Caller:  AH = handler ID
                                        ;          AL = function number
                                        ; Returns for function 0:  AL = 0FFH
                                        ; for all other functions:  nothing

                cmp     ah,MultiplexID
                je      L70             ; jump if this handler is requested

                jmp     cs:PrevISR2F    ; chain to previous INT 2FH handler
```

Figure 11-4. Continued. *(more)*

```
L70:            test    al,al
                jnz     MultiplexIRET   ; jump if reserved or undefined function

; Function 0:  get installed state

                mov     al,0FFh         ; AL = 0FFH (this handler is installed)

MultiplexIRET:  iret                    ; return from interrupt

ISR2F           ENDP

                PAGE
;
;
; AuxInt21--sets ErrorMode while executing INT 21H to force use of the
;       AuxStack instead of the IOStack.
;
;

AuxInt21        PROC    near            ; Caller:     registers for INT 21H
                                        ; Returns:    registers from INT 21H

                push    ds
                push    bx
                lds     bx,ErrorModeAddr
                inc     byte ptr [bx]   ; ErrorMode is now nonzero
                pop     bx
                pop     ds

                int     21h             ; perform MS-DOS function

                push    ds
                push    bx
                lds     bx,ErrorModeAddr
                dec     byte ptr [bx]   ; restore ErrorMode
                pop     bx
                pop     ds
                ret

AuxInt21        ENDP


Int21v          PROC    near            ; perform INT 21H or AuxInt21,
                                        ; depending on MS-DOS version

                cmp     DOSVersion,30Ah
                jb      L80             ; jump if earlier than 3.1

                int     21h             ; versions 3.1 and later
                ret
```

Figure 11-4. Continued.

(more)

```
L80:            call    AuxInt21        ; versions earlier than 3.1
                ret

Int21v          ENDP

                PAGE
;-----------------------------------------------------------------------------
; RAM-resident application
;-----------------------------------------------------------------------------

TSRapp          PROC    near

; Set up a safe stack

                push    ds              ; save previous DS on previous stack

                push    cs
                pop     ds              ; DS -> RESIDENT_GROUP

                mov     PrevSP,sp       ; save previous SS:SP
                mov     PrevSS,ss

                mov     ss,TSRSS        ; SS:SP -> RESIDENT_STACK
                mov     sp,TSRSP

                push    es              ; preserve remaining registers
                push    ax
                push    bx
                push    cx
                push    dx
                push    si
                push    di
                push    bp

                cld                     ; clear direction flag

; Set break and critical error traps

                mov     cx,NTrap
                mov     si,offset RESIDENT_GROUP:StartTrapList

L90:            lodsb                   ; AL = interrupt number
                                        ; DS:SI -> byte past interrupt number

                mov     byte ptr [si],FALSE     ; zero the trap flag

                push    ax              ; preserve AX
                mov     ah,35h          ; INT 21H function 35H
                                        ; (get interrupt vector)
                int     21h             ; ES:BX = previous interrupt vector
                mov     [si+1],bx       ; save offset and segment ..
                mov     [si+3],es       ;  .. of previous handler
```

Figure 11-4. Continued. *(more)*

```
                pop     ax              ; AL = interrupt number
                mov     dx,[si+5]       ; DS:DX -> this TSR's trap
                mov     ah,25h          ; INT 21H function 25H
                int     21h             ; (set interrupt vector)
                add     si,7            ; DS:SI -> next in list

                loop    L90

; Disable MS-DOS break checking during disk I/O

                mov     ax,3300h        ; AH = INT 21H function number
                                        ; AL = 00H (request current break state)
                int     21h             ; DL = current break state
                mov     PrevBreak,dl    ; preserve current state

                xor     dl,dl           ; DL = 00H (disable disk I/O break
                                        ; checking)
                mov     ax,3301h        ; AL = 01H (set break state)
                int     21h

; Preserve previous extended error information

                cmp     DOSVersion,30Ah
                jb      L91             ; jump if MS-DOS version earlier
                                        ; than 3.1
                push    ds              ; preserve DS
                xor     bx,bx           ; BX = 00H (required for function 59H)
                mov     ah,59h          ; INT 21H function 59H
                call    Int21v          ; (get extended error info)

                mov     cs:PrevExtErrDS,ds
                pop     ds
                mov     PrevExtErrAX,ax ; preserve error information
                mov     PrevExtErrBX,bx ; in data structure
                mov     PrevExtErrCX,cx
                mov     PrevExtErrDX,dx
                mov     PrevExtErrSI,si
                mov     PrevExtErrDI,di
                mov     PrevExtErrES,es

; Inform MS-DOS about current PSP

L91:            mov     ah,51h          ; INT 21H function 51H (get PSP address)
                call    Int21v          ; BX = foreground PSP

                mov     PrevPSP,bx      ; preserve previous PSP

                mov     bx,TSRPSP       ; BX = resident PSP
                mov     ah,50h          ; INT 21H function 50H (set PSP address)
                call    Int21v
```

Figure 11-4. Continued. *(more)*

```
; Inform MS-DOS about current DTA (not really necessary in this application
; because DTA is not used)

                mov     ah,2Fh          ; INT 21H function 2FH
                int     21h             ; (get DTA address) into ES:BX
                mov     PrevDTAoffs,bx
                mov     PrevDTAseg,es

                push    ds              ; preserve DS
                mov     ds,TSRPSP
                mov     dx,80h          ; DS:DX -> default DTA at PSP:0080H
                mov     ah,1Ah          ; INT 21H function 1AH
                int     21h             ; (set DTA address)
                pop     ds              ; restore DS

; Open a file, write to it, and close it

                mov     ax,0E07h        ; AH = INT 10H function number
                                        ; (write teletype)
                                        ; AL = 07H (bell character)
                int     10h             ; emit a beep

                mov     dx,offset RESIDENT_GROUP:SnapFile
                mov     ah,3Ch          ; INT 21H function 3CH
                                        ; (create file handle)
                mov     cx,0            ; file attribute
                int     21h
                jc      L94             ; jump if file not opened

                push    ax              ; push file handle
                mov     ah,0Fh          ; INT 10H function 0FH (get video status)
                int     10h             ; AL = video mode number
                                        ; AH = number of character columns
                pop     bx              ; BX = file handle

                cmp     ah,80
                jne     L93             ; jump if not 80-column mode

                mov     dx,0B800h       ; DX = color video buffer segment
                cmp     al,3
                jbe     L92             ; jump if color alphanumeric mode

                cmp     al,7
                jne     L93             ; jump if not monochrome mode

                mov     dx,0B000h       ; DX = monochrome video buffer segment

L92:            push    ds
                mov     ds,dx
                xor     dx,dx           ; DS:DX -> start of video buffer
                mov     cx,80*25*2      ; CX = number of bytes to write
                mov     ah,40h          ; INT 21H function 40H (write file)
```

Figure 11-4. Continued. *(more)*

```
                int     21h
                pop     ds

L93:            mov     ah,3Eh          ; INT 21H function 3EH (close file)
                int     21h

                mov     ax,0E07h        ; emit another beep
                int     10h

; Restore previous DTA

L94:            push    ds              ; preserve DS
                lds     dx,PrevDTA      ; DS:DX -> previous DTA
                mov     ah,1Ah          ; INT 21H function 1AH (set DTA address)
                int     21h
                pop     ds

; Restore previous PSP

                mov     bx,PrevPSP      ; BX = previous PSP
                mov     ah,50h          ; INT 21H function 50H
                call    Int21v          ; (set PSP address)

; Restore previous extended error information

                mov     ax,DOSVersion
                cmp     ax,30Ah
                jb      L95             ; jump if MS-DOS version earlier than 3.1
                cmp     ax,0A00h
                jae     L95             ; jump if MS OS/2-DOS 3.x box

                mov     dx,offset RESIDENT_GROUP:PrevExtErrInfo
                mov     ax,5D0Ah
                int     21h             ; (restore extended error information)

; Restore previous MS-DOS break checking

L95:            mov     dl,PrevBreak    ; DL = previous state
                mov     ax,3301h
                int     21h

; Restore previous break and critical error traps

                mov     cx,NTrap
                mov     si,offset RESIDENT_GROUP:StartTrapList
                push    ds              ; preserve DS

L96:            lods    byte ptr cs:[si] ; AL = interrupt number
                                        ; ES:SI -> byte past interrupt number

                lds     dx,cs:[si+1]    ; DS:DX -> previous handler
                mov     ah,25h          ; INT 21H function 25H
                int     21h             ; (set interrupt vector)
```

Figure 11-4. Continued. *(more)*

```
                add     si,7            ; DS:SI -> next in list
                loop    L96
                pop     ds              ; restore DS

; Restore all registers

                pop     bp
                pop     di
                pop     si
                pop     dx
                pop     cx
                pop     bx
                pop     ax
                pop     es

                mov     ss,PrevSS       ; SS:SP -> previous stack
                mov     sp,PrevSP
                pop     ds              ; restore previous DS

; Finally, reset status flag and return

                mov     byte ptr cs:HotFlag,FALSE
                ret

TSRapp          ENDP

RESIDENT_TEXT   ENDS

RESIDENT_DATA   SEGMENT word public 'DATA'

ErrorModeAddr   DD      ?               ; address of MS-DOS ErrorMode flag
InDOSAddr       DD      ?               ; address of MS-DOS InDOS flag

NISR            DW      (EndISRList-StartISRList)/8 ; number of installed ISRs

StartISRList    DB      05h             ; INT number
InISR5          DB      FALSE           ; flag
PrevISR5        DD      ?               ; address of previous handler
                DW      offset RESIDENT_GROUP:ISR5

                DB      08h
InISR8          DB      FALSE
PrevISR8        DD      ?
                DW      offset RESIDENT_GROUP:ISR8

                DB      09h
InISR9          DB      FALSE
PrevISR9        DD      ?
                DW      offset RESIDENT_GROUP:ISR9

                DB      10h
InISR10         DB      FALSE
```

Figure 11-4. Continued. *(more)*

```
PrevISR10       DD      ?
                DW      offset RESIDENT_GROUP:ISR10

                DB      13h
InISR13         DB      FALSE
PrevISR13       DD      ?
                DW      offset RESIDENT_GROUP:ISR13

                DB      28h
InISR28         DB      FALSE
PrevISR28       DD      ?
                DW      offset RESIDENT_GROUP:ISR28

                DB      2Fh
InISR2F         DB      FALSE
PrevISR2F       DD      ?
                DW      offset RESIDENT_GROUP:ISR2F

EndISRList      LABEL   BYTE

TSRPSP          DW      ?               ; resident PSP
TSRSP           DW      TSRStackSize    ; resident SS:SP
TSRSS           DW      seg RESIDENT_STACK
PrevPSP         DW      ?               ; previous PSP
PrevSP          DW      ?               ; previous SS:SP
PrevSS          DW      ?

HotIndex        DW      0               ; index of next scan code in sequence
HotSeqLen       DW      EndHotSeq-HotSequence   ; length of hot-key sequence

HotSequence     DB      SCEnter         ; hot sequence of scan codes
EndHotSeq       LABEL   BYTE

HotKBFlag       DB      KBAlt           ; hot value of ROM BIOS KB_FLAG
HotKBMask       DB      (KBIns OR KBCaps OR KBNum OR KBScroll) XOR 0FFh
HotFlag         DB      FALSE

ActiveTSR       DB      FALSE

DOSVersion      LABEL   WORD
                DB      ?               ; minor version number
MajorVersion    DB      ?               ; major version number

; The following data is used by the TSR application:

NTrap           DW      (EndTrapList-StartTrapList)/8   ; number of traps

StartTrapList   DB      1Bh
Trap1B          DB      FALSE
PrevISR1B       DD      ?
                DW      offset RESIDENT_GROUP:ISR1B

                DB      23h
```

Figure 11-4. Continued.

(more)

```
Trap23          DB      FALSE
PrevISR23       DD      ?
                DW      offset RESIDENT_GROUP:ISR23

                DB      24h
Trap24          DB      FALSE
PrevISR24       DD      ?
                DW      offset RESIDENT_GROUP:ISR24

EndTrapList     LABEL   BYTE

PrevBreak       DB      ?               ; previous break-checking flag

PrevDTA         LABEL   DWORD           ; previous DTA address
PrevDTAoffs     DW      ?
PrevDTAseg      DW      ?

PrevExtErrInfo  LABEL   BYTE            ; previous extended error information
PrevExtErrAX    DW      ?
PrevExtErrBX    DW      ?
PrevExtErrCX    DW      ?
PrevExtErrDX    DW      ?
PrevExtErrSI    DW      ?
PrevExtErrDI    DW      ?
PrevExtErrDS    DW      ?
PrevExtErrES    DW      ?
                DW      3 dup(0)

SnapFile        DB      '\snap.img'     ; output filename in root directory

RESIDENT_DATA   ENDS

RESIDENT_STACK  SEGMENT word stack 'STACK'

                DB      TSRStackSize dup(?)

RESIDENT_STACK  ENDS

                PAGE
;-----------------------------------------------------------------------------
;
; Transient installation routines
;
;-----------------------------------------------------------------------------

TRANSIENT_TEXT  SEGMENT para public 'TCODE'
                ASSUME  cs:TRANSIENT_TEXT,ds:RESIDENT_DATA,ss:RESIDENT_STACK

InstallSnapTSR  PROC    far             ; At entry:  CS:IP -> InstallSnapTSR
                                        ;            SS:SP -> stack
                                        ;            DS,ES -> PSP
```

Figure 11-4. Continued. *(more)*

```
; Save PSP segment

                mov     ax,seg RESIDENT_DATA
                mov     ds,ax           ; DS  -> RESIDENT_DATA

                mov     TSRPSP,es       ; save PSP segment

; Check the MS-DOS version

                call    GetDOSVersion   ; AH = major version number
                                        ; AL = minor version number

; Verify that this TSR is not already installed
;
;       Before executing INT 2FH in MS-DOS versions 2.x, test whether INT 2FH
;       vector is in use.  If so, abort if PRINT.COM is using it.
;
;       (Thus, in MS-DOS 2.x, if both this program and PRINT.COM are used,
;       this program should be made resident before PRINT.COM.)

                cmp     ah,2
                ja      L101            ; jump if version 3.0 or later

                mov     ax,352Fh        ; AH = INT 21H function number
                                        ; AL = interrupt number
                int     21h             ; ES:BX = INT 2FH vector

                mov     ax,es
                or      ax,bx           ; jump if current INT 2FH vector ..
                jnz     L100            ; .. is nonzero

                push    ds
                mov     ax,252Fh        ; AH = INT 21H function number
                                        ; AL = interrupt number
                mov     dx,seg RESIDENT_GROUP
                mov     ds,dx
                mov     dx,offset RESIDENT_GROUP:MultiplexIRET

                int     21h             ; point INT 2FH vector to IRET
                pop     ds
                jmp     short L103      ; jump to install this TSR

L100:           mov     ax,0FF00h       ; look for PRINT.COM:
                int     2Fh             ; if resident, AH = print queue length;
                                        ; otherwise, AH is unchanged

                cmp     ah,0FFh         ; if PRINT.COM is not resident ..
                je      L101            ; .. use multiplex interrupt

                mov     al,1
                call    FatalError      ; abort if PRINT.COM already installed
```

Figure 11-4. Continued. *(more)*

```
L101:           mov     ah,MultiplexID  ; AH = multiplex interrupt ID value
                xor     al,al           ; AL = 00H
                int     2Fh             ; multiplex interrupt

                test    al,al
                jz      L103            ; jump if ok to install

                cmp     al,0FFh
                jne     L102            ; jump if not already installed

                mov     al,2
                call    FatalError      ; already installed

L102:           mov     al,3
                call    FatalError      ; can't install

; Get addresses of InDOS and ErrorMode flags

L103:           call    GetDOSFlags

; Install this TSR's interrupt handlers

                push    es              ; preserve PSP segment

                mov     cx,NISR
                mov     si,offset StartISRList

L104:           lodsb                   ; AL = interrupt number
                                        ; DS:SI -> byte past interrupt number
                push    ax              ; preserve AX
                mov     ah,35h          ; INT 21H function 35H
                int     21h             ; ES:BX = previous interrupt vector
                mov     [si+1],bx       ; save offset and segment ..
                mov     [si+3],es       ; .. of previous handler

                pop     ax              ; AL = interrupt number
                push    ds              ; preserve DS
                mov     dx,[si+5]
                mov     bx,seg RESIDENT_GROUP
                mov     ds,bx           ; DS:DX -> this TSR's handler
                mov     ah,25h          ; INT 21H function 25H
                int     21h             ; (set interrupt vector)
                pop     ds              ; restore DS
                add     si,7            ; DS:SI -> next in list
                loop    L104

; Free the environment

                pop     es              ; ES = PSP segment
                push    es              ; preserve PSP segment
                mov     es,es:[2Ch]     ; ES = segment of environment
```

Figure 11-4. Continued. *(more)*

```
                mov     ah,49h          ; INT 21H function 49H
                int     21h             ; (free memory block)

; Terminate and stay resident

                pop     ax              ; AX = PSP segment
                mov     dx,cs           ; DX = paragraph address of start of
                                        ; transient portion (end of resident
                                        ; portion)
                sub     dx,ax           ; DX = size of resident portion

                mov     ax,3100h        ; AH = INT 21H function number
                                        ; AL = 00H (return code)
                int     21h

InstallSnapTSR  ENDP

GetDOSVersion   PROC    near            ; Caller:   DS = seg RESIDENT_DATA
                                        ;           ES = PSP
                                        ; Returns:  AH = major version
                                        ;           AL = minor version
                ASSUME  ds:RESIDENT_DATA

                mov     ah,30h          ; INT 21H function 30H:
                                        ; (get MS-DOS version)
                int     21h
                cmp     al,2
                jb      L110            ; jump if versions 1.x

                xchg    ah,al           ; AH = major version
                                        ; AL = minor version
                mov     DOSVersion,ax   ; save with major version in
                                        ; high-order byte
                ret

L110:           mov     al,00h
                call    FatalError      ; abort if versions 1.x

GetDOSVersion   ENDP
GetDOSFlags     PROC    near            ; Caller:   DS = seg RESIDENT_DATA
                                        ; Returns:  InDOSAddr -> InDOS
                                        ;           ErrorModeAddr -> ErrorMode
                                        ; Destroys: AX,BX,CX,DI
                ASSUME  ds:RESIDENT_DATA

; Get InDOS address from MS-DOS

                push    es

                mov     ah,34h          ; INT 21H function number
                int     21h             ; ES:BX -> InDOS
```

Figure 11-4. Continued. *(more)*

```
                mov     word ptr InDOSAddr,bx
                mov     word ptr InDOSAddr+2,es

; Determine ErrorMode address

                mov     word ptr ErrorModeAddr+2,es     ; assume ErrorMode is
                                                        ; in the same segment
                                                        ; as InDOS

                mov     ax,DOSVersion
                cmp     ax,30Ah
                jb      L120            ; jump if MS-DOS version earlier
                                        ; than 3.1 ..
                cmp     ax,0A00h
                jae     L120            ; .. or MS OS/2-DOS 3.x box

                dec     bx              ; in MS-DOS 3.1 and later, ErrorMode
                mov     word ptr ErrorModeAddr,bx       ; is just before InDOS
                jmp     short L125

L120:                                   ; scan MS-DOS segment for ErrorMode

                mov     cx,0FFFFh       ; CX = maximum number of bytes to scan
                xor     di,di           ; ES:DI -> start of MS-DOS segment

L121:           mov     ax,word ptr cs:LF2  ; AX = opcode for INT 28H

L122:           repne   scasb           ; scan for first byte of fragment
                jne     L126            ; jump if not found

                cmp     ah,es:[di]              ; inspect second byte of opcode
                jne     L122                    ; jump if not INT 28H

                mov     ax,word ptr cs:LF1 + 1  ; AX = opcode for CMP
                cmp     ax,es:[di][LF1-LF2]
                jne     L123                    ; jump if opcode not CMP

                mov     ax,es:[di][(LF1-LF2)+2] ; AX = offset of ErrorMode
                jmp     short L124              ; in DOS segment

L123:           mov     ax,word ptr cs:LF3 + 1  ; AX = opcode for TEST
                cmp     ax,es:[di][LF3-LF4]
                jne     L121                    ; jump if opcode not TEST

                mov     ax,es:[di][(LF3-LF4)+2] ; AX = offset of ErrorMode

L124:           mov     word ptr ErrorModeAddr,ax

L125:           pop     es
                ret
```

Figure 11-4. Continued. *(more)*

```
; Come here if address of ErrorMode not found

L126:           mov     al,04h
                call    FatalError

; Code fragments for scanning for ErrorMode flag

LFnear          LABEL   near            ; dummy labels for addressing
LFbyte          LABEL   byte
LFword          LABEL   word
                                        ; MS-DOS versions earlier than 3.1
LF1:            cmp     ss:LFbyte,0     ; CMP ErrorMode,0
                jne     LFnear
LF2:            int     28h
                                        ; MS-DOS versions 3.1 and later
LF3:            test    ss:LFbyte,0FFh  ; TEST ErrorMode,0FFH
                jne     LFnear
                push    ss:LFword
LF4:            int     28h

GetDOSFlags     ENDP

FatalError      PROC    near            ; Caller:   AL = message number
                                        ;           ES = PSP
                ASSUME  ds:TRANSIENT_DATA

                push    ax              ; save message number on stack

                mov     bx,seg TRANSIENT_DATA
                mov     ds,bx

; Display the requested message

                mov     bx,offset MessageTable
                xor     ah,ah           ; AX = message number
                shl     ax,1            ; AX = offset into MessageTable
                add     bx,ax           ; DS:BX -> address of message
                mov     dx,[bx]         ; DS:DX -> message
                mov     ah,09h          ; INT 21H function 09H (display string)
                int     21h             ; display error message

                pop     ax              ; AL = message number
                or      al,al
                jz      L130            ; jump if message number is zero
                                        ; (MS-DOS versions 1.x)

; Terminate (MS-DOS 2.x and later)

                mov     ah,4Ch          ; INT 21H function 4CH
                int     21h             ; (terminate process with return code)
```

Figure 11-4. Continued.

(more)

```
; Terminate (MS-DOS 1.x)

L130            PROC    far

                push    es              ; push PSP:0000H
                xor     ax,ax
                push    ax
                ret                     ; far return (jump to PSP:0000H)

L130            ENDP

FatalError      ENDP


TRANSIENT_TEXT  ENDS

                PAGE
;
;
; Transient data segment
;
;

TRANSIENT_DATA  SEGMENT word public 'DATA'

MessageTable    DW   Message0           ; MS-DOS version error
                DW   Message1           ; PRINT.COM found in MS-DOS 2.x
                DW   Message2           ; already installed
                DW   Message3           ; can't install
                DW   Message4           ; can't find flag

Message0        DB   CR,LF,'TSR requires MS-DOS 2.0 or later version',CR,LF,'$'
Message1        DB   CR,LF,'Can''t install TSR:  PRINT.COM active',CR,LF,'$'
Message2        DB   CR,LF,'This TSR is already installed',CR,LF,'$'
Message3        DB   CR,LF,'Can''t install this TSR',CR,LF,'$'
Message4        DB   CR,LF,'Unable to locate MS-DOS ErrorMode flag',CR,LF,'$'

TRANSIENT_DATA  ENDS

                END     InstallSnapTSR
```

Figure 11-4. Continued.

When installed, the SNAP program monitors keyboard input until the user types the hot-key sequence Alt-Enter. When the hot-key sequence is detected, the monitoring routine waits until the operating environment is stable and then activates the RAM-resident application, which dumps the current contents of the computer's video buffer into the file SNAP.IMG. Figure 11-5 is a block diagram of the RAM-resident and transient components of this TSR.

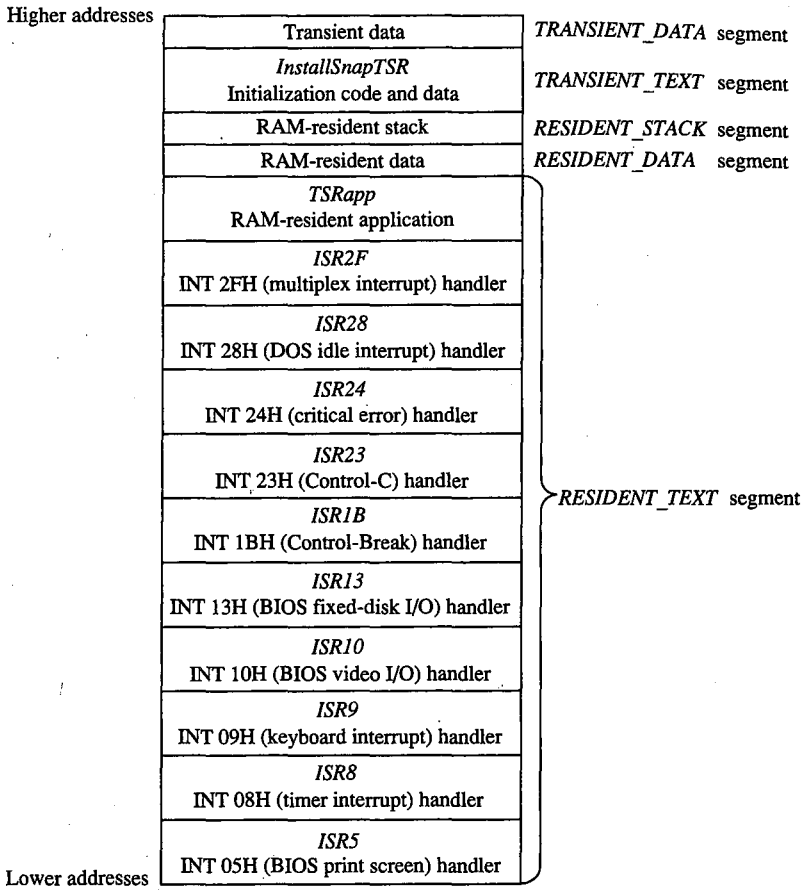


Figure 11-5. Block structure of the TSR program SNAP.EXE when loaded into memory. (Compare with Figure 11-1.)

Installing the program

When SNAP.EXE is run, only the code in the transient portion of the program is executed. The transient code performs several operations before it finally executes Interrupt 21H Function 31H (Terminate and Stay Resident). First it determines which MS-DOS version is in use. Then it executes the multiplex interrupt (Interrupt 2FH) to discover whether the resident portion has already been installed. If an MS-DOS version earlier than 2.0 is in use or if the resident portion has already been installed, the program aborts with an error message.

Otherwise, installation continues. The addresses of the InDOS and critical error flags are saved in the resident data segment. The interrupt service routines in the RAM-resident portion of the program are installed by updating all relevant interrupt vectors. The transient code then frees the RAM occupied by the program's environment, because the resident

portion of this program never uses the information contained there. Finally, the transient portion of the program, which includes the *TRANSIENT_TEXT* and *TRANSIENT_DATA* segments, is discarded and the program is terminated using Interrupt 21H Function 31H.

Detecting a hot key

The SNAP program detects the hot-key sequence (Alt-Enter) by monitoring each keypress. On IBM PCs and PS/2s, each keystroke generates a hardware interrupt on IRQ1 (Interrupt 09H). The TSR's Interrupt 09H handler compares the keyboard scan code corresponding to each keypress with a predefined sequence. The TSR's handler also inspects the shift-key status flags maintained by the ROM BIOS Interrupt 09H handler. When the predetermined sequence of keypresses is detected at the same time as the proper shift keys are pressed, the handler sets a global status flag (*HotFlag*).

Note how the TSR's handler transfers control to the previous Interrupt 09H ISR before it performs its own work. If the TSR's Interrupt 09H handler did not chain to the previous handler(s), essential system processing of keystrokes (particularly in the ROM BIOS Interrupt 09H handler) might not be performed.

Activating the application

The TSR monitors the status of *HotFlag* by regularly testing its value within a timer-tick handler. On IBM PCs and PS/2s, the timer-tick interrupt occurs on IRQ0 (Interrupt 08H) roughly 18.2 times per second. This hardware interrupt occurs regardless of what else the system is doing, so an Interrupt 08H ISR a convenient place to check whether *HotFlag* has been set.

As in the case of the Interrupt 09H handler, the TSR's Interrupt 08H handler passes control to previous Interrupt 08H handlers before it proceeds with its own work. This procedure is particularly important with Interrupt 08H because the ROM BIOS Interrupt 08H handler, which maintains the system's time-of-day clock and resets the system's Intel 8259A Programmable Interrupt Controller, must execute before the next timer tick can occur. The TSR's handler therefore defers its own work until control has returned after previous Interrupt 08H handlers have executed.

The only function of the TSR's Interrupt 08H handler is to attempt to transfer control to the RAM-resident application. The routine *VerifyTSRState* performs this task. It first examines the contents of *HotFlag* to determine whether a hot-key sequence has been detected. If so, it examines the state of the MS-DOS InDOS and critical error flags, the current status of hardware interrupts, and the current status of any non-reentrant ROM BIOS routines that might be executing.

If *HotFlag* is nonzero, the InDOS and critical error flags are both zero, no hardware interrupts are currently being serviced, and no non-reentrant ROM BIOS code has been interrupted, the Interrupt 08H handler activates the RAM-resident utility. Otherwise, nothing happens until the next timer tick, when the handler executes again.

While *HotFlag* is nonzero, the Interrupt 08H handler continues to monitor system status until MS-DOS, the ROM BIOS, and the hardware interrupts are all in a stable state. Often

the system status is stable at the time the hot-key sequence is detected, so the RAM-resident application runs immediately. Sometimes, however, system activities such as prolonged disk reads or writes can preclude the activation of the RAM-resident utility for several seconds after the hot-key sequence has been detected. The handler could be designed to detect this situation (for example, by decrementing *HotFlag* on each timer tick) and return an error status or display a message to the user.

A more serious difficulty arises when the MS-DOS default command processor (COMMAND.COM) is waiting for keyboard input. In this situation, Interrupt 21H Function 01H (Character Input with Echo) is executing, so InDOS is nonzero and the Interrupt 08H handler can never detect a state in which it can activate the RAM-resident utility. This problem is solved by providing a custom handler for Interrupt 28H (the MS-DOS idle interrupt), which is executed by Interrupt 21H Function 01H each time it loops as it waits for a keypress. The only difference between the Interrupt 28H handler and the Interrupt 08H handler is that the Interrupt 28H handler can activate the RAM-resident application when the value of InDOS is 1, which is reasonable because InDOS must have been incremented when Interrupt 21H Function 01H started to execute.

The interrupt service routines for ROM BIOS Interrupts 05H, 10H, and 13H do nothing more than increment and decrement flags that indicate whether these interrupts are being processed by ROM BIOS routines. These flags are inspected by the TSR's Interrupt 08H and 28H handlers.

Executing the RAM-resident application

When the RAM-resident application is first activated, it runs in the context of the program that was interrupted; that is, the contents of the registers, the video display mode, the current PSP, and the current DTA all belong to the interrupted program. The resident application is responsible for preserving the registers and updating MS-DOS with its PSP and DTA values.

The RAM-resident application preserves the previous contents of the CPU registers on its own stack to avoid overflowing the interrupted program's stack. It then installs its own handlers for Control-Break (Interrupt 1BH), Control-C (Interrupt 23H), and critical error (Interrupt 24H). (Otherwise, the interrupted program's handlers would take control if the user pressed Ctrl-Break or Ctrl-C or if an MS-DOS critical error occurred.) These handlers perform no action other than setting flags that can be inspected later by the RAM-resident application, which could then take appropriate action.

The application uses Interrupt 21H Functions 50H and 51H to update MS-DOS with the address of its PSP. If the application is running under MS-DOS versions 2.x, the critical error flag is set before Functions 50H and 51H are executed so that *AuxStack* is used for the call instead of *IOStack*, to avoid corrupting *IOStack* in the event that InDOS is 1.

The application preserves the current extended error information with a call to Interrupt 21H Function 59H. Otherwise, the RAM-resident application might be activated immediately after a critical error occurred in the interrupted program but before the interrupted

program had executed Function 59H and, if a critical error occurred in the TSR application, the interrupted program's extended error information would inadvertently be destroyed.

This example also shows how to update the MS-DOS default DTA using Interrupt 21H Functions 1AH and 2FH, although in this case this step is not necessary because the DTA is never used within the application. In practice, the DTA should be updated only if the RAM-resident application includes calls to Interrupt 21H functions that use a DTA (Functions 11H, 12H, 14H, 15H, 21H, 22H, 27H, 28H, 4EH, and 4FH).

After the resident interrupt handlers are installed and the PSP, DTA, and extended error information have been set up, the RAM-resident application can safely execute any Interrupt 21H function calls except those that use *IOStack* (Functions 01H through 0CH). These functions cannot be used within a RAM-resident application even if the application sets the critical error flag to force the use of the auxiliary stack, because they also use other non-reentrant data structures such as input/output buffers. Thus, a RAM-resident utility must rely either on user-written console input/output functions or, as in the example, on ROM BIOS functions.

The application terminates by returning the interrupted program's extended error information, DTA, and PSP to MS-DOS, restoring the previous Interrupt 1BH, 23H, and 24H handlers, and restoring the previous CPU registers and stack.

Richard Wilton

Article 12
Exception Handlers

Exceptions are system events directly related to the execution of an application program; they ordinarily cause the operating system to abort the program. Exceptions are thus different from errors, which are minor unexpected events (such as failure to find a file on disk) that the program can be expected to handle appropriately. Likewise, they differ from external hardware interrupts, which are triggered by events (such as a character arriving at the serial port) that are not directly related to the program's execution.

The computer hardware assists MS-DOS in the detection of some exceptions, such as an attempt to divide by zero, by generating an internal hardware interrupt. Exceptions related to peripheral devices, such as an attempt to read from a disk drive that is not ready or does not exist, are called *critical* errors. Instead of causing a hardware interrupt, these exceptions are typically reported to the operating system by device drivers. MS-DOS also supports a third type of exception, which is triggered by the entry of a Control-C or Control-Break at the keyboard and allows the user to signal that the current program should be terminated immediately.

MS-DOS contains built-in handlers for each type of exception and so guarantees a minimum level of system stability that requires no effort on the part of the application programmer. For some applications, however, these default handlers are inadequate. For example, if a communications program that controls the serial port directly with custom interrupt handlers is terminated by the operating system without being given a chance to turn off serial-port interrupts, the next character that arrives on the serial line will trigger an interrupt for which a handler is no longer present in memory. The result will be a system crash. Accordingly, MS-DOS allows application programs to install custom exception handlers so that they can shut down operations in an orderly way when an exception occurs.

This article examines the default exception handlers provided by MS-DOS and discusses methods programmers can use to replace those routines with handlers that are more closely matched to specific application requirements.

Overview

Two major exception handlers of importance to application programmers are supported under all versions of MS-DOS. The first, the Control-C exception handler, terminates the program and is invoked when the user enters a Ctrl-C or Ctrl-Break keystroke; the address

of this handler is found in the vector for Interrupt 23H. The second, the critical error exception handler, is invoked if MS-DOS detects a critical error while servicing an I/O request. (A critical error is a hardware error that makes normal completion of the request impossible.) This exception handler displays the familiar *Abort, Retry, Ignore* prompt; its address is saved in the vector for Interrupt 24H.

When a program begins executing, the addresses in the Interrupt 23H and 24H vectors usually point to the system's default Control-C and critical error handlers. If the program is a child process, however, the vectors might point to exception handlers that belong to the parent process, if the immediate parent is not COMMAND.COM. In any case, the application program can install its own custom handler for Control-C or critical error exceptions simply by changing the address in the vector for Interrupt 23H or Interrupt 24H so that the vector points to the application's own routine. When the program performs a final exit by means of Interrupt 21H Function 00H (Terminate Process), Function 31H (Terminate and Stay Resident), Function 4CH (Terminate Process with Return Code), Interrupt 20H (Terminate Process), or Interrupt 27H (Terminate and Stay Resident), MS-DOS restores the previous contents of the Interrupt 23H and 24H vectors.

Note that Interrupts 23H and 24H *never* occur as externally generated hardware interrupts in an MS-DOS system. The vectors for these interrupts are used simply as storage areas for the addresses of the exception handlers.

MS-DOS also contains default handlers for the Control-Break event detected by the ROM BIOS in IBM PCs and compatible computers and for some of the Intel microprocessor exceptions that generate actual hardware interrupts. These exception handlers are not replaced by application programs as often as the Control-C and critical error handlers. The interrupt vectors that contain the addresses of these handlers are *not* restored by MS-DOS when a program exits.

The address of the Control-Break handler is saved in the vector for Interrupt 1BH and is invoked by the ROM BIOS whenever the Ctrl-Break key combination is detected. The default MS-DOS handler normally flushes the keyboard input buffer and substitutes Control-C for Control-Break, and the Control-C is later handled by the Control-C exception handler. The default handlers for exceptions that generate hardware interrupts either abort the current program (as happens with Divide by Zero) or bring the entire system to a halt (as for a memory parity error).

The Control-C Handler

The vector for Interrupt 23H points to code that is executed whenever MS-DOS detects a Control-C character in the keyboard input buffer. When the character is detected, MS-DOS executes a software Interrupt 23H.

In response to Interrupt 23H, the default Control-C exception handler aborts the current process. Files that were opened with handles are closed (FCB-based files are not), but no

other cleanup is performed. Thus, unsaved data can be left in buffers, some files might not be processed, and critical addresses, such as the vectors for custom interrupt handlers, might be left pointing into free RAM. If more complete control over process termination is wanted, the application should replace the default Control-C handler with custom code. *See* Customizing Control-C Handling below.

The Control-Break exception handler, pointed to by the vector for Interrupt 1BH, is closely related to the Control-C exception handler in MS-DOS systems on the IBM PC and close compatibles but is called by the ROM BIOS keyboard driver on detection of the Ctrl-Break keystroke combination. Because the Control-Break exception is generated by the ROM BIOS, it is present only on IBM PC-compatible machines and is not a standard feature of MS-DOS. The default ROM BIOS handler for Control-Break is a simple interrupt return — in other words, no action is taken to handle the keystroke itself, other than converting the Ctrl-Break scan code to an extended character and passing it through to MS-DOS as normal keyboard input.

To account for as many hardware configurations as possible, MS-DOS redirects the ROM BIOS Control-Break interrupt vector to its own Control-Break handler during system initialization. The MS-DOS Control-Break handler sets an internal flag that causes the Ctrl-Break keystroke to be interpreted as a Ctrl-C keystroke and thus causes Interrupt 23H to occur.

Customizing Control-C handling

The exception handlers most often neglected by application programmers — and most often responsible for major program failures — are the default exception handlers invoked by the Ctrl-C and Ctrl-Break keystrokes. Although the user must be able to recover from a runaway condition (the reason for Ctrl-C capability in the first place), any exit from a complex program must also be orderly, with file buffers flushed to disk, directories and indexes updated, and so on. The default Control-C and Control-Break handlers do not provide for such an orderly exit.

The simplest and most direct way to deal with Ctrl-C and Ctrl-Break keystrokes is to install new exception handlers that do nothing more than an IRET and thus take MS-DOS out of the processing loop entirely. This move is not as drastic as it sounds: It allows an application to check for and handle the Ctrl-C and Ctrl-Break keystrokes at its convenience when they arrive through the normal keyboard input functions and prevents MS-DOS from terminating the program unexpectedly.

The following example sets the Interrupt 23H and Interrupt 1BH vectors to point to an IRET instruction. When the user presses Ctrl-C or Ctrl-Break, the keystroke combination is placed into the keyboard buffer like any other keystroke. When it detects the Ctrl-C or Ctrl-Break keystroke, the executing program should exit properly (if that is the desired action) after an appropriate shutdown procedure.

To install the new exception handlers, the following procedure (*set_int*) should be called while the main program is initializing:

```
_DATA   segment para public 'DATA'
oldint1b dd     0               ; original INT 1BH vector
oldint23 dd     0               ; original INT 23H vector
_DATA   ends
_TEXT   segment byte public 'CODE'
        assume cs:_TEXT,ds:_DATA,es:NOTHING
myint1b:                        ; handler for Ctrl-Break
myint23:                        ; handler for Ctrl-C
        iret

set_int proc    near
        mov     ax,351bh        ; get current contents of
        int     21h             ; Int 1BH vector and save it
        mov     word ptr oldint1b,bx
        mov     word ptr oldint1b+2,es
        mov     ax,3523h        ; get current contents of
        int     21h             ; Int 23H vector and save it
        mov     word ptr oldint23,bx
        mov     word ptr oldint23+2,es
        push    ds              ; save our data segment
        push    cs              ; let DS point to our
        pop     ds              ; code segment
        mov     dx,offset myint1b
        mov     ax,251bh        ; set interrupt vector 1BH
        int     21h             ; to point to new handler
        mov     dx,offset myint23
        mov     ax,2523h        ; set interrupt vector 23H
        int     21h             ; to point to new handler
        pop     ds              ; restore our data segment
        ret                     ; back to caller
set_int endp
_TEXT   ends
```

The application can use the following routine to restore the original contents of the vectors pointing to the Control-C and Control-Break exception handlers before making a final exit back to MS-DOS. Note that, although MS-DOS restores the Interrupt 23H vector to its previous contents, the application *must* restore the Interrupt 1BH vector itself.

```
rest_int proc   near
        push    ds              ; save our data segment
        mov     dx,word ptr oldint23
        mov     ds,word ptr oldint23+2
        mov     ax,2523h        ; restore original contents
        int     21h             ; of Int 23H vector
        pop     ds              ; restore our data segment
        push    ds              ; then save it again
        mov     dx,word ptr oldint1B
        mov     ds,word ptr oldint1B+2
        mov     ax,251Bh        ; restore original contents
        int     21h             ; of Int 1BH vector
        pop     ds              ; get back our data segment
        ret                     ; return to caller
rest_int endp
```

The preceding example simply prevents MS-DOS from terminating an application when a Ctrl-C or Ctrl-Break keystroke is detected. Program termination is still often the ultimate goal, but after a more orderly shutdown than is provided by the MS-DOS default Control-C handler. The following exception handler allows the program to exit more gracefully:

```
myint1b:                        ; Control-Break exception handler
        iret                    ; do nothing
myint23:                        ; Control-C exception handler
        call    safe_shut_down  ; release interrupt vectors,
                                ; close files, etc.
        jmp     program_exit_point
```

Note that because the Control-Break handler is invoked by the ROM BIOS keyboard driver and MS-DOS is not reentrant, MS-DOS services (such as closing files and terminating with return code) cannot be invoked during processing of a Control-Break exception. In contrast, any MS-DOS Interrupt 21H function call can be used during the processing of a Control-C exception. Thus, the Control-Break handler in the preceding example does nothing, whereas the Control-C handler performs orderly shutdown of the application.

Most often, however, neither a handler that does nothing nor a handler that shuts down and terminates is sufficient for processing a Ctrl-C (or Ctrl-Break) keystroke. Rather than simply prevent Control-C processing, software developers usually prefer to have a Ctrl-C keystroke signal some important action without terminating the program. Using methods similar to those above, the programmer can replace Interrupts 1BH and 23H with a routine like the following:

```
myint1b:                        ; Control-Break exception handler
myint23:                        ; Control-C exception handler
        call    control_c_happened
        iret
```

Notes on processing Control-C

The preceding examples assume the programmer wants to treat Control-C and Control-Break the same way, but this is not always desirable. Control-C and Control-Break are not the same, and the difference between the two should be kept in mind: The Control-Break handler is invoked by a keyboard-input interrupt and can be called at any time; the Control-C handler is called only at "safe" points during the processing of MS-DOS Interrupt 21H functions. Also, even though MS-DOS restores the Interrupt 23H vector on exit from a program, the *application* must restore the previous contents of the Interrupt 1BH vector before exiting. If this interrupt vector is not restored, the next Ctrl-Break keystroke will cause the machine to attempt to execute an undetermined piece of code or data and will probably crash the system.

Although it is generally desirable to take control of the Control-C and Control-Break interrupts, control should be retained only as long as necessary. For example, a RAM-resident pop-up application should take over Control-C and Control-Break handling only when it is activated, and it should restore the previous contents of the Interrupt 1BH and Interrupt 23H vectors before it returns control to the foreground process.

The Critical Error Handler

When MS-DOS detects a critical error—an error that prevents successful completion of an I/O operation—it calls the exception handler whose address is stored in the vector for Interrupt 24H. Information about the operation in progress and the nature of the error is passed to the exception handler in the CPU registers. In addition, the contents of all the registers at the point of the original MS-DOS call are pushed onto the stack for inspection by the exception handler.

The action of MS-DOS's default critical error handler is to present a message such as

```
Error type error action device
Abort, Retry, Ignore?
```

This message signals a hardware error from which MS-DOS cannot recover without user intervention. For example, if the user enters the command

```
C>DIR A:  <Enter>
```

but drive A either does not contain a disk or the disk drive door is open, the MS-DOS critical error handler displays the message

```
Not ready error reading drive A
Abort, Retry, Ignore?
```

I (*Ignore*) simply tells MS-DOS to forget that an error occurred and continue on its way. (Of course, if the error occurred during the writing of a file to disk, the file is generally corrupted; if the error occurred during reading, the data might be incorrect.)

R (*Retry*) gives the application a second chance to access the device. The critical error handler returns information to MS-DOS that says, in effect, "Try again; maybe it will work this time." Sometimes, the attempt succeeds (as when the user closes an open drive door), but more often the same or another critical error occurs.

A (*Abort*) is the problem child of Interrupt 24H. If the user responds with *A*, the application is terminated immediately. The directory structure is not updated for open files, interrupt vectors are left pointing to inappropriate locations, and so on. In many cases, restarting the system is the only safe thing to do at this point.

Note: Beginning with version 3.3, an *F* (*Fail*) option appears in the message displayed by MS-DOS's default critical error handler. When *Fail* is selected, the current MS-DOS function is terminated and an error condition is returned to the calling program. For example, if a program calls Interrupt 21H Function 3DH to open a file on drive A but the drive door is open, choosing *F* in response to the error message causes the function call to return with the carry flag set, indicating that an error occurred but processing continues.

Like the Control-C exception handler, the default critical error exception handler can and should be replaced by an application program when complete control of the system is desired. The program installs its own handler simply by placing the address of the new handler in the vector for Interrupt 24H; MS-DOS restores the previous contents of the Interrupt 24H vector when the program terminates.

Unlike the Control-C handler, however, the critical error handler must be kept within carefully defined limits to preserve the stability of the operating system. Programmers must rigidly adhere to the structure described in the following pages for passing information to and from an Interrupt 24H handler.

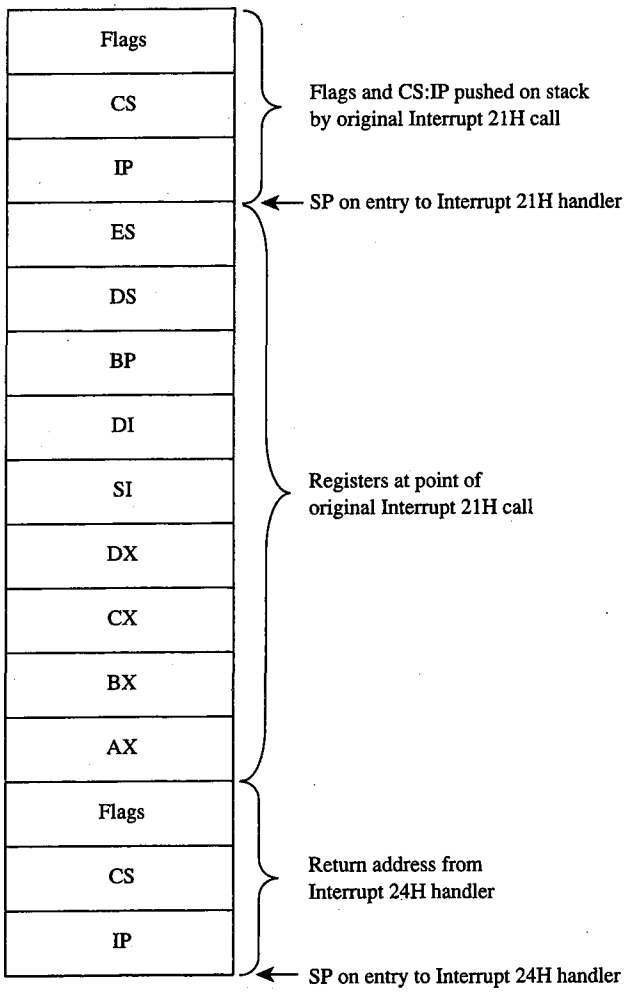


Figure 12-1. The stack contents at entry to a critical error exception handler.

Mechanics of critical error handling

MS-DOS critical error handling has two components: the exception handler, whose address is saved in the Interrupt 24H vector and which can be replaced by an application program; and an internal routine inside MS-DOS. The internal routine sets up the information to be passed to the exception handler on the stack and in registers and, in turn, calls the exception handler itself. The internal routine also responds to the values returned by the critical error handler when that handler executes an IRET to return to the MS-DOS kernel.

Before calling the exception handler, MS-DOS arranges the stack (Figure 12-1 on the preceding page) so the handler can inspect the location of the error and register contents at the point in the original MS-DOS function call that led to the critical error.

When the critical error handler is called by the internal routine, four registers may contain important information: AX, DI, BP, and SI. (With MS-DOS versions 1.x, only the AX and DI registers contain significant information.) The information passed to the handler in the registers differs somewhat, depending on whether a character device or a block device is causing the error.

Block-device (disk-based) errors

If the critical error handler is entered in response to a block-device (disk-based) error, registers BP:SI contain the segment:offset of the device driver header for the device causing the error and bit 7 (the high-order bit) of the AH register is zero. The remaining bits of the AH register contain the following information (bits 3 through 5 apply only to MS-DOS versions 3.1 and later):

Bit	Value	Meaning
0	0	Read operation
	1	Write operation
1-2		Indicate the affected disk area:
	00	MS-DOS
	01	File allocation table
	10	Root directory
	11	Files area
3	0	Fail response not allowed
	1	Fail response allowed
4	0	Retry response not allowed
	1	Retry response allowed
5	0	Ignore response not allowed
	1	Ignore response allowed
6	0	Undefined

The AL register contains the designation of the drive where the error occurred; for example, AL = 00H (drive A), AL = 01H (drive B), and so on.

The lower half of the DI register contains the following error codes (the upper half of this register is undefined):

Error Code	Meaning
00H	Write-protected disk
01H	Unknown unit
02H	Drive not ready
03H	Invalid command
04H	Data error (CRC)
05H	Length of request structure invalid
06H	Seek error
07H	Non-MS-DOS disk
08H	Sector not found
09H	Printer out of paper
0AH	Write fault
0BH	Read fault
0CH	General failure
0FH	Invalid disk change (version 3.0 or later)

Note: With versions 1.x, the only valid error codes are 00H, 02H, 04H, 06H, 08H, 0AH, and 0CH.

Before calling the critical error handler for a disk-based error, MS-DOS tries from one to five times to perform the requested read or write operation, depending on the type of operation. Critical disk errors result only from Interrupt 21H operations, not from failed sector-read and sector-write operations attempted with Interrupts 25H and 26H.

Character-device errors

If the critical error handler is called from the MS-DOS kernel with bit 7 of the AH register set to 1, either an error occurred on a character device or the memory image of the file allocation table is bad (a rare occurrence). Again, registers BP:SI contain the segment and offset of the device driver header for the device causing the critical error. The exception handler can inspect bit 15 of the device attribute word at offset 04H in the device header to confirm that the error was caused by a character device—this bit is 0 for block devices and 1 for character devices. *See also* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Installable Device Drivers.

If the error was caused by a character device, the lower half of the DI register contains error codes as described above and the contents of the AL register are undefined. The exception handler can inspect the other fields of the device header to obtain the logical name of the character device; to determine whether that device is the standard input, standard output, or both; and so on.

Critical error processing

The critical error exception handler is entered from MS-DOS with interrupts disabled. Because an MS-DOS system call is already in progress and MS-DOS is not reentrant, the

handler cannot request any MS-DOS system services other than Interrupt 21H Functions 01 through 0CH (character I/O functions), Interrupt 21H Function 30H (Get MS-DOS Version Number), and Interrupt 21H Function 59H (Get Extended Error Information). These functions use a special stack so that they can be called during error processing.

In general, the critical error handler must preserve all but the AL register. It must not change the contents of the device header pointed to by BP:SI. The handler must return to the MS-DOS kernel with an IRET, passing an action code in register AL as follows:

Value in AL	Meaning
00H	Ignore
01H	Retry
02H	Terminate process
03H	Fail current system call

These values correspond to the options presented by the MS-DOS default critical error handler. The default handler prompts the user for input, places the appropriate return information in the AL register, and immediately issues an IRET instruction.

Note: Although the *Fail* option is displayed by the MS-DOS default critical error handler in versions 3.3 and later, the *Fail* option inside the handler was added in version 3.1.

With MS-DOS versions 3.1 and later, if the handler returns an action code in AL that is not allowed for the error in question (bits 3 through 5 of the AH register at the point of call), MS-DOS reacts according to the following rules:

If *Ignore* is specified by AL = 00H but is not allowed because bit 5 of AH = 0, the response defaults to *Fail* (AL = 03H).

If *Retry* is specified by AL = 01H but is not allowed because bit 4 of AH = 0, the response defaults to *Fail* (AL = 03H).

If *Fail* is specified by AL = 03H but is not allowed because bit 3 of AH = 0, the response defaults to *Abort*.

Custom critical error handlers

Each time it receives control, COMMAND.COM restores the Interrupt 24H vector to point to the system's default critical error handler and displays a prompt to the user. Consequently, a single custom handler cannot terminate and stay resident to provide critical error handling services for subsequent application programs. Each program that needs better critical error handling than MS-DOS provides must contain its own critical error handler.

Figure 12-2 contains a simple critical error handler, INT24.ASM, written in assembly language. In the form shown, INT24.ASM is no more than a functional replacement for the MS-DOS default critical error handler, but it can be used as the basis for more sophisticated handlers that can be incorporated into application programs.

INT24.ASM contains three routines:

Routine	Action
get24	Saves the previous contents of the Interrupt 24H critical error handler vector and stores the address of the new critical error handler into the vector.
res24	Restores the address of the previous critical error handler, which was saved by a call to *get24*, into the Interrupt 24 vector.
int24	Replaces the MS-DOS critical error handler.

A program wishing to substitute the new critical error handler for the system's default handler should call the *get24* routine during its initialization sequence. If the program wishes to revert to the system's default handler during execution, it can accomplish this with a call to the *res24* routine. Otherwise, a call to *res24* (and the presence of the routine itself in the program) is not necessary, because MS-DOS automatically restores the Interrupt 24H vector to its previous value when the program exits, from information stored in the program segment prefix (PSP).

The replacement critical error handler, *int24*, is simple. First it saves all registers; then it displays a message that a critical error has occurred and prompts the user to enter a key selecting one of the four possible options: *Abort*, *Retry*, *Ignore*, or *Fail*. If an illegal key is entered, the prompt is displayed again; otherwise, the action code corresponding to the key is extracted from a table and placed in the AL register, the other registers are restored, and control is returned to the MS-DOS kernel with an IRET instruction.

Note that the handle read and write functions (Interrupt 21H Functions 3FH and 40H), which would normally be preferred for interaction with the display and keyboard, cannot be used in a critical error handler.

```
        name    int24
        title   INT24 Critical Error Handler

;
; INT24.ASM - Replacement critical error handler
; by Ray Duncan, September 1987
;

cr      equ     0dh             ; ASCII carriage return
lf      equ     0ah             ; ASCII linefeed

DGROUP  group   _DATA

_DATA   segment word public 'DATA'

save24  dd      0               ; previous contents of Int 24H
                                ; critical error handler vector
```

Figure 12-2. INT24.ASM, a replacement Interrupt 24H handler. *(more)*

```
                                ; prompt message used by
                                ; critical error handler
prompt  db      cr,lf,'Critical Error Occurred: '
        db      'Abort, Retry, Ignore, Fail? $'

keys    db      'aArRiIfF'      ; possible user response keys
keys_len equ    $-keys          ; (both cases of each allowed)

codes   db      2,2,1,1,0,0,3,3 ; codes returned to MS-DOS kernel
                                ; for corresponding response keys

_DATA   ends


_TEXT   segment word public 'CODE'

        assume  cs:_TEXT,ds:DGROUP

        public  get24
get24   proc    near            ; set Int 24H vector to point
                                ; to new critical error handler

        push    ds              ; save segment registers
        push    es

        mov     ax,3524h        ; get address of previous
        int     21h             ; INT 24H handler and save it

        mov     word ptr save24,bx
        mov     word ptr save24+2,es

        push    cs              ; set DS:DX to point to
        pop     ds              ; new INT 24H handler
        mov     dx,offset _TEXT:int24
        mov     ax,2524h        ; then call MS-DOS to
        int     21h             ; set the INT 24H vector

        pop     es              ; restore segment registers
        pop     ds
        ret                     ; and return to caller

get24   endp


        public  res24
res24   proc    near            ; restore original contents
                                ; of Int 24H vector

        push    ds              ; save our data segment
```

Figure 12-2. Continued. *(more)*

```
        lds     dx,save24       ; put address of old handler
        mov     ax,2524h        ; back into INT 24H vector
        int     21h

        pop     ds              ; restore data segment
        ret                     ; and return to caller

res24   endp

;
; This is the replacement critical error handler.  It
; prompts the user for Abort, Retry, Ignore, or Fail and
; returns the appropriate code to the MS-DOS kernel.
;
int24   proc    far             ; entered from MS-DOS kernel

        push    bx              ; save registers
        push    cx
        push    dx
        push    si
        push    di
        push    bp
        push    ds
        push    es

int24a: mov     ax,DGROUP       ; display prompt for user
        mov     ds,ax           ; using function 09H (print string
        mov     es,ax           ; terminated by $ character)
        mov     dx,offset prompt
        mov     ah,09h
        int     21h

        mov     ah,01h          ; get user's response
        int     21h             ; function 01H = read one character

        mov     di,offset keys  ; look up code for response key
        mov     cx,keys_len
        cld
        repne scasb
        jnz     int24a          ; prompt again if bad response

                                ; set AL = action code for MS-DOS
                                ; according to key that was entered:
                                ; 0 = ignore, 1 = retry, 2 = abort, 3 = fail
        mov     al,[di+keys_len-1]

        pop     es              ; restore registers
        pop     ds
        pop     bp
        pop     di
        pop     si
```

Figure 12-2. Continued.

(more)

```
        pop     dx
        pop     cx
        pop     bx
        iret                    ; exit critical error handler

int24   endp

_TEXT   ends

        end
```

Figure 12-2. Continued.

Hardware-generated Exception Interrupts

Intel reserved the vectors for Interrupts 00H through 1FH (Table 12-1) for exceptions generated by the execution of various machine instructions. Handling of these chip-dependent internal interrupts can vary from one make of MS-DOS machine to another; some such differences are mentioned in the discussion.

Table 12-1. Intel Reserved Exception Interrupts.

Interrupt Number	Definition
00H	Divide by Zero
01H	Single-Step
02H	Nonmaskable Interrupt (NMI)
03H	Breakpoint Trap
04H	Overflow Trap
05H	BOUND Range Exceeded*
06H	Invalid Opcode*
07H	Coprocessor not Available†
08H	Double-Fault Exception†
09H	Coprocessor Segment Overrun†
0AH	Invalid Task State Segment (TSS)†
0BH	Segment not Present†
0CH	Stack Exception†
0DH	General Protection Exception†
0EH	Page Fault‡
0FH	(Reserved)
10H	Coprocessor Error†
11–1FH	(Reserved)

*The 80186, 80286, and 80386 microprocessors only.
†The 80286 and 80386 microprocessors only.
‡The 80386 microprocessor only.

Note: A number of these reserved exception interrupts generally do not occur in MS-DOS because they are generated only when the 80286 or 80386 microprocessor is operating in protected mode. The following discussions do not cover these interrupts.

Divide by Zero (Interrupt 00H)

An Interrupt 00H occurs whenever a DIV or IDIV operation fails to terminate within a reasonable period of time. The interrupt is triggered by a mathematical anomaly: Division by zero is inherently undefined. To handle such situations, Intel built special processing into the DIV and IDIV instructions to ensure that the condition does not cause the processor to lock up. Although the assumption underlying Interrupt 00H is an attempt to divide by zero (a condition that will never terminate), the interrupt can also be triggered by other error conditions, such as a quotient that is too large to fit in the designated register (AX or AL).

The ROM BIOS handler for Interrupt 00H in the IBM PC and close compatibles is a simple IRET instruction. During the MS-DOS startup process, however, MS-DOS modifies the interrupt vector to point to its own handler — a routine that issues the warning message *Divide by Zero* and aborts the current application. This abort procedure can leave the computer and operating system in an extremely unstable state. If the default handler is used, the system should be restarted immediately and an attempt should be made to find and eliminate the cause of the error. A better approach, however, is to provide a replacement handler that treats Interrupt 00H much as MS-DOS treats Interrupt 24H.

Single-Step (Interrupt 01H)

If the trap flag (bit 8 of the microprocessor's 16-bit flags register) is set, Interrupt 01H occurs at the end of every instruction executed by the processor. By default, Interrupt 01H points to a simple IRET instruction, so the net effect is as if nothing happened. However, debugging programs, which are the only applications that use this interrupt, modify the interrupt vector to point to their own handlers. The interrupt can then be used to allow a debugger to single-step through the machine instructions of the program being debugged, as DEBUG does with its T (Trace) command.

Nonmaskable Interrupt, or NMI (Interrupt 02H)

In the hardware architecture of IBM PCs and close compatibles, Interrupt 02H is invoked whenever a memory parity error is detected. MS-DOS provides no handler, because this error, as a hardware-related problem, is in the domain of the ROM BIOS.

In response to the Interrupt 02H, the default ROM BIOS handler displays a message and locks the machine, on the assumption that bad memory prevents reliable system operation. Many programmers, however, prefer to include code that permits orderly shutdown of the system. Replacing the ROM BIOS parity trap routine can be dangerous, though, because a parity error detected in memory means the contents of RAM are no longer reliable — even the memory locations containing the NMI handler itself might be defective.

Breakpoint Trap (Interrupt 03H)

Interrupt 03H, which is used in conjunction with Interrupt 01H for debugging, is invoked by a special 1-byte opcode (0CCH). During a debugging session, a debugger modifies the vector for Interrupt 03H to point to its own handler and then replaces 1 byte of program opcode with the 0CCH opcode at any location where a breakpoint is needed.

When a breakpoint is reached, the 0CCH opcode triggers Interrupt 03H and the debugger regains control. The debugger then restores the original opcode in the program being debugged and issues a prompt so that the user can display or alter the contents of memory or registers. The use of Interrupt 03H is illustrated by DEBUG and SYMDEB's breakpoint capabilities.

Overflow Trap (Interrupt 04H)

If the overflow bit (bit 11) in the microprocessor's flags register is set, Interrupt 04H occurs when the INTO (Interrupt on Overflow) instruction is executed. The overflow bit can be set during prior execution of any arithmetic instruction (such as MUL or IMUL) that can produce an overflow error.

The ROM BIOS of the IBM PC and close compatibles initializes the Interrupt 04H vector to point to an IRET, so this interrupt becomes invisible to the user if it is executed. MS-DOS does not have its own handler for Interrupt 04H. However, because the Intel microprocessors also include JO (Jump if Overflow) and JNO (Jump if No Overflow) instructions, applications rarely need the INTO instruction and, hence, seldom have to provide their own Interrupt 04H handlers.

BOUND Range Exceeded (Interrupt 05H)

Interrupt 05H is generated on 80186, 80286, and 80386 microprocessors if a BOUND instruction is executed to test the value of an array index and the index falls outside the limits specified by the instruction's operand. The exception handler is expected to alter the index so that it is correct—when the handler performs an interrupt return (IRET), the CPU reexecutes the BOUND instruction that caused the interrupt.

On IBM PC/AT-compatible machines, the ROM BIOS assignment of the PrtSc (print screen) routine to Interrupt 05H is in conflict with the CPU's use of Interrupt 05H for BOUND exceptions.

Invalid opcode (Interrupt 06H)

Interrupt 06H is generated by the 80186, 80286, and 80386 microprocessors if the current instruction is not a valid opcode—for example, if the machine tries to execute a data statement.

On IBM PC/ATs, Interrupt 06H simply points to an IRET instruction. The ROM BIOS routines of some IBM PC/AT-compatibles, however, provide an interrupt handler that reports an unexpected software Interrupt 06H and asks if the user wants to continue. A *Y* response causes the interrupt handler to skip over the invalid opcode. Unfortunately, because the succeeding opcode is often invalid as well, the user may have the feeling of being trapped in a loop.

Extended Error Information

Under MS-DOS versions 1.x, the operating system provided limited information about errors that occurred during calls to the Interrupt 21H system functions. For example, if a program called Function 0FH to open a file, there were only two possible results: On return, the AL register either contained 00H for a successful open or 0FFH for failure. No further detail was available from the operating system. Although some of these early system calls (such as the read and write functions) returned somewhat more information, the 1.x versions of MS-DOS were essentially limited to success/failure return codes.

Beginning with version 2.0 and the introduction of the handle concept, additional error information became available. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management. For example, if a program attempts to open a file with Interrupt 21H Function 3DH (Open File with Handle), it can check the status of the carry flag on return to detect whether an error occurred. If the carry flag is not set, the call was successful and the AX register contains the file handle. If the carry flag is set, the AX register contains one of the following possible error codes:

Error Code	Meaning
01H	Invalid function code
02H	File not found
03H	Path not found
04H	Too many open files (no more handles available)
05H	Access denied
0CH	Invalid access code

In some circumstances, however, even these error codes do not provide enough information. Therefore, beginning with version 3.0, MS-DOS made extended error information available through Interrupt 21H Function 59H (Get Extended Error Information). This function can be called after any other Interrupt 21H function fails, or it can be called from a critical error handler. The extended error codes, briefly described below, maintain compatibility with the MS-DOS versions 2.x error returns and are grouped as follows:

Error Code	Error Group
00H	No error encountered.
01–12H	MS-DOS versions 2.x and 3.x Interrupt 21H errors. These error codes are identical to those returned in the AX register by Functions 38H through 57H if the carry flag is set on return from the function call.
13–1FH	MS-DOS versions 2.x and 3.x Interrupt 24H errors. These error codes are 13H (19) greater than the codes passed to a critical error handler in the lower half of the DI register; that is, if the critical error handler receives error code 04H (CRC error), Interrupt 21H Function 59H returns 17H.
20–58H	Extended error codes, many related to networking and file sharing, for MS-DOS versions 3.0 and later.

Note: The contents of the CPU registers (except CS:IP and SS:SP) are destroyed by a call to Function 59H. Also, as mentioned earlier, this function is available only with MS-DOS versions 3.x, even though it maintains compatibility with error returns in versions 2.x.

On return, Function 59H provides the extended error code in the AX register, the error class (type) in the BH register, a code for the suggested corrective action in the BL register, and the locus of the error in the CH register. These values are defined in the following paragraphs. With MS-DOS or PC-DOS versions 3.x, if an error 22H (invalid disk change) occurs and if the capability is supported by the system's block-device drivers, ES:DI points to an ASCIIZ volume label that designates the disk to be inserted in the drive before the operation is retried.

Error Code (AX register). This value is defined as follows:

Value in AX	Meaning
Interrupt 21H errors (MS-DOS versions 2.0 and later):	
01H	Invalid function number
02H	File not found
03H	Path not found
04H	Too many open files (no handles available)
05H	Access denied
06H	Invalid handle
07H	Memory control blocks destroyed
08H	Insufficient memory
09H	Invalid memory-block address
0AH	Invalid environment
0BH	Invalid format
0CH	Invalid access code
0DH	Invalid data
0EH	Reserved
0FH	Invalid disk drive specified
10H	Attempt to remove the current directory
11H	Not same device
12H	No more files
Interrupt 24H errors (MS-DOS versions 2.0 and later):	
13H	Attempt to write on write-protected disk
14H	Unknown unit
15H	Drive not ready
16H	Invalid command
17H	Data error based on cyclic redundancy check (CRC)
18H	Length of request structure invalid
19H	Seek error

(more)

Value in AX	Meaning
Interrupt 24H errors *(continued)*	
1AH	Unknown media type (non-MS-DOS disk)
1BH	Sector not found
1CH	Printer out of paper
10H	Write fault
1EH	Read fault
1FH	General failure
MS-DOS versions 3.x extended errors:	
20H	Sharing violation
21H	Lock violation
22H	Invalid disk change
23H	FCB unavailable
24H	Sharing buffer exceeded
25H–31H	Reserved
32H	Network request not supported
33H	Remote computer not listening
34H	Duplicate name on network
35H	Network name not found
36H	Network busy
37H	Device no longer exists on network
38H	Net BIOS command limit exceeded
39H	Error in network adapter hardware
3AH	Incorrect response from network
3BH	Unexpected network error
3CH	Incompatible remote adapter
3DH	Print queue full
3EH	Queue not full
3FH	Not enough room for print file
40H	Network name deleted
41H	Access denied
42H	Incorrect network device type
43H	Network name not found
44H	Network name limit exceeded
45H	Net BIOS session limit exceeded
46H	Temporary pause
47H	Network request not accepted
48H	Print or disk redirection paused
49H–4FH	Reserved
50H	File already exists
51H	Reserved

(more)

Value in AX	Meaning
MS-DOS versions 3.x extended errors *(continued)*	
52H	Cannot make directory
53H	Failure on Interrupt 24H
54H	Out of structures
55H	Already assigned
56H	Invalid password
57H	Invalid parameter
58H	Network write fault

Locus (CH register). This value provides information on the location of the error:

Value in CH	Meaning
01H	Location unknown
02H	Block device; generally caused by a disk error
03H	Network
04H	Serial device; generally caused by a timeout from a character device
05H	Memory; caused by an error in RAM

Error Class (BH register). This value gives the general category of the error:

Value in BH	Meaning
01H	Out of resource; out of storage space or I/O channels.
02H	Temporary situation; expected to clear, as in a file or record lock—generally occurs only in a network environment.
03H	Authorization; a problem with permission to access the requested device.
04H	Internal error in system software; generally reflects a system software bug rather than an application or system failure.
05H	Hardware failure; a serious hardware-related problem not the fault of the user program.
06H	System failure; a serious failure of the system software, not directly the fault of the application—generally occurs if configuration files are missing or incorrect.
07H	Application-program error; generally caused by inconsistent function requests from the user program.
08H	File or item not found.
09H	File or item of invalid format or type detected, or an otherwise unsuitable or invalid item requested.
0AH	File or item interlocked by the system.

(more)

Value in BH	Meaning
0BH	Media failure; generally occurs with a bad disk in a drive, a bad spot on the disk, or the like.
0CH	Already exists; generally occurs when application tries to declare a machine name or device that already exists.
0DH	Unknown.

Suggested Action (BL register). One of the most useful returns from Function 59H, this value suggests a corrective action to try:

Value in BL	Meaning
01H	Retry a few times before prompting the user to choose *Ignore* for the program to continue or *Abort* to terminate.
02H	Pause for a few seconds between retries and then prompt user as above.
03H	Ask user to reenter the input. In most cases, this solution applies when an incorrect drive specifier or filename was entered. Of course, if the value was hard-coded into the program, the user should not be prompted for input.
04H	Clean up as well as possible, then abort the application. This solution applies when the error is destructive enough that the application cannot safely proceed, but the system is healthy enough to try an orderly shut-down of the application.
05H	Exit from the application as soon as possible, without trying to close files and clean up. This means something is seriously wrong with either the application or the system.
06H	Ignore; error is informational.
07H	Prompt user to perform some action, such as changing floppy disks in a drive and then retry.

Function 59H and older system calls

The Interrupt 21H functions — primarily the FCB-related file and record calls — that return 0FFH in the AL register to indicate that an error has occurred but provide no further information about the type of error include

Function	Name
0FH	Open File with FCB
10H	Close File with FCB
11H	Find First File
12H	Find Next File

(more)

Function	Name
13H	Delete File
16H	Create File with FCB
17H	Rename File
23H	Get File Size

These function calls now exist only to maintain compatibility with MS-DOS versions 1.x. The preferred choices are the handle-style calls available in MS-DOS versions 2.0 and later, which offer full path support and much better error reporting. *See also* SYSTEM CALLS.

If the older calls *must* be used, the program can use Function 59H to obtain more detailed information under MS-DOS version 3.0 or later. For example:

```
myfcb   db      0               ; drive = default
        db      'MYFILE  '      ; filename, 8 chars
        db      'DAT'           ; extension, 3 chars
        db      25 dup (0)      ; remainder of FCB
        .
        .
        .
        mov     dx,seg myfcb    ; DS:DX = FCB
        mov     ds,dx
        mov     dx,offset myfcb
        mov     ah,0fh          ; function 0FH = Open FCB

        int     21h             ; transfer to MS-DOS
        or      al,al           ; test status
        jz      success         ; jump, open succeeded
                                ; open failed, get
                                ; extended error info
        mov     bx,0            ; BX = 00H for ver. 2.x-3.x
        mov     ah,59h          ; function 59H = Get Info
        int     21h             ; transfer to MS-DOS
        or      ax,ax           ; really an error?
        jz      success         ; no error, jump
                                ; test recommended actions
        cmp     bl,01h
        jz      retry           ; if BL = 01H retry operation
        cmp     bl,04h
        jz      cleanup         ; if BL = 04H clean up and exit
        cmp     bl,05h
        jz      panic           ; if BL = 05H exit immediately
        .
        .
        .
```

Function 59H and newer system calls

The function calls listed below were added in MS-DOS versions 2.0 and later. These calls return with the carry flag set if an error occurs; in addition, the AX register contains an error value corresponding to error codes 01H through 12H of the extended error return codes:

Function	Name
MS-DOS versions 2.0 and later:	
38H	Get/Set Current Country
39H	Create Directory
3AH	Remove Directory
3BH	Change Current Directory
3CH	Create File with Handle
3DH	Open File with Handle
3EH	Close File
3FH	Read File or Device
40H	Write File or Device
41H	Delete File
42H	Move File Pointer
43H	Get/Set File Attributes
44H	IOCTL (I/O Control for Devices)
45H	Duplicate File Handle
46H	Force Duplicate File Handle
47H	Get Current Directory
48H	Allocate Memory Block
49H	Free Memory Block
4AH	Resize Memory Block
4BH	Load and Execute Program (EXEC)
4EH	Find First File
4FH	Find Next File
56H	Rename File
57H	Get/Set Date/Time of File
MS-DOS versions 3.0 and later:	
58H	Get/Set Allocation Strategy
5AH	Create Temporary File
5BH	Create New File
5CH	Lock/Unlock File Region
MS-DOS versions 3.1 and later:	
5EH	Network Machine Name/Printer Setup
5FH	Get/Make Assign List Entry

Although these newer functions have much better error reporting than the older FCB functions, Function 59H is still useful. Regardless of the version of MS-DOS that is running, the error code returned in the AX register from an Interrupt 21H function call is always in the range 0–12H. If a program is running under MS-DOS versions 3.x and wants to obtain one or more of the more specific error codes in the range 20–58H, the program must

follow the failed Interrupt 21H call with a subsequent call to Interrupt 21H Function 59H. The program can then use the code returned by Function 59H in the BL register as a guide to the action to take in response to the error. For example:

```
myfile  db      'MYFILE.DAT',0  ; ASCIIZ filename
        .
        .
        .
        mov     dx,seg myfile   ; DS:DX = ASCIIZ filename
        mov     ds,dx
        mov     dx,offset myfile
        mov     ax,3d02h        ; open, read/write
        int     21h             ; transfer to MS-DOS
        jnc     success         ; jump, open succeeded
                                ; open failed, get
                                ; extended error info
        mov     bx,0            ; BX = 00H for ver. 2.x-3.x
        mov     ah,59h          ; function 59H = Get Info
        int     21h             ; transfer to MS-DOS
        or      ax,ax           ; really an error?
        jz      success         ; no error, jump
                                ; test recommended actions
        cmp     bl,01h
        jz      retry           ; if BL = 01H retry operation
        .
        .
        .
```

If the standard critical error handler is replaced with a customized critical handler, Function 59H can also be used to obtain more detailed information about an error inside the handler before either returning control to the application or aborting. The value in the BL register should be used to determine the appropriate action to take or the message to display to the user.

Jim Kyle
Chip Rabinowitz

Article 13
Hardware Interrupt Handlers

Unlike software interrupts, which are service requests initiated by a program, hardware interrupts occur in response to electrical signals received from a peripheral device such as a serial port or a disk controller, or they are generated internally by the microprocessor itself. Hardware interrupts, whether external or internal to the microprocessor, are given prioritized servicing by the Intel CPU architecture.

The 8086 family of microprocessors (which includes the 8088, 8086, 80186, 80286, and 80386) reserves the first 1024 bytes of memory (addresses 0000:0000H through 0000:03FFH) for a table of 256 interrupt vectors, each a 4-byte far pointer to a specific interrupt service routine (ISR) that is carried out when the corresponding interrupt is processed. The design of the 8086 family requires certain of these interrupt vectors to be used for specific functions (Table 13-1). Although Intel actually reserves the first 32 interrupts, IBM, in the original PC, redefined usage of Interrupts 05H to 1FH. Most, but not all, of these reserved vectors are used by software, rather than hardware, interrupts; the redefined IBM uses are listed in Table 13-2.

Table 13-1. Intel Reserved Exception Interrupts.

Interrupt Number	Definition
00H	Divide by zero
01H	Single step
02H	Nonmaskable interrupt (NMI)
03H	Breakpoint trap
04H	Overflow trap
05H	BOUND range exceeded*
06H	Invalid opcode*
07H	Coprocessor not available†
08H	Double-fault exception†
09H	Coprocessor segment overrun†
0AH	Invalid task state segment (TSS)†
0BH	Segment not present†
0CH	Stack exception†
0DH	General protection exception†
0EH	Page fault‡

(more)

Table 13-1. *Continued.*

Interrupt Number	Definition
0FH	(Reserved)
10H	Coprocessor error†

*The 80186, 80286, and 80386 microprocessors only.
†The 80286 and 80386 microprocessors only.
‡The 80386 microprocessor only.

Table 13-2. IBM Interrupt Usage.

Interrupt Number	Definition
05H	Print screen
06H	Unused
07H	Unused
08H	Hardware IRQ0 (timer-tick)*
09H	Hardware IRQ1 (keyboard)
0AH	Hardware IRQ2 (reserved)†
0BH	Hardware IRQ3 (COM2)
0CH	Hardware IRQ4 (COM1)
0DH	Hardware IRQ5 (fixed disk)
0EH	Hardware IRQ6 (floppy disk)
0FH	Hardware IRQ7 (printer)
10H	Video service
11H	Equipment information
12H	Memory size
13H	Disk I/O service
14H	Serial-port service
15H	Cassette/network service
16H	Keyboard service
17H	Printer service
18H	ROM BASIC
19H	Restart system
1AH	Get/Set time/date
1BH	Control-Break (user defined)
1CH	Timer tick (user defined)
1DH	Video parameter pointer
1EH	Disk parameter pointer
1FH	Graphics character table

*IRQ = Interrupt request line.
†*See* Table 13-4.

Nestled in the middle of Table 13-2 are the eight hardware interrupt vectors (08-0FH) IBM implemented in the original PC design. These eight vectors provide the maskable interrupts for the IBM PC-family and close compatibles. Additional IRQ lines built into the IBM PC/AT are discussed under The IRQ Levels below.

The conflicting uses of the interrupts listed in Tables 13-1 and 13-2 have created compatibility problems as the 8086 family of microprocessors has developed. For complete compatibility with IBM equipment, the IBM usage must be followed even when it conflicts with the chip design. For example, a BOUND error occurs if an array index exceeds the specified upper and lower limits (bounds) of the array, causing an Interrupt 05H to be generated. But the 80286 processor used in all AT-class computers will, if a BOUND error occurs, send the contents of the display to the printer, because IBM uses Interrupt 05H for the Print Screen function.

Hardware Interrupt Categories

The 8086 family of microprocessors can handle three types of hardware interrupts. First are the internal, microprocessor-generated exception interrupts (Table 13-1). Second is the nonmaskable interrupt, or NMI (Interrupt 02H), which is generated when the NMI line (pin 17 on the 8088 and 8086, pin 59 on the 80286, pin B8 on the 80386) goes high (active). In the IBM PC family (except the PCjr and the Convertible), the nonmaskable interrupt is designated for memory parity errors. Third are the maskable interrupts, which are usually generated by external devices.

Maskable interrupts are routed to the main processor through a chip called the 8259A Programmable Interrupt Controller (PIC). When it receives an interrupt request, the PIC signals the microprocessor that an interrupt needs service by driving the interrupt request (INTR) line of the main processor to high voltage level. This article focuses on the maskable interrupts and the 8259A because it is through the PIC that external I/O devices (disk drives, serial communication ports, and so forth) gain access to the interrupt system.

Interrupt priorities in the 8086 family

The Intel microprocessors have a built-in priority system for handling interrupts that occur simultaneously. Priority goes to the internal instruction exception interrupts, such as Divide by Zero and Invalid Opcode, because priority is determined by the interrupt number: Interrupt 00H takes priority over all others, whereas the last possible interrupt, 0FFH, would, if present, never be allowed to break in while another interrupt was being serviced. However, if interrupt service is enabled (the microprocessor's interrupt flag is set), any hardware interrupt takes priority over any software interrupt (INT instruction).

The priority sequencing by interrupt number must not be confused with the priority resolution performed by hardware external to the microprocessor. The numeric priority discussed here applies only to interrupts generated within the 8086 family of microprocessor chips and is totally independent of system interrupt priorities established for components external to the microprocessor itself.

Interrupt service routines

For the most part, programmers need not write hardware-specific program routines to service the hardware interrupts. The IBM PC BIOS routines, together with MS-DOS services, are usually sufficient. In some cases, however, MS-DOS and the ROM BIOS do not provide enough assistance to ensure adequate performance of a program. Most notable in this category is communications software, for which programmers usually must access the 8259A and the 8250 Universal Asynchronous Receiver and Transmitter (UART) directly. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Interrupt-Driven Communications.

Characteristics of Maskable Interrupts

Two major characteristics distinguish maskable interrupts from all other events that can occur in the system: They are totally unpredictable, and they are highly volatile. In general, a hardware interrupt occurs when a peripheral device requires the full attention of the system and data will be irretrievably lost unless the system responds rapidly.

All things are relative, however, and this is especially true of the speed required to service an interrupt request. For example, assume that two interrupt requests occur at essentially the same time. One is from a serial communications port receiving data at 300 bps; the other is from a serial port receiving data at 9600 bps. Data from the first serial port will not change for at least 30 milliseconds, but the second serial port must be serviced within one millisecond to avoid data loss.

Unpredictability

Because maskable interrupts generally originate in response to external physical events, such as the receipt of a byte of data over a communications line, the exact time at which such an interrupt will occur cannot be predicted. Even the timer interrupt request, which by default occurs approximately 18.2 times per second, cannot be predicted by any program that happens to be executing when the interrupt request occurs.

Because of this unpredictability, the system must, if it allows any interrupts to be recognized, be prepared to service all maskable interrupt requests. Conversely, if interrupts cannot be serviced, they must all be disabled. The 8086 family of microprocessors provides the Set Interrupt Flag (STI) instruction to enable maskable interrupt response and the Clear Interrupt Flag (CLI) instruction to disable it. The interrupt flag is also cleared automatically when a hardware interrupt response begins; the interrupt handler should execute STI as quickly as possible to allow higher priority interrupts to be serviced.

Volatility

As noted earlier, a maskable interrupt request must normally be serviced immediately to prevent loss of data, but the concept of immediacy is relative to the data transfer rate of the device requesting the interrupt. The rule is that the currently available unit of data must be processed (at least to the point of being stored in a buffer) before the next such item can

arrive. Except for such devices as disk drives, which always require immediate response, interrupts for devices that receive data are normally much more critical than interrupts for devices that transmit data.

The problems imposed by data volatility during hardware interrupt service are solved by establishing service priorities for interrupts generated outside the microprocessor chip itself. Devices with the slowest transfer rates are assigned lower interrupt service priorities, and the most time-critical devices are assigned the highest priority of interrupt service.

Handling Maskable Interrupts

The microprocessor handles all interrupts (maskable, nonmaskable, and software) by pushing the contents of the flags register onto the stack, disabling the interrupt flag, and pushing the current contents of the CS:IP registers onto the stack.

The microprocessor then takes the interrupt number from the data bus, multiplies it by 4 (the size of each vector in bytes), and uses the result as an offset into the interrupt vector table located in the bottom 1 KB (segment 0000H) of system RAM. The 4-byte address at that location is then used as the new CS:IP value (Figure 13-1).

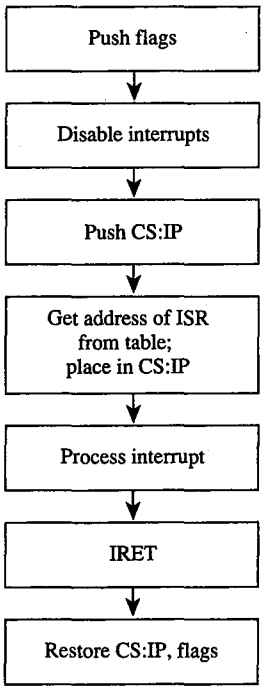


Figure 13-1. General interrupt sequence.

External devices are assigned dedicated interrupt request lines (IRQs) associated with the 8259A. *See* The IRQ Levels below. When a device requires attention, it sends a signal to the PIC via its IRQ line. The PIC, which functions as an "executive secretary" for the external devices, operates as shown in Figure 13-2. It evaluates the service request and, if appropriate, causes the microprocessor's INTR line to go high. The microprocessor then checks whether interrupts are enabled (whether the interrupt flag is set). If they are, the flags are pushed onto the stack, the interrupt flag is disabled, and CS:IP is pushed onto the stack.

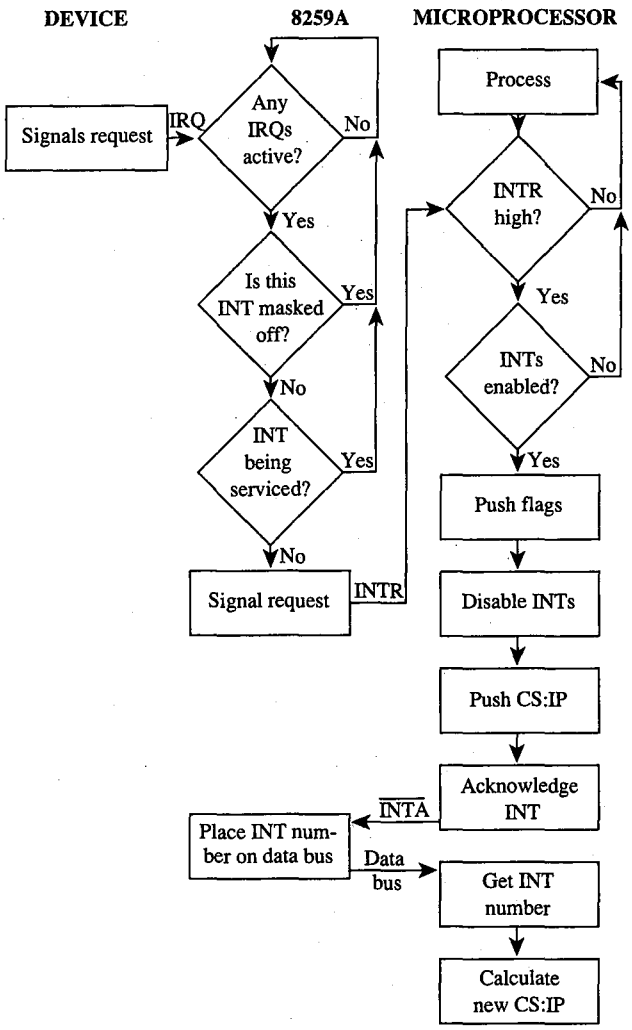


Figure 13-2. Maskable interrupt service.

The microprocessor acknowledges the interrupt request by signaling the 8259A via the interrupt acknowledge (INTA) line. The 8259A then places the interrupt number on the data bus. The microprocessor gets the interrupt number from the data bus and services the interrupt. Before issuing the IRET instruction, the interrupt service routine must issue an end-of-interrupt (EOI) sequence to the 8259A so that other interrupts can be processed. This is done by sending 20H to port 20H. (The similarity of numbers is pure coincidence.) The EOI sequence is covered in greater detail elsewhere. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Interrupt-Driven Communications.

The 8259A Programmable Interrupt Controller

The 8259A (Figure 13-3) has a number of internal components, many of them under software control. Only the default settings for the IBM PC family are covered here.

Three registers influence the servicing of maskable interrupts: the interrupt request register (IRR), the in-service register (ISR), and the interrupt mask register (IMR).

The IRR is used to keep track of the devices requesting attention. When a device causes its IRQ line to go high to signal the 8259A that it needs service, a bit is set in the IRR that corresponds to the interrupt level of the device.

The ISR specifies which interrupt levels are currently being serviced; an ISR bit is set when an interrupt has been acknowledged by the CPU (via INTA) and the interrupt number has been placed on the data bus. The ISR bit associated with a particular IRQ remains set until an EOI sequence is received.

The IMR is a read/write register (at port 21H) that masks (disables) specific interrupts. When a bit is set in this register, the corresponding IRQ line is masked and no servicing for it is performed until the bit is cleared. Thus, a particular IRQ can be disabled while all others continue to be serviced.

The fourth major block in Figure 13-3, labeled *Priority resolver*, is a complex logical circuit that forms the heart of the 8259A. This component combines the statuses of the IMR, the ISR, and the IRR to determine which, if any, pending interrupt request should be serviced and then causes the microprocessor's INTR line to go high. The priority resolver can be programmed in a number of modes, although only the mode used in the IBM PC and close compatibles is described here.

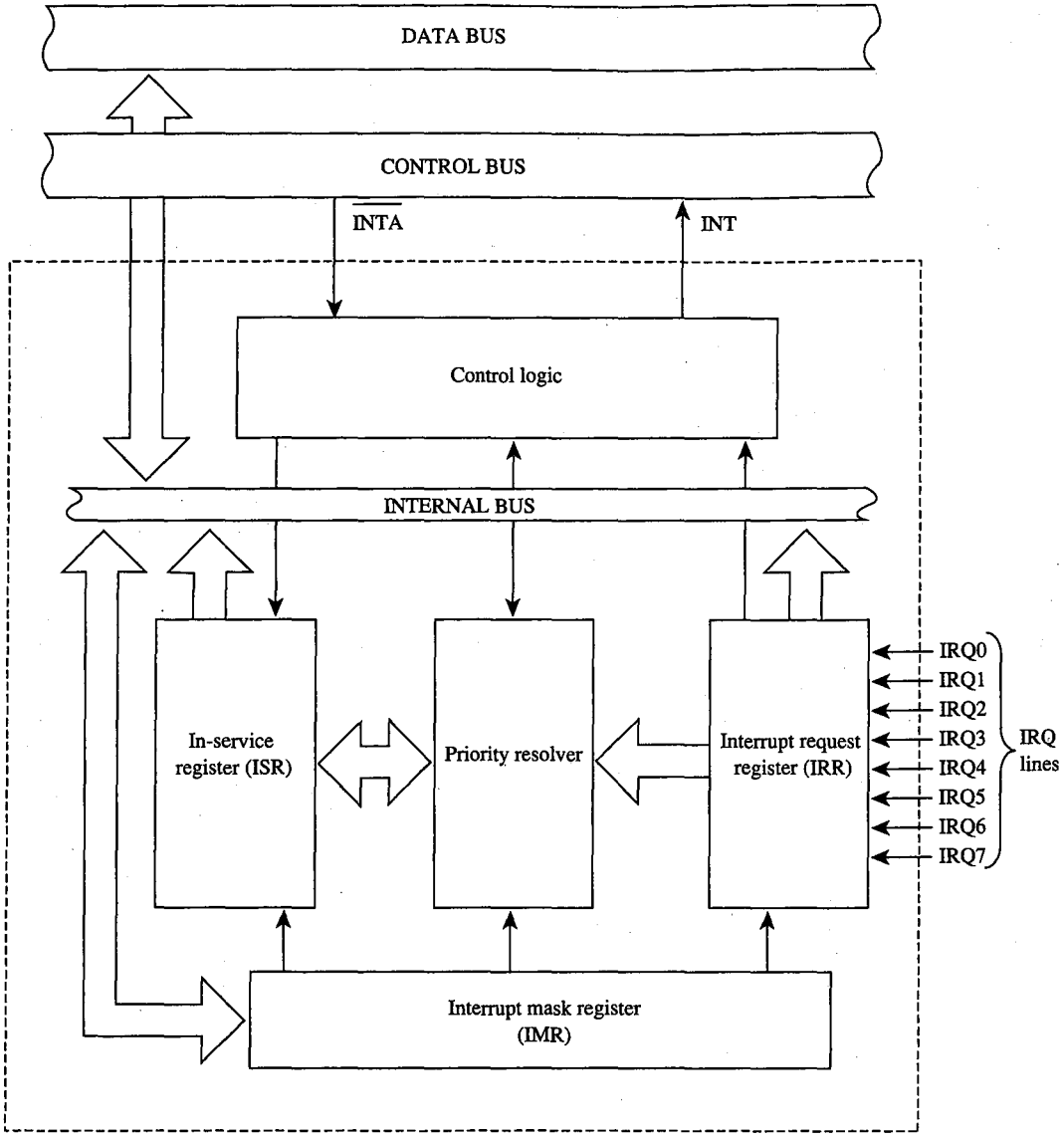


Figure 13-3. Block diagram of the 8259A Programmable Interrupt Controller.

The IRQ levels

When two or more unserviced hardware interrupts are pending, the 8259A determines which should be serviced first. The standard mode of operation for the PIC is the fully nested mode, in which IRQ lines are prioritized in a fixed sequence. Only IRQ lines with higher priority than the one currently being serviced are permitted to generate new interrupts.

The highest priority is IRQ0, and the lowest is IRQ7. Thus, if an Interrupt 09H (signaled by IRQ1) is being serviced, only an Interrupt 08H (signaled by IRQ0) can break in. All other interrupt requests are delayed until the Interrupt 09H service routine is completed and has issued an EOI sequence.

Eight-level designs

The IBM PC, PCjr, and PC/XT (and port-compatible computers) have eight IRQ lines to the PIC chip — IRQ0 through IRQ7. These lines are mapped into interrupt vectors for Interrupts 08H through 0FH (that is, 8 + IRQ level). These eight IRQ lines and their associated interrupts are listed in Table 13-3.

Table 13-3. Eight-Level Interrupt Map.

IRQ Line	Interrupt	Description
IRQ0	08H	Timer tick, 18.2 times per second
IRQ1	09H	Keyboard service required
IRQ2	0AH	I/O channel (unused on IBM PC/XT)
IRQ3	0BH	COM1 service required
IRQ4	0CH	COM2 service required
IRQ5	0DH	Fixed-disk service required
IRQ6	0EH	Floppy-disk service required
IRQ7	0FH	Data request from parallel printer*

*This request cannot be reliably generated by older versions of the IBM Monochrome/Printer Adapter and compatibles. Printer drivers that depend on this signal for operation with these cards are subject to failure.

Sixteen-level designs

In the IBM PC/AT, 8 more IRQ levels have been added by using a second 8259A PIC (the "slave") and a cascade effect, which gives 16 priority levels.

The cascade effect is accomplished by connecting the INT line of the slave to the IRQ2 line of the first, or "master," 8259A instead of to the microprocessor. When a device connected to one of the slave's IRQ lines makes an interrupt request, the INT line of the slave goes high and causes the IRQ2 line of the master 8259A to go high, which, in turn, causes the INT line of the master to go high and thus interrupts the microprocessor.

The microprocessor, ignorant of the second 8259A's presence, simply generates an interrupt acknowledge signal on receipt of the interrupt from the master 8259A. This signal initializes *both* 8259As and also causes the master to turn control over to the slave. The slave then completes the interrupt request.

On the IBM PC/AT, the eight additional IRQ lines are mapped to Interrupts 70H through 77H (Table 13-4). Because the eight additional lines are effectively connected to the master

8259A's IRQ2 line, they take priority over the master's IRQ3 through IRQ7 events. The cascade effect is graphically represented in Figure 13-4.

Table 13-4. Sixteen-Level Interrupt Map.

IRQ Line	Interrupt	Description
IRQ0	08H	Timer tick, 18.2 times per second
IRQ1	09H	Keyboard service required
IRQ2	0AH	INT from slave 8259A:
IRQ8	70H	Real-time clock service
IRQ9	71H	Software redirected to IRQ2
IRQ10	72H	Reserved
IRQ11	73H	Reserved
IRQ12	74H	Reserved
IRQ13	75H	Numeric coprocessor
IRQ14	76H	Fixed-disk controller
IRQ15	77H	Reserved
IRQ3	0BH	COM2 service required
IRQ4	0CH	COM1 service required
IRQ5	0DH	Data request from LPT2
IRQ6	0EH	Floppy-disk service required
IRQ7	0FH	Data request from LPT1

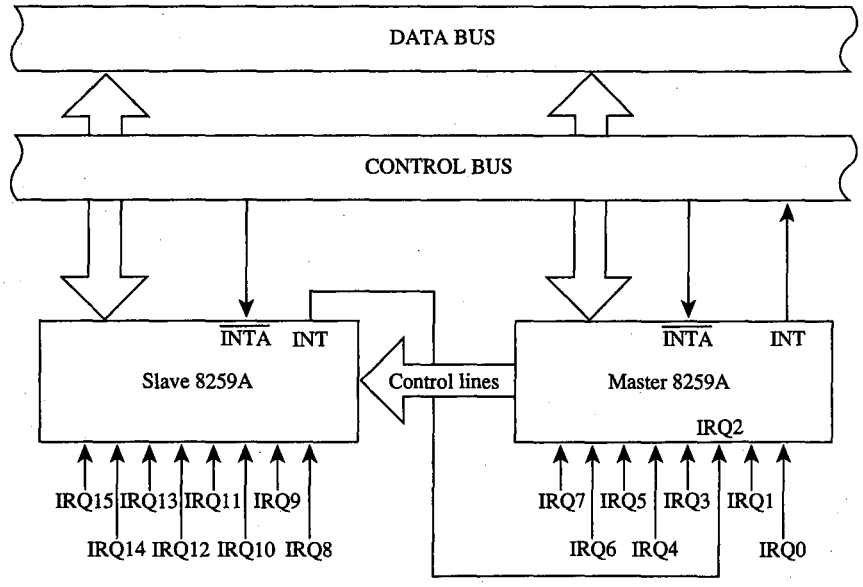


Figure 13-4. A graphic representation of the cascade effect for IRQ priorities.

Note: During the INTA sequence, the corresponding bit in the ISR register of both 8259As is set, so two EOIs must be issued to complete the interrupt service — one for the slave and one for the master.

Programming for the Hardware Interrupts

Any program that modifies an interrupt vector must restore the vector to its original condition before returning control to MS-DOS (or to its parent process). Any program that totally replaces an existing hardware interrupt handler with one of its own must perform all the handshaking and terminating actions of the original — re-enable interrupt service, signal EOI to the interrupt controller, and so forth. Failure to follow these rules has led to many hours of programmer frustration. *See also* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Exception Handlers.

When an existing interrupt handler is completely replaced with a new, customized routine, the existing vector must be saved so it can be restored later. Although it is possible to modify the 4-byte vector by directly addressing the vector table in low RAM (and many published programs have followed this practice), any program that does so runs the risk of causing system failure when the program is used with multitasking or multiuser enhancements or with future versions of MS-DOS. The only technique that can be recommended for either obtaining the existing vector values or changing them is to use the MS-DOS functions provided for this purpose: Interrupt 21H Functions 25H (Set Interrupt Vector) and 35H (Get Interrupt Vector).

After the existing vector has been saved, it can be replaced with a far pointer to the replacement routine. The new routine must end with an IRET instruction. It should also take care to preserve all microprocessor registers and conditions at entry and restore them before returning.

A sample replacement handler

Suppose a program performs many mathematical calculations of random values. To prevent abnormal termination of the program by the default MS-DOS Interrupt 00H handler when a DIV or IDIV instruction is attempted and the divisor is zero, a programmer might want to replace the Interrupt 00H (Divide by Zero) routine with one that informs the user of what has happened and then continues operation without abnormal termination. The .COM program DIVZERO.ASM (Figure 13-5) does just that. (Another example is included in the article on interrupt-driven communications. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Interrupt-Driven Communications.)

```
        name    divzero
        title   'DIVZERO - Interrupt 00H Handler'
;
; DIVZERO.ASM: Demonstration Interrupt 00H Handler
;
; To assemble, link, and convert to COM file:
;
;       C>MASM DIVZERO;  <Enter>
;       C>LINK DIVZERO;  <Enter>
;       C>EXE2BIN DIVZERO.EXE DIVZERO.COM <Enter>
;       C>DEL DIVZERO.EXE  <Enter>
;

cr      equ     0dh             ; ASCII carriage return
lf      equ     0ah             ; ASCII linefeed
eos     equ     '$'             ; end of string marker

_TEXT   segment word public 'CODE'

        assume  cs:_TEXT,ds:_TEXT,es:_TEXT,ss:_TEXT

        org     100h

entry:  jmp     start           ; skip over data area

intmsg  db      'Divide by Zero Occurred!',cr,lf,eos

divmsg  db      'Dividing '     ; message used by demo
par1    db      '0000h'         ; dividend goes here
        db      ' by '
par2    db      '00h'           ; divisor goes here
        db      ' equals '
par3    db      '00h'           ; quotient here
        db      ' remainder '
par4    db      '00h'           ; and remainder here
        db      cr,lf,eos

oldint0 dd      ?               ; save old Int 00H vector

intflag db      0               ; nonzero if divide by
                                ; zero interrupt occurred

oldip   dw      0               ; save old IP value


;
; The routine 'int0' is the actual divide by zero
; interrupt handler.  It gains control whenever a
; divide by zero or overflow occurs.  Its action
; is to set a flag and then increment the instruction
; pointer saved on the stack so that the failing
```

(more)

Figure 13-5. The Divide by Zero replacement handler, DIVZERO.ASM. This code is specific to 80286 and 80386 microprocessors. (See *Appendix M: 8086/8088 Software Compatibility Issues.*)

```
; divide will not be reexecuted after the IRET.
;
; In this particular case we can call MS-DOS to
; display a message during interrupt handling
; because the application triggers the interrupt
; intentionally. Thus, it is known that MS-DOS or
; other interrupt handlers are not in control
; at the point of interrupt.
;

int0:   pop     cs:oldip        ; capture instruction pointer

        push    ax
        push    bx
        push    cx
        push    dx
        push    di
        push    si
        push    ds
        push    es

        push    cs              ; set DS = CS
        pop     ds

        mov     ah,09h          ; print error message
        mov     dx,offset _TEXT:intmsg
        int     21h

        add     oldip,2         ; bypass instruction causing
                                ; divide by zero error

        mov     intflag,1       ; set divide by 0 flag

        pop     es              ; restore all registers
        pop     ds
        pop     si
        pop     di
        pop     dx
        pop     cx
        pop     bx
        pop     ax

        push    cs:oldip        ; restore instruction pointer

        iret                    ; return from interrupt


;
; The code beginning at 'start' is the application
; program.  It alters the vector for Interrupt 00H to
; point to the new handler, carries out some divide
```

Figure 13-5. Continued.

(more)

```
; operations (including one that will trigger an
; interrupt) for demonstration purposes, restores
; the original contents of the Interrupt 00H vector,
; and then terminates.
;
start:  mov     ax,3500h        ; get current contents
        int     21h             ; of Int 00H vector

                                ; save segment:offset
                                ; of previous Int 00H handler
        mov     word ptr oldint0,bx
        mov     word ptr oldint0+2,es

                                ; install new handler...
        mov     dx,offset int0  ; DS:DX = handler address
        mov     ax,2500h        ; call MS-DOS to set
        int     21h             ; Int 00H vector

                                ; now our handler is active,
                                ; carry out some test divides.

        mov     ax,20h          ; test divide
        mov     bx,1            ; divide by 1
        call    divide

        mov     ax,1234h        ; test divide
        mov     bx,5eh          ; divide by 5EH
        call    divide

        mov     ax,5678h        ; test divide
        mov     bx,7fh          ; divide by 127
        call    divide

        mov     ax,20h          ; test divide
        mov     bx,0            ; divide by 0
        call    divide          ; (triggers interrupt)

                                ; demonstration complete,
                                ; restore old handler

        lds     dx,oldint0      ; DS:DX = handler address
        mov     ax,2500h        ; call MS-DOS to set
        int     21h             ; Int 00H vector

        mov     ax,4c00h        ; final exit to MS-DOS
        int     21h             ; with return code = 0

;
; The routine 'divide' carries out a trial division,
; displaying the arguments and the results.  It is
```

Figure 13-5. Continued.

(more)

```
; called with AX = dividend and BL = divisor.
;
divide  proc    near

        push    ax              ; save arguments
        push    bx

        mov     di,offset par1  ; convert dividend to
        call    wtoa            ; ASCII for display

        mov     ax,bx           ; convert divisor to
        mov     di,offset par2  ; ASCII for display
        call    btoa

        pop     bx              ; restore arguments
        pop     ax

        div     bl              ; perform the division
        cmp     intflag,0       ; divide by zero detected?
        jne     nodiv           ; yes, skip display

        push    ax              ; no, convert quotient to
        mov     di,offset par3  ; ASCII for display
        call    btoa

        pop     ax              ; convert remainder to
        xchg    ah,al           ; ASCII for display
        mov     di,offset par4
        call    btoa

        mov     ah,09h          ; show arguments, results
        mov     dx,offset divmsg
        int     21h

nodiv:  mov     intflag,0       ; clear divide by 0 flag
        ret                     ; and return to caller

divide  endp


wtoa    proc    near            ; convert word to hex ASCII
                                ; call with AX = binary value
                                ;           DI = addr for string
                                ; returns AX, CX, DI destroyed

        push    ax              ; save original value
        mov     al,ah
        call    btoa            ; convert upper byte
        add     di,2            ; increment output address
```

Figure 13-5. Continued.

(more)

```
        pop     ax
        call    btoa            ; convert lower byte
        ret                     ; return to caller

wtoa    endp


btoa    proc    near            ; convert byte to hex ASCII
                                ; call with AL = binary value
                                ;           DI = addr to store string
                                ; returns AX, CX destroyed

        mov     ah,al           ; save lower nibble
        mov     cx,4            ; shift right 4 positions
        shr     al,cl           ; to get upper nibble
        call    ascii           ; convert 4 bits to ASCII
        mov     [di],al         ; store in output string
        mov     al,ah           ; get back lower nibble

        and     al,0fh          ; blank out upper one
        call    ascii           ; convert 4 bits to ASCII
        mov     [di+1],al       ; store in output string
        ret                     ; back to caller

btoa    endp


ascii   proc    near            ; convert AL bits 0-3 to
                                ; ASCII {0...9,A...F}
        add     al,'0'          ; and return digit in AL
        cmp     al,'9'
        jle     ascii2
        add     al,'A'-'9'-1    ; "fudge factor" for A-F
ascii2: ret                     ; return to caller

ascii   endp

_TEXT   ends

        end     entry
```

Figure 13-5. Continued.

Supplementary handlers

In many cases, a custom interrupt handler augments, rather than replaces, the existing routine. The added routine might process some data before passing the data to the existing routine, or it might do the processing afterward. These cases require slightly different coding for the handler.

If the added routine is to process data before the existing handler does, the routine need only jump to the original handler after completing its processing. This jump can be done

indirectly, with the same pointer used to save the original content of the vector for restoration at exit. For example, a replacement Interrupt 08H handler that merely increments an internal flag at each timer tick can look something like the following:

```
        .
        .
        .
myflag  dw      ?                       ; variable to be incremented
                                        ; on each timer-tick interrupt

oldint8 dd      ?                       ; contains address of previous
                                        ; timer-tick interrupt handler
        .
        .                               ; get the previous contents
        .                               ; of the Interrupt 08H vector...
        mov     ax,3508h                ; AH = 35H (Get Interrupt Vector)
        int     21h                     ; AL = Interrupt number (08H)
        mov     word ptr oldint8,bx     ; save the address of
        mov     word ptr oldint8+2,es   ; the previous Int 08H Handler
        mov     dx,seg myint8           ; put address of the new
        mov     ds,dx                   ; interrupt handler into DS:DX
        mov     dx,offset myint8        ; and call MS-DOS to set vector
        mov     ax,2508h                ; AH = 25H (Set Interrupt Vector)
        int     21h                     ; AL = Interrupt number (08H)
        .
        .
        .
myint8:                                 ; this is the new handler
                                        ; for Interrupt 08H

        inc     cs:myflag               ; increment variable on each
                                        ; timer-tick interrupt

        jmp     dword ptr cs:[oldint8]  ; then chain to the
                                        ; previous interrupt handler
```

The added handler must preserve all registers and machine conditions, except those machine conditions it will modify, such as the value of *myflag* in the example (and the flags register, which is saved by the interrupt action), and it must restore those registers and conditions before performing the jump to the original handler.

A more complex situation arises when a replacement handler does some processing *after* the original routine executes, especially if the replacement handler is not reentrant. To allow for this processing, the replacement handler must prevent nested interrupts, so that even if the old handler (which is chained to the replacement handler by a CALL instruction) issues an EOI, the replacement handler will not be interrupted during postprocessing. For example, instead of using the preceding Interrupt 08H example routine, the programmer could use the following code to implement *myflag* as a semaphore and use the XCHG instruction to test it:

```
myint8:                                 ; this is the new handler
                                        ; for Interrupt 08H

        mov     ax,1                    ; test and set interrupt-
        xchg    cs:myflag,ax            ; handling-in-progress semaphore

        push    ax                      ; save the semaphore

        pushf                           ; simulate interrupt, allowing
        call    dword ptr cs:oldint8    ; the previous handler for the
                                        ; Interrupt 08H vector to run

        pop     ax                      ; get the semaphore back
        or      ax,ax                   ; is our interrupt handler
                                        ; already running?

        jnz     myint8x                 ; yes, skip this one

        .                               ; now perform our interrupt
        .                               ; processing here...
        .

        mov     cs:myflag,0             ; clear the interrupt-handling-
                                        ; in-progress flag

myint8x:
        iret                            ; return from interrupt
```

Note that an interrupt handler of this type must simulate the original call to the interrupt routine by first doing a PUSHF, followed by a far CALL via the saved pointer to execute the original handler routine. The flags register pushed onto the stack is restored by the IRET of the original handler. Upon return from the original code, the new routine can preserve the machine state and do its own processing, finally returning to the caller by means of its own IRET.

The flags inside the new routine need not be preserved, as they are automatically restored by the IRET instruction. Because of the nature of interrupt servicing, the service routine should not depend on any information in the flags register, nor can it return any information in the flags register. Note also that the previous handler (invoked by the indirect CALL) will almost certainly have dismissed the interrupt by sending an EOI to the 8259A PIC. Thus, the machine state is not the same as in the first *myint8* example.

To remove the new vector and restore the original, the program simply replaces the new vector (in the vector table) with the saved copy. If the substituted routine is part of an application program, the original vector must be restored for every possible method of exiting from the program (including Control-Break, Control-C, and critical-error *Abort* exits). Failure to observe this requirement invariably results in system failure. Even though the system failure might be delayed for some time after the exit from the offending program, when some subsequent program overlays the interrupt handler code the crash will be imminent.

Summary

Hardware interrupt handler routines, although not strictly a part of MS-DOS, form an integral part of many MS-DOS programs and are tightly constrained by MS-DOS requirements. Routines of this type play important roles in the functioning of the IBM personal computers, and, with proper design and programming, significantly enhance product reliability and performance. In some instances, no other practical method exists for meeting performance requirements.

Jim Kyle
Chip Rabinowitz

Article 14
Writing MS-DOS Filters

A filter is, essentially, a program that operates on a stream of characters. The source and destination of the character stream can be files, another program, or almost any character device. The transformation applied by the filter to the character stream can range from an operation as simple as substituting a character set to an operation as elaborate as generating splines from sets of coordinates.

The standard MS-DOS package includes three simple filters: SORT, which alphabetically sorts text on a line-by-line basis; FIND, which searches a text stream to match a specified string; and MORE, which displays text one screenful at a time. This article describes how filters work and how new ones can be constructed. *See also* USER COMMANDS: FIND; MORE; SORT.

System Support for Filters

The operation of a filter program relies on two features that appeared in MS-DOS version 2.0: standard devices and redirectable I/O.

The standard devices are represented by five handles that are originally established when the system is initialized. Each process inherits these handles from its immediate parent. Thus, the standard device handles are already opened when a process acquires control of the system, and the process can use the handles with Interrupt 21H Functions 3FH and 40H for read and write operations without further preliminaries. The default assignments of the standard device handles are

Handle	Name	Default Device
0	*stdin* (standard input)	CON
1	*stdout* (standard output)	CON
2	*stderr* (standard error)	CON
3	*stdaux* (standard auxiliary)	AUX
4	*stdlst* (standard list)	PRN

The CON device is assigned by default to the system's keyboard and video display. AUX is assigned by default to COM1 (the first physical serial port), and PRN is assigned by default to LPT1 (the first physical parallel printer port); in some systems these assignments can be altered with the MODE command. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Character Device Input and Output; USER COMMANDS: MODE; CTTY.

When a program is executed by entering its name at the system (COMMAND.COM) prompt, the user can redirect either or both of the standard input and standard output handles from their default device (CON) to another file, a character device, or a process. This redirection is accomplished by including one of the special characters <, >, >>, or ¦ in the command line, in the following form:

Redirection	Result
< *file*	Contents of the specified *file* are used instead of the keyboard as the program's standard input.
< *device*	Program takes its standard input from the named *device* instead of from the keyboard.
> *device*	Program sends its standard output to the named *device* instead of to the video display.
> *file*	Program sends its standard output to the specified *file* instead of to the video display.
>> *file*	Program appends its standard output to the current contents of the specified *file* instead of to the video display.
p1 ¦ *p2*	Standard output of program *p1* is routed to become the standard input of program *p2* (output of *p1* is said to be piped to *p2*).

For example, the command

```
C>SORT < MYFILE.TXT > PRN  <Enter>
```

causes the SORT filter to read its input from the file MYFILE.TXT, sort the lines alphabetically, and write resulting text to the character device PRN (the logical name for the system's list device).

The redirection requested by the <, >, >>, or ¦ characters takes place at the level of COMMAND.COM and is invisible to the program it affects. Such redirection can also be put into effect by another process. *See* Using a Filter as a Child Process below.

Note that if a program "goes around" MS-DOS to perform its input and output, either by calling ROM BIOS functions or by manipulating the keyboard or video controller directly, redirection commands placed in the program's command line do not have the expected effect.

How Filters Work

By convention, a filter program reads its text from standard input and writes the results of its operations to standard output. When the end of the input stream is reached, the filter simply terminates, optionally writing an end-of-file mark (1AH) to the output stream. As a result, filters are both flexible and simple.

Filter programs are flexible because they do not know, and do not care, about the source of the data they process or the destination of their output. Any redirection that the user

specifies in the command line is invisible to the filter. Thus, any character device that has a logical name within the system (CON, AUX, COM1, COM2, PRN, LPT1, LPT2, LPT3, and so on), any file on any block device (local or network) known to the system, or any other program can supply a filter's input or accept its output. If necessary, several functionally simple filters can be concatenated with pipes to perform very complex operations.

Although flexible, filters are also simple because they rely on their parent process to supply standard input and standard output handles that have already been appropriately redirected. The parent is responsible for opening or creating any necessary files, checking the validity of logical character device names, and loading and executing the preceding or following process in a pipe. The filter need only concern itself with the transformation it will apply to the data; it can leave the I/O details to the operating system and to its parent.

Building a Filter

Creating a new filter for MS-DOS is a straightforward process. In its simplest form, a filter need only use the handle-oriented read (Interrupt 21H Function 3FH) and write (Interrupt 21H Function 40H) functions to get characters or lines from standard input and send them to standard output, performing any desired alterations on the text stream on a character-by-character or line-by-line basis.

Figures 14-1 through 14-4 contain template character-oriented and line-oriented filters in both assembly language and C. The C version of the character filter runs much faster than the assembly-language version, because the C run-time library provides hidden blocking and deblocking (buffering) of character reads and writes; the assembly-language program actually makes two calls to MS-DOS for each character processed. (Of course, if buffering is added to the assembly-language version it will be both faster and smaller than the C filter.) The C and assembly-language versions of the line-oriented filter run at roughly the same speed.

```
        name    protoc
        title   'PROTOC.ASM --- template character filter'
;
; PROTOC.ASM: a template for a character-oriented filter.
;
; Ray Duncan, June 1987
;

stdin   equ     0               ; standard input
stdout  equ     1               ; standard output
stderr  equ     2               ; standard error

cr      equ     0dh             ; ASCII carriage return
lf      equ     0ah             ; ASCII linefeed
```

Figure 14-1. Assembly-language template for a character-oriented filter (file PROTOC.ASM). *(more)*

```
DGROUP  group   _DATA,STACK     ; 'automatic data group'

_TEXT   segment byte public 'CODE'

        assume  cs:_TEXT,ds:DGROUP,ss:STACK

main    proc    far             ; entry point from MS-DOS

        mov     ax,DGROUP       ; set DS = our data segment
        mov     ds,ax

main1:                          ; read a character from standard input
        mov     dx,offset DGROUP:char   ; address to place character
        mov     cx,1            ; length to read = 1
        mov     bx,stdin        ; handle for standard input
        mov     ah,3fh          ; function 3FH = read from file or device
        int     21h             ; transfer to MS-DOS
        jc      main3           ; error, terminate
        cmp     ax,1            ; any character read?
        jne     main2           ; end of file, terminate program

        call    translt         ; translate character if necessary

                                ; now write character to standard output
        mov     dx,offset DGROUP:char   ; address of character
        mov     cx,1            ; length to write = 1
        mov     bx,stdout       ; handle for standard output
        mov     ah,40h          ; function 40H = write to file or device
        int     21h             ; transfer to MS-DOS
        jc      main3           ; error, terminate
        cmp     ax,1            ; was character written?
        jne     main3           ; disk full, terminate program
        jmp     main1           ; go process another character

main2:  mov     ax,4c00h        ; end of file reached, terminate
        int     21h             ; program with return code = 0

main3:  mov     ax,4c01h        ; error or disk full, terminate
        int     21h             ; program with return code = 1

main    endp                    ; end of main procedure

;
; Perform any necessary translation on character from input,
; stored in 'char'.  Template action: leave character unchanged.
;
translt proc    near

        ret                     ; template action: do nothing

translt endp
```

Figure 14-1. Continued. *(more)*

```
_TEXT   ends

_DATA   segment word public 'DATA'

char    db      0               ; temporary storage for input character

_DATA   ends

STACK   segment para stack 'STACK'

        dw      64 dup (?)

STACK   ends

        end     main            ; defines program entry point
```

Figure 14-1. Continued.

```
/*
        PROTOC.C: a template for a character-oriented filter.

        Ray Duncan, June 1987
*/

#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];
{       char ch;

        while ( (ch=getchar())!=EOF )   /* read a character */
        {       ch=translate(ch);       /* translate it if necessary */
                putchar(ch);            /* write the character */
        }
        exit(0);                        /* terminate at end of file */
}

/*
        Perform any necessary translation on character from
        input file.  Template action just returns same character.
*/

int translate(ch)
char ch;
{       return (ch);
}
```

Figure 14-2. C template for a character-oriented filter (file PROTOC.C).

```
        name    protol
        title   'PROTOL.ASM --- template line filter'
;
; PROTOL.ASM:  a template for a line-oriented filter.
;
; Ray Duncan, June 1987
;

stdin   equ     0               ; standard input
stdout  equ     1               ; standard output
stderr  equ     2               ; standard error

cr      equ     0dh             ; ASCII carriage return
lf      equ     0ah             ; ASCII linefeed


DGROUP  group   _DATA,STACK     ; 'automatic data group'


_TEXT   segment byte public 'CODE'

        assume  cs:_TEXT,ds:DGROUP,es:DGROUP,ss:STACK

main    proc    far             ; entry point from MS-DOS

        mov     ax,DGROUP       ; set DS = ES = our data segment
        mov     ds,ax
        mov     es,ax

main1:                          ; read a line from standard input
        mov     dx,offset DGROUP:input  ; address to place data
        mov     cx,256          ; max length to read = 256
        mov     bx,stdin        ; handle for standard input
        mov     ah,3fh          ; function 3FH = read from file or device
        int     21h             ; transfer to MS-DOS
        jc      main3           ; if error, terminate
        or      ax,ax           ; any characters read?
        jz      main2           ; end of file, terminate program

        call    translt         ; translate line if necessary
        or      ax,ax           ; anything to output after translation?
        jz      main1           ; no, get next line

                                ; now write line to standard output
        mov     dx,offset DGROUP:output ; address of data
        mov     cx,ax           ; length to write
        mov     bx,stdout       ; handle for standard output
        mov     ah,40h          ; function 40H = write to file or device
        int     21h             ; transfer to MS-DOS
        jc      main3           ; if error, terminate
```

Figure 14-3. Assembly-language template for a line-oriented filter (file PROTOL.ASM). *(more)*

```
        cmp     ax,cx           ; was entire line written?
        jne     main3           ; disk full, terminate program
        jmp     main1           ; go process another line

main2:  mov     ax,4c00h        ; end of file reached, terminate
        int     21h             ; program with return code = 0

main3:  mov     ax,4c01h        ; error or disk full, terminate
        int     21h             ; program with return code = 1

main    endp                    ; end of main procedure

;
; Perform any necessary translation on line stored in
; 'input' buffer, leaving result in 'output' buffer.
;
; Call with:    AX = length of data in 'input' buffer.
;
; Return:       AX = length to write to standard output.
;
; Action of template routine is just to copy the line.
;
translt proc    near

                                ; just copy line from input to output
        mov     si,offset DGROUP:input
        mov     di,offset DGROUP:output
        mov     cx,ax
        rep movsb
        ret                     ; return length in AX unchanged

translt endp


_TEXT   ends


_DATA   segment word public 'DATA'

input   db      256 dup (?)     ; storage for input line
output  db      256 dup (?)     ; storage for output line

_DATA   ends


STACK   segment para stack 'STACK'

        dw      64 dup (?)

STACK   ends


        end     main            ; defines program entry point
```

Figure 14-3. Continued.

```
/*
        PROTOL.C: a template for a line-oriented filter.

        Ray Duncan, June 1987.
*/

#include <stdio.h>

static char input[256];                 /* buffer for input line */
static char output[256];                /* buffer for output line */

main(argc,argv)
int argc;
char *argv[];
{       while( gets(input) != NULL )    /* get a line from input stream */
                                        /* perform any necessary translation
                                           and possibly write result */
        {       if (translate()) puts(output);
        }
        exit(0);                        /* terminate at end of file */
}

/*
        Perform any necessary translation on input line, leaving
        the resulting text in output buffer.  Value of function
        is 'true' if output buffer should be written to standard output
        by main routine, 'false' if nothing should be written.
*/

translate()
{       strcpy(output,input);           /* template action is copy input */
        return(1);                      /* line and return true flag */
}
```

Figure 14-4. C template for a line-oriented filter (file PROTOL.C).

Each of the four template filters can be assembled or compiled, linked, and run exactly as they are shown in Figures 14-1 through 14-4. Of course, in this form they function like an incredibly slow COPY command.

To obtain a filter that does something useful, a routine that performs some modification of the text stream that is flowing by must be inserted between the reads and writes. For example, Figures 14-5 and 14-6 contain the assembly-language and C source code for a character-oriented filter named LC. This program converts all uppercase input characters (A–Z) to lowercase (a–z) output, leaving other characters unchanged. The only difference between LC and the template character filter is the translation subroutine that operates on the text stream.

```
        name    lc
        title   'LC.ASM --- lowercase filter'
;
; LC.ASM:       a simple character-oriented filter to translate
;               all uppercase (A-Z) to lowercase (a-z).
;
; Ray Duncan, June 1987
;

stdin   equ     0               ; standard input
stdout  equ     1               ; standard output
stderr  equ     2               ; standard error

cr      equ     0dh             ; ASCII carriage return
lf      equ     0ah             ; ASCII linefeed


DGROUP  group   _DATA,STACK     ; 'automatic data group'


_TEXT   segment byte public 'CODE'

        assume  cs:_TEXT,ds:DGROUP,ss:STACK

main    proc    far             ; entry point from MS-DOS

        mov     ax,DGROUP       ; set DS = our data segment
        mov     ds,ax

main1:                          ; read a character from standard input
        mov     dx,offset DGROUP:char   ; address to place character
        mov     cx,1            ; length to read = 1
        mov     bx,stdin        ; handle for standard input
        mov     ah,3fh          ; function 3FH = read from file or device
        int     21h             ; transfer to MS-DOS
        jc      main3           ; error, terminate
        cmp     ax,1            ; any character read?
        jne     main2           ; end of file, terminate program

        call    translt         ; translate character if necessary

                                ; now write character to standard output
        mov     dx,offset DGROUP:char   ; address of character
        mov     cx,1            ; length to write = 1
        mov     bx,stdout       ; handle for standard output
        mov     ah,40h          ; function 40H = write to file or device
        int     21h             ; transfer to MS-DOS
        jc      main3           ; error, terminate
        cmp     ax,1            ; was character written?
        jne     main3           ; disk full, terminate program
        jmp     main1           ; go process another character
```

Figure 14-5. Assembly-language source code for the LC filter (file LC.ASM). *(more)*

```
main2:  mov     ax,4c00h        ; end of file reached, terminate
        int     21h             ; program with return code = 0

main3:  mov     ax,4c01h        ; error or disk full, terminate
        int     21h             ; program with return code = 1

main    endp                    ; end of main procedure

;
; Translate uppercase (A-Z) characters to corresponding
; lowercase characters (a-z).  Leave other characters unchanged.
;
translt proc    near

        cmp     byte ptr char,'A'
        jb      transx
        cmp     byte ptr char,'Z'
        ja      transx
        add     byte ptr char,'a'-'A'
transx: ret

translt endp


_TEXT   ends


_DATA   segment word public 'DATA'

char    db      0               ; temporary storage for input character

_DATA   ends


STACK   segment para stack 'STACK'

        dw      64 dup (?)

STACK   ends


        end     main            ; defines program entry point
```

Figure 14-5. Continued.

```
/*
        LC:     a simple character-oriented filter to translate
                all uppercase (A-Z) to lowercase (a-z) characters.

        Usage:  LC [< source] [> destination]
```

Figure 14-6. C source code for the LC filter (file LC.C). *(more)*

```
        Ray Duncan, June 1987

*/

#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];
{       char ch;
                                        /* read a character */
        while ( (ch=getchar() ) != EOF )
        {       ch=translate(ch);       /* perform any necessary
                                           character translation */
                putchar(ch);            /* then write character */
        }
        exit(0);                        /* terminate at end of file */
}

/*
        Translate characters A-Z to lowercase equivalents
*/

int translate(ch)
char ch;
{       if (ch >= 'A' && ch <= 'Z') ch += 'a'-'A';
        return (ch);
}
```

Figure 14-6. Continued.

As another example, Figure 14-7 contains the C source code for a line-oriented filter called FIND. This simple filter is invoked with a command line in the form

FIND "*pattern*" < *source* > *destination*

FIND searches the input stream for lines containing the pattern specified in the command line. The line number and text of any line containing a match is sent to standard output, with any tabs expanded to eight-column tab stops.

```
/*
        FIND.C          Searches text stream for a string.

        Usage:          FIND "pattern" [< source] [> destination]

        by Ray Duncan, June 1987

*/

#include <stdio.h>
```

Figure 14-7. C source code for a new FIND filter (file FIND.C). *(more)*

```
#define TAB     '\x09'                  /* ASCII tab character (^I) */
#define BLANK   '\x20'                  /* ASCII space character */

#define TAB_WIDTH 8                     /* columns per tab stop */

static char input[256];                 /* buffer for line from input */
static char output[256];                /* buffer for line to output */
static char pattern[256];               /* buffer for search pattern */

main(argc,argv)
int argc;
char *argv[];
{       int line=0;                     /* initialize line variable */

        if ( argc < 2 )                 /* was search pattern supplied? */
        {       puts("find: missing pattern.");
                exit(1);                /* abort if not */
        }
        strcpy(pattern,argv[1]);        /* save copy of string to find */
        strupr(pattern);                /* fold it to uppercase */
        while( gets(input) != NULL )    /* read a line from input */
        {       line++;                 /* count lines */
                strcpy(output,input);   /* save copy of input string */
                strupr(input);          /* fold input to uppercase */
                                        /* if line contains pattern */
                if( strstr(input,pattern) )
                                        /* write it to standard output */
                        writeline(line,output);
        }
        exit(0);                        /* terminate at end of file */
}

/*
        WRITELINE: Write line number and text to standard output,
        expanding any tab characters to stops defined by TAB_WIDTH.
*/

writeline(line,p)
int line;
char *p;
{       int i=0;                        /* index to original line text */
        int col=0;                      /* actual output column counter */
        printf("\n%4d: ",line);         /* write line number */
        while( p[i]!=NULL )             /* while end of line not reached */
        {       if(p[i]==TAB)           /* if current char = tab, expand it */
                {       do putchar(BLANK);
                        while((++col % TAB_WIDTH) != 0);
                }
                else                    /* otherwise just send character */
                {       putchar(p[i]);
                        col++;          /* count columns */
                }
```

Figure 14-7. Continued. *(more)*

```
                i++;                    /* advance through output line */
        }
}
```

Figure 14-7. Continued.

This sample FIND filter differs from the FIND filter supplied by Microsoft with MS-DOS in several respects. It is not case sensitive, so the pattern "foobar" will match "FOOBAR", "FooBar", and so forth. Second, this filter supports no switches; these are left as an exercise for the reader. Third, unlike the Microsoft version of FIND, this program always reads from standard input; it is not able to open its own files.

Using a Filter as a Child Process

Instead of incorporating all the code necessary to do the job itself, an application program can load and execute a filter as a child process to carry out a specific task. Before the child filter is loaded, the parent must arrange for the standard input and standard output handles that will be inherited by the child to be attached to the files or character devices that will supply the filter's input and receive its output. This redirection is accomplished with the following steps using Interrupt 21H functions:

1. The parent process uses Function 45H (Duplicate File Handle) to create duplicates of its standard input and standard output handles and then saves the duplicates.
2. The parent opens (with Function 3DH) or creates (with Function 3CH) the files or devices that the child process will use for input and output.
3. The parent uses Function 46H (Force Duplicate File Handle) to force its own standard device handles to track the new file or device handles acquired in step 2.
4. The parent uses Function 4B00H (Load and Execute Program [EXEC]) to load and execute the child process. The child inherits the redirected standard input and standard output handles and uses them to do its work. The parent regains control after the child filter terminates.
5. The parent uses the duplicate handles created in step 1, together with Function 46H (Force Duplicate File Handle), to restore its own standard input and standard output handles to their original meanings.
6. The parent closes (with Function 3EH) the duplicate handles created in step 1, because they are no longer needed.

It might seem as though the parent process could just as easily close its own standard input and standard output (handles 0 and 1), open the input and output files needed by the child, load and execute the child, close the files upon regaining control, and then reopen the CON device twice. Because the open operation always assigns the first free handle, this approach would have the desired effect as far as the child process is concerned. However, it would throw away any redirection that had been established for the parent process by its parent. Thus, the need to preserve any preexisting redirection of the parent's standard

input and standard output, along with the desire to preserve the parent's usual output channel for informational messages right up to the actual point of the EXEC call, is the reason for the elaborate procedure outlined above in steps 1 through 6.

The program EXECSORT.ASM in Figure 14-8 demonstrates this redirection of input and output for a filter run as a child process. The parent, which is called EXECSORT, saves duplicates of its current standard input and standard output handles and then redirects those handles respectively to the files MYFILE.DAT (which it opens) and MYFILE.SRT (which it creates). EXECSORT then uses Interrupt 21H Function 4BH (EXEC) to run the SORT.EXE filter that is supplied with MS-DOS (this file must be in the current drive and directory for the demonstration to work correctly).

```
        name    execsort
        title   'EXECSORT --- demonstrate EXEC of filter'
        .sall
;
; EXECSORT.ASM --- demonstration of use of EXEC to run the SORT
; filter as a child process, redirecting its input and output.
; This program requires the files SORT.EXE and MYFILE.DAT in
; the current drive and directory.
;
; Ray Duncan, June 1987
;

stdin   equ     0                       ; standard input
stdout  equ     1                       ; standard output
stderr  equ     2                       ; standard error

stksize equ     128                     ; size of stack

cr      equ     0dh                     ; ASCII carriage return
lf      equ     0ah                     ; ASCII linefeed

jerr    macro   target                  ;; Macro to test carry flag
        local   notset                  ;; and jump if flag set.
        jnc     notset                  ;; Uses JMP DISP16 to avoid
        jmp     target                  ;; branch out of range errors
notset:
        endm


DGROUP  group   _DATA,_STACK            ; 'automatic data group'
```

(more)

Figure 14-8. Assembly-language source code demonstrating use of a filter as a child process. This code redirects the standard input and standard output handles to files, invokes the EXEC function (Interrupt 21H Function 4BH) to run the SORT.EXE program, and then restores the original meaning of the standard input and standard output handles (file EXECSORT.ASM).

```
_TEXT   segment byte public 'CODE'      ; executable code segment

        assume  cs:_TEXT,ds:DGROUP,ss:_STACK


stk_seg dw      ?                       ; original SS contents
stk_ptr dw      ?                       ; original SP contents


main    proc    far                     ; entry point from MS-DOS

        mov     ax,DGROUP               ; set DS = our data segment
        mov     ds,ax

                                        ; now give back extra memory so
                                        ; child SORT has somewhere to run...
        mov     ax,es                   ; let AX = segment of PSP base
        mov     bx,ss                   ; and BX = segment of stack base
        sub     bx,ax                   ; reserve seg stack - seg psp
        add     bx,stksize/16           ; plus paragraphs of stack
        mov     ah,4ah                  ; fxn 4AH = modify memory block
        int     21h                     ; transfer to MS-DOS
        jerr    main1                   ; jump if resize block failed

                                        ; prepare stdin and stdout
                                        ; handles for child SORT process

        mov     bx,stdin                ; dup the handle for stdin
        mov     ah,45h
        int     21h                     ; transfer to MS-DOS
        jerr    main1                   ; jump if dup failed
        mov     oldin,ax                ; save dup'd handle

        mov     dx,offset DGROUP:infile ; now open the input file
        mov     ax,3d00h                ; mode = read-only
        int     21h                     ; transfer to MS-DOS
        jerr    main1                   ; jump if open failed

        mov     bx,ax                   ; force stdin handle to
        mov     cx,stdin                ; track the input file handle
        mov     ah,46h
        int     21h                     ; transfer to MS-DOS
        jerr    main1                   ; jump if force dup failed

        mov     bx,stdout               ; dup the handle for stdout
        mov     ah,45h
        int     21h                     ; transfer to MS-DOS
        jerr    main1                   ; jump if dup failed
        mov     oldout,ax               ; save dup'd handle

        mov     dx,offset dGROUP:outfile ; now create the output file
```

Figure 14-8. Continued.

(more)

```
mov     cx,0                    ; normal attribute
mov     ah,3ch
int     21h                     ; transfer to MS-DOS
jerr    main1                   ; jump if create failed

mov     bx,ax                   ; force stdout handle to
mov     cx,stdout               ; track the output file handle
mov     ah,46h
int     21h                     ; transfer to MS-DOS
jerr    main1                   ; jump if force dup failed

                                ; now EXEC the child SORT,
                                ; which will inherit redirected
                                ; stdin and stdout handles

push    ds                      ; save EXECSORT's data segment
mov     stk_seg,ss              ; save EXECSORT's stack pointer
mov     stk_ptr,sp

mov     ax,ds                   ; set ES = DS
mov     es,ax
mov     dx,offset DGROUP:cname  ; DS:DX = child pathname
mov     bx,offset DGROUP:pars   ; EX:BX = parameter block
mov     ax,4b00h                ; function 4BH, subfunction 00H
int     21h                     ; transfer to MS-DOS

cli                             ; (for bug in some early 8088s)
mov     ss,stk_seg              ; restore execsort's stack pointer
mov     sp,stk_ptr
sti                             ; (for bug in some early 8088s)
pop     ds                      ; restore DS = our data segment

jerr    main1                   ; jump if EXEC failed

mov     bx,oldin                ; restore original meaning of
mov     cx,stdin                ; standard input handle for
mov     ah,46h                  ; this process
int     21h
jerr    main1                   ; jump if force dup failed

mov     bx,oldout               ; restore original meaning
mov     cx,stdout               ; of standard output handle
mov     ah,46h                  ; for this process
int     21h
jerr    main1                   ; jump if force dup failed

mov     bx,oldin                ; close dup'd handle of
mov     ah,3eh                  ; original stdin
int     21h                     ; transfer to MS-DOS
```

Figure 14-8. Continued. *(more)*

```
        jerr    main1                   ; jump if close failed

        mov     bx,oldout               ; close dup'd handle of
        mov     ah,3eh                  ; original stdout
        int     21h                     ; transfer to MS-DOS
        jerr    main1                   ; jump if close failed

                                        ; display success message
        mov     dx,offset DGROUP:msg1   ; address of message
        mov     cx,msg1_len             ; message length
        mov     bx,stdout               ; handle for standard output
        mov     ah,40h                  ; fxn 40H = write file or device
        int     21h                     ; transfer to MS-DOS
        jerr    main1

        mov     ax,4c00h                ; no error, terminate program
        int     21h                     ; with return code = 0

main1:  mov     ax,4c01h                ; error, terminate program
        int     21h                     ; with return code = 1

main    endp                            ; end of main procedure

_TEXT   ends


_DATA   segment para public 'DATA'      ; static & variable data segment

infile  db      'MYFILE.DAT',0          ; input file for SORT filter
outfile db      'MYFILE.SRT',0          ; output file for SORT filter

oldin   dw      ?                       ; dup of old stdin handle
oldout  dw      ?                       ; dup of old stdout handle

cname   db      'SORT.EXE',0            ; pathname of child SORT process

pars    dw      0                       ; segment of environment block
                                        ; (0 = inherit parent's)
        dd      tail                    ; long address, command tail
        dd      -1                      ; long address, default FCB #1
                                        ; (-1 = none supplied)
        dd      -1                      ; long address, default FCB #2
                                        ; (-1 = none supplied)

tail    db      0,cr                    ; empty command tail for child

msg1    db      cr,lf,'SORT was executed as child.',cr,lf
msg1_len equ    $-msg1

_DATA   ends
```

Figure 14-8. Continued.

(more)

```
_STACK  segment para stack 'STACK'

        db      stksize dup (?)

_STACK  ends


        end     main                    ; defines program entry point
```

Figure 14-8. Continued.

The MS-DOS SORT program reads the file MYFILE.DAT via its standard input handle, sorts the file alphabetically, and writes the sorted data to MYFILE.SRT via its standard output handle. When SORT terminates, MS-DOS closes SORT's inherited handles for standard input and standard output, which forces an update of the directory entries for the associated files. The program EXECSORT then resumes execution, restores its own standard input and standard output handles (which are still open) to their original meanings, displays a success message on standard output, and exits to MS-DOS.

Ray Duncan

Article 15
Installable Device Drivers

The software that runs on modern computer systems is, by convention, organized into layers with varied degrees of independence from the underlying computer hardware. The purpose of this layering is threefold:

- To minimize the impact on programs of differences between hardware devices or changes in the hardware.
- To allow the code for common operations to be centralized and optimized.
- To ease the task of moving programs and their data from one machine to another.

The top and most hardware-independent layer is usually the transient, or application, program, which performs a specific job and deals with data in terms of files and records within those files. Such programs are called transient because they are brought into RAM for execution when needed and are discarded from memory when their job is finished. Examples of such programs are Microsoft Word, various programming tools such as the Microsoft Macro Assembler (MASM) and the Microsoft Object Linker (LINK), and even some of the standard MS-DOS utility programs such as CHKDSK and FORMAT.

The middle layer is the operating-system kernel, which manages the allocation of system resources such as memory and disk storage, provides a battery of services to application programs, and implements disk directories and the other housekeeping details of disk storage. The MS-DOS kernel is brought into memory from the file MSDOS.SYS (or IBMDOS.COM with PC-DOS) when the system is turned on or restarted and remains fixed in memory until the system is turned off. The system's default command processor, COMMAND.COM, and system manager programs such as Microsoft Windows bridge the categories of application program and operating system: Parts of them remain resident in memory at all times, but they rely on the MS-DOS kernel for services such as file I/O. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: Components of MS-DOS.

The modules in the lowest layer are called device drivers. These drivers are the components of the operating system that manage the controller, or adapter, of a peripheral device — a piece of hardware that the computer uses for such purposes as storage or communicating with the outside world. Thus, device drivers are responsible for transferring data between a peripheral device and the computer's RAM memory, where other programs can work on it. Drivers shield the operating-system kernel from the need to deal with hardware I/O port addresses, operating characteristics, and the peculiarities of a particular peripheral device, just as the kernel, in turn, shields application programs from the details of file management.

In MS-DOS versions 1.x, device drivers were integrated into the operating system and could be extended or replaced only by patching the files that contained the operating system itself. Because every third-party peripheral manufacturer evolved a different method of modifying these files to get its product to work, conflicts between products from different manufacturers were frequent and expansion of a PC with new disk drives and other devices (especially fixed disks) was often a chancy proposition.

In MS-DOS versions 2.0 and later, there is a clean separation between device drivers and the MS-DOS kernel. Device drivers have a straightforward structure and are interfaced to the kernel through a simple and clearly defined scheme that consists of far calls, function codes, and data packets. Given adequate information about the hardware, a programmer can write a new device driver that follows this structure and interface for almost any conceivable peripheral device; such a driver can subsequently be installed and used without any changes to the underlying operating system.

This article explains the anatomy, operation, and creation of drivers for MS-DOS versions 2.0 and later. Device drivers for versions 1.x are not discussed further here.

Resident and Installable Drivers

Every MS-DOS system contains built-in device drivers for the console (keyboard and video display), the serial port, the parallel printer port, the real-time clock, and at least one disk storage device (the system boot device). These drivers, known as the resident drivers, are loaded as a set from the file IO.SYS (or IBMBIO.COM with PC-DOS) when the system is turned on or restarted.

Drivers for additional peripheral devices occupy individual files on the disk. These drivers, called installable drivers, are loaded and linked into the system during its initialization as a result of DEVICE directives in the CONFIG.SYS file. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: Components of MS-DOS. Examples of such drivers are the ANSI.SYS and RAMDISK.SYS files included with MS-DOS version 3.2. In all other respects, installable drivers have the same structure and relationship to the MS-DOS kernel as the resident drivers. All drivers in the system are chained together so that MS-DOS can rapidly search the entire set to find a specific block or character device when an I/O operation is requested.

Device drivers as a whole are categorized into two groups: block-device drivers and character-device drivers. A driver's membership in one of these two groups determines how the associated device is viewed by MS-DOS and what functions the driver itself must support.

Character-device drivers

Character-device drivers control peripheral devices, such as a terminal or a printer, that perform input and output one character (or byte) at a time. Each character-device driver

ordinarily supports a single hardware unit. The device has a one-character to eight-character logical name that can be used by an application program to "open" the device for input or output as though it were a file. The logical name is strictly a means of identifying the driver to MS-DOS and has no physical equivalent on the device (unlike a volume label for block devices).

The three resident character-device drivers for the console, serial port, and printer carry the logical device names CON, AUX, and PRN, respectively. These three drivers receive special treatment by MS-DOS that allows application programs to address the associated devices in three different ways:

- They can be opened by name for input and output (like any other character device).
- They are supported by special-purpose MS-DOS function calls (Interrupt 21H Functions 01–0CH).
- They are assigned to default handles (standard input, standard output, standard error, standard auxiliary, and standard list) that need not be opened to be used.

See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Character Device Input and Output.

Other character devices can be supported by simply installing additional character-device drivers. The only significant restriction on the total number of devices that can be supported, other than the memory required to hold the drivers, is that each driver must have a unique logical name. When MS-DOS receives an open request for a character device, it searches the chain of device drivers in order from the last driver loaded to the first. Thus, if more than one driver uses the same logical name, the last driver to be loaded supersedes any others and receives all I/O requests addressed to that logical name. This behavior can be used to advantage in some situations. For example, it allows the more powerful ANSI.SYS display driver to supersede the system's default console driver, which does not support cursor positioning and character attributes.

The MS-DOS kernel's buffering and filtering of the characters that pass between it and a character-device driver are affected by whether MS-DOS regards the device to be in cooked mode or raw mode. During cooked mode input, MS-DOS requests characters one at a time from the driver and places them in its own internal buffer, echoing each character to the screen (if the input device is the keyboard) and checking each character for a Control-C (03H) or a Return (0DH). When either the number of characters requested by the application program has been received or a Return is detected, the input is terminated and the data is copied from MS-DOS's internal buffer into the requesting program's buffer. When a Control-C is detected, MS-DOS aborts the input operation and transfers to the routine whose address is stored in the Interrupt 23H (Control-C Handler Address) vector. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Exception Handlers. Similarly, during output in cooked mode, MS-DOS checks between each character for a Control-C pending at the keyboard and aborts the output operation if one is detected.

In raw mode, the exact number of bytes requested by the application program is read or written, without regard to any control characters such as Return or Control-C. MS-DOS passes the entire I/O request to the driver in a single operation, instead of breaking the request into single-character reads or writes, and the characters are transferred directly to or from the requesting program's buffer.

The mode for a specific device can be queried by an application program with the IOCTL Get Device Data function (Interrupt 21H Function 44H Subfunction 00H); the mode can be selected with the Set Device Data function (Interrupt 21H Function 44H Subfunction 01H). *See* SYSTEM CALLS: INTERRUPT 21H: Function 44H. The driver itself is not usually aware of its mode and the mode does not affect its operation.

Block-Device Drivers

Block-device drivers control peripheral devices that transfer data in chunks rather than 1 byte at a time. Block devices are usually randomly addressable devices such as floppy- or fixed-disk drives, but they can also be sequential devices such as magnetic-tape drives. A block driver can support more than one physical unit and can also map two or more logical units onto a single physical unit, as with a partitioned fixed disk.

MS-DOS assigns single-letter drive identifiers (A, B, and so forth) to block devices, instead of logical names. The first letter assigned to a block-device driver is determined solely by the driver's position in the chain of all drivers — that is, by the number of units supported by the block drivers loaded before it; the total number of letters assigned to the driver is determined by the number of logical drive units the driver supports.

MS-DOS does not associate a mode (cooked or raw) with block-device drivers. A block-device driver always reads or writes exactly the number of sectors requested (barring hardware or addressing errors) and never filters or otherwise manipulates the contents of the blocks being transferred.

Structure of an MS-DOS Device Driver

A device driver has three major components (Figure 15-1):

- The device header
- The Strategy routine (*Strat*)
- The Interrupt routine (*Intr*)

The device header

The device header (Figure 15-2) always lies at the beginning of the driver. It contains a link to the next driver in the chain, a word (16 bits) of device attribute flags, offsets to the executable Strategy and Interrupt routines for the device, and the logical device name if it is a character device such as PRN or COM1 or the number of logical units if it is a block device.

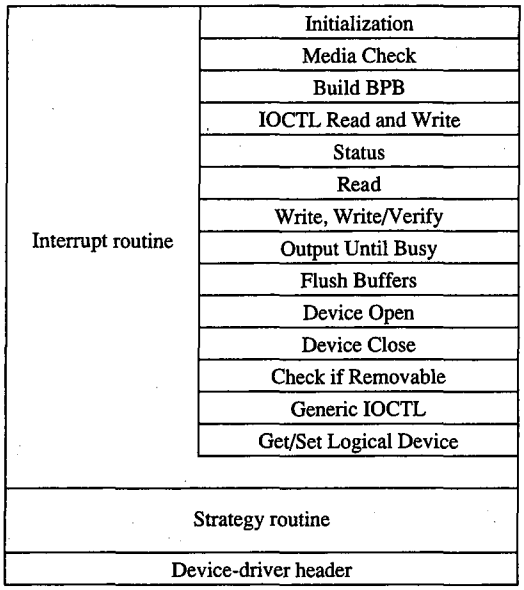


Figure 15-1. General structure of an MS-DOS installable device driver.

Offset	
00H	Link to next driver, offset
02H	Link to next driver, segment
04H	Device attribute word
06H	Offset, Strategy entry point
08H	Offset, Interrupt entry point
0AH	Logical name (8 bytes) if character device or Number of units (1 byte) followed by 7 bytes of reserved space if block device
12H	

Figure 15-2. Device header. The offsets to the Strat *and* Intr *routines are offsets from the same segment used to point to the device header.*

The device attribute flags word (Table 15-1) defines whether a driver controls a character or a block device, which of the optional subfunctions added in MS-DOS versions 3.0 and 3.2 are supported by the driver, and, in the case of block drivers, whether the driver supports IBM-compatible disk media. The least significant 4 bits of the device attribute flags word control whether MS-DOS should use the driver as the standard input, standard output, clock, or NUL device; each of these 4 bits should be set on only one driver in the system at a time.

Table 15-1. Device Attribute Word in Device Header.

Bit	Setting
15*	1 if character device, 0 if block device
14*	1 if IOCTL Read and Write supported
13*	1 if non-IBM format (block device) 1 if Output Until Busy supported (character device)
12	0 (reserved)
11*	1 if Open/Close/Removable Media supported (versions 3.0 and later)
10	0 (reserved)
9	0 (reserved)
8	0 (reserved)
7	0 (reserved)
6*	1 if Generic IOCTL and Get/Set Logical Drive supported (version 3.2)
5	0 (reserved)
4	1 if special fast output function for CON device supported
3	1 if current CLOCK device
2	1 if current NUL device
1	1 if current standard output (*stdout*)
0	1 if current standard input (*stdin*)

* Only bits 6, 11, and 13-15 have significance on block devices; the remainder should be zero.

The information in the device header is ordinarily used only by the MS-DOS kernel and is not available to application programs. However, the IOCTL subfunctions Get and Set Device Data (Interrupt 21H Function 44H Subfunctions 00H and 01H) can be used to inspect or modify some of the bits in the device attribute flags word. Note that there is not a one-to-one correspondence between the bits defined for those functions and the bits in the device header. For example, in the device information word used by the IOCTL subfunctions, bit 7 indicates a block or character device; in the device attribute word of the device header, bit 15 indicates a block or character device.

The Strategy routine (*Strat*)

MS-DOS calls the driver's Strategy routine as the first step of any operation, passing it the segment and offset of a data structure called a request header in registers ES:BX. The Strategy routine saves this pointer for subsequent processing by the Interrupt routine and returns to MS-DOS.

A request header is essentially a small buffer used for private communication between MS-DOS and the device driver. Both MS-DOS and the device driver read and write information in the request header.

The first 13 bytes of a request header are the same for all device-driver functions and are therefore referred to as the static portion of the header. The number and contents of the subsequent bytes vary according to the type of operation being requested by the MS-DOS

kernel (Figure 15-3). The request header's most important component is the command code passed in its third byte; this code selects a driver function such as Read or Write. Other information passed to the driver in the request header includes unit numbers, transfer addresses, and sector or byte counts.

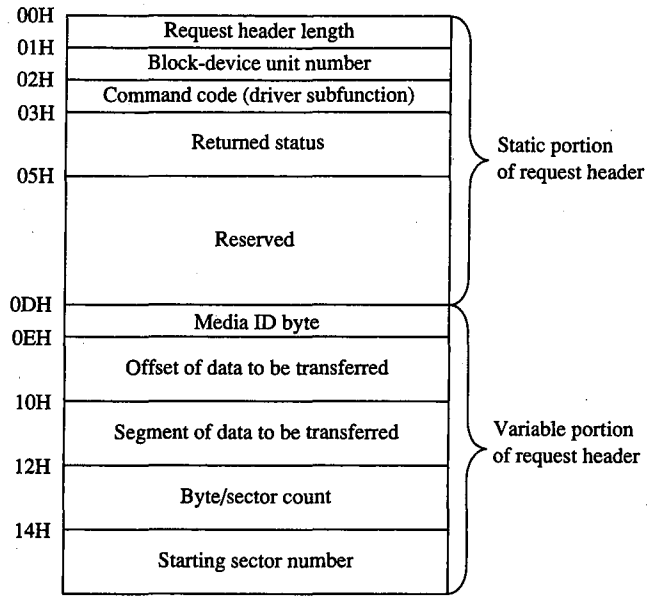


Figure 15-3. A typical driver request header. The bytes following the static portion are the format used for driver Read, Write, Write with Verify, IOCTL Read, and IOCTL Write operations.

The Interrupt routine (*Intr*)

The last and most complex part of a device driver is the Interrupt routine, which is called by MS-DOS immediately after the call to the Strategy routine. The bulk of the Interrupt routine is a collection of functions or subroutines, sometimes called command-code routines, that carry out each of the various operations the MS-DOS kernel requires a driver to support.

When the Interrupt routine receives control from MS-DOS, it saves any affected registers, examines the request header whose address was previously passed in the call to the Strategy routine, determines which command-code routine is needed, and branches to the appropriate function. When the operation is completed, the Interrupt routine stores the status (Table 15-2), error (Table 15-3), and any other applicable information into the request header, restores the previous contents of the affected registers, and returns to the MS-DOS kernel.

Table 15-2. The Request Header Status Word.

Bits	Meaning
15	Error
12-14	Reserved
9	Busy
8	Done
0-7	Error code if bit 15 = 1

Table 15-3. Device-Driver Error Codes.*

Code	Meaning
00H	Write-protect violation
01H	Unknown unit
02H	Drive not ready
03H	Unknown command
04H	CRC error
05H	Bad drive request structure length
06H	Seek error
07H	Unknown media
08H	Sector not found
09H	Printer out of paper
0AH	Write fault
0BH	Read fault
0CH	General failure
0DH	Reserved
0EH	Reserved
0FH	Invalid disk change (versions 3.x)

*Returned in bits 0-7 of the request header status word.

The Interrupt routine's name is misleading in that it is never entered asynchronously as a hardware interrupt. The division of function between the Strategy and Interrupt routines is present for symmetry with UNIX/XENIX and MS OS/2 drivers but is essentially meaningless in single-tasking MS-DOS because there is never more than one I/O request in progress at a time.

The command-code functions

A total of twenty command codes are defined for MS-DOS device drivers. The command codes and the names of their associated Interrupt routines are shown in the following list:

Code	Routine
0	Init (initialization)
1	Media Check (block devices only)
2	Build BIOS Parameter Block (block devices only)
3	IOCTL Read
4	Read (Input)
5	Nondestructive Read (character devices only)
6	Input Status (character devices only)
7	Flush Input Buffers (character devices only)
8	Write (Output)
9	Write with Verify
10	Output Status (character devices only)
11	Flush Output Buffers (character devices only)
12	IOCTL Write
13*	Device Open
14*	Device Close
15*	Removable Media (block devices only)
16*	Output Until Busy (character devices only)
19†	Generic IOCTL Request
23†	Get Logical Device (block devices only)
24†	Set Logical Device (block devices only)

*MS-DOS versions 3.0 and later
†MS-DOS version 3.2

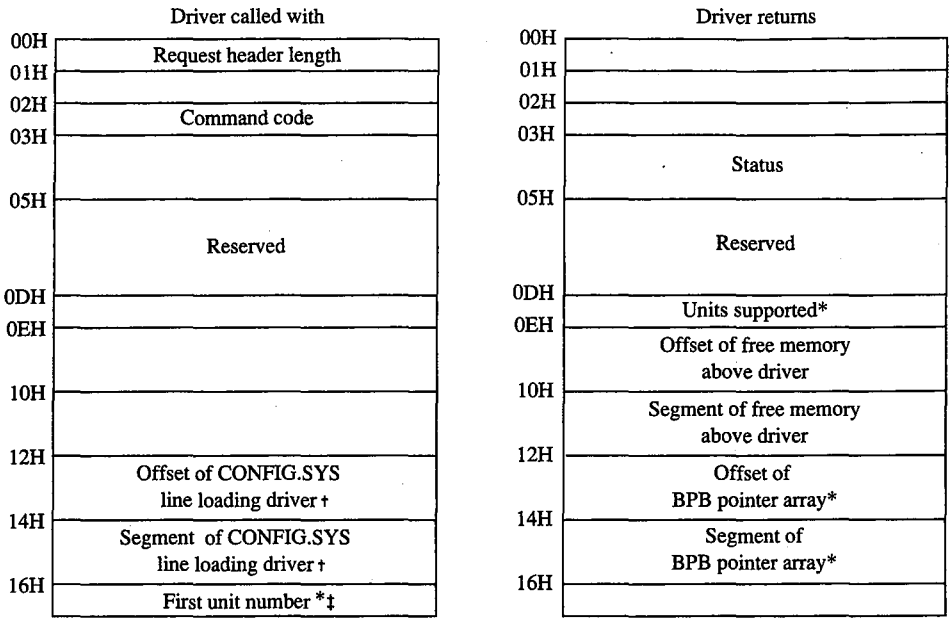
Functions 0 through 12 must be supported by a driver's Interrupt section under all versions of MS-DOS. Drivers tailored for versions 3.0 and 3.1 can optionally support an additional 4 functions defined under those versions of the operating system and drivers designed for version 3.2 can support 3 more, for a total of 20. MS-DOS inspects the bits in the device attribute word of the device header to determine which of the optional version 3.x functions a driver supports, if any.

As noted in the list above, some of the functions are relevant only for character drivers, some only for block drivers, and some for both. In any case, there must be an executable routine present for each function, even if the routine does nothing but set the done flag in the status word of the request header. The general requirements for each function routine are described below.

The Init function

The Init (initialization) function (command code 0) for a driver is called only once, when the driver is loaded (Figure 15-4). Init is responsible for checking that the hardware device controlled by the driver is present and functional, performing any necessary hardware initialization (such as a reset on a printer or a seek to the home track on a disk device), and capturing any interrupt vectors that the driver will need later.

The Init function is passed a pointer in the request header to the text of the DEVICE line in CONFIG.SYS that caused the driver to be loaded—specifically, the address of the next byte after the equal sign (=). The line is read-only and is terminated by a linefeed or carriage-return character; it can be scanned by the driver for switches or other parameters that might influence the driver's operation. (Alphabetic characters in the line are folded to uppercase.) With versions 3.0 and later, block drivers are also passed the drive number that will be assigned to their first unit (0 = A, 1 = B, and so on).



* Block-device drivers only
† Points to the character after DEVICE=
‡ MS-DOS 3.0 and later only

Figure 15-4. Initialization request header (command code 0).

When it returns to the kernel, the Init function must set the done flag in the status word of the request header and return the address of the start of free memory after the driver (sometimes called the break address). This address tells the kernel where it can build certain control structures of its own associated with the driver and then load the next driver. The Init routine of a block-device driver must also return the number of logical units supported by the driver and the address of a BPB pointer array.

The number of units returned by a block driver is used to assign device identifiers. For example, if at the time the driver is loaded there are already drivers present for four block devices (drive codes 0–3, corresponding to drive identifiers A through D) and the driver being initialized supports four units, it will be assigned the drive numbers 4 through 7

(corresponding to the drive names E through H). (Although there is also a field in the device header for the number of units, it is not inspected by MS-DOS; rather, it is set by MS-DOS from the information returned by the Init function.)

The BPB pointer array is an array of word offsets to BIOS parameter blocks. *See* The Build BIOS Parameter Block Function below; PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: MS-DOS Storage Devices. The array must contain one entry for each unit defined by the driver, although all entries can point to the same BPB to conserve memory. During the operating-system boot sequence, MS-DOS scans all the BPBs defined by all the units in all the resident block-device drivers to determine the largest sector size that exists on any device in the system; this information is used to set MS-DOS's cache buffer size. Thus, the sector size in the BPB of any installable block driver must be no larger than the largest sector size used by the resident block drivers.

If the Init routine finds that its hardware device is missing or defective, it can bypass the installation of the driver completely by returning the following values in the request header:

Item	Value
Number of units	0
Address of free memory	Segment and offset of the driver's own device header

A character-device driver must also clear bit 15 of the device attribute word in the device header so that MS-DOS will load the next driver in the same location as the one that just terminated itself.

The operating-system services that can be invoked by the Init routine are very limited. Only MS-DOS Interrupt 21H Functions 01–0CH (various character input and output services), 25H (Set Interrupt Vector), 30H (Get MS-DOS Version Number), and 35H (Get Interrupt Vector) can be called by the Init code. These functions assist the driver in configuring itself for the version of the host operating system it is to run under, capturing vectors for hardware interrupts, and displaying informational or error messages.

The amount of RAM required by a device driver can be reduced by positioning the Init routine at the end of the driver and returning that routine's starting address as the location of the first free memory.

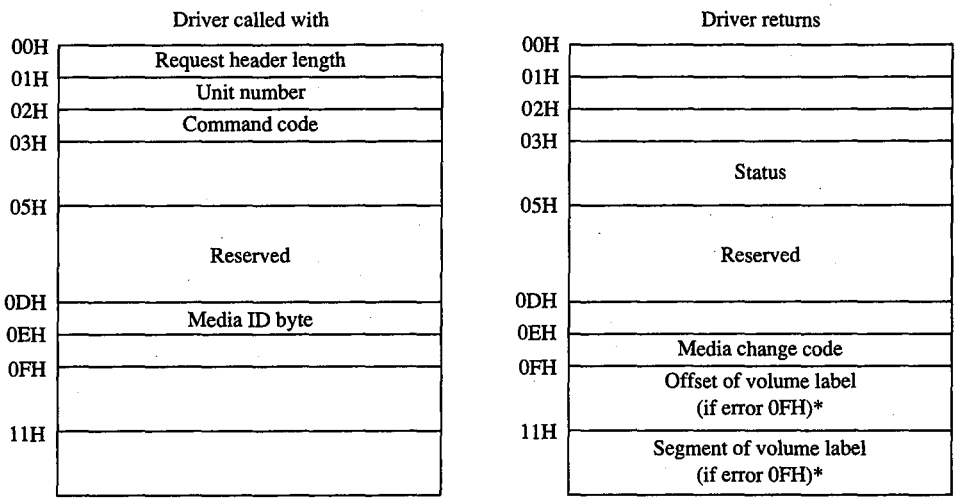
The Media Check function

The Media Check function (command code 1) is used only in block-device drivers. It is called by the MS-DOS kernel when there is a pending drive access call other than a simple file read or write (for example, a file open, close, rename, or delete), passing the media ID byte (Figure 15-5) for the disk that MS-DOS assumes is in the drive:

Description	Medium
0F9H	5.25-inch double-sided, 15 sectors
0FCH	5.25-inch single-sided, 9 sectors
0FDH	5.25-inch double-sided, 9 sectors
0FEH	5.25-inch single-sided, 8 sectors
0FFH	5.25-inch double-sided, 8 sectors
0F9H	3.5-inch double-sided, 9 sectors
0F0H	3.5-inch double-sided, 18 sectors
0F8H	Fixed disk

The function returns a code indicating whether the medium has been changed since the last transfer:

Code	Meaning
−1	Medium changed
0	Don't know if medium changed
1	Medium not changed



* MS-DOS 3.0 and later only

Figure 15-5. Media Check request header (command code 1).

If the Media Check routine asserts that the disk has not been changed, MS-DOS bypasses rereading the FAT and proceeds with the disk access. If the returned code indicates that the disk has been changed, MS-DOS invalidates all buffers associated with the drive, including buffers containing data waiting to be written (this data is simply lost), performs a Build BPB call, and then reads the disk's FAT and directory.

The action taken by MS-DOS when *Don't know* is returned depends on the state of its internal buffers. If data that needs to be written out is present in the buffers associated with the drive, MS-DOS assumes that no disk change has occurred. If the buffers are empty or have all been previously flushed to the disk, MS-DOS assumes that the disk was changed and proceeds as described above for the *Medium changed* return code.

If bit 11 of the device attribute word is set (that is, the driver supports the optional Open/Close/Removable Media functions), the host system is MS-DOS version 3.0 or later, and the function returns the *Medium changed* code (–1), the function must also return the segment and offset of the ASCIIZ volume label for the previous disk in the drive. (If the driver does not have the volume label, it can return a pointer to the ASCIIZ string *NO NAME.*) If MS-DOS determines that the disk was changed with unwritten data still present in the buffers, it issues a critical error 0FH (Invalid Disk Change). Application programs can trap this critical error and prompt the user to replace the original disk.

In character-device drivers, the Media Change function should simply set the done flag in the status word of the request header and return.

The Build BIOS Parameter Block function

The Build BPB function (command code 2) is supported only on block devices. MS-DOS calls this function when the *Medium changed* code has been returned by the Media Check routine or when the *Don't know* code has been returned and there are no dirty buffers (buffers that have not yet been written to disk). Thus, a call to this function indicates that the disk has been legally changed.

The Build BPB call receives a pointer to a one-sector buffer in the request header (Figure 15-6). If the non-IBM-format bit (bit 13) in the device attribute word in the device header is zero, the buffer contains the first sector of the disk's FAT, with the media ID byte in the first byte of the buffer. In this case, the contents of the buffer should not be modified by the driver. However, if the non-IBM-format bit is set, the buffer can be used by the driver as scratch space.

The Build BPB function must return the segment and offset of a BIOS parameter block (Table 15-4) for the disk format indicated by the media ID byte and set the done flag in the status word of the request header. The information in the BPB is used by the kernel to interpret the disk structure and is also used by the driver itself to translate logical sector addresses into physical track, sector, and head addresses. If bit 11 of the device attribute word is set (that is, the driver supports the optional Open/Close/Removable Media functions) and the host system is MS-DOS version 3.0 or later, this routine should also read the volume label from the disk and save it.

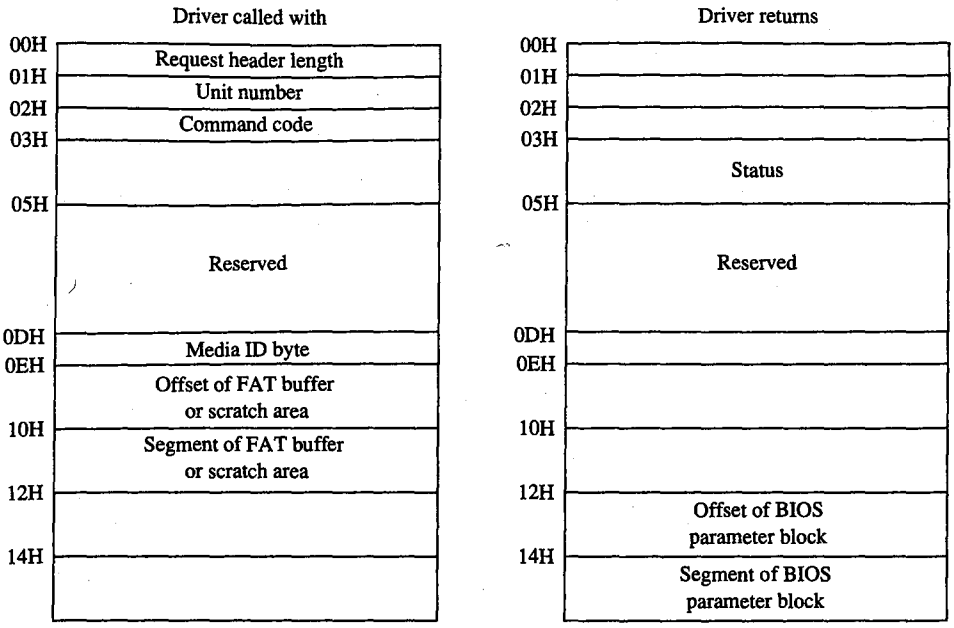


Figure 15-6. Build BPB request header (command code 2).

Table 15-4. Format of a BIOS Parameter Block (BPB).

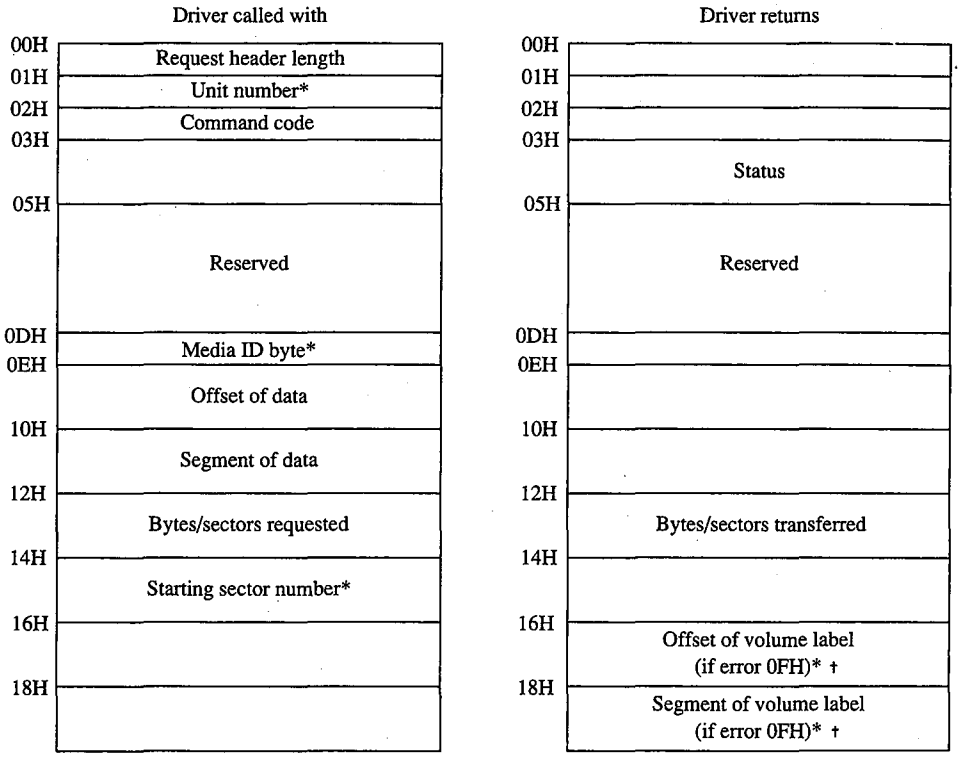
Bytes	Contents
00–01H	Bytes per sector
02H	Sectors per allocation unit (must be power of 2)
03–04H	Number of reserved sectors (starting at sector 0)
05H	Number of file allocation tables (FATs)
06–07H	Maximum number of root-directory entries
08–09H	Total number of sectors in medium
0AH	Media ID byte
0B–0CH	Number of sectors occupied by a single FAT
0D–0EH	Sectors per track (versions 3.0 and later)
0F–10H	Number of heads (versions 3.0 and later)
11–12H	Number of hidden sectors (versions 3.0 and later)
13–14H	High-order word of number of hidden sectors (version 3.2)
15–18H	If bytes 8–9 are zero, total number of sectors in medium (version 3.2)

In character-device drivers, the Build BPB function should simply set the done flag in the status word of the request header and return.

The Read, Write, and Write with Verify functions

The Read (Input) function (command code 4) transfers data from the device into a specified memory buffer. The Write (Output) function (command code 8) transfers data from a specified memory buffer to the device. The Write with Verify function (command code 9) works like the Write function but, if feasible, also performs a read-after-write verification that the data was transferred correctly. The MS-DOS kernel calls the Write with Verify function, instead of the Write function, whenever the system's global verify flag has been turned on with the VERIFY command or with Interrupt 21H Function 2EH (Set Verify Flag).

All three of these driver functions are called by the MS-DOS kernel with the address and length of the buffer for the data to be transferred. In the case of block-device drivers, the kernel also passes the drive unit code, the starting logical sector number, and the media ID byte for the disk (Figure 15-7).



* Block-device drivers only

† MS-DOS 3.0 and later, command codes 4, 8, and 9 only

Figure 15-7. The request header for IOCTL Read (command code 3), Read (command code 4), Write (command code 8), Write with Verify (command code 9), IOCTL Write (command code 12), and Output Until Busy (command code 16).

The Read and Write functions must perform the requested I/O, first translating each logical sector number for a block device into a physical track, head, and sector with the aid of the BIOS parameter block. Then the functions must return the number of bytes or sectors actually transferred in the appropriate field of the request header and also set the done flag in the request header status word. If an error is encountered during an operation, the functions must set the done flag, the error flag, and the error type in the status word and also report the number of bytes or sectors successfully transferred before the error; it is not sufficient to simply report the error.

Under MS-DOS versions 3.0 and later, the Read and Write functions can optionally use the reference count of open files maintained by the driver's Device Open and Device Close functions, together with the media ID byte, to determine whether the medium has been illegally changed. If the medium was changed with files open, the driver can return the error code 0FH and the segment and offset of the volume label for the correct disk so that the user can be prompted to replace the disk.

The Nondestructive Read function

The Nondestructive Read function (command code 5) is supported only on character devices. It allows MS-DOS to look ahead in the character stream by one character and is used to check for Control-C characters pending at the keyboard.

The function is called by the kernel with no parameters other than the command code itself (Figure 15-8). It must set the done bit in the status word of the request header and also set the busy bit in the status word to reflect whether the device's input buffer is empty (busy bit = 1) or contains at least one character (busy bit = 0). If the latter, the function must also return the next character that would be obtained by a kernel call to the Read function, without removing that character from the buffer (hence the term nondestructive).

In block-device drivers, the Nondestructive Read function should simply set the done flag in the status word of the request header and return.

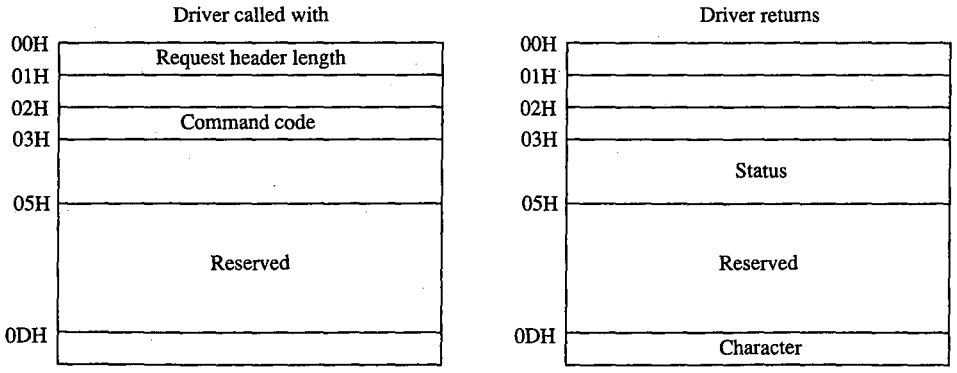


Figure 15-8. The Nondestructive Read request header.

The Input Status and Output Status functions

The Input Status and Output Status functions (command codes 6 and 10) are defined only for character devices. They are called with no parameters in the request header other than the command code itself and return their results in the busy bit of the request header status word (Figure 15-9). These functions constitute the driver-level support for the services the MS-DOS kernel provides to application programs by means of Interrupt 21H Function 44H Subfunctions 06H and 07H (Check Input Status and Check Output Status).

MS-DOS calls the Input Status function to determine whether there are characters waiting in a type-ahead buffer. The function sets the done bit in the status word of the request header and sets the busy bit to 0 if at least one character is already in the input buffer or to 1 if no characters are in the buffer and a read request would wait on a character from the physical device. If the character device does not have a type-ahead buffer, the Input Status routine should always return the busy bit set to 0 so that MS-DOS will not wait for something to arrive in the buffer before calling the Read function.

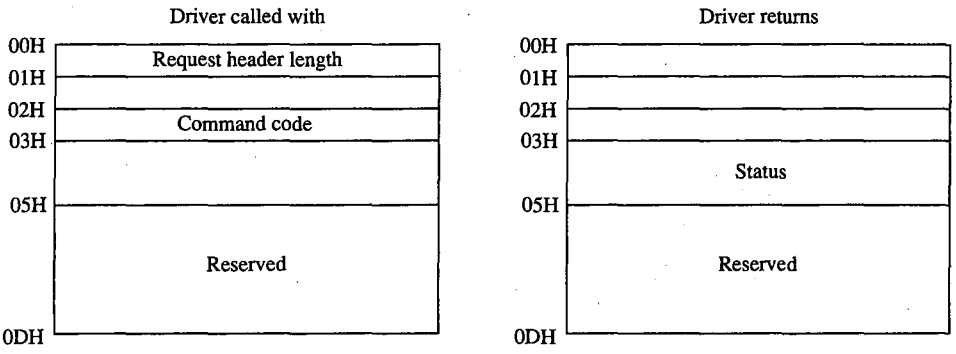


Figure 15-9. The request header for Input Status (command code 6), Flush Input Buffers (command code 7), Output Status (command code 10), and Flush Output Buffers (command code 11).

MS-DOS uses the Output Status function to determine whether a write operation is already in progress for the device. The function must set the done bit and the busy bit (0 if the device is idle and a write request would start immediately; 1 if a write is already in progress and a new write request would be delayed) in the status word of the request header.

In block-device drivers, the Input Status and Output Status functions should simply set the done flag in the status word of the request header and return.

The Flush Input Buffer and Flush Output Buffer functions

The Flush Input Buffer and Flush Output Buffer functions (command codes 7 and 11) are defined only for character devices. They simply terminate any read (for Flush Input) or write (for Flush Output) operations that are in progress and empty the associated buffer. The Flush Input Buffer function is used by MS-DOS to discard characters waiting in the type-ahead queue. This driver action corresponds to the MS-DOS service provided to application programs by means of Interrupt 21H Function 0CH (Flush Buffer, Read Keyboard).

These functions are called with no parameters in the request header other than the command code itself (*see* Figure 15-9) and return only the status word.

In block-device drivers, the Flush Buffer functions have no meaning. They should simply set the done flag in the status word of the request header and return.

The IOCTL Read and IOCTL Write functions

The IOCTL (I/O Control) Read and IOCTL Write functions (command codes 3 and 12) allow control information to be passed directly between a device driver and an application program. The IOCTL Read and Write driver functions are called by the MS-DOS kernel only if the IOCTL flag (bit 14) is set in the device attribute word of the device header.

The MS-DOS kernel passes the address and length of the buffer that contains or will receive the IOCTL information (*see* Figure 15-7). The driver must return the actual count of bytes transferred and set the done flag in the request header status word. Any error code returned by the driver is ignored by the kernel.

IOCTL Read and IOCTL Write operations are typically used to configure a driver or device or to report driver or device status and do not usually result in the transfer of data to or from the physical device. These functions constitute the driver support for the services provided to application programs by the MS-DOS kernel through Interrupt 21H Function 44H Subfunctions 02H, 03H, 04H, and 05H (Receive Control Data from Character Device, Send Control Data to Character Device, Receive Control Data from Block Device, and Send Control Data to Block Device).

The Device Open and Device Close functions

The Device Open and Device Close functions (command codes 13 and 14) are supported only in MS-DOS versions 3.0 and later and are called only if the open/close/removable media flag (bit 11) is set in the device attribute word of the device header. The Device Open and Device Close functions have no parameters in the request header other than the unit code for block devices and return nothing except the done flag and, if applicable, the error flag and number in the request header status word (Figure 15-10).

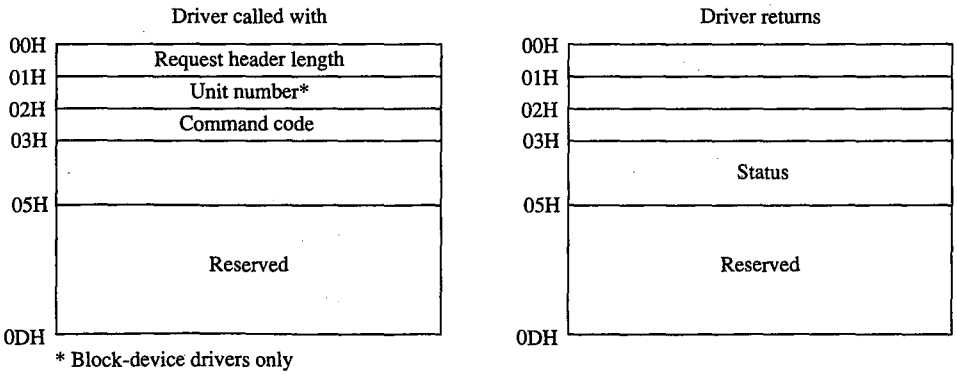


Figure 15-10. The request header for Device Open (command code 13), Device Close (command code 14), and Removable Media (command code 15).

Each Interrupt 21H request by an application to open or create a file or to open a character device for input or output results in a Device Open call by the kernel to the corresponding device driver. Similarly, each Interrupt 21H call by an application to close a file or device results in a Device Close call by the kernel to the appropriate device driver. These Device Open and Device Close calls are in addition to any directory read or write calls that may be necessary.

On block devices, the Device Open and Device Close functions can be used to manage local buffering and to maintain a reference count of the number of open files on a device. Whenever this reference count is decremented to zero, all files on the disk have been closed and the driver should flush any internal buffers so that data is not lost, as the user may be about to change disks. The reference count can also be used together with the media ID byte by the Read and Write functions to determine whether the disk has been changed while files are still open.

The reference count should be forced to zero when a Media Check call that returns the *Medium changed* code is followed by a Build BPB call, to provide for those programs that use FCBs to open files and then never close them. This problem does not arise with programs that use the handle functions for file management, because all handles are always closed automatically by MS-DOS on behalf of the program when it terminates. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management.

On character devices, the Device Open and Device Close functions can be used to send hardware-dependent initialization and post-I/O strings to the associated device (for example, a reset sequence or formfeed character to precede new output and a formfeed to follow it). Although these strings can be written directly by an application using ordinary write function calls, they can also be previously passed to the driver by application programs with IOCTL Write calls (Interrupt 21H Function 44H Subfunction 05H), which in turn are translated by the MS-DOS kernel into driver command code 12 (IOCTL Write) requests. The latter method makes the driver responsible for sending the proper control strings to the device each time a Device Open or Device Close is executed, but this method can be used only with drivers specifically written to support it.

The Removable Media function

The Removable Media function (command code 15) is defined only for block devices. It is supported in MS-DOS versions 3.0 and later and is called by MS-DOS only if the open/close/removable media flag (bit 11) is set in the device attribute word of the device header. This function constitutes the driver-level support for the service provided to application programs by MS-DOS by means of Interrupt 21H Function 44H Subfunction 08H (Check If Block Device Is Removable).

The only parameter for the Removable Media function is the unit code (*see* Figure 15-10). The function sets the done bit in the request header status word and sets the busy bit to 1 if the disk is not removable or to 0 if the disk is removable. This information can be used by MS-DOS to optimize its accesses to the disk and to eliminate unnecessary FAT and directory reads.

In character-device drivers, the Removable Media function should simply set the done flag in the status word of the request header and return.

The Output Until Busy function

The Output Until Busy function (command code 16) is defined only for character devices under MS-DOS versions 3.0 and later and is called by the MS-DOS kernel only if the corresponding flag (bit 13) is set in the device attribute word of the device header. This function is an optional driver-optimization function included specifically for the benefit of background print spoolers driving printers that have internal memory buffers. Such printers can accept data at a rapid rate until the buffer is full.

The Output Until Busy function is called with the address and length of the data to be written to the device (*see* Figure 15-7). It transfers data continuously to the device until the device indicates that it is busy or until the data is exhausted. The function then must set the done flag in the request header status word and return the actual number of bytes transferred in the appropriate field of the request header.

For this function to return a count of bytes transferred that is less than the number of bytes requested is not an error. MS-DOS will adjust the address and length of the data passed in the next Output Until Busy function request so that all characters are sent.

In block-device drivers, the Output Until Busy function should simply set the done flag in the status word of the request header and return.

The Generic IOCTL function

The Generic IOCTL function (command code 19) is defined under MS-DOS version 3.2 and is called only if the 3.2-functions-supported flag (bit 6) is set in the device attribute word of the device header. This driver function corresponds to the MS-DOS generic IOCTL service supplied to application programs by means of Interrupt 21H Function 44H Subfunctions 0CH (Generic I/O Control for Handles) and 0DH (Generic I/O Control for Block Devices).

In addition to the usual information in the static portion of the request header, the Generic IOCTL function is passed a category (major) code, a function (minor) code, the contents of the SI and DI registers at the point of the IOCTL call, and the segment and offset of a data buffer (Figure 15-11). This buffer in turn contains other information whose format depends on the major and minor IOCTL codes passed in the request header. The driver must interpret the major and minor codes in the request header and the contents of the additional buffer to determine which operation it will carry out and then set the done flag in the request header status word and return any other applicable information in the request header or the data buffer.

Services that can be invoked by the Generic IOCTL function, if the driver supports them, include configuring the driver for nonstandard disk formats, reading and writing entire disk tracks of data, and formatting and verifying tracks. The Generic IOCTL function has been designed to be open-ended so that it can be used to easily extend the device driver definition in future versions of MS-DOS.

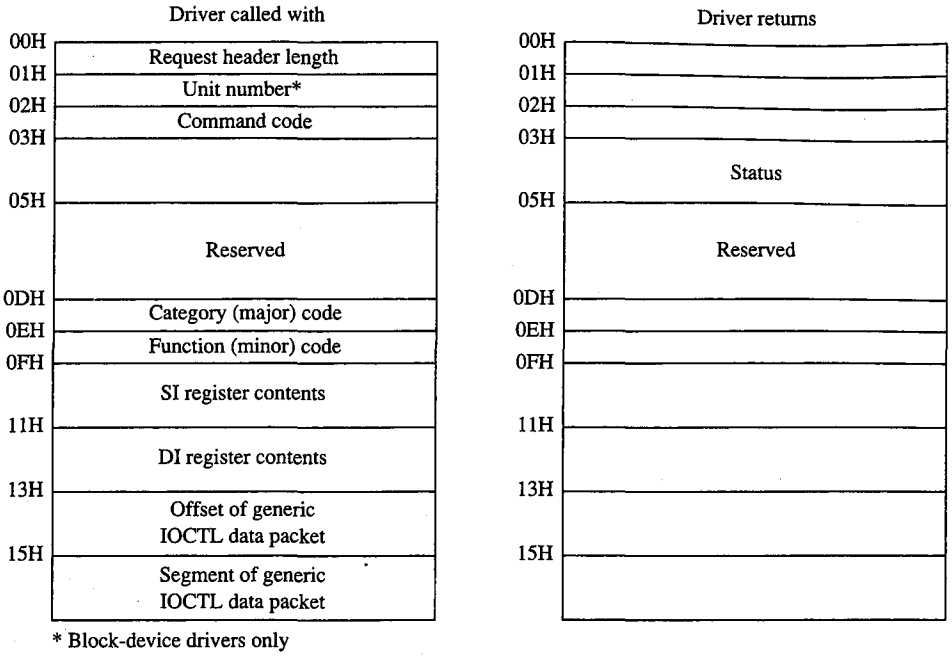


Figure 15-11. Generic IOCTL request header.

The Get Logical Device and Set Logical Device functions

The Get and Set Logical Device functions (command codes 23 and 24) are defined only for block devices under MS-DOS version 3.2 and are called only if the 3.2-functions-supported flag (bit 6) is set in the device attribute word of the device header. They correspond to the Get and Set Logical Drive Map services supplied by MS-DOS to application programs by means of Interrupt 21H Function 44H Subfunctions 0EH and 0FH.

The Get and Set Logical Device functions are called with a drive unit number in the request header (Figure 15-12). Both functions return a status word for the operation in the request header; the Get Logical Device function also returns a unit number.

The Get Logical Device function is called to determine whether more than one drive letter is assigned to the same physical device. It returns a code for the last drive letter used to reference the device (1 = A, 2 = B, and so on); if only one drive letter is assigned to the device, the returned unit code should be 0.

The Set Logical Device function is called to inform the driver of the next logical drive identifier that will be used to reference the device. The unit code passed by the MS-DOS kernel in this case is zero based relative to the logical drives supported by this particular driver. For example, if the driver supports two logical floppy-disk-drive units (A and B), only one physical disk drive exists in the system, and Set Logical Device is called with a unit number of 1, the driver is being informed that the next read or write request from the MS-DOS kernel will be directed to drive B.

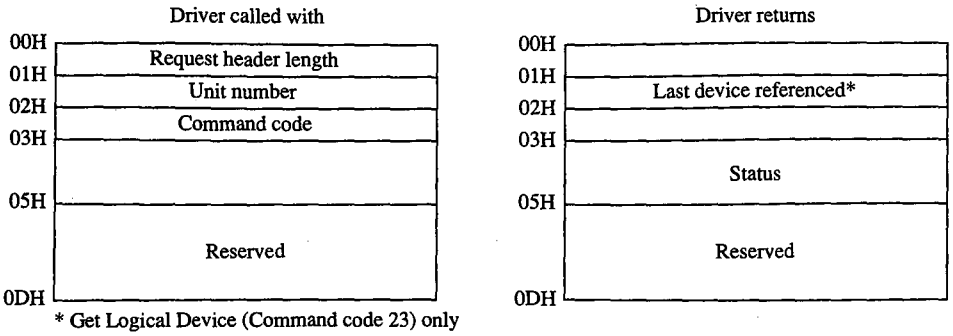


Figure 15-12. Get Logical Device and Set Logical Device request header.

In character-device drivers, the Get Logical Device and Set Logical Device functions should simply set the done flag in the status word of the request header and return.

The Processing of a Typical I/O Request

An application program requests an I/O operation from MS-DOS by loading registers with the appropriate values and addresses and executing a software Interrupt 21H. MS-DOS inspects its internal tables, searches the chain of device headers if necessary, and determines which device driver should receive the I/O request.

MS-DOS then creates a request header data packet in a reserved area of memory. Disk I/O requests are transformed from file and record information into logical sector requests by MS-DOS's interpretation of the disk directory and file allocation table. (MS-DOS locates these disk structures using the information returned by the driver from a previous Build BPB call and issues additional driver read requests, if necessary, to bring their sectors into memory.)

After the request header is prepared, MS-DOS calls the device driver's Strategy entry point, passing the address of the request header in registers ES:BX. The Strategy routine saves the address of the request header and performs a far return to MS-DOS.

MS-DOS then immediately calls the device driver's Interrupt entry point. The Interrupt routine saves all registers, retrieves the address of the request header that was saved by the Strategy routine, extracts the command code, and branches to the appropriate function to perform the operation requested by MS-DOS. When the requested function is complete, the Interrupt routine sets the done flag in the status word and places any other required information into the request header, restores all registers to their state at entry, and performs a far return.

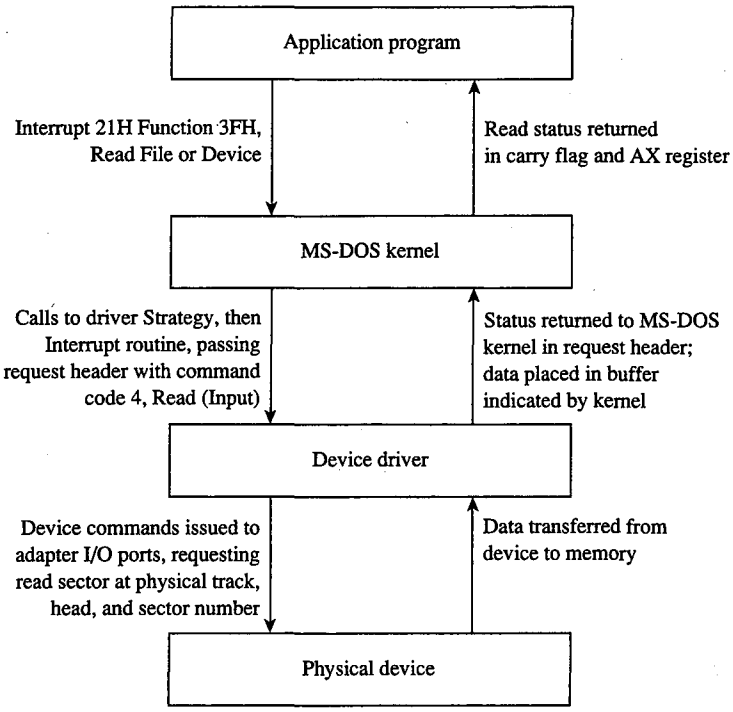


Figure 15-13. The processing of a typical I/O request from an application program.

MS-DOS translates the driver's returned status into the appropriate carry flag status, register values, and (possibly) error code for the MS-DOS Interrupt 21H function that was requested and returns control to the application program. Figure 15-13 sketches this entire flow of control and data.

Note that a single Interrupt 21H function request by an application program can result in many operation requests by MS-DOS to the device driver. For example, if the application invokes Interrupt 21H Function 3DH (Open File with Handle) to open a file, MS-DOS may have to issue multiple sector read requests to the driver while searching the directory for the filename. Similarly, an application program's request to write a string to the screen in cooked mode with Interrupt 21H Function 40H (Write File or Device) will result in a write request to the driver for each character in the string, because MS-DOS filters the characters and polls the keyboard for a pending Control-C between each character output.

Writing Device Drivers

Device drivers are traditionally coded in assembly language, both because of the rigid structural requirements and because of the need to keep driver execution speed high and memory overhead low. Although MS-DOS versions 3.0 and later are capable of loading

drivers in .EXE format, versions 2.x can load only pure memory-image device drivers that do not require relocation. Therefore, drivers are typically written as though they were .COM programs with an "origin" of zero and converted with EXE2BIN to .BIN or .SYS files so that they will be compatible with any version of MS-DOS (2.0 or later). *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program.

The device header must be located at the beginning of the file (offset 0). Both words in the header's link field should be set to –1, thus allowing MS-DOS to fix up the link field when the driver is loaded during system initialization so that it points to the next driver in the chain. When a single file contains more than one driver, the offset portion of each header link field should point to the next header in that file, all using the same segment base of zero, and only the link field of the last header in the file should be set to –1, –1.

The device attribute word must reflect the device-driver type (character or block) and the bits that indicate support for the various optional command codes must have appropriate values. The device header's offsets to the Strategy and Interrupt routines must be relative to the same segment base as the device header itself. If the driver is for a character device, the name field should be filled in properly with the device's logical name, which can be any legal eight-character uppercase filename padded with spaces and without a colon. Duplication of existing character-device names or existing disk-file names should be avoided (unless a resident character-device driver is being intentionally superseded).

The Strategy and Interrupt routines for the device are called by MS-DOS by means of an intersegment call (CALL FAR) and must return to MS-DOS with a far return. Both routines must preserve all CPU registers and flags. The MS-DOS kernel's stack has room for 40 to 50 bytes when the driver is called; if the driver makes heavy use of the stack, it should switch to an internal stack of adequate depth.

The Strategy routine is, of course, very simple. It need only save the address of the request header that is passed to it in registers ES:BX and exit back to the kernel.

The logic of the Interrupt routine is necessarily more complex. It must save the CPU registers and flags, extract the command code from the request header whose address was previously saved by the Strategy routine, and dispatch the appropriate command-code function. When that function is finished, the Interrupt routine must ensure that the appropriate status and other information is placed in the request header, restore the CPU registers and flags, and return control to the kernel.

Although the interface between the MS-DOS kernel and the command-code routines is fairly simple, it is also strict. The command-code functions must behave exactly as they are defined or the system will behave erratically. Even a very subtle discrepancy in the action of a driver function can have unexpectedly large global effects. For example, if a block driver Read function returns an error but does not return a correct value for the number of sectors successfully transferred, the MS-DOS kernel will be misled in its attempts to retry the read for only the failing sectors and disk data might be corrupted.

Example character driver: TEMPLATE

Figure 15-14 contains the source code for a skeleton character-device driver called TEMPLATE.ASM. This driver does nothing except display a sign-on message when it is loaded, but it demonstrates all the essential driver components, including the device header, Strategy routine, and Interrupt routine. The command-code functions take no action other than to set the done flag in the request header status word.

```
        name    template
        title   'TEMPLATE --- installable driver template'

;
; TEMPLATE.ASM:  A program skeleton for an installable
;                device driver (MS-DOS 2.0 or later)
;
; The driver command-code routines are stubs only and have
; no effect but to return a nonerror "Done" status.
;
; Ray Duncan, July 1987
;

_TEXT   segment byte public 'CODE'

        assume  cs:_TEXT,ds:_TEXT,es:NOTHING

        org     0

MaxCmd  equ     24              ; maximum allowed command code
                                ; 12 for MS-DOS 2.x
                                ; 16 for MS-DOS 3.0-3.1
                                ; 24 for MS-DOS 3.2-3.3

cr      equ     0dh             ; ASCII carriage return
lf      equ     0ah             ; ASCII linefeed
eom     equ     '$'             ; end-of-message signal


Header:                         ; device driver header
        dd      -1              ; link to next device driver
        dw      0c840h          ; device attribute word
        dw      Strat           ; "Strategy" routine entry point
        dw      Intr            ; "Interrupt" routine entry point
        db      'TEMPLATE'      ; logical device name


RHPtr   dd      ?               ; pointer to request header, passed
                                ; by MS-DOS kernel to Strategy routine
```

Figure 15-14. TEMPLATE.ASM, the source file for the TEMPLATE.SYS driver. *(more)*

```
Dispatch:                       ; Interrupt routine command-code
                                ; dispatch table
        dw      Init            ;  0 = initialize driver
        dw      MediaChk        ;  1 = media check on block device
        dw      BuildBPB        ;  2 = build BIOS parameter block
        dw      IoctlRd         ;  3 = I/O control read
        dw      Read            ;  4 = read (input) from device
        dw      NdRead          ;  5 = nondestructive read
        dw      InpStat         ;  6 = return current input status
        dw      InpFlush        ;  7 = flush device input buffers
        dw      Write           ;  8 = write (output) to device
        dw      WriteVfy        ;  9 = write with verify
        dw      OutStat         ; 10 = return current output status
        dw      OutFlush        ; 11 = flush output buffers
        dw      IoctlWt         ; 12 = I/O control write
        dw      DevOpen         ; 13 = device open        (MS-DOS 3.0+)
        dw      DevClose        ; 14 = device close       (MS-DOS 3.0+)
        dw      RemMedia        ; 15 = removable media    (MS-DOS 3.0+)
        dw      OutBusy         ; 16 = output until busy  (MS-DOS 3.0+)
        dw      Error           ; 17 = not used
        dw      Error           ; 18 = not used
        dw      GenIOCTL        ; 19 = generic IOCTL      (MS-DOS 3.2+)
        dw      Error           ; 20 = not used
        dw      Error           ; 21 = not used
        dw      Error           ; 22 = not used
        dw      GetLogDev       ; 23 = get logical device (MS-DOS 3.2+)
        dw      SetLogDev       ; 24 = set logical device (MS-DOS 3.2+)


Strat   proc    far             ; device driver Strategy routine,
                                ; called by MS-DOS kernel with
                                ; ES:BX = address of request header

                                ; save pointer to request header
        mov     word ptr cs:[RHPtr],bx
        mov     word ptr cs:[RHPtr+2],es

        ret                     ; back to MS-DOS kernel

Strat   endp


Intr    proc    far             ; device driver Interrupt routine,
                                ; called by MS-DOS kernel immediately
                                ; after call to Strategy routine

        push    ax              ; save general registers
        push    bx
        push    cx
        push    dx
        push    ds
```

Figure 15-14. Continued.

(more)

```
        push    es
        push    di
        push    si
        push    bp

        push    cs              ; make local data addressable
        pop     ds              ; by setting DS = CS

        les     di,[RHPtr]      ; let ES:DI = request header

                                ; get BX = command code
        mov     bl,es:[di+2]
        xor     bh,bh
        cmp     bx,MaxCmd       ; make sure it's valid
        jle     Intr1           ; jump, function code is ok
        call    Error           ; set error bit, "Unknown Command" code
        jmp     Intr2

Intr1:  shl     bx,1            ; form index to dispatch table
                                ; and branch to command-code routine
        call    word ptr [bx+Dispatch]

        les     di,[RHPtr]      ; ES:DI = address of request header

Intr2:  or      ax,0100h        ; merge Done bit into status and
        mov     es:[di+3],ax    ; store status into request header

        pop     bp              ; restore general registers
        pop     si
        pop     di
        pop     es
        pop     ds
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret                     ; return to MS-DOS kernel


; Command-code routines are called by the Interrupt routine
; via the dispatch table with ES:DI pointing to the request
; header. Each routine should return AX = 00H if function was
; completed successfully or AX = 8000H + error code if
; function failed.


MediaChk proc   near            ; function 1 = Media Check

        xor     ax,ax
        ret

MediaChk endp
```

Figure 15-14. Continued.

(more)

```
BuildBPB proc   near            ; function 2 = Build BPB

        xor     ax,ax
        ret

BuildBPB endp


IoctlRd proc    near            ; function 3 = I/O Control Read

        xor     ax,ax
        ret

IoctlRd endp


Read    proc    near            ; function 4 = Read (Input)

        xor     ax,ax
        ret

Read    endp


NdRead  proc    near            ; function 5 = Nondestructive Read

        xor     ax,ax
        ret

NdRead  endp


InpStat proc    near            ; function 6 = Input Status

        xor     ax,ax
        ret

InpStat endp


InpFlush proc   near            ; function 7 = Flush Input Buffers

        xor     ax,ax
        ret

InpFlush endp
```

Figure 15-14. Continued.

(more)

```
Write   proc    near            ; function 8 = Write (Output)

        xor     ax,ax
        ret

Write   endp


WriteVfy proc   near            ; function 9 = Write with Verify

        xor     ax,ax
        ret

WriteVfy endp


OutStat proc    near            ; function 10 = Output Status

        xor     ax,ax
        ret

OutStat endp


OutFlush proc   near            ; function 11 = Flush Output Buffers

        xor     ax,ax
        ret

OutFlush endp


IoctlWt proc    near            ; function 12 = I/O Control Write

        xor     ax,ax
        ret

IoctlWt endp


DevOpen proc    near            ; function 13 = Device Open

        xor     ax,ax
        ret

DevOpen endp
```

Figure 15-14. Continued.

(more)

```
DevClose proc   near            ; function 14 = Device Close

        xor     ax,ax
        ret

DevClose endp


RemMedia proc   near            ; function 15 = Removable Media

        xor     ax,ax
        ret

RemMedia endp


OutBusy proc    near            ; function 16 = Output Until Busy

        xor     ax,ax
        ret

OutBusy endp


GenIOCTL proc   near            ; function 19 = Generic IOCTL

        xor     ax,ax
        ret

GenIOCTL endp


GetLogDev proc  near            ; function 23 = Get Logical Device

        xor     ax,ax
        ret

GetLogDev endp


SetLogDev proc  near            ; function 24 = Set Logical Device

        xor     ax,ax
        ret

SetLogDev endp
```

Figure 15-14. Continued.

(more)

```
Error   proc    near            ; bad command code in request header

        mov     ax,8003h        ; error bit + "Unknown Command" code
        ret

Error   endp


Init    proc    near            ; function 0 = initialize driver

        push    es              ; save address of request header
        push    di

        mov     ah,9            ; display driver sign-on message
        mov     dx,offset Ident
        int     21h

        pop     di              ; restore request header address
        pop     es

                                ; set address of free memory
                                ; above driver (break address)
        mov     word ptr es:[di+14],offset Init
        mov     word ptr es:[di+16],cs

        xor     ax,ax           ; return status
        ret

Init    endp

Ident   db      cr,lf,lf
        db      'TEMPLATE Example Device Driver'
        db      cr,lf,eom

Intr    endp


_TEXT   ends

        end
```

Figure 15-14. Continued.

TEMPLATE.ASM can be assembled, linked, and converted into a loadable driver with the following commands:

```
C>MASM TEMPLATE;  <Enter>
C>LINK TEMPLATE;  <Enter>
C>EXE2BIN TEMPLATE.EXE TEMPLATE.SYS  <Enter>
```

The Microsoft Object Linker (LINK) will display the warning message *No Stack Segment*; this message can be ignored. The driver can then be installed by adding the line

```
DEVICE=TEMPLATE.SYS
```

to the CONFIG.SYS file and restarting the system. The fact that the TEMPLATE.SYS driver also has the logical character-device name TEMPLATE allows the demonstration of an interesting MS-DOS effect: After the driver is installed, the file that contains it can no longer be copied, renamed, or deleted. The reason for this limitation is that MS-DOS always searches its list of character-device names first when an open request is issued, before it inspects the disk directory. The only way to erase the TEMPLATE.SYS file is to modify the CONFIG.SYS file to remove the associated DEVICE statement and then restart the system.

For a complete example of a character-device driver for interrupt-driven serial communications, *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Interrupt-Driven Communications.

Example block driver: TINYDISK

Figure 15-15 contains the source code for a simple 64 KB virtual disk (RAMdisk) called TINYDISK.ASM. This code provides a working example of a simple block-device driver. When its Initialization routine is called by the kernel, TINYDISK allocates itself 64 KB of RAM and maps a disk structure onto the RAM in the form of a boot sector containing a valid BPB, a FAT, a root directory, and a files area. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: MS-DOS Storage Devices.

```
        name    tinydisk
        title   TINYDISK example block-device driver

; TINYDISK.ASM --- 64 KB RAMdisk
;
; Ray Duncan, July 1987
; Example of a simple installable block-device driver.

_TEXT   segment public 'CODE'

        assume  cs:_TEXT,ds:_TEXT,es:_TEXT

        org     0

MaxCmd  equ     12              ; max driver command code
                                ; (no MS-DOS 3.x functions)

cr      equ     0dh             ; ASCII carriage return
lf      equ     0ah             ; ASCII linefeed
blank   equ     020h            ; ASCII space code
eom     equ     '$'             ; end-of-message signal

Secsize equ     512             ; bytes/sector, IBM-compatible media
```

Figure 15-15. TINYDISK.ASM, the source file for the TINYDISK.SYS driver. *(more)*

```
                                ; device-driver header
Header  dd      -1              ; link to next driver in chain
        dw      0               ; device attribute word
        dw      Strat           ; "Strategy" routine entry point
        dw      Intr            ; "Interrupt" routine entry point
        db      1               ; number of units, this device
        db      7 dup (0)       ; reserved area (block-device drivers)

RHPtr   dd      ?               ; segment:offset of request header

Secseg  dw      ?               ; segment base of sector storage

Xfrsec  dw      0               ; current sector for transfer
Xfrcnt  dw      0               ; sectors successfully transferred
Xfrreq  dw      0               ; number of sectors requested
Xfraddr dd      0               ; working address for transfer

Array   dw      BPB             ; array of pointers to BPB
                                ; for each supported unit


Bootrec equ     $

        jmp     $               ; phony JMP at start of
        nop                     ; boot sector; this field
                                ; must be 3 bytes

        db      'MS   2.0'      ; OEM identity field

                                ; BIOS Parameter Block (BPB)
BPB     dw      Secsize         ; 00H - bytes per sector
        db      1               ; 02H - sectors per cluster
        dw      1               ; 03H - reserved sectors
        db      1               ; 05H - number of FATs
        dw      32              ; 06H - root directory entries
        dw      128             ; 08H - sectors = 64 KB/secsize
        db      0f8h            ; 0AH - media descriptor
        dw      1               ; 0BH - sectors per FAT

Bootrec_len equ $-Bootrec


Strat   proc    far             ; RAMdisk strategy routine

                                ; save address of request header
        mov     word ptr cs:RHPtr,bx
        mov     word ptr cs:[RHPtr+2],es
        ret                     ; back to MS-DOS kernel

Strat   endp
```

Figure 15-15. Continued.

(more)

```
Intr    proc    far             ; RAMdisk interrupt routine

        push    ax              ; save general registers
        push    bx
        push    cx
        push    dx
        push    ds
        push    es
        push    di
        push    si
        push    bp

        mov     ax,cs           ; make local data addressable
        mov     ds,ax

        les     di,[RHPtr]      ; ES:DI = request header

        mov     bl,es:[di+2]    ; get command code
        xor     bh,bh
        cmp     bx,MaxCmd       ; make sure it's valid
        jle     Intr1           ; jump, function code is ok
        mov     ax,8003h        ; set Error bit and
        jmp     Intr3           ; "Unknown Command" error code

Intr1:  shl     bx,1            ; form index to dispatch table and
                                ; branch to command-code routine
        call    word ptr [bx+Dispatch]
                                ; should return AX = status

        les     di,[RHPtr]      ; restore ES:DI = request header

Intr3:  or      ax,0100h        ; merge Done bit into status and store
        mov     es:[di+3],ax    ; status into request header

Intr4:  pop     bp              ; restore general registers
        pop     si
        pop     di
        pop     es
        pop     ds
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret                     ; return to MS-DOS kernel

Intr    endp
```

Figure 15-15. Continued. *(more)*

```
Dispatch:                       ; command-code dispatch table
                                ; all command-code routines are
                                ; entered with ES:DI pointing
                                ; to request header and return
                                ; the operation status in AX
        dw      Init            ;  0 = initialize driver
        dw      MediaChk        ;  1 = media check on block device
        dw      BuildBPB        ;  2 = build BIOS parameter block
        dw      Dummy           ;  3 = I/O control read
        dw      Read            ;  4 = read (input) from device
        dw      Dummy           ;  5 = nondestructive read
        dw      Dummy           ;  6 = return current input status
        dw      Dummy           ;  7 = flush device input buffers
        dw      Write           ;  8 = write (output) to device
        dw      Write           ;  9 = write with verify
        dw      Dummy           ; 10 = return current output status
        dw      Dummy           ; 11 = flush output buffers
        dw      Dummy           ; 12 = I/O control write


MediaChk proc  near             ; command code 1 = Media Check

                                ; return "not changed" code
        mov     byte ptr es:[di+0eh],1
        xor     ax,ax           ; and success status
        ret

MediaChk endp


BuildBPB proc  near             ; command code 2 = Build BPB

                                ; put BPB address in request header
        mov     word ptr es:[di+12h],offset BPB
        mov     word ptr es:[di+14h],cs
        xor     ax,ax           ; return success status
        ret

BuildBPB endp


Read    proc    near            ; command code 4 = Read (Input)

        call    Setup           ; set up transfer variables

Read1:  mov     ax,Xfrcnt       ; done with all sectors yet?
        cmp     ax,Xfrreq
        je      Read2           ; jump if transfer completed
        mov     ax,Xfrsec       ; get next sector number
        call    Mapsec          ; and map it
```

Figure 15-15. Continued.

(more)

```
        mov     ax,es
        mov     si,di
        les     di,Xfraddr      ; ES:DI = requester's buffer
        mov     ds,ax           ; DS:SI = RAMdisk address
        mov     cx,Secsize      ; transfer logical sector from
        cld                     ; RAMdisk to requestor
        rep movsb
        push    cs              ; restore local addressing
        pop     ds
        inc     Xfrsec          ; advance sector number
                                ; advance transfer address
        add     word ptr Xfraddr,Secsize
        inc     Xfrcnt          ; count sectors transferred
        jmp     Read1

Read2:                          ; all sectors transferred
        xor     ax,ax           ; return success status
        les     di,RHPtr        ; put actual transfer count
        mov     bx,Xfrcnt       ; into request header
        mov     es:[di+12h],bx
        ret

Read    endp


Write   proc    near            ; command code 8 = Write (Output)
                                ; command code 9 = Write with Verify

        call    Setup           ; set up transfer variables

Write1: mov     ax,Xfrcnt       ; done with all sectors yet?
        cmp     ax,Xfrreq
        je      Write2          ; jump if transfer completed

        mov     ax,Xfrsec       ; get next sector number
        call    Mapsec          ; and map it
        lds     si,Xfraddr
        mov     cx,Secsize      ; transfer logical sector from
        cld                     ; requester to RAMdisk
        rep movsb
        push    cs              ; restore local addressing
        pop     ds
        inc     Xfrsec          ; advance sector number
                                ; advance transfer address
        add     word ptr Xfraddr,Secsize
        inc     Xfrcnt          ; count sectors transferred
        jmp     Write1

Write2:                         ; all sectors transferred
        xor     ax,ax           ; return success status
        les     di,RHPtr        ; put actual transfer count
```

Figure 15-15. Continued. *(more)*

```
        mov     bx,Xfrcnt       ; into request header
        mov     es:[di+12h],bx
        ret

Write   endp

Dummy   proc    near            ; called for unsupported functions

        xor     ax,ax           ; return success flag for all
        ret

Dummy   endp

Mapsec  proc    near            ; map sector number to memory address
                                ; call with AX = logical sector no.
                                ; return ES:DI = memory address

        mov     di,Secsize/16   ; paragraphs per sector
        mul     di              ; * logical sector number
        add     ax,Secseg       ; + segment base of sector storage
        mov     es,ax
        xor     di,di           ; now ES:DI points to sector
        ret

Mapsec  endp

Setup   proc    near            ; set up for read or write
                                ; call ES:DI = request header
                                ; extracts address, start, count

        push    es              ; save request header address
        push    di
        mov     ax,es:[di+14h]  ; starting sector number
        mov     Xfrsec,ax
        mov     ax,es:[di+12h]  ; sectors requested
        mov     Xfrreq,ax
        les     di,es:[di+0eh]  ; requester's buffer address
        mov     word ptr Xfraddr,di
        mov     word ptr Xfraddr+2,es
        mov     Xfrcnt,0        ; initialize sectors transferred count
        pop     di              ; restore request header address
        pop     es
        ret

Setup   endp
```

Figure 15-15. Continued. *(more)*

```
Init    proc    near            ; command code 0 = Initialize driver
                                ; on entry ES:DI = request header

        mov     ax,cs           ; calculate segment base for sector
        add     ax,Driver_len   ; storage and save it
        mov     Secseg,ax
        add     ax,1000h        ; add 1000H paras (64 KB) and
        mov     es:[di+10h],ax  ; set address of free memory
        mov     word ptr es:[di+0eh],0

        call    Format          ; format the RAMdisk

        call    Signon          ; display driver identification

        les     di,cs:RHPtr     ; restore ES:DI = request header
                                ; set logical units = 1
        mov     byte ptr es:[di+0dh],1
                                ; set address of BPB array
        mov     word ptr es:[di+12h],offset Array
        mov     word ptr es:[di+14h],cs

        xor     ax,ax           ; return success status
        ret

Init    endp


Format  proc    near            ; format the RAMdisk area

        mov     es,Secseg       ; first zero out RAMdisk
        xor     di,di
        mov     cx,8000h        ; 32 K words = 64 KB
        xor     ax,ax
        cld
        rep stosw

        mov     ax,0            ; get address of logical
        call    Mapsec          ; sector zero
        mov     si,offset Bootrec
        mov     cx,Bootrec_len
        rep movsb               ; and copy boot record to it

        mov     ax,word ptr BPB+3
        call    Mapsec          ; get address of 1st FAT sector
        mov     al,byte ptr BPB+0ah
        mov     es:[di],al      ; put media ID byte into it
        mov     word ptr es:[di+1],-1

        mov     ax,word ptr BPB+3
        add     ax,word ptr BPB+0bh
        call    Mapsec          ; get address of 1st directory sector
```

Figure 15-15. Continued. *(more)*

```
        mov     si,offset Volname
        mov     cx,Volname_len
        rep movsb               ; copy volume label to it

        ret                     ; done with formatting

Format  endp


Signon  proc    near            ; driver identification message

        les     di,RHPtr        ; let ES:DI = request header
        mov     al,es:[di+22]   ; get drive code from header,
        add     al,'A'          ; convert it to ASCII, and
        mov     drive,al        ; store into sign-on message

        mov     ah,30h          ; get MS-DOS version
        int     21h
        cmp     al,2
        ja      Signon1         ; jump if version 3.0 or later
        mov     Ident1,eom      ; version 2.x, don't print drive

Signon1:                        ; print sign-on message
        mov     ah,09H          ; Function 09H = print string
        mov     dx,offset Ident ; DS:DX = address of message
        int     21h             ; transfer to MS-DOS

        ret                     ; back to caller

Signon  endp


Ident   db      cr,lf,lf        ; driver sign-on message
        db      'TINYDISK 64 KB RAMdisk'
        db      cr,lf
Ident1  db      'RAMdisk will be drive '
Drive   db      'X:'
        db      cr,lf,eom


Volname db      'DOSREF_DISK'   ; volume label for RAMdisk
        db      08h             ; attribute byte
        db      10 dup (0)      ; reserved area
        dw      0               ; time = 00:00
        dw      0f01h           ; date = August 1, 1987
        db      6 dup (0)       ; reserved area

Volname_len equ $-volname

Driver_len dw (($-header)/16)+1 ; driver size in paragraphs

_TEXT   ends

        end
```

Figure 15-15. Continued.

Subsequent driver Read and Write calls by the kernel to TINYDISK function as though they were transferring sectors to and from a physical storage device but actually only copy data from one area in memory to another. A programmer can learn a great deal about the operation of block-device drivers and MS-DOS's relationship to those drivers (such as the order and frequency of Media Change, Build BPB, Read, Write, and Write With Verify calls) by inserting software probes into TINYDISK at appropriate locations and monitoring its behavior.

TINYDISK.ASM can be assembled, linked, and converted into a loadable driver with the following commands:

```
C>MASM TINYDISK;  <Enter>
C>LINK TINYDISK;  <Enter>
C>EXE2BIN TINYDISK.EXE TINYDISK.SYS  <Enter>
```

The linker will display the warning message *No Stack Segment*; this message can be ignored. The driver can then be installed by adding the line

```
DEVICE=TINYDISK.SYS
```

to the CONFIG.SYS file and restarting the system. When it is loaded, TINYDISK displays a sign-on message and the drive letter that it was assigned if it is running under MS-DOS version 3.0 or later. (If the host system is MS-DOS version 2.x, this information is not provided to the driver.) Files can then be copied to the RAMdisk as though it were a small but extremely fast disk drive.

Ray Duncan

Part D
Directions of MS-DOS

Article 16
Writing Applications for Upward Compatibility

One of the major concerns of the designers of Microsoft OS/2 was that it be backwardly compatible — that is, that programs written to run under MS-DOS versions 2 and 3 be able to run on MS OS/2. A major concern for present application programmers is that their programs run not only on current versions of MS-DOS (and MS OS/2) but also on future versions of MS-DOS. Ensuring such upward compatibility involves both hardware issues and operating-system issues.

Hardware Issues

A basic requirement for ensuring upward compatibility is hardware-independent code. If you bypass system services and directly program the hardware — such as the system interrupt controller, the system clock, and the enhanced graphics adapter (EGA) registers — your application will not run on future versions of MS-DOS.

Protected mode compatibility

The 80286 and the 80386 microprocessors can operate in two incompatible modes: real mode and protected mode. When either chip is operating in real mode, it is perceived by the operating system and programs as a fast 8088 chip. Applications written for the 8086 and 8088 run the same on the 80286 and the 80386 — only faster. They cannot, however, take advantage of 80286 and 80386 features unless they can run in protected mode.

Following the guidelines below will minimize the work necessary to convert a real mode program to protected mode and will also allow a program to use a special subset of the MS OS/2 Applications Program Interface (API) — Family API. A binary program (.EXE) that uses the family API can run in either protected mode or real mode under MS OS/2 and subsequent systems, but it can run only in real mode under MS-DOS version 3.

Family API

The Family API requires that the application use a subset of the MS OS/2 Dynamic Link System API. Special tools link the application with a special library that implements the subset MS OS/2 system services in the MS-DOS version 3 environment. Many of these services are implemented by calling the appropriate Interrupt 21H subfunction; some are implemented in the special library itself.

When a Family API application is loaded under MS OS/2 protected mode, MS OS/2 ignores the special library code and loads only the application itself. MS OS/2 then provides the requested services in the normal fashion. However, MS-DOS version 3 loads the entire package — the application and the special library — because the Family API .EXE file is constructed to look like an MS-DOS 3 .EXE file.

Linear *vs* segmented memory

The protected mode and the real mode of the 80286 and the 80386 are compatible except in the area of segmentation. The 8086 has been described as a segmented machine, but it is actually a linear memory machine with offset registers. When a memory address is generated, the value in one of the "segment" registers is multiplied by 16 and added as a displacement to the offset value supplied by the instruction's addressing mode. No length information is associated with each "segment"; the "segment" register supplies only a 20-bit addressing offset. Programs routinely use this by computing a 20-bit address and then decomposing it into a 16-bit "segment" value and a 16-bit displacement value so that the address can be referenced.

The protected mode of the 80286 and the 80386, however, is truly segmented. A value placed in a segment register selects an entry from a descriptor table; that entry contains the addressing offset, a segment length, and permission bits. On the 8086, the so-called segment component of an address is multiplied by 16 and added to the offset component, producing a 20-bit physical address. Thus, if you take an address in the *segment:offset* form, add 4 to the segment value, and subtract 64 (that is, 4*16) from the offset value, the new address references exactly the same location as the old address. On the 80286 and the 80386 in protected mode, however, segment values, called segment selectors, have no direct correspondence to physical addresses. In other words, in 8086 mode, the two address forms

$1000_{16}:0345_{16}$

and

$1004_{16}:0305_{16}$

reference the same memory location, but in protected mode these two forms reference totally different locations.

Creating segment values

This architectural difference gives rise to the most common cause of incompatibility — the program performs addressing arithmetic to compute "segment" values. Any program that uses the 20-bit addressing scheme to create or to compute a value to be loaded in a segment register cannot be converted to run in protected mode. To be protected mode compatible, a program must treat the 8086's so-called segments as true segments.

To create a program that does this, write according to the following guidelines:

1. Do not generate any segment values. Use only the segment values supplied by MS-DOS calls and those placed in the segment registers when MS-DOS loaded your program. The exception is "huge objects" — memory objects larger than 64 KB. In

this case, MS OS/2 provides a base segment number and a "segment offset value." The returned segment number selects the first 64 KB of the object and the segment number, plus the segment offset value address the second 64 KB of the object. Likewise, the returned segment value plus N*(segment offset value) selects the N+1 64 KB piece of the huge object. Write real mode code in this same fashion, using 4096 as the segment offset value. When you convert your program, you can substitute the value provided by MS OS/2.

2. Do not address beyond the allocated length of a segment.
3. Do not use segment registers as scratch registers by placing general data in them. Place only valid segment values, supplied by MS-DOS, in a segment register. The one exception is that you can place a zero value in a segment register, perhaps to indicate "no address." You can place the zero in the segment register, but you cannot reference memory using that register; you can only load/store or push/pop it.
4. Do not use CS: overrides on instructions that store into memory. It is impossible to store into a code segment in protected mode.

CPU speed

Because various microprocessors and machine configurations execute at different speeds, a program should not contain timing loops that depend on CPU speed. Specifically, a program should not establish CPU speed during initialization and then use that value for timing loops because the preemptive scheduling of MS OS/2 and future operating systems can "take away" the CPU at any time for arbitrary and unpredictable lengths of time. (In any case, time should not be wasted in a timing loop when other processes could be using system resources.)

Program timing

Programs must measure the passage of time carefully. They can use the system clock-tick interrupt while directly interfacing with the user, but no clock ticks will be seen by real mode programs when the user switches the screen interface to another program.

It is recommended that applications use the time-of-day system interface to determine elapsed time. To facilitate conversion to MS OS/2 protected mode, programs should encapsulate time-of-day or elapsed-time functions into subroutines.

BIOS

Avoid BIOS interrupt interfaces except for Interrupt 10H (the screen display functions) and Interrupt 16H (the keyboard functions). Interrupt 10H functions are contained in the MS OS/2 VIO package, and Interrupt 16H functions are in the MS OS/2 KBD package. Other BIOS interrupts provide functions that are available under MS OS/2 only in considerably modified forms.

Special operations

Uncommon, or special, operations and instructions can produce varied results, depending on the microprocessor. For example, when a "divide by 0" trap is taken on an 8086, the stack frame points to the instruction after the fault; when such action is taken on the 80286 and 80386, the return address points to the instruction that caused the fault. The effect of

pushing the SP register is different between the 80286 and the 80386 as well. *See* Appendix M: 8086/8088 Software Compatibility Issues. Write your program to avoid these problem areas.

Operating-System Issues

Basic to writing programs that will run on future operating systems is writing code that is not version specific. Incorporating special version-specific features in a program will virtually ensure that the program will be incompatible with future versions of MS-DOS and MS OS/2.

Following the guidelines below will not necessarily ensure your program's compatibility, but it will facilitate converting the program or using the Family API to produce a dual-mode binary program.

Filenames

MS-DOS versions 2 and 3 silently truncate a filename that is longer than eight characters or an extension that is longer than three characters. MS-DOS generates no error message when performing this task. In real mode, MS OS/2 also silently truncates a filename or extension that exceeds the maximum length; in protected mode, however, it does not. Therefore, a real mode application program needs to perform this truncating function. The program should check the length of the filenames that it generates or that it obtains from a user and refuse names that are longer than the eight-character maximum. This prevents improperly formatted names from becoming embedded in data and control files—a situation that could cause a protected mode version of the application to fail when it presents that invalid name to the operating system.

When you convert your program to protected mode API, remove the length-checking code; MS OS/2 will check the length and return an error code as appropriate. Future file systems will support longer filenames, so it's important that protected mode programs simply present filenames to the operating system, which is then responsible for judging their validity.

Other MS-DOS version 2 and 3 elements have fixed lengths, including the current directory path. To be upwardly compatible, your program should accept whatever length is provided by the user or returned from a system call and rely on MS OS/2 to return an error message if a length is inappropriate. The exception is filename length in real mode non-Family API programs: These programs should enforce the eight-character maximum because MS-DOS versions 2 and 3 fail to do so.

File truncation

Files are truncated by means of a zero-length write under MS-DOS versions 2 and 3; under MS OS/2 in protected mode, files are truncated with a special API. File truncation operations should be encapsulated in a special routine to facilitate conversion to MS OS/2 protected mode or the Family API.

File searches

MS-DOS versions 2 and 3 never close file-system searches (Find First File/Find Next File). The returned search contains the information necessary for MS-DOS to continue the search later, and if the search is never continued, no harm is done.

MS OS/2, however, retains the necessary search continuation information in an internal structure of limited size. For this reason, your program should not depend on more than about 10 simultaneous searches and it should be able to close searches when it is done. If your program needs to perform more than about 10 searches simultaneously, it should be able to close a search, restart it later, and advance to the place where the program left off, rather than depending on MS OS/2 to continue the search.

MS OS/2 further provides a Find Close function that releases the internal search information. Protected mode programs should use this call at the end of every search sequence. Because MS-DOS versions 2 and 3 have no such call, your program should call a dummy procedure by this name at the appropriate locations. Then you can convert your program to the protected mode API or to the Family API without reexamining your algorithms.

Note: Receiving a "No more files" return code from a search does not implicitly close the search; all search closes must be explicit.

The Family API allows only a single search at a time. To circumvent this restriction, code two different Find Next File routines in your program — one for MS OS/2 protected mode and one for MS-DOS real mode — and use the Family API function that determines the program's current environment to select the routine to execute.

MS-DOS calls

A program that uses only the Interrupt 21H functions listed below is guaranteed to work in the Compatibility Box of MS OS/2 and will be relatively easy to modify for MS OS/2 protected mode.

Function	Name
0DH	Disk Reset
0EH	Select Disk
19H	Get Current Disk
1AH	Set DTA Address
25H	Set Interrupt Vector
2AH	Get Date
2BH	Set Date
2CH	Get Time
2EH	Set/Reset Verify Flag
2FH	Get DTA Address

(more)

Function	Name
30H	Get MS-DOS Version Number
33H	Get/Set Control-C Check Flag
35H	Get Interrupt Vector
36H	Get Disk Free Space
38H	Get/Set Current Country
39H	Create Directory
3AH	Remove Directory
3BH	Change Current Directory
3CH	Create File with Handle
3DH	Open File with Handle
3EH	Close File
3FH	Read File or Device
40H	Write File or Device
41H	Delete File
42H	Move File Pointer
43H	Get/Set File Attributes
44H	IOCTL (all subfunctions)
45H	Duplicate File Handle
46H	Force Duplicate File Handle
47H	Get Current Directory
48H	Allocate Memory Block
49H	Free Memory Block
4AH	Resize Memory Block
4BH	Load and Execute Program (EXEC)
4CH	Terminate Process with Return Code
4DH	Get Return Code of Child Process
4EH	Find First File
4FH	Find Next File
54H	Get Verify Flag
56H	Rename File
57H	Get/Set Date/Time of File
59H	Get Extended Error Information
5AH	Create Temporary File
5BH	Create New File
5CH	Lock/Unlock File Region

FCBs

FCBs are not supported in MS OS/2 protected mode. Use handle-based calls instead.

Interrupt calls

MS-DOS versions 2 and 3 use an interrupt-based interface; MS OS/2 protected mode uses a procedure-call interface. Write your code to accommodate this difference by encapsulating the interrupt-based interfaces into individual subroutines that can then easily be modified to use the MS OS/2 procedure-call interface.

System call register usage

The MS OS/2 procedure-call interface preserves all registers except AX and FLAGS. Write your program to assume that the contents of AX and the contents of any register modified by MS-DOS version 2 and 3 interrupt interfaces are destroyed at each system call, regardless of the success or failure of that call.

Flush/Commit calls

Your program should issue Flush/Commit calls where necessary—for example, after writing out the user's work file—but no more than necessary. Because MS OS/2 is multitasking, the floppy disk that contains the files to be flushed may not be in the drive. In such a case, MS OS/2 prompts the user to insert the proper floppy disk. As a result, too frequent flushes could generate a great many *Insert disk* messages and degrade the system's usability.

Seeks

Seeks to negative offsets and to devices also create compatibility issues.

To negative offsets

Your program should not attempt to seek to a negative file location. A negative seek offset is permissible as long as the sum of the seek offset and the current file position is positive. MS-DOS versions 2 and 3 allow seeking to a negative offset as long as you do not attempt to read or write the file at that offset. MS OS/2 and subsequent systems return an error code for negative net offsets.

On devices

Your program should not issue seeks to devices (such as AUX, COM, and so on). Doing so produces an error under MS OS/2.

Error codes

Because future releases of the operating system may return new error codes to system calls, you should write code that is open-ended about error codes—that is, write your program to deal with error codes beyond those currently defined. You can generally do this by including special handling for any codes that require special treatment, such as "File not found," and by taking a generic course of action for all other errors. The MS OS/2 protected mode API and the Family API have an interface that contains a message describing the error; this message can be displayed to the user. The interface also returns error classification information and a recommended action.

Multitasking concerns

Multitasking is a feature of MS OS/2 and will be a feature of all future versions of MS-DOS. The following guidelines apply to all programs, even to those written for MS-DOS version 3, because they may run in compatibility mode under MS OS/2.

Disabling interrupts

Do not disable interrupts, typically with the CLI instruction. The consequences of doing so depend on the environment.

In real mode programs under MS OS/2, disabling interrupts works normally but has a negative impact on the system's ability to maintain proper system throughput. Communications programs or networking applications might lose data. In a future version of real mode MS OS/2-80386, the operating system will disregard attempts to disable interrupts.

Protected mode programs under MS OS/2 can disable interrupts only in special Ring 2 segments. Disabling interrupts for longer than 100 microseconds might cause communications programs or networking applications to lose data or break connection. A future 80386-specific version of MS OS/2 will ignore attempts to disable interrupts in protected mode programs.

Measuring system resources

Do not attempt to measure system resources by exhausting them, and do not assume that because a resource is available at one time it will be available later. Remember: System resources are being shared with other programs.

For example, it is common for an MS-DOS version 3 application to request 1 MB of memory. The system cannot fulfill this request, so it returns the largest amount of memory available. The application then requests that amount of memory. Typically, applications do not even check for an error code from the second request. They routinely request all available memory because their creators knew that no other application could be in the system at the same time. This practice will work in real mode MS OS/2, although it is inefficient because MS OS/2 must allocate memory to a program that has no effective use for it. However, this practice will *not* work under MS OS/2 protected mode or under the Family API.

Another typical resource-exhaustion technique is opening files until an open is refused and then closing unneeded file handles. All applications, even those that run only in an MS OS/2 real mode environment, must use only the resources they need and not waste system resources; in a multitasking environment, other programs in the system usually need those resources.

Sharing rules

Because multiple programs can run under MS OS/2 simultaneously and because the system can be networked, conflicts can occur when two programs try to access the same file. MS OS/2 handles this situation with special file-sharing support. Although programs

ignorant of file-sharing rules can run in real mode, you should explicitly specify file-sharing rules in your program. This will reduce the number of file-access conflicts the user will encounter.

Miscellaneous guidelines

Do not use undocumented features of MS-DOS or undocumented fields such as those in the Find First File buffer. Also, do not modify or store your own values in such areas.

Maintain at least 2048 free bytes on the stack at all times. Future releases of MS-DOS may require extra stack space at system call and at interrupt time.

Print using conventional handle writes to the LPT device(s). For example:

```
fd = open("LPT1");
write(fd, data, datalen);
```

Do not use Interrupt 17H (the IBM ROM BIOS printer services), writes to the *stdprn* handle (handle 3), or special-purpose Interrupt 21H functions such as 05H (Printer Output). These methods are not supported under MS OS/2 protected mode or in the Family API.

Do not use the MS-DOS standard handles *stdaux* and *stdprn* (handles 3 and 4); these handles are not supported in MS OS/2 protected mode. Use only *stdin* (handle 0), *stdout* (handle 1), and *stderr* (handle 2). Do use these latter handles where appropriate and avoid opening the CON device directly. Avoid Interrupt 21H Functions 03H (Auxiliary Input) and 04H (Auxiliary Output), which are polling operations on *stdaux*.

Summary

A tenet of MS OS/2 design was flexibility: Each component was constructed in anticipation of massive changes in a future release and with an eye toward existing versions of MS-DOS. Writing applications that are upwardly and backwardly compatible in such an environment is essential—and challenging. Following the guidelines in this article will ensure that your programs function appropriately in the MS-DOS/OS/2 operating-system family.

Gordon Letwin

Article 17
Windows

Microsoft Windows is an operating environment that runs under MS-DOS versions 2.0 and later. The current version of Windows, version 2.0, requires either a fixed disk or two double-sided floppy-disk drives, at least 320 KB of memory, and a video display board and monitor capable of graphics and a screen resolution of at least 640 (horizontal) by 200 (vertical) pixels. A fixed disk and 640 KB of memory provide the best environment for running Windows; a mouse or other pointing device is optional but recommended.

For the user, Windows provides a multitasking, graphics-based windowing environment for running programs. In this environment, users can easily switch among several programs and transfer data between them. Because programs specially designed to run under Windows usually have a consistent user interface, the time spent learning a new program is greatly diminished. Furthermore, the user can carry out command functions using only the keyboard, only the mouse, or some combination of the two. In some cases, Windows (and Windows applications) provides several different ways to execute the same command.

For the program developer, Windows provides a wealth of high-level routines that make it easy to incorporate menus, scroll bars, and dialog boxes (which contain controls, such as push buttons and list boxes) into programs. Windows' graphics interface is device independent, so programs developed for Windows work with every video display adapter and printer that has a Windows driver (usually supplied by the hardware manufacturer). Windows also includes features that facilitate the translation of programs into foreign languages for international markets.

When Windows is running, it shares responsibility for managing system resources with MS-DOS. Thus, programs that run under Windows continue to use MS-DOS function calls for all file input and output and for executing other programs, but they do not use MS-DOS for display or printer output, keyboard or mouse input, or memory management. Instead, they use functions provided by Windows.

Program Categories

Programs that run under Windows can be divided into three categories:

1. Programs specially designed for the Windows environment. Examples of such programs include Clock and Calculator, which come with Windows. Microsoft Excel is also specially designed for Windows. Other programs of this type (such as Aldus's Pagemaker) are available from software vendors other than Microsoft. Programs in this category cannot run under MS-DOS without Windows.
2. Programs designed to run under MS-DOS but that can usually be run in a window along with programs designed specially for Windows. These programs do not require

large amounts of memory, do not write directly to the display, do not use graphics, and do not alter the operation of the keyboard interrupt. They cannot use the mouse, the Windows application-program interface (such as menus and dialog boxes), or the graphics services that Windows provides. MS-DOS utilities, such as EDLIN and CHKDSK, are examples of programs in this category.

3. Programs designed to run under MS-DOS but that require large amounts of memory, write directly to the display, use graphics, or alter the operation of the keyboard interrupt. When Windows runs such a program, it must suspend operation of all other programs running in Windows and allow the program to use the full screen. In some cases, Windows cannot switch back to its normal display until the program terminates. Microsoft Word and Lotus 1-2-3 are examples of programs in this category.

The programs in categories 2 and 3 are sometimes called standard applications. To run one of these programs in Windows, the user must create a PIF file (Program Information File) that describes how much memory the program requires and how it uses the computer's hardware.

Although the ability to run existing MS-DOS programs under Windows benefits the user, the primary purpose of Windows is to provide an environment for specially designed programs that take full advantage of the Windows interface. This discussion therefore concentrates almost exclusively on programs written for the Windows 2.0 environment.

The Windows Display

Figure 17-1 shows a typical Windows display running several programs that are included with the retail version of Windows 2.0.

The display is organized as a desktop, with each program occupying one or more rectangular windows that, unlike the tiled (juxtaposed) windows typical of earlier versions, can be overlapped. Only one program is active at any time — usually the program that is currently receiving keyboard input. Windows displays the currently active program on top of (overlying) the others. Programs such as CLOCK and TERMINAL that are not active continue to run normally, but do not receive keyboard input.

The user can make another program active by pressing and releasing (clicking) the mouse button when the mouse cursor is positioned in the new program's window or by pressing either the Alt-Tab or Alt-Esc key combination. Windows then brings the new active program to the top.

Most Windows programs allow their windows to be moved to another part of the display or to be resized to occupy smaller or larger areas. Most of these programs can also be maximized to fill the entire screen or minimized — generally as a small icon displayed at the bottom of the screen — to occupy a small amount of display space.

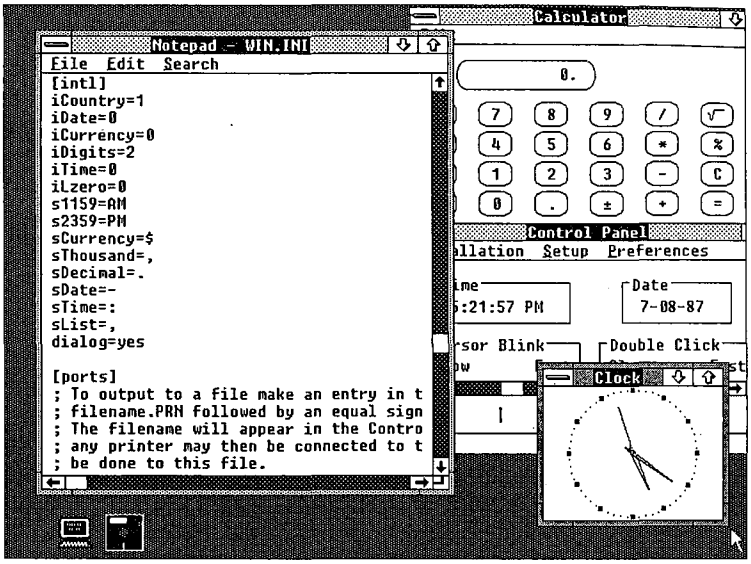


Figure 17-1. A typical Windows display.

Parts of the window

Figure 17-2 shows the Windows NOTEPAD program, with the different parts of the window identified. NOTEPAD is a small ASCII text editor limited to files of 16 KB. The various parts of the NOTEPAD window (similar to all Windows programs) are described in this section.

Title bar (or caption bar). The title bar identifies the program and, if applicable, the data file currently loaded into the program. For example, the NOTEPAD window shown in Figure 17-2 on the next page has the file WIN.INI loaded into memory. Windows uses different title-bar colors to distinguish the active window from inactive windows. The user can move a window to another part of the display by pressing the mouse button when the mouse pointer is positioned anywhere on the title bar and dragging (moving) the mouse while the button is pressed.

System-menu icon. When the user clicks a system-menu icon with the mouse (or presses Alt-Spacebar), Windows displays a system menu like that shown in Figure 17-3. (Most Windows programs have identical system menus.) The user selects a menu item in one of several ways: clicking on the item; moving the highlight bar to the item with the cursor-movement keys and then pressing Enter; or pressing the letter that is underlined in the menu item (for example, *n* for *Minimize*).

The keyboard combinations (Alt plus function key) at the right of the system menu are keyboard accelerators. Using a keyboard accelerator, the user can select system-menu options without first displaying the system menu.

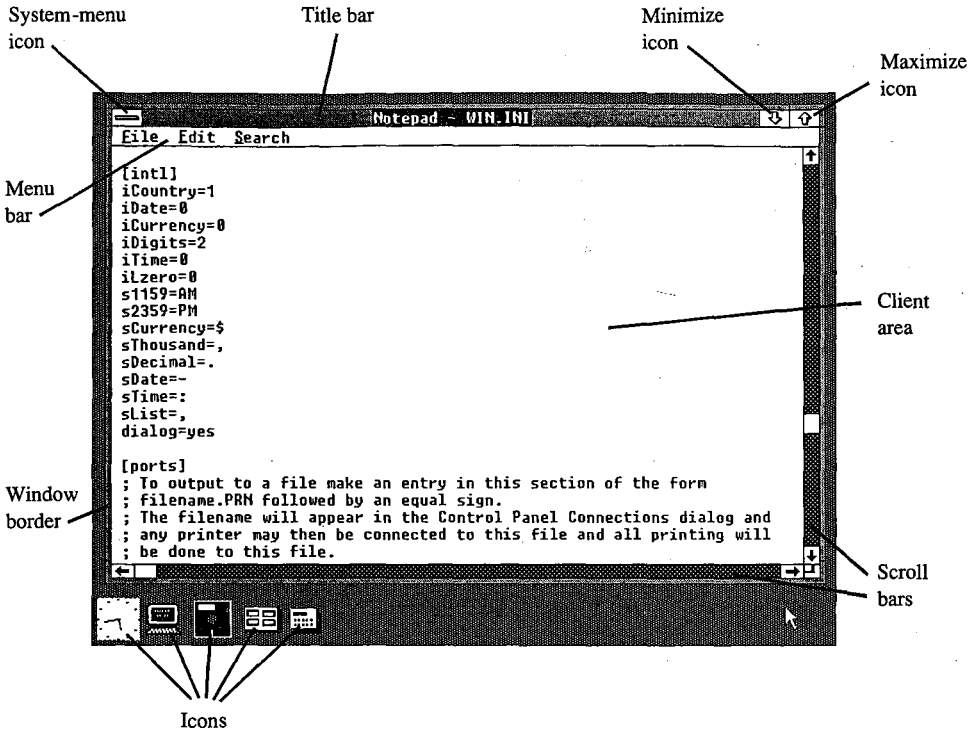


Figure 17-2. The Windows NOTEPAD program, with different parts of the display labeled.

The six options on the standard system menu are

- *Restore*: Return the window to its previous position and size after it has been minimized or maximized.
- *Move*: Allow the window to be moved with the cursor-movement keys.
- *Size*: Allow the window to be resized with the cursor-movement keys.
- *Minimize*: Display the window in its iconic form.
- *Maximize*: Allow the window to occupy the full screen.
- *Close*: End the program.

Windows displays an option on the system menu in grayed text to indicate that the option is not currently valid. In the system menu shown in Figure 17-3, for example, the *Restore* option is grayed because the window is not in a minimized or maximized form.

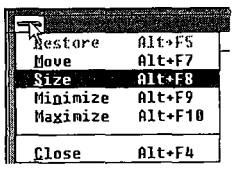


Figure 17-3. A system menu, displayed either when the user clicks the system-menu icon (top left corner) or presses Alt-Spacebar.

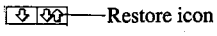


Figure 17-4. The restore icon, which replaces the maximize icon when a window is expanded to fill the entire screen.

Minimize icon. When the user clicks on the minimize icon with the mouse, Windows displays the program in its iconic form.

Maximize icon. Clicking on the maximize icon expands the window to fill the full screen. Windows then replaces the maximize icon with a restore icon (shown in Figure 17-4). Clicking on the restore icon restores the window to its previous size and position.

Programs that use a window of a fixed size (such as the CALC.EXE calculator program included with Windows) do not have a maximize icon.

Menu bar. The menu bar, sometimes called the program's main or top-level menu, displays keywords for several sets of commands that differ from program to program.

When the user clicks on a main-menu item with the mouse or presses the Alt key and the underlined letter in the menu text, Windows displays a pop-up menu for that item. The pop-up menu for NOTEPAD's keyword *File* is shown in Figure 17-5. Items are selected from a pop-up menu in the same way they are selected from the system menu.

A Windows program can display options on the menu in grayed text to indicate that they are not currently valid. The program can also display checkmarks to the left of pop-up menu items to indicate which of several options have been selected by the user.

In addition, items on a pop-up menu can be followed by an ellipsis (...) to indicate that selecting the item invokes a dialog box that prompts the user for additional information—more than can be provided by the menu.

Client area. The client area of the window is where the program displays data. In the case of the NOTEPAD program shown in Figure 17-2, the client area displays the file currently being edited. A program's handling of keyboard and mouse input within the client area depends on the type of work it does.

Scroll bars. Programs that cannot display all the data in a file within the client area of the window often have a horizontal scroll bar across the bottom and a vertical scroll bar down the right edge. Both types of scroll bars have a small, boxed arrow at each end to indicate the direction in which to scroll. In the NOTEPAD window in Figure 17-2, for example, clicking on the up arrow of the vertical scroll bar moves the data within the window down

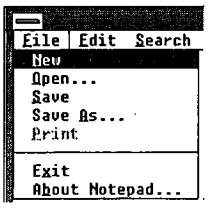


Figure 17-5. The NOTEPAD program's pop-up file menu.

one line. Clicking on the shaded part of the vertical scroll bar above the thumb (the box near the middle) moves the data within the client area of the window down one screen; clicking below the thumb moves the data up one screen. The user can also drag the thumb with the mouse to move to a relative position within the file.

Windows programs often include a keyboard interface (generally relying on the cursor-movement keys) to duplicate the mouse-based scroll-bar commands.

Window border. The window border is a thick frame surrounding the entire window. It is segmented into eight sections that represent the four sides and four corners of the window. The user can change the size of a window by dragging the window border with the mouse. Dragging a corner section moves two adjacent sides of the border.

When a program is maximized to fill the full screen, Windows does not draw the window border. Programs that use a window of a fixed size do not have a window border either.

Dialog boxes

When a pop-up menu is not adequate for all the command options a program requires, the program can display a dialog box. A dialog box is a pop-up window that contains various controls in the form of push buttons, check boxes, radio buttons, list boxes, and text and edit fields. Programmers can also design their own controls for use in dialog boxes. A user fills in a dialog box and then clicks on a button, such as *OK*, or presses Enter to indicate that the information can be processed by the program.

Most Windows programs use a dialog box to open an existing data file and load it into the program. The program displays the dialog box when the user selects the *Open* option on the *File* pop-up menu. The sample dialog box shown in Figure 17-6 is from the NOTEPAD program.

The list box displays a list of all valid disk drives, the subdirectories of the current directory, and all the filenames in the current directory, including the filename extension used by the program. (NOTEPAD uses the extension .TXT for its data files.) The user can scroll through this list box and change the current drive or subdirectory or select a filename with the keyboard or the mouse. The user can also perform these actions by typing the name directly into the edit field.

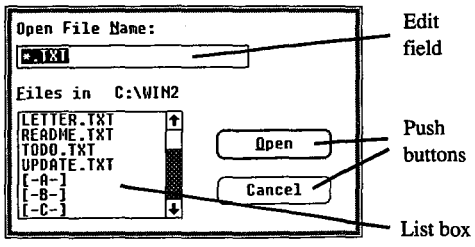


Figure 17-6. A dialog box from the NOTEPAD program, with parts labeled.

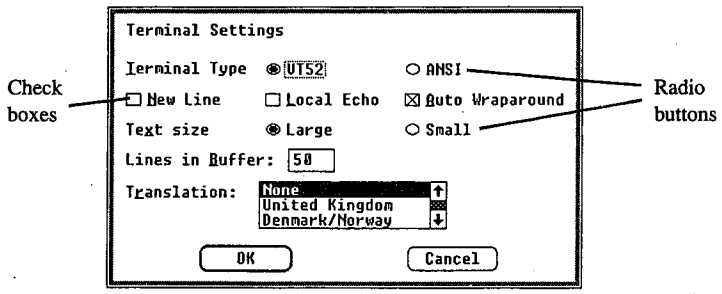


Figure 17-7. A dialog box from the TERMINAL program, with parts labeled.

Clicking the *Open* button (or pressing Enter) indicates to NOTEPAD that a file has been selected or that a new drive or subdirectory has been chosen (in this case, the program displays the files on the new drive or subdirectory). Clicking the *Cancel* button (or pressing Esc) tells NOTEPAD to close the dialog box without loading a new file.

Figure 17-7 shows a different dialog box—this one from the Windows TERMINAL communications program. The check boxes turn options on (indicated by an *X*) and off. The circular radio buttons allow the user to select from a set of mutually exclusive options.

Another, simple form of a dialog box is called a message box. This box displays one or more lines of text, an optional icon such as an exclamation point or an asterisk, and one or more buttons containing the words *OK*, *Yes*, *No*, or *Cancel*. Programs sometimes use message boxes for warnings or error messages.

The MS-DOS Executive

Within Windows, the MS-DOS Executive program (shown in Figure 17-8) serves much the same function as the COMMAND.COM program in the MS-DOS environment.

The top of the MS-DOS Executive client area displays all valid disk drives. The current disk drive is highlighted. Below or to the right of the disk drives is a display of the full path of the current directory. Below this is an alphabetic listing of all subdirectories in the current directory, followed by an alphabetic listing of all files in the current directory. Subdirectory names are displayed in boldface to distinguish them from filenames.

The user can change the current drive by clicking on the disk drive with the mouse or by pressing Ctrl and the key corresponding to the disk drive letter.

To change to one of the parent directories, the user double-clicks (clicks the mouse button twice in succession) on the part of the text string corresponding to the directory name. Pressing the Backspace key moves up one directory level toward the root directory. The user can also change the current directory to a child subdirectory by double-clicking on the subdirectory name in the list or by pressing the Enter key when the cursor highlight is on the subdirectory name. In addition, the menu also contains an option for changing the current directory.

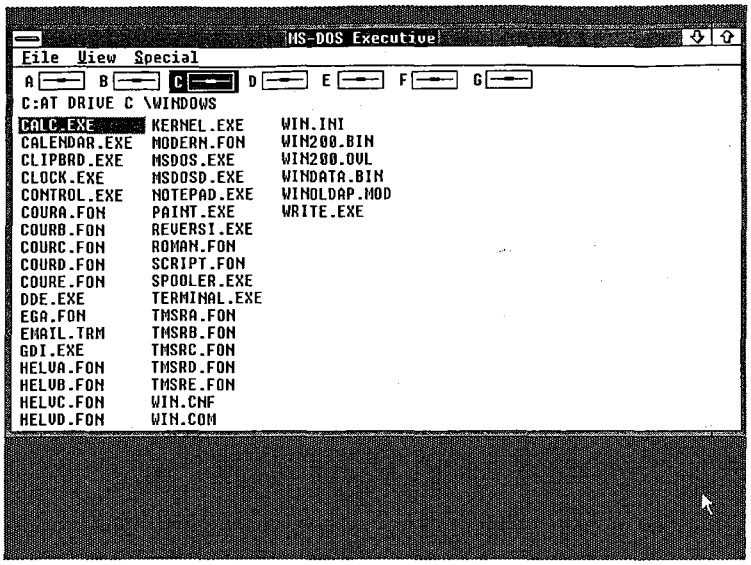


Figure 17-8. The MS-DOS Executive.

The user can run a program by double-clicking on the program filename, by pressing the Enter key when the highlight is on the program name, or by selecting it from a menu.

Other menu options allow the user to display the file and subdirectory lists in a variety of ways. A long format includes the same information displayed by the MS-DOS DIR command, or the user can choose to display a select group of files. Menu options also enable the user to specify whether the files should be listed in alphabetic order by filename, by filename extension, or by date or size.

The remaining options on the MS-DOS Executive menu allow the user to run programs; copy, rename, and delete files; format a floppy disk; change a volume name; make a system disk; create a subdirectory; and print a text file.

Other Windows Programs

Windows 2.0 also includes a number of application and utility programs. The application programs are CALC (a calculator), CALENDAR, CARDFILE (a database arranged as a series of index cards), CLOCK, NOTEPAD, PAINT (a drawing and painting program), REVERSI (a game), TERMINAL, and WRITE (a word processor).

The utility programs include

CLIPBRD. This program displays the current contents of the Clipboard, which is a storage facility that allows users to transfer data from one program to another.

CONTROL. The Control Panel utility allows the user to add or delete font files and printer drivers and to change the following: current printer, printer output port, communications parameters, date and time, cursor blink rate, screen colors, border width, mouse double-click time and options, and country-specific information, such as time and date formats. The Control Panel stores much of this information in the file named WIN.INI (Windows Initialization), so the information is available to other Windows programs.

PIFEDIT. The PIF editor allows the user to create or modify the PIFs that contain information about standard applications that have not been specially designed to run under Windows. This information allows Windows to adjust the environment in which the program runs.

SPOOLER. Windows uses the print-spooler utility to print files without suspending the operation of other programs. Most printer-directed output from Windows programs goes to the print spooler, which then prints the files while other programs run. SPOOLER enables the user to change the priority of print jobs or to cancel them.

The Structure of Windows

When programs run under MS-DOS, they make requests of the operating system through MS-DOS software interrupts (such as Interrupt 21H), through BIOS software interrupts, or by directly accessing the machine hardware.

When programs run under Windows, they use MS-DOS function calls only for file input and output and (more rarely) for executing other programs. Windows programs do not use MS-DOS function calls for memory management, keyboard input, display or printer output, or RS232 communications. Nor do Windows programs use BIOS routines or direct access to the hardware.

Instead, Windows provides application programs with access to more than 450 functions that allow programs to create and manipulate windows on the display; use menus, dialog boxes, and scroll bars; display text and graphics within the client area of a window; use the printer and RS232 communications port; and allocate memory.

The Windows modules

The functions provided by Windows are largely handled by three main modules named KERNEL, GDI, and USER. The KERNEL module is responsible for scheduling and multitasking, and it provides functions for memory management and some file I/O. The GDI module provides Windows' Graphics Device Interface functions, and the USER module does everything else.

The USER and GDI modules, in turn, call functions in various driver modules that are also included with Windows. Drivers control the display, printer, keyboard, mouse, sound, RS232 port, and timer. In most cases, these driver modules access the hardware of the computer directly. Windows includes different driver files for various hardware configurations. Hardware manufacturers can also develop Windows drivers specifically for their products.

A block diagram showing the relationships of an application program, the KERNEL, USER, and GDI modules, and the driver modules is shown in Figure 17-9. The figure shows each of these modules as a separate file — KERNEL, USER, and GDI have the extension .EXE; the driver files have the extension .DRV. Some program developers install Windows with these modules in separate files, as in Figure 17-9, but most users install Windows by running the SETUP program included with Windows.

SETUP combines most of these modules into two larger files called WIN200.BIN and WIN200.OVL. Printer drivers are a little different from the other driver files, however, because the Windows SETUP program does not include them in WIN200.BIN and WIN200.OVL. The name of the driver file identifies the printer. For example, IBMGRX.DRV is a printer driver file for the IBM Personal Computer Graphics Printer.

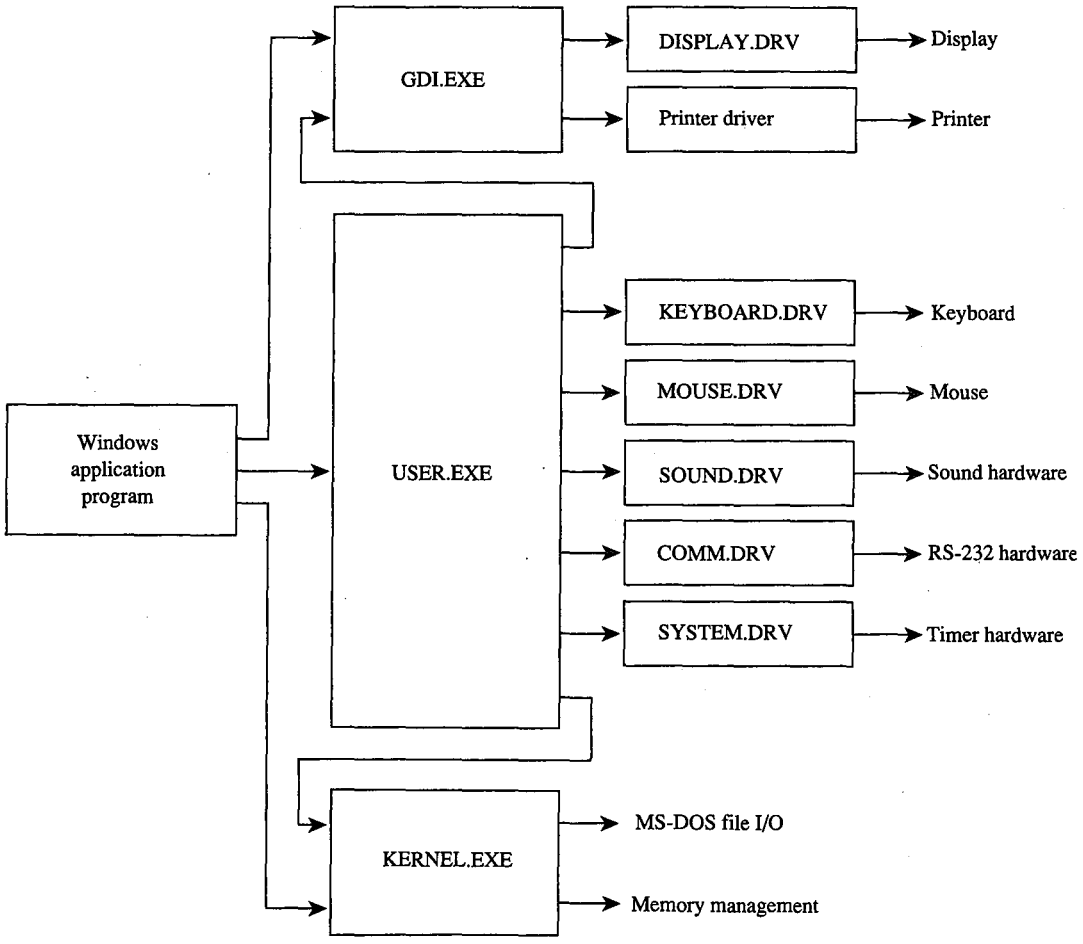


Figure 17-9. A simplified block diagram showing the relationships of an application program, Windows modules (GDI, USER, and KERNEL), driver modules, and system hardware.

The diagram in Figure 17-9 is somewhat simplified. In reality, a Windows application program can also make direct calls to the KEYBOARD.DRV and SOUND.DRV modules, and USER.EXE calls the DISPLAY.DRV and printer driver modules directly. The GDI.EXE module and driver modules can also call routines in KERNEL.EXE, and drivers sometimes call routines in SYSTEM.DRV.

Also, Figure 17-9 omits the various font files provided with Windows, the WIN.INI file that contains Windows initialization information and user preferences, and the files WINOLDAP.MOD and WINOLDAP.GRB, which Windows uses to run standard MS-DOS applications.

Libraries and programs

The USER.EXE, GDI.EXE, and KERNEL.EXE files, all driver files with the extension .DRV, and all font files with the extension .FON are called Windows libraries or, sometimes, dynamic link libraries to distinguish them from Windows programs. Programs and libraries both use a file format called the New Executable format.

From the user's perspective, a Windows program and a Windows library are very different. The user cannot run a Windows library directly: Windows loads a part of a library into memory only when a program needs to use a function that the library provides.

The user can also run multiple instances of the same Windows program. Windows uses the same code segments for the different instances but creates a unique data segment for each. Windows never runs multiple instances of a Windows library.

From the programmer's perspective, a Windows program is a task that creates and manages windows on the display. Libraries are modules that assist the task. A programmer can write additional library modules, which one or more programs can use. For the developer, one important distinction between programs and libraries is that a Windows library does not have its own stack; instead, the library uses the stack of the program that calls the routine in the library.

The New Executable format used for both programs and libraries gives Windows much more information about the module than is provided by the current MS-DOS .EXE format. In particular, the module contains information that allows Windows to make links between program modules and library modules when a program is run.

When a module (such as a library) contains functions that can be called from another module (such as a program), the functions are said to be exported from the module that contains them. Each exported function in a module is identified either by a name (generally the name of the function) or by an ordinal (positive) number. A list of all exported functions in a module is included in the New Executable format header section of the module.

Conversely, when a module (such as a program) contains code that calls a function in another module (such as a library), the function is said to be imported to the module that makes the call. This call appears in the .EXE file as an unresolved reference to an external function. The New Executable format identifies the module and the function name or ordinal number that the call references.

When Windows loads a program or a library into memory, it must resolve all calls the module makes to functions in other modules. Windows does this by inserting the addresses of the functions into the code—a process called dynamic linking.

For example, many Windows programs use the function TextOut to display text in the client area. In the code segment of the program's .EXE file, a call to TextOut appears as an unresolved far (intersegment) call. The code segment's relocation table shows that this call is to an imported function in the GDI module identified by the ordinal number 33. The header section of the GDI module lists TextOut as an exported function with the ordinal number 33. When Windows loads the program, it resolves all references to TextOut by inserting the address of the function into the program's code segment in each place where TextOut is called.

Although Windows programs reference many functions that are exported from the standard Windows libraries, Windows programs also often include at least one exported function, called a window function. While the program is running, Windows calls this function to pass messages to the program's window. *See* The Structure of a Windows Program below.

Memory Management

Windows' memory management is based on the segmented-memory architecture of the Intel 8086 family of microprocessors. The memory space controlled by Windows is divided into segments of various lengths. Windows uses separate segments for nearly everything kept in memory—such as the code and data segments of programs and libraries—and for resources, such as fonts and bitmaps.

Windows programs and libraries contain one or more code segments, which are usually both movable and discardable. Windows can move a code segment in memory in order to consolidate free memory space. It can also discard a code segment from memory and later reload the code segment from the program's or library's .EXE file when it is needed again. This capability is called demand loading.

Windows programs usually contain only one data segment; Windows libraries are limited to one data segment. In most cases, Windows can move data segments in memory. However, it cannot usually discard data segments, because they can contain data that changes after the program begins executing. When a user runs multiple copies of a program, the different instances share the same code segments but have separate data segments.

The use of movable and discardable segments allows Windows to run several large programs in a memory space that might be inadequate for even one of the programs if the entire program were kept in memory, as is typical under MS-DOS without Windows. The ability of Windows to use memory in this way is called memory overcommitment.

The moving and discarding of code segments requires Windows to make special provisions so that intersegment calls continue to reference the correct address when a code

segment is moved. These provisions are another part of dynamic linking. When Windows resolves a far call from one code segment to a function in another code segment that is movable and discardable, the call actually references a fixed area of memory. This fixed area of memory contains a small section of code called a thunk. If the code segment containing the function is currently in memory, the thunk simply jumps to the function. If the code segment with the function is not currently in memory, the thunk calls a loader that loads the segment into memory. This process is called dynamic loading. When Windows moves or discards a code segment, it must alter the thunks appropriately.

Windows and Windows programs generally reference data structures stored in Windows' memory space by using 16-bit unsigned integers known as handles. The data structure that a handle references can be movable and discardable, so when Windows or the Windows program needs to access the data directly, it must lock the handle to cause the data to become fixed in memory. The function that locks the segment returns a pointer to the program.

During the time the handle is locked, Windows cannot move or discard the data. The data can then be referenced directly with the pointer. When Windows (or the Windows program) finishes using the data, it unlocks the segment so that it can be moved (or in some cases discarded) to free up memory space if necessary.

Programmers can choose to allocate nonmovable data segments, but the practice is not recommended, because Windows cannot relocate the segments to make room for segments required by other programs.

The Structure of a Windows Program

During development, a Windows program includes several components that are combined later into a single executable file with the extension .EXE for execution under Windows. Although the Windows executable file has the same .EXE filename extension as MS-DOS executable files, the format is different. Among other things, the New Executable format includes Windows-specific information required for dynamic linking and the discarding and reloading of the program's code segments.

Programmers generally use C, Pascal, or assembly language to create applications specially designed to run under Windows. Also required are several header files and development tools, which are included in the Microsoft Windows Software Development Kit.

The Microsoft Windows Software Development Kit

The Windows Software Development Kit contains reference material, a special linker (LINK4), the Windows Resource Compiler (RC), special versions of the SYMDEB and CodeView debuggers, header files, and several programs that aid development and testing. These programs include

- DIALOG: Used for creating dialog boxes.
- ICONEDIT: Used for creating a program's icon, customized cursors, and bitmap images.

- FONTEDIT: Used for creating customized fonts derived from an existing font file with the extension .FNT.
- HEAPWALK: Used for displaying the organization of code and data segments in Windows' memory space and for testing programs under low memory conditions.
- SHAKER: Used for randomly allocating memory to force segment movement and discarding. SHAKER tests a program's response to movement in memory and is useful for exposing program bugs involving pointers to unlocked segments.

The Windows Software Development Kit also provides several *include* and header files that contain declarations of all Windows functions, definitions of many macro identifiers that the programmer can use, and structure definitions. Import libraries included in the kit allow LINK4 to resolve calls to Windows functions and to prepare the program's .EXE file for dynamic linking.

Work with the Windows Software Development Kit requires one of the following compilers or assemblers:

- Microsoft C Compiler version 4.0 or later
- Microsoft Pascal Compiler version 3.31 or later
- Microsoft Macro Assembler version 4.0 or later

Other software manufacturers also provide compilers that are suitable for compiling Windows programs.

Components of a Windows program

The discussion in this section is illustrated by a program called SAMPLE, which displays the word *Windows* in its client area. In response to a menu selection, the program

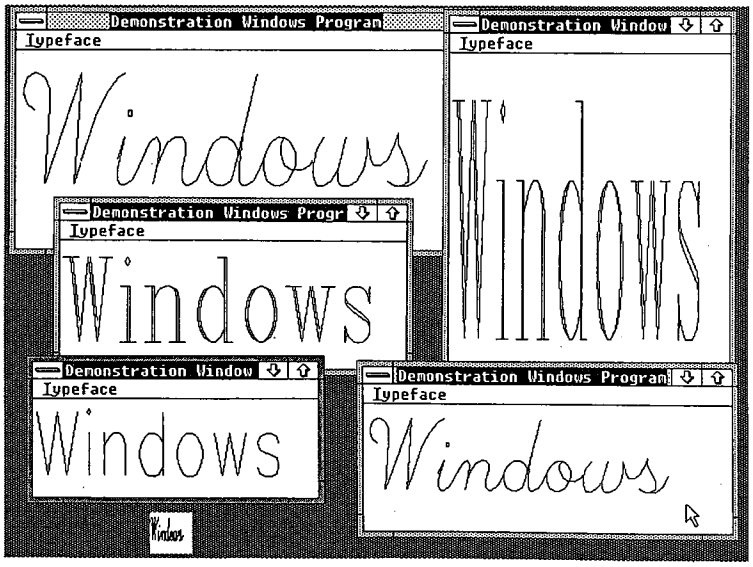


Figure 17-10. A display produced by the example program SAMPLE.

displays this text in any of the three vector fonts — Script, Modern, and Roman — that are included with Windows. Sometimes also called stroke or graphics fonts, these vector fonts are defined by a series of line segments, rather than by the pixel patterns that make up the more common raster fonts. The SAMPLE program picks a font size that fits the client area.

Figure 17-10 shows several instances of this program running under Windows.

Five separate files go into the making of this program:

1. Source-code file: This is the main part of the program, generally written in C, Pascal, or assembly language. The SAMPLE program was written in C, which is the most popular language for Windows programs because of its flexibility in using pointers and structures. The SAMPLE.C source-code file is shown in Figure 17-11.

```
/* SAMPLE.C -- Demonstration Windows Program */

#include <windows.h>
#include "sample.h"

long FAR PASCAL WndProc (HWND, unsigned, WORD, LONG) ;

int PASCAL WinMain (hInstance, hPrevInstance, lpszCmdLine, nCmdShow)
     HANDLE      hInstance, hPrevInstance ;
     LPSTR       lpszCmdLine ;
     int         nCmdShow ;
     {
     WNDCLASS    wndclass ;
     HWND        hWnd ;
     MSG         msg ;
     static char szAppName [] = "Sample" ;

               /*----------------------------*/
               /* Register the Window Class */
               /*----------------------------*/

     if (!hPrevInstance)
          {
          wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
          wndclass.lpfnWndProc   = WndProc ;
          wndclass.cbClsExtra    = 0 ;
          wndclass.cbWndExtra    = 0 ;
          wndclass.hInstance     = hInstance ;
          wndclass.hIcon         = NULL ;
          wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
          wndclass.hbrBackground = GetStockObject (WHITE_BRUSH) ;
          wndclass.lpszMenuName  = szAppName ;
          wndclass.lpszClassName = szAppName ;

          RegisterClass (&wndclass) ;
          }
```

Figure 17-11. The SAMPLE.C source code. *(more)*

```
                /*----------------------------------*/
                /* Create the window and display it */
                /*----------------------------------*/

     hWnd = CreateWindow (szAppName, "Demonstration Windows Program",
                          WS_OVERLAPPEDWINDOW,
                          (int) CW_USEDEFAULT, 0,
                          (int) CW_USEDEFAULT, 0,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hWnd, nCmdShow) ;
     UpdateWindow (hWnd) ;

                /*----------------------------------------------*/
                /* Stay in message loop until a WM_QUIT message */
                /*----------------------------------------------*/

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

long FAR PASCAL WndProc (hWnd, iMessage, wParam, lParam)
     HWND         hWnd ;
     unsigned     iMessage ;
     WORD         wParam ;
     LONG         lParam ;
     {
     PAINTSTRUCT  ps ;
     HFONT        hFont ;
     HMENU        hMenu ;
     static short xClient, yClient, nCurrentFont = IDM_SCRIPT ;
     static BYTE  cFamily [] = { FF_SCRIPT, FF_MODERN, FF_ROMAN } ;
     static char  *szFace [] = { "Script",  "Modern",  "Roman"  } ;

     switch (iMessage)
          {

                    /*---------------------------------------------*/
                    /* WM_COMMAND message: Change checkmarked font */
                    /*---------------------------------------------*/

          case WM_COMMAND:
               hMenu = GetMenu (hWnd) ;
               CheckMenuItem (hMenu, nCurrentFont, MF_UNCHECKED) ;
               nCurrentFont = wParam ;
               CheckMenuItem (hMenu, nCurrentFont, MF_CHECKED) ;
               InvalidateRect (hWnd, NULL, TRUE) ;
               break ;
```

Figure 17-11. Continued. *(more)*

```
                    /*----------------------------------------*/
                    /* WM_SIZE message: Save dimensions of window */
                    /*----------------------------------------*/

               case WM_SIZE:
                    xClient = LOWORD (lParam) ;
                    yClient = HIWORD (lParam) ;
                    break ;

                    /*-----------------------------------------------*/
                    /* WM_PAINT message: Display "Windows" in Script */
                    /*-----------------------------------------------*/

               case WM_PAINT:
                    BeginPaint (hWnd, &ps) ;

                    hFont = CreateFont (yClient, xClient / 8,
                                        0, 0, 400, 0, 0, 0, OEM_CHARSET,
                                        OUT_STROKE_PRECIS, OUT_STROKE_PRECIS,
                                        DRAFT_QUALITY, (BYTE) VARIABLE_PITCH |
                                        cFamily [nCurrentFont - IDM_SCRIPT],
                                        szFace  [nCurrentFont - IDM_SCRIPT]) ;

                    hFont = SelectObject (ps.hdc, hFont) ;
                    TextOut (ps.hdc, 0, 0, "Windows", 7) ;

                    DeleteObject (SelectObject (ps.hdc, hFont)) ;
                    EndPaint (hWnd, &ps) ;
                    break ;

                    /*---------------------------------------*/
                    /* WM_DESTROY message: Post Quit message */
                    /*---------------------------------------*/

               case WM_DESTROY:
                    PostQuitMessage (0) ;
                    break ;

                    /*---------------------------------------*/
                    /* Other messages: Do default processing */
                    /*---------------------------------------*/

               default:
                    return DefWindowProc (hWnd, iMessage, wParam, lParam) ;
               }
          return 0L ;
          }
```

Figure 17-11. Continued.

2. Resource script: The resource script is an ASCII file that generally has the extension .RC. This file contains definitions of menus, dialog boxes, string tables, and keyboard accelerators used by the program. The resource script can also reference other files that contain icons, cursors, bitmaps, and fonts in binary form, as well as other read-only data defined by the programmer. When a program is running, Windows loads resources into memory only when they are needed and in most cases can discard them if additional memory space is required.

   SAMPLE.RC, the resource script for the SAMPLE program, is shown in Figure 17-12; it contains only the definition of the menu used in the program.

```
#include "sample.h"

Sample MENU
     BEGIN
          POPUP "&Typeface"
               BEGIN
                    MENUITEM "&Script", IDM_SCRIPT, CHECKED
                    MENUITEM "&Modern", IDM_MODERN
                    MENUITEM "&Roman",  IDM_ROMAN
               END
     END
```

Figure 17-12. The resource script for the SAMPLE program.

3. Header (or *include*) file: This file, with the extension .H, can contain definitions of constants or macros, as is customary in C programming. For Windows programs, the header file also reconciles constants used in both the resource script and the program source-code file. For example, in the SAMPLE.RC resource script, each item in the pop-up menu (*Script*, *Modern*, and *Roman*) also includes an identifier—IDM_SCRIPT, IDM_MODERN, and IDM_ROMAN, respectively. These identifiers are merely numbers that Windows uses to notify the program of the user's selection of a menu item. The same names are used to identify the menu selection in the C source-code file. And, because both the resource compiler and the source-code compiler must have access to these identifiers, the header file is included in both the resource script and the source-code file.

   The header file for the SAMPLE program, SAMPLE.H, is shown in Figure 17-13.

```
#define IDM_SCRIPT 1
#define IDM_MODERN 2
#define IDM_ROMAN  3
```

Figure 17-13. The SAMPLE.H header file.

4. Module-definition file: The module-definition file generally has a .DEF extension. The Windows linker uses this file in creating the executable .EXE file. The module-definition file specifies various attributes of the program's code and data segments, and it lists all imported and exported functions in the source-code file. In large programs that are divided into multiple code segments, the module-definition file allows the programmer to specify different attributes for each code segment.

The module-definition file for the SAMPLE program is named SAMPLE.DEF and is shown in Figure 17-14.

```
NAME           SAMPLE
DESCRIPTION    'Demonstration Windows Program'
STUB           'WINSTUB.EXE'
CODE           MOVABLE
DATA           MOVABLE MULTIPLE
HEAPSIZE       1024
STACKSIZE      4096
EXPORTS        WndProc
```

Figure 17-14. The SAMPLE.DEF module-definition file.

5. Make file: To facilitate construction of the executable file from these different components, Windows programmers often use the MAKE program to compile only those files that have changed since the last time the program was linked. To do this, the programmer first creates an ASCII text file called a make file. By convention, the make file has no extension.

   The make file for the SAMPLE program is named SAMPLE and is shown in Figure 17-15. The programmer can create the SAMPLE.EXE executable file by executing

   ```
   C>MAKE SAMPLE  <Enter>
   ```

   A make file often contains several sections, each beginning with a target filename, followed by a colon and one or more dependent filenames, such as

   ```
   sample.obj : sample.c sample.h
   ```

   If either or both the SAMPLE.C and SAMPLE.H files have a later creation time than SAMPLE.OBJ, then MAKE runs the program or programs listed immediately below. In the case of the SAMPLE make file, the program is the C compiler, and it compiles the SAMPLE.C source code:

   ```
   cl -c -Gsw -W2 -Zdp sample.c
   ```

   Thus, if the programmer changes only one of the several files used in the development of SAMPLE, then running MAKE ensures that the executable file is brought up to date, while carrying out only the required steps.

   ```
   sample.obj : sample.c sample.h
        cl -c -Gsw -W2 -Zdp sample.c

   sample.res : sample.rc sample.h
        rc -r sample.rc

   sample.exe : sample.obj sample.def sample.res
        link4 sample, /align:16, /map /line, slibw, sample
        rc sample.res
        mapsym sample
   ```

   Figure 17-15. The make file for the SAMPLE program.

Construction of a Windows program

The make file shows the steps that create a program's .EXE file from the various components:

1. Compiling the source-code file:

```
cl -c -Gsw -W2 -Zdp sample.c
```

   This step uses the CL.EXE C compiler to create a .OBJ object-module file. The command line switches are
   - -c: Compiles the program but does not link it. Windows programs must be linked with Windows' LINK4 linker, rather than with the LINK program the C compiler would normally invoke.
   - -Gsw: Includes two switches, -Gs and -Gw. The -Gs switch removes stack checks from the program. The -Gw switch inserts special prologue and epilogue code in all far functions defined in the program. This special code is required for Windows' memory management.
   - -W2: Compiles with warning level 2. This is the highest warning level, and it causes the compiler to display messages for conditions that may be acceptable in normal C programs but that can cause serious errors in a Windows program.
   - -Zdp: Includes two switches, -Zd and -Zp. The -Zd switch includes line numbers in the .OBJ file — helpful for debugging at the source-code level. The -Zp switch packs structures on byte boundaries. The -Zp switch is required, because data structures used within Windows are in a packed format.
2. Compiling the resource script:

```
rc -r sample.rc
```

   This step runs the resource compiler and converts the ASCII .RC resource script into a binary .RES form. The -r switch indicates that the resource script should be compiled but the resources should not yet be added to the program's .EXE file.
3. Linking the program:

```
link4 sample, /align:16, /map /line, slibw, sample
```

   This step uses the special Windows linker, LINK4. The first parameter listed is the name of the .OBJ file. The /align:16 switch instructs LINK4 to align segments in the .EXE file on 16-byte boundaries. The /map and /line switches cause LINK4 to create a .MAP file that contains program line numbers — again, useful for debugging source code. Next, slibw is a reference to the SLIBW.LIB file, which is an import library that contains module names and ordinal numbers for all Windows functions. The last parameter, sample, is the program's module-definition file, SAMPLE.DEF.
4. Adding the resources to the .EXE file:

```
rc sample.res
```

This step runs the resource compiler a second time, using the compiled resource file, SAMPLE.RES. This time, the resource compiler adds the resources to the .EXE file.

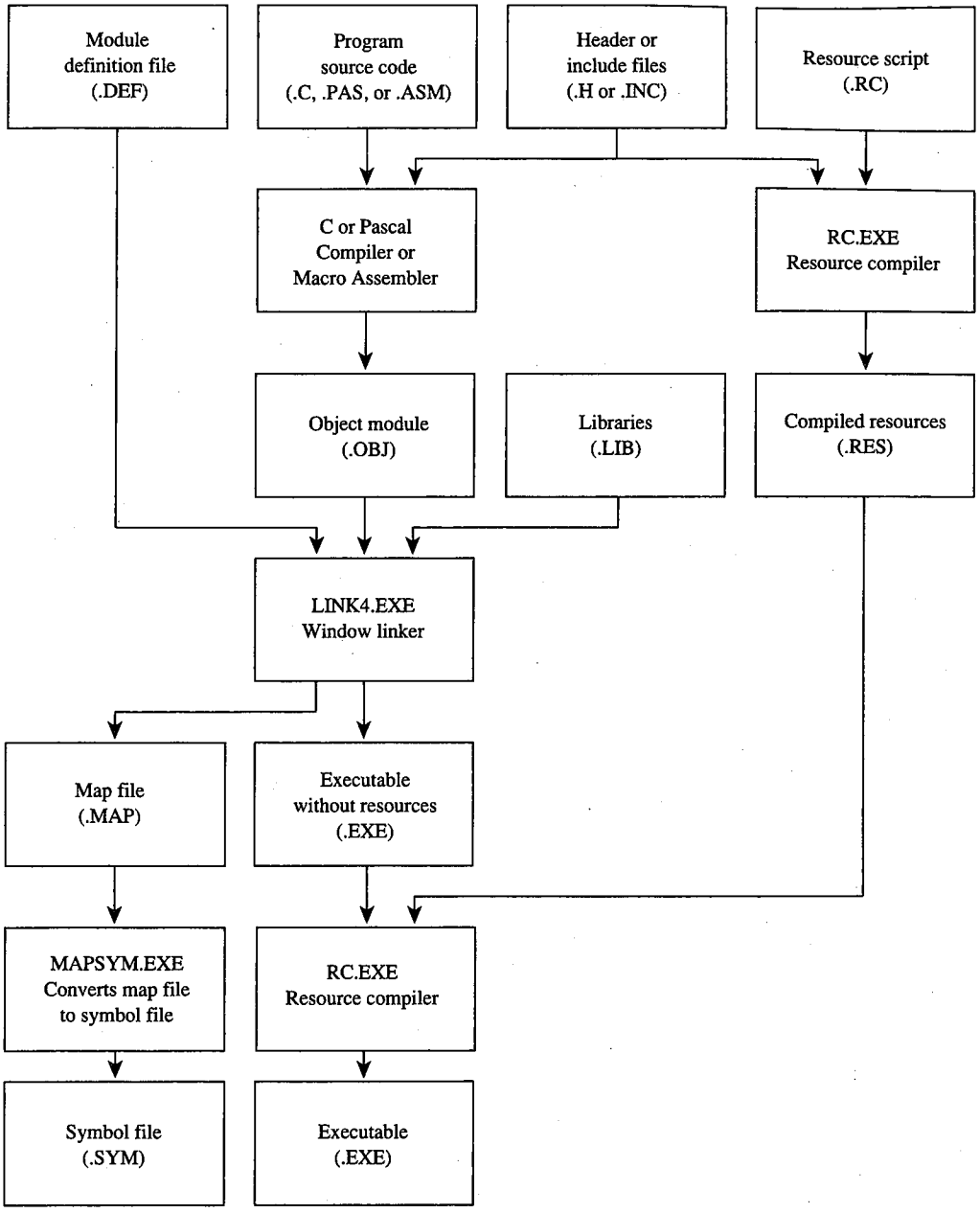


Figure 17-16. A block diagram showing the creation of a Windows .EXE file.

5. Creating a symbol (.SYM) file from the linker's map (.MAP) file:

```
mapsym sample
```

This step is required for symbolic debugging with SYMDEB.

Figure 17-16 on the preceding page shows how the various components of a Windows program fit into the creation of a .EXE file.

Program initialization

The SAMPLE.C program shown in Figure 17-11 contains some code that appears in almost every Windows program. The statement

```
#include <windows.h>
```

appears at the top of every Windows source-code file written in C. The WINDOWS.H file, provided with the Microsoft Windows Software Development Kit, contains templates for all Windows functions, structure definitions, and #define statements for many mnemonic identifiers.

Some of the variable names in SAMPLE.C may look unusual to C programmers because they begin with a prefix notation that denotes the data type of the variable. Windows programmers are encouraged to use this type of notation. Some of the more common prefixes are

Prefix	Data Type
i *or* n	Integer (16-bit signed integer)
w	Word (16-bit unsigned integer)
l	Long (32-bit signed integer)
dw	Doubleword (32-bit unsigned integer)
h	Handle (16-bit unsigned integer)
sz	Null-terminated string
lpsz	Long pointer to null-terminated string
lpfn	Long pointer to a function

The program's entry point (following some startup code) is the WinMain function, which is passed the following parameters: a handle to the current instance of the program (hInstance), a handle to the most recent previous instance of the program (hPrevInstance), a long pointer to the program's command line (lpszCmdLine), and a number (nCmdShow) that indicates whether the program should initially be displayed as a normally sized window or as an icon.

The first job SAMPLE performs in the WinMain function is to register a window class—a structure that describes characteristics of the windows that will be created in the class. These characteristics include background color, the type of cursor to be displayed in the window, the window's initial menu and icon, and the window function (the structure member called lpfnWndProc).

Multiple instances of a program can share the same window class, so SAMPLE registers the window class only for the first instance of the program:

```
if (!hPrevInstance)
          {
          wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
          wndclass.lpfnWndProc   = WndProc ;
          wndclass.cbClsExtra    = 0 ;
          wndclass.cbWndExtra    = 0 ;
          wndclass.hInstance     = hInstance ;
          wndclass.hIcon         = NULL ;
          wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
          wndclass.hbrBackground = GetStockObject (WHITE_BRUSH) ;
          wndclass.lpszMenuName  = szAppName ;
          wndclass.lpszClassName = szAppName ;

          RegisterClass (&wndclass) ;
          }
```

The SAMPLE program then creates a window using the CreateWindow call, displays it to the screen by calling ShowWindow, and updates the client area by calling UpdateWindow:

```
hWnd = CreateWindow (szAppName, "Demonstration Windows Program",
                     WS_OVERLAPPEDWINDOW,
                     (int) CW_USEDEFAULT,0,
                     (int) CW_USEDEFAULT,0,
                     NULL, NULL, hInstance, NULL) ;
ShowWindow (hWnd, nCmdShow) ;
UpdateWindow (hWnd) ;
```

The first parameter to CreateWindow is the name of the window class. The second parameter is the actual text that appears in the window's title bar. The third parameter is the style of the window — in this case, the WINDOWS.H identifier WS_OVERLAPPEDWINDOW. The WS_OVERLAPPEDWINDOW is the most common window style. The fourth through seventh parameters specify the initial position and size of the window. The identifier CW_USEDEFAULT tells Windows to position and size the window according to the default rules.

After creating and displaying a Window, the SAMPLE program enters a piece of code called the message loop:

```
while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
return msg.wParam ;
```

This loop continues to execute until the GetMessage call returns zero. When that happens, the program instance terminates and the memory required for the instance is freed.

The Windows messaging system

Interactive programs written for the normal MS-DOS environment generally obtain user input only from the keyboard, using either an MS-DOS or a ROM BIOS software interrupt to check for keystrokes. When the user types something, such programs act on the keystroke and then return to wait for the next keystroke.

Programs written for Windows, however, can receive user input from a variety of sources, including the keyboard, the mouse, the Windows timer, menus, scroll bars, and controls, such as buttons and edit boxes.

Moreover, a Windows program must be informed of other events occurring within the system. For instance, the user of a Windows program might choose to make its window smaller or larger. Windows must make the program aware of this change so that the program can adjust its screen output to fit the new window size. Thus, for example, if a Windows program is minimized as an icon and the user maximizes its window to fill the full screen, Windows must inform the program that the size of the client area has changed and needs to be re-created.

Windows carries out this job of keeping a program informed of other events through the use of formatted messages. In effect, Windows sends these messages to the program. The Windows program receives and acts upon the messages.

This messaging makes the relationship between Windows and a Windows program much different from the relationship between MS-DOS and an MS-DOS program. Whereas MS-DOS does not provide information until a program requests it through an MS-DOS function call, Windows must continually notify a program of all the events that affect its window.

Window messages can be separated into two major categories: queued and nonqueued.

Queued messages are similar to the keyboard information an MS-DOS program obtains from MS-DOS. When the Windows user presses a key on the keyboard, moves the mouse, or presses one of the mouse buttons, Windows saves information about the event (in the form of a data structure) in the system message queue. Each message is destined for a particular window in a particular instance of a Windows program. Windows therefore determines which window should get the information and then places the message in the instance's own message queue.

A Windows program retrieves information from its queue in the message loop:

```
while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
```

The *msg* variable is a structure. During the GetMessage call, Windows fills in the fields of this structure with information about the message. The fields are as follows:

- *hwnd*: The handle for the window that is to receive the message.
- *iMessage*: A numeric code identifying the type of message (for example, keyboard or mouse).
- *wParam*: A 16-bit value containing information specific to the message. *See* The Windows Messages below.
- *lParam*: A 32-bit value containing information specific to the message.
- *time*: The time, in milliseconds, that the message was placed in the queue. The time is a 32-bit value relative to the time at which the current Windows session began.
- *pt.x*: The horizontal coordinate of the mouse cursor at the time the event occurred.
- *pt.y*: The vertical coordinate of the mouse cursor at the time the event occurred.

GetMessage always returns a nonzero value except when it receives a quit message. The quit message causes the message loop to end. The program should then terminate and return control to Windows.

Within the message loop, the TranslateMessage function translates physical keystrokes into character-code messages. Windows places these translated messages into the program's message queue.

The DispatchMessage function essentially makes a call to the window function of the window specified by the hwnd field. This window function (WndProc in SAMPLE) is indicated in the lpfnWndProc field of the window class structure.

When DispatchMessage passes the message to the window function, Windows uses the first four fields of the message structure as parameters to the function. The window function can then process the message. In SAMPLE, for instance, the four fields passed to WndProc are *hwnd* (the handle to the window), *iMessage* (the numeric message identifier), *wParam*, and *lParam*. Although Windows does not pass the time and mouse-position information fields as parameters to the window function, this information is available through the Windows functions GetMessageTime and GetMessagePos.

A Windows program obtains only a few specific types of messages through its message queue. These are keyboard messages, mouse messages, timer messages, the paint message that tells the program it must re-create the client area of its window, and the quit message that tells the program it is being terminated.

In addition to the queued messages, however, a program's window function also receives many nonqueued messages. Windows sends these nonqueued messages by bypassing the message loop and calling the program's window function directly.

Many of these nonqueued messages are derived from queued messages. For example, when the user clicks the mouse on the menu bar, a mouse-click message is placed in the program's message queue. The GetMessage function retrieves the message and the DispatchMessage function sends it to the program's window function. However, because this mouse message affects a nonclient area of the window (an area outside the window's client area), the window function normally does not process it. Instead, the function passes the message back to Windows. In this example, the message tells Windows to invoke a pop-up menu. Windows calls up the menu and then sends the window function several nonqueued messages to inform the program of this action.

A Windows program is thus message driven. Once a program reaches the message loop, it acts only when the window function receives a message. And, although a program receives many messages that affect the window, the program usually processes only some of them, sending the rest to Windows for normal default processing.

The Windows messages

Windows can send a window function more than 100 different messages. The WINDOWS.H header file includes identifiers for all these messages so that C programmers do not have to remember the message numbers. Some of the more common messages and the meanings of the *wParam* and *lParam* parameters are discussed here:

WM_CREATE. Windows sends a window function this nonqueued message while processing the CreateWindow call. The *lParam* parameter is a pointer to a creation structure. A window function can perform some program initialization during the WM_CREATE message.

WM_MOVE. Windows sends a window function the nonqueued WM_MOVE message when the window has been moved to another part of the display. The *lParam* parameter gives the new coordinates of the window relative to the upper left corner of the screen.

WM_SIZE. This nonqueued message indicates that the size of the window has been changed. The new size is encoded in the *lParam* parameter. Programs often save this window size for later use.

WM_PAINT. This queued message indicates that a region in the window's client area needs repainting. (The message queue can contain only one WM_PAINT message.)

WM_COMMAND. This nonqueued message signals a program that a user has selected a menu item or has triggered a keyboard accelerator. Child-window controls also use WM_COMMAND to send messages to the parent window.

WM_KEYDOWN. The *wParam* parameter of this queued message is a virtual key code that identifies the key being pressed. The *lParam* parameter includes flags that indicate the previous key state and the number of keypresses the message represents.

WM_KEYUP. This queued message tells a window function that a key has been released. The *wParam* parameter is a virtual key code.

WM_CHAR. This queued message is generated from WM_KEYDOWN messages during the TranslateMessage call. The *wParam* parameter is the ASCII code of a keyboard key.

WM_MOUSEMOVE. Windows uses this queued message to tell a program about mouse movement. The *lParam* parameter contains the coordinates of the mouse relative to the upper left corner of the client area of the window. The *wParam* parameter contains flags that indicate whether any mouse buttons or the Shift or Ctrl keys are currently pressed.

WM_xBUTTONDOWN. This queued message tells a program that a button on the mouse was depressed while the mouse was in the window's client area. The *x* can be either L, R, or M for the left, right, or middle mouse button. The *wParam* and *lParam* parameters are the same as for WM_MOUSEMOVE.

WM_xBUTTONUP. This queued message tells a program that the user has released a mouse button.

WM_xBUTTONDBLCLK. When the user double-clicks a mouse button, Windows generates a WM_*x*BUTTONDOWN message for the first click and a queued WM_*x*BUTTONDBLCLK message for the second click.

WM_TIMER. When a Windows program sets a timer with the SetTimer function, Windows places a WM_TIMER message in the message queue at periodic intervals. The *wParam* parameter is a timer ID. (If the message queue already contains a WM_TIMER message, Windows does not add another one to the queue.)

WM_VSCROLL. A Windows program that includes a vertical scroll bar in its window receives nonqueued WM_VSCROLL messages indicating various types of scroll-bar manipulation.

WM_HSCROLL. This nonqueued message indicates a user is manipulating a horizontal scroll bar.

WM_CLOSE. Windows sends a window function this nonqueued message when the user has selected *Close* from the window's system menu. A program can query the user to determine whether any action, such as saving a file to disk, is needed before the program is terminated.

WM_QUERYENDSESSION. This nonqueued message indicates that the user is shutting down Windows by selecting *Close* from the MS-DOS Executive system menu. A program can request the user to verify that the program should be ended. If the window function returns a zero value from the message, Windows does not end the session.

WM_DESTROY. This nonqueued message is the last message a window function receives before the program ends. A window function can perform some last-minute cleanup while processing WM_DESTROY.

WM_QUIT. This is a queued message that never reaches the window function because it causes GetMessage to return a zero value that causes the program to exit the message loop.

Message processing

Programmers can choose to process some messages and ignore others in the window function. Messages that are ignored are generally passed on to the function DefWindowProc for default processing within Windows.

Because Windows eventually has access to messages that a window function does not process, it can send a program messages that might otherwise be regarded as pertaining to system functions—for example, mouse messages that occur in a nonclient area of the window, or system keyboard messages that affect the menu. Unless these messages are passed on to DefWindowProc, the menu and other system functions do not work properly.

A program can, however, trap some of these messages to override Windows' default processing. For example, when Windows needs to repaint the nonclient area of a window (the title bar, system-menu box, and scroll bars), it sends the window function a WM_NCPAINT

(nonclient paint) message. The window function normally passes this message to DefWindowProc, which then calls routines to update the nonclient areas of the window. The program can, however, choose to process the WM_NCPAINT message and paint the nonclient area itself. A program that does this can, for example, draw its own scroll bars.

The Windows messaging system also notifies a program of important events occurring outside its window. For example, if the MS-DOS Executive were simply to end the Windows session when the user selects the *Close* option from its system menu, then applications that were still running would not have a chance to save changed files to disk. Instead, when the user selects *Close* from the last instance of the MS-DOS Executive's system menu, the MS-DOS Executive sends a WM_QUERYENDSESSION message to each currently running application. If any application responds by returning a zero value, the MS-DOS Executive does not end the Windows session.

Before responding, an application can process the WM_QUERYENDSESSION message and display a message box asking the user if a file should be saved. The message box should include three buttons labeled *Yes*, *No*, and *Cancel*. If the user answers *Yes*, the program can save the file and then return a nonzero value to the WM_QUERYENDSESSION message. If the user answers *No*, the program can return a nonzero value without saving the file. But if the user answers *Cancel*, the program should return a zero value so that the Windows session will not be ended. If a program does not process the WM_QUERYENDSESSION message, DefWindowProc returns a nonzero value.

When a user selects *Close* from the system menu of a particular instance of an application, rather than from the MS-DOS Executive's menu, Windows sends the window function a WM_CLOSE message. If the program has an unsaved file loaded, it can query the user with a message box—possibly the same one displayed when WM_QUERYENDSESSION is processed. If the user responds *Yes* to the query, the program can save the file and then call DestroyWindow. If the user responds *No*, the program can call DestroyWindow without saving the file. If the user responds *Cancel*, the window function does not call DestroyWindow and the program will not be terminated. If a program does not process WM_CLOSE messages, DefWindowProc calls DestroyWindow.

Finally, a window function can send messages to other window functions, either within the same program or in other programs, with the Windows SendMessage function. This function returns control to the calling program after the message has been processed. A program can also place messages in a program's message queue with the PostMessage function. This function returns control immediately after posting the message.

For example, when a program makes changes to the WIN.INI file (a file containing Windows initialization information), it can notify all currently running instances of these changes by sending them a WM_WININICHANGE message:

```
SendMessage (-1, WM_WININICHANGE, 0, 0L) ;
```

The -1 parameter indicates that the message is to be sent to all window functions of all currently running instances. Windows calls the window functions with the WM_WININICHANGE message and then returns control to the program that sent the message.

SAMPLE's message processing

The SAMPLE program shown in Figure 17-11 processes only four messages: WM_COMMAND, WM_SIZE, WM_PAINT, and WM_DESTROY. All other messages are passed to DefWindowProc. As is typical with most Windows programs written in C, SAMPLE uses a switch and case construction for processing messages.

The WM_COMMAND message signals the program that the user has selected a new font from the menu. SAMPLE first obtains a handle to the menu and removes the checkmark from the previously selected font:

```
hMenu = GetMenu (hWnd) ;
CheckMenuItem (hMenu, nCurrentFont, MF_UNCHECKED) ;
```

The value of *wParam* in the WM_COMMAND message is the menu ID of the newly selected font. SAMPLE saves that value in a static variable (nCurrentFont) and then places a checkmark on the new menu choice:

```
nCurrentFont = wParam ;
CheckMenuItem (hMenu, nCurrentFont, MF_CHECKED) ;
```

Because the typeface has changed, SAMPLE must repaint its display. The program does not repaint it immediately, however. Instead, it calls the InvalidateRect function:

```
InvalidateRect (hWnd, NULL, TRUE) ;
```

This causes a WM_PAINT message to be placed in the program's message queue. The NULL parameter indicates that the entire client area should be repainted. The TRUE parameter indicates that the background should be erased.

The WM_SIZE message indicates that the size of SAMPLE's client area has changed. SAMPLE simply saves the new dimensions of the client area in two static variables:

```
xClient = LOWORD (lParam) ;
yClient = HIWORD (lParam) ;
```

The LOWORD and HIWORD macros are defined in WINDOWS.H.

Windows also places a WM_PAINT message in SAMPLE's message queue when the size of the client area has changed. As is the case with WM_COMMAND, the program does not have to repaint the client area immediately, because the WM_PAINT message is in the message queue.

SAMPLE can receive a WM_PAINT message for many reasons. The first WM_PAINT message it receives results from calling UpdateWindow in the WinMain function. Later, if the current font is changed from the menu, the program itself causes a WM_PAINT message to be placed in the message queue by calling InvalidateRect. Windows also sends a window function a WM_PAINT message whenever the user changes the size of the window or when part of the window previously covered by another window is uncovered.

Programs begin processing WM_PAINT messages by calling BeginPaint:

```
BeginPaint (hWnd, &ps) ;
```

The SAMPLE program then creates a font based on the current size of the client area and the current typeface selected from the menu:

```
hFont = CreateFont (yClient, xClient / 8,
                    0, 0, 400, 0, 0, 0, OEM_CHARSET,
                    OUT_STROKE_PRECIS, OUT_STROKE_PRECIS,
                    DRAFT_QUALITY, (BYTE) VARIABLE_PITCH |
                    cFamily [nCurrentFont - IDM_SCRIPT],
                    szFace  [nCurrentFont - IDM_SCRIPT]) ;
```

The font is selected into the device context (a data structure internal to Windows that describes the characteristics of the output device); the program also saves the original device-context font:

```
hFont = SelectObject (ps.hdc, hFont) ;
```

And the word *Windows* is displayed:

```
TextOut (ps.hdc, 0, 0, "Windows", 7) ;
```

The original font in the device context is then selected, and the font that was created is now deleted:

```
DeleteObject (SelectObject (ps.hdc, hFont)) ;
```

Finally, SAMPLE calls EndPaint to signal Windows that the client area is now updated and valid:

```
EndPaint (hWnd, &ps) ;
```

Although the processing of the WM_PAINT message in this program is simple, the method used is common to all Windows programs. The BeginPaint and EndPaint functions always occur in pairs, first to get information about the area that needs repainting and then to mark that area as valid.

SAMPLE will display this text even when the program is minimized to be displayed as an icon at the bottom of the screen. Although most Windows programs use a customized icon for this purpose, the window-class structure in SAMPLE indicates that the program's icon is NULL, meaning that the program is responsible for drawing its own icon. SAMPLE does not, however, make any special provisions for drawing the icon. To it, the icon is simply a small client area. As a result, SAMPLE displays the word *Windows* in its "icon," using a small font size.

Windows sends the window function the WM_DESTROY message as a result of the DestroyWindow function that DefWindowProc calls when processing a WM_CLOSE message. The standard processing involves placing a WM_QUIT message in the message queue:

```
PostQuitMessage (0) ;
```

When the GetMessage function retrieves WM_QUIT from the message queue, GetMessage returns 0. This terminates the message loop and the program.

For all other messages, SAMPLE calls DefWindowProc and exits the window function by returning the value from the call:

```
return DefWindowProc (hWnd, iMessage, wParam, lParam) ;
```

This allows Windows to perform default processing on the messages SAMPLE ignores.

Windows' multitasking

Most operating systems or operating environments that allow multitasking use what is called a preemptive scheduler. Generally, the procedure involves use of the computer's clock to switch rapidly between programs and allow each a small time slice. When switching between programs, the operating system must preserve the machine state.

Windows is different. It is a nonpreemptive multitasking environment. Although Windows allows several programs to run simultaneously, it never switches from one program to allow another to run. It switches between programs only when the currently running program calls the GetMessage function or the related PeekMessage and WaitMessage functions.

When a Windows program calls GetMessage and the program's message queue contains a message other than WM_PAINT or WM_TIMER, Windows returns control to the program with the next message. However, if the program's message queue contains only a WM_PAINT or WM_TIMER message and another program's queue contains a message other than WM_PAINT or WM_TIMER, Windows returns control to the other program, which is also waiting for its GetMessage call to return.

(Windows also switches between programs temporarily when a program uses SendMessage to send a message to a window function in another program, but control returns to the calling program after the window function has processed the message sent to it.)

To cooperate with Windows' nonpreemptive multitasking, programmers should try to perform message processing as quickly as possible. Programs can, for example, split a large amount of processing into several smaller pieces to allow other programs to run in the interval. During long processing a program can also periodically call PeekMessage to allow other programs to run.

Graphics Device Interface

Programs receive input through the Windows message system. For program output, Windows provides a device-independent interface to graphics output devices, such as the video display, printers, and plotters. This interface is called the Graphics Device Interface, or GDI.

The device context (DC)

When a Windows program needs to send output to the video screen, the printer, or another graphics output device, it must first obtain a handle to the device's device context, or DC. Windows provides a number of functions for obtaining this device-context handle:

BeginPaint. Used for obtaining a video device-context handle during processing of a WM_PAINT message. This device context applies only to the rectangular section of the client area that is invalid (needs repainting). This region is also a clipping region, meaning that a program cannot paint outside this rectangle. BeginPaint fills in the fields of a PAINTSTRUCT structure. This structure contains the coordinates of the invalid rectangle and a byte that indicates if the background of the invalid rectangle has been erased.

GetDC. Generally used for obtaining a video device-context handle during processing of messages other than WM_PAINT. The handle obtained with this function references only the client area of the window.

GetWindowDC. Used for obtaining a video device-context handle that encompasses the entire window, including the title bar, menu bar, and scroll bars. A Windows program can use this function if it is necessary to paint over areas of the window outside the client area.

CreateDC. Used for obtaining a device-context handle for the entire display or for a printer, a plotter, or other graphics output device.

CreateIC. Used for obtaining an information-context handle, which is similar to a device-context handle but can be used only for obtaining information about the output device, not for drawing.

CreateCompatibleDC. Used for obtaining a device-context handle to a memory device context compatible with a particular graphics output device. This function is generally used for transferring bitmaps to a graphics output device.

CreateMetaFile. Used for obtaining a metafile device-context handle. A metafile is a collection of GDI calls encoded in binary form.

The Windows program uses the device-context handle when calling GDI functions. In addition to drawing, the various GDI functions can change the attributes of the device context, select different drawing objects (such as pens and fonts) into the device context, and determine the characteristics of the device context.

Device-independent programming

Windows supports such a wide variety of video displays, printers, and plotters that programs cannot make assumptions about the size and resolution of the device. Furthermore, because the user can generally alter the size of a program's window, the program must be able to adjust its output appropriately. The SAMPLE program, for example, showed how the window function can use the WM_SIZE message to obtain the current size of a window to create a font that fits text within the window's client area.

Programs can also use other Windows functions to determine the physical characteristics of a device. For instance, a program can use the GetDeviceCaps function to obtain

information about the device context, including the resolution of the device, its physical dimensions, and its relative pixel height and width.

Then, too, the GetTextMetrics function returns information about the current font selected in the device context. In the default device context, this is the system font. Many Windows programs base the size of their display output on the size of a system-font character.

Device-context attributes

The device context includes attributes that define how the graphics output functions work on the device. When a program first obtains a handle to a device context, Windows sets these attributes to default values, but the program can change them. Some of these device-context attributes are as follows:

Pen. Windows uses the current pen for drawing lines. The default pen produces a solid black line 1 pixel wide. A program can change the pen color, change to a dotted or dashed line, or make the pen draw a solid line wider than 1 pixel.

Brush. Windows uses the current brush (sometimes called a pattern) for filling areas. A brush is an 8-pixel-by-8-pixel bitmap. The default brush is solid white. Programs can create colored brushes, hatched brushes, and customized brushes based on bitmaps.

Background color. Windows uses the background color to fill the spaces in and between characters when drawing text and to color the open areas in hatched brushstrokes and dotted or dashed pen lines. Windows uses the background color only if the background mode (another attribute of the display context) is opaque. If the background mode is transparent, Windows leaves the background unaltered. The default background color is white.

Text color. Windows uses this color for drawing text. The default is black.

Font. Windows uses the font to determine the shape of text characters. The default is called the system font, a fixed-pitch font that also appears in menus, caption bars, and dialog boxes.

Additional device-context attributes (such as mapping modes) are described in the following sections.

Mapping modes

Most GDI drawing functions in Windows have parameters that specify the coordinates or size of an object. For instance, the Rectangle function has five parameters:

```
Rectangle (hDC, x1, y1, x2, y2) ;
```

The first parameter is the handle to the device context. The others are

- *x1*: horizontal coordinate of the upper left corner of the rectangle.
- *y1*: vertical coordinate of the upper left corner of the rectangle.
- *x2*: horizontal coordinate of the lower right corner of the rectangle.
- *y2*: vertical coordinate of the lower right corner of the rectangle.

In the Rectangle and most other GDI functions, coordinates are logical coordinates, which are not necessarily the same as the physical coordinates (pixels) of the device. To translate logical coordinates into physical coordinates, Windows uses the current mapping mode.

In actuality, the mapping mode defines a transformation of coordinates between a window, which is defined in terms of logical coordinates, and a viewport, which is defined in terms of physical coordinates. For any mapping mode, a program can define separate window and viewport origins. The logical point defined as the window origin is then mapped to the physical point defined as the viewport origin. For some mapping modes, a program can also define window and viewport extents, which determine how the logical coordinates are scaled to the physical coordinates.

Windows programs can select one of eight mapping modes. The first six are sometimes called fully constrained, because the ratio between the window and viewport extents is fixed and cannot be changed.

In MM_TEXT, the default mapping mode, coordinates on the x axis increase from left to right, and coordinates on the y axis increase from the top downward. In the other five fully constrained mapping modes, coordinates on the x axis also increase from left to right, but coordinates on the y axis increase from the bottom upward. The six fully constrained mapping modes are

- *MM_TEXT*: Logical coordinates are the same as physical coordinates.
- *MM_LOMETRIC*: Logical coordinates are in units of 0.1 millimeter.
- *MM_HIMETRIC*: Logical coordinates are in units of 0.01 millimeter.
- *MM_LOENGLISH*: Logical coordinates are in units of 0.01 inch.
- *MM_HIENGLISH*: Logical coordinates are in units of 0.001 inch.
- *MM_TWIPS*: Logical coordinates are in units of $\frac{1}{1440}$ inch. (These units are $\frac{1}{20}$ of a typographic point, which is approximately $\frac{1}{72}$ inch.)

The seventh mapping mode is called partially constrained, because a program can change the window and viewport extents but Windows adjusts the values to ensure that equal horizontal and vertical logical coordinates translate to equal horizontal and vertical physical dimensions:

- *MM_ISOTROPIC*: Logical coordinates represent the same physical distance on both the x and y axes.

The MM_ISOTROPIC mapping mode is useful for drawing circles and squares. The MM_LOMETRIC, MM_HIMETRIC, MM_LOENGLISH, MM_HIENGLISH, and MM_TWIPS mapping modes are also isotropic, because equal logical coordinates map to the same physical dimensions on both axes.

The final mapping mode is sometimes called unconstrained because a program is free to set different window and viewport extents on the x and y axes.

- *MM_ANISOTROPIC*: Logical coordinates are mapped to arbitrarily scaled physical coordinates.

Functions for drawing

Windows includes several functions that programs can use to draw in the client area of a window. The most common of these functions are

SetPixel. Sets a point to a particular color.

LineTo. Draws a line from the current position to a point specified in the LineTo function. The current position is defined in the device context and can be altered before the call to LineTo with the MoveTo function, which changes the current position but does not draw anything. Windows uses the current pen and the current drawing mode (*see* below) for drawing the line.

Polyline. Draws multiple lines much like a series of LineTo calls but does not alter the current position on completion.

Rectangle. Draws a filled rectangle with a border. Parameters to the Rectangle function specify the coordinates of the upper left and lower right corners of the rectangle. Windows draws the border of the rectangle with the current pen and current drawing mode defined in the device context, just as if it were using the Polyline function then Windows fills the rectangle with the current brush defined in the device context.

Ellipse. Uses the same parameters as Rectangle but draws an ellipse within the rectangular area.

RoundRect. Draws a rectangle with rounded corners. Two parameters to this function define the height and width of an ellipse that Windows uses for drawing the rounded corners.

Polygon. Draws a polygon connecting a series of points and fills the enclosed areas in either an alternate or winding mode. The winding mode causes Windows to fill every area within the polygon. The alternate mode fills every other area. For a polygon that defines a five-pointed star, for instance, the center is filled if the mode is winding but is not filled if the mode is alternate.

Arc. Draws a curved line that is part of the circumference of an ellipse.

Chord. Similar to the Arc function, but Windows connects the beginning and ending points of the arc with a straight line. The area is filled with the current brush defined in the device context.

Pie. Similar to the Arc function, but Windows draws lines from the beginning and ending points of the arc to the center of the ellipse. The area is filled with the current brush defined in the device context.

TextOut. Writes text with the current font, text color, background color, and background mode (transparent or opaque).

Windows also includes other drawing functions for filling areas, formatting text, and transferring bitmaps.

Raster operations for pens

When Windows uses a pen to write to a device context, it must first determine which pixels of the destination are to be altered by the pen (the foreground) and which pixels will not be affected (the background). With dotted and dashed pens, the background—the space between the dots or dashes—is left unaltered if the drawing mode is transparent and is filled with the background color if the drawing mode is opaque.

When Windows alters the pixels of the destination that correspond to the foreground of the pen, the most obvious result is that the color of the current pen defined in the display context is used to color the destination. But this is not the only possible result. Windows also generalizes the process by using a logical operation to combine the pixels of the pen and the pixels of the destination.

This logical operation is defined by the drawing mode attribute of the device context. This drawing mode can be set to one of 16 binary raster operations (abbreviated ROP2).

The following table shows the 16 binary raster operation codes defined in WINDOWS.H. The column headed "Resultant Destination" shows how the destination changes, depending on the bit pattern of the pen and the bit pattern of the destination before the line is drawn. The words OR, AND, XOR, and NOT are the logical operations.

Binary Raster Operation	Resultant Destination
R2_BLACK	0
R2_COPYPEN	pen
R2_MERGEPEN	pen OR destination
R2_MASKPEN	pen AND destination
R2_XORPEN	pen XOR destination
R2_NOTCOPYPEN	NOT pen
R2_NOTMERGEPEN	NOT (pen OR destination)
R2_NOTMASKPEN	NOT (pen AND destination)
R2_NOTXORPEN	NOT (pen XOR destination)
R2_MERGEPENNOT	pen OR (NOT destination)
R2_MASKPENNOT	pen AND (NOT destination)
R2_MERGENOTPEN	(NOT pen) OR destination
R2_MASKNOTPEN	(NOT pen) AND destination
R2_NOP	destination
R2_NOT	NOT destination
R2_WHITE	1

The default drawing mode defined in a device context is R2_COPYPEN, which simply copies the pen to the destination. However, if the pen color is blue, the destination is red, and the drawing mode is R2_MERGEPEN, then the drawn line appears as magenta, which

results from combining the pen and destination colors. If the pen color is blue, the destination is red, and the drawing mode is R2_NOTMERGEPEN, then the drawn line is green, because the blue pen and the red destination are combined into magenta, which Windows then inverts to make green.

Bit-block transfers

Windows also uses logical operations when transferring a rectangular pixel pattern (a bit block) from one device context to another or from one area of a device context to another area of the same device context.

While line drawing involves a logical combination of two sets of pixels (the pen and the destination), the bit-block transfer functions perform a logical combination of three sets of pixels: a source bitmap, a destination bitmap, and the brush currently selected in the destination device context. As shown in the preceding section, there are 16 different ROP2 drawing modes for all the possible combinations of two sets of pixels. The tertiary raster operations (abbreviated ROP3) for bit-block transfers require 256 different operations for all possible combinations.

Windows defines three functions for transferring rectangular pixel patterns: BitBlt (bit-block transfer), StretchBlt (stretch-block transfer), and PatBlt (pattern-block transfer). Of these three functions, StretchBlt is the most generalized. StretchBlt transfers a bitmap from a source device context to a destination device context. Function parameters specify the origin, width, and height of the bitmap. If the source and destination widths and heights are different, Windows stretches or compresses the bitmap appropriately. Negative values of widths and heights cause Windows to draw a mirror image of the bitmap.

The BitBlt function transfers a bitmap from a source device context to a destination device context, but the width and height of the source and destination must be the same. If the source and destination device contexts have different mapping modes, Windows uses StretchBlt instead.

In both BitBlt and StretchBlt, Windows performs a bit-by-bit logical operation with the bit block in the source device context, the bit block in the destination area of the destination device context, and the brush currently selected in the destination device context. Although Windows supports all 256 possible raster operations with these three bitmaps, only a few have been given WINDOWS.H identifiers:

Raster Operation	Resultant Destination
BLACKNESS	0
SRCCOPY	source
SRCAND	source AND destination
SRCPAINT	source OR destination

(more)

Raster Operation	Resultant Destination
SRCINVERT	source XOR destination
SRCERASE	source AND (NOT destination)
MERGEPAINT	source OR (NOT destination)
NOTSRCCOPY	NOT source
NOTSRCERASE	NOT (source OR destination)
DSTINVERT	NOT destination
PATCOPY	pattern
MERGECOPY	source AND pattern
PATINVERT	destination XOR pattern
PATPAINT	source OR (NOT destination) OR pattern
WHITENESS	1

The PatBlt function is similar to BitBlt and StretchBlt but performs a logical operation only between the currently selected brush and a destination area of the device context. Thus, only 16 raster operations can be used with PatBlt; these are equivalent to the binary raster operations used with line drawing.

Text and fonts

Windows supports file-based text fonts in two different formats: raster and vector. The raster fonts, such as Courier, Helvetica, and Times Roman, are defined by digital representations of the bit patterns of the characters. Font files usually contain several different sizes for each typeface. The vector fonts, such as Modern, Script, and Roman, are defined by points that are connected to form the letters and can be scaled to different sizes.

When using a device such as a printer, which has built-in fonts, Windows can also use these device-based fonts.

To specify a font, a Windows program uses the CreateFont function to create a logical font—a detailed description of the desired font. When this logical font is selected into a device context, Windows finds the actual font that best fits this description. In many cases, this match is not exact. The program can then call GetTextMetrics to determine the characteristics of the actual font that the device will use to display text.

Windows supports both fixed-width and variable-width fonts, as well as such attributes as italics, underlining, and boldfacing. Programs can also justify text with the GetTextExtent call, which obtains the width of a particular text string. The program can then insert extra spaces between words with SetTextJustification or it can insert extra spaces between letters with SetTextCharacterExtra.

Metafiles

As explained earlier, a metafile is a collection of GDI function calls stored in a binary coded form. A program can create a metafile by calling CreateMetaFile and giving it either

an MS-DOS filename or NULL as a parameter. If CreateMetaFile is given an MS-DOS filename, Windows creates a disk-based metafile; if the parameter is NULL, Windows creates a metafile in memory. The CreateMetaFile call returns a handle to a metafile device context. Any GDI calls that reference this device-context handle become part of the metafile.

When the program calls CloseMetaFile, Windows closes the metafile device context and returns a handle to the metafile. The program can then "play" this metafile on another device context (such as the video display) without calling the GDI functions directly.

Metafiles provide a useful way to transfer device-independent pictures between programs.

Data Sharing and Data Exchange

Windows includes a variety of methods by which programs can share and exchange data. These methods are discussed in the following sections.

Sharing local data among instances

Multiple instances of the same program can share data in the static data area of the program's data segment. Later instances of a program can thus call GetInstanceData and copy configuration options established by the user in the first instance. Multiple instances of programs can also share resources, such as dialog-box templates.

The Windows Clipboard

The Windows Clipboard is a general-purpose mechanism that allows a user to transfer data from one program to another. Programs that support the Clipboard generally include a top-level menu item called *Edit,* which invokes a pop-up menu that offers at least these three options:

- *Cut*: Copies the current selection to the Clipboard and deletes the selection from the current program file.
- *Copy*: Copies the current selection to the Clipboard without deleting the selection from the current program file.
- *Paste*: Copies the contents of the Clipboard to the current program file.

The Clipboard can hold only one item at a time. A program can transfer data to the Clipboard through the function call SetClipboardData. With this function, the program passes the Clipboard a handle to a global memory block, which then becomes the property of the Clipboard. A program can access Clipboard data through the complementary function GetClipboardData.

The Clipboard supports several formats:

- Text: ASCII text; each line ends with a carriage return and linefeed, and the text is terminated with a NULL character.
- Bitmap: A collection of bits in the GDI bitmap format.

- Metafile Picture: A structure that contains a handle to a metafile along with other information suggesting the mapping mode and aspect ratio of the picture.
- SYLK: Microsoft's Symbolic Link format.
- DIF: Software Arts' Data Interchange Format.

Programs can also use the Clipboard for storing data in private formats.

Some programs, such as the CLIPBRD program included with Windows, can also become Clipboard viewers. Such programs receive a message whenever the contents of the Clipboard change.

Dynamic Data Exchange (DDE)

Dynamic Data Exchange (DDE) is a protocol that cooperating programs can use to exchange data without user intervention. DDE makes use of the facilities in Windows that enable programs to send messages among themselves.

In DDE, the program that needs data from another program is called the client. The client sends a WM_DDE_INITIATE message either to a dedicated server program or to all currently running programs. Parameters to the WM_DDE_INITIATE message are *atoms*, which are numbers referring to text strings. A server application that has the data the client needs sends a WM_DDE_ACK message back to the client. The client can then be more specific about the data it needs by sending the server a WM_DDE_ADVISE message. The server can then pass global memory handles to the client with the WM_DDE_DATA message.

Internationalization

Windows includes several features that ease the conversion and translation of programs for international markets. Among these features are keyboard drivers appropriate for many European languages and use of the ANSI character set, which provides a richer set of accented letters than does the character set resident in the IBM PC and compatibles.

Windows also includes several functions that assist in language-independent coding. The AnsiUpper and AnsiLower functions translate characters or strings to uppercase or lowercase in the full ANSI character set, rather than the more limited ASCII character set. In addition, the AnsiNext and AnsiPrev functions allow scanning of text strings that may contain 2 or more bytes per character.

Windows programmers can also help in program translation by defining all text strings used within the program as resources contained in the resource script file. Because the resource script file also contains menu templates and dialog-box templates, it thus becomes the only file that needs alteration when a foreign-language version of the program is created.

Charles Petzold

Part E
Programming Tools

Article 18
Debugging in the MS-DOS Environment

It is axiomatic that any program will need debugging at some time in its development cycle, and programs written to run under MS-DOS are no exception. This article provides an introduction to the debugging tools and techniques available to the serious programmer developing code in the MS-DOS environment. Space does not permit a thorough investigation of the philosophy, psychology, and science of debugging computer programs; instead, a brief and practical discussion of the basic debugging approaches is presented, along with some rules-of-thumb for choosing the best approach. Nor are the details of every single utility and command included in this article; these are described in detail in the reference sections of this volume. The commands and utility programs that are most useful for debugging are discussed and illustrated with examples and case histories that also serve as models for the various debugging methods.

The reader of this article is assumed to be a programmer with sufficient experience to understand an assembly-language program. The reader is also assumed to be familiar with MS-DOS — terms like FCB and PSP are not explained. A reader without this background in MS-DOS need not be deterred, however; these terms are thoroughly explained elsewhere in this book. Besides assembly language, examples in this article are written in Microsoft QuickBASIC and Microsoft C. A detailed knowledge of these languages is not required; the examples are short and straightforward.

The reader should also keep in mind that the examples given here are real but not necessarily realistic. To avoid the tedium that accompanies debugging, the examples have been designed to reveal their bugs fairly quickly. All the methods and techniques shown are accurate in detail but not always in scale. Most of the debugging examples presented here would require one-half to one hour of work. It is possible for real debugging sessions to last for hours or days, especially if the wrong approach or tool is chosen. One of the purposes of this article is to help the programmer choose the correct tool and, thus, to reduce the tedium.

The Programs

There are more than a dozen listings in this article. Some of them are correct and others contain errors for use in illustrating debugging techniques. Many of the programs serve as examples in multiple sections of the article. The following summary of the programs (Table 18-1) is given to avoid confusion and to provide a common location to consult for explanations of the programs.

Table 18-1. Summary of Example Programs.

Name:	EXP.BAS
Figure:	18-1
Status:	Incorrect—do not use.
Purpose:	Computes EXP(x) (the exponential of x) to a specified precision using an infinite series.
Compiling:	QB EXP; LINK EXP;
Parameters:	Prompts for value for x and a convergence criterion. Enter zero to quit.

Name:	EXP.BAS
Figure:	18-3
Status:	Correct version of Figure 18-1.
Purpose:	Computes EXP(x) (the exponential of x) to a specified precision using an infinite series.
Compiling:	QB EXP; LINK EXP;
Parameters:	Prompts for value for x and a convergence criterion. Enter zero to quit.

Name:	COMMSCOP.ASM
Figure:	18-4
Status:	Correct.
Purpose:	Monitors the activity on a specified COM port and places a copy of all transmitted and received data in a RAM buffer. Each entry in the buffer is tagged to indicate whether the byte was sent by or received by the application program under test. Control is provided to start, stop, and resume tracing by means of a control interrupt. When tracing is stopped and resumed, a marker is left in the buffer. COMMSCOP is a terminate-and-stay-resident (TSR) program.
Compiling:	MASM COMMSCOP; LINK COMMSCOP; EXE2BIN COMMSCOP.EXE COMMSCOP.COM DEL COMMSCOP.EXE
Parameters:	Installed by entering *COMMSCOP*; no parameters for installation. The TSR is controlled by passing parameter data in registers with an Interrupt 60H call. The registers can have the following values:

AH:	Command:
00H	STOP
01H	FLUSH AND START
02H	RESUME TRACE
03H	RETURN TRACE BUFFER ADDRESS

(more)

DX:	COM port:
00H	COM1
01H	COM2

Interrupt 60H returns the following in response to function 3:

CX	Buffer count in bytes
DX	Segment address of buffer
BX	Offset address of buffer

Name: COMMSCMD.C
Figure: 18-5
Status: Correct.
Purpose: Controls the COMMSCOP program by issuing Interrupt 60H calls. C version.
COMPILING: MSC COMMSCMD;
LINK COMMSCMD;
Parameters: Commands are issued by
COMMSCMD [[*cmd*][*port*]]
where: *cmd* is the command to be executed:
STOP Stop trace
START Flush buffer and start trace
RESUME Resume a stopped trace
port is the COM port (1 = COM1, 2 = COM2)
If *cmd* is omitted, STOP is assumed; if *port* is omitted, 1 is assumed.

Name: COMMSCMD.BAS
Figure: 18-6
Status: Correct.
Purpose: Controls the COMMSCOP program by issuing Interrupt 60H calls. QuickBASIC version.
Compiling: QB COMMSCMD;
LINK COMMSCMD USERLIB;
Parameters: Commands are issued by
COMMSCMD [[*cmd*][,*port*]]
where: *cmd* is the command to be executed:
STOP Stop trace
START Flush buffer and start trace
RESUME Resume a stopped trace
port is the COM port (1 = COM1, 2 = COM2)
If *cmd* is omitted, STOP is assumed; if *port* is omitted, 1 is assumed.

Name COMMDUMP.BAS
Figure: 18-7
Status: Correct.
Purpose: Produces a formatted dump of the communications trace buffer.

(more)

Compiling:	QB COMMDUMP; LINK COMMDUMP USERLIB;
Parameters:	No parameters. When COMMDUMP is invoked, it formats and dumps the entire buffer.

Name:	TESTCOMM.ASM
Figure:	18-9
Status:	Incorrect — do not use.
Purpose:	Provides test data for the COMMSCOP routine.
Compiling:	MASM TESTCOMM; LINK TESTCOMM;
Parameters:	No parameters. TESTCOMM reads data from the keyboard and writes to COM1 and reads COM1 data and displays it on the screen. Ctrl-C cancels.

Name:	TESTCOMM.ASM
Figure:	18-10
Status:	Correct version of Figure 18-9.
Purpose:	Provides test data for the COMMSCOP routine.
Compiling:	MASM TESTCOMM; LINK TESTCOMM;
Parameters:	No parameters. TESTCOMM reads data from the keyboard and writes to COM1 and reads COM1 data and displays it on the screen. Ctrl-C cancels.

Name:	BADSCOP.ASM
Figure:	18-11
Status:	Incorrect version of Figure 18-4 — do not use.
Purpose:	Monitors the activity on a specified COM port and places a copy of all transmitted and received data in a RAM buffer. Each entry in the buffer is tagged to indicate whether the byte was sent by or received by the application program under test. Control is provided to start, stop, and resume tracing by means of a control interrupt. When tracing is stopped and resumed, a marker is left in the buffer. BADSCOP is a terminate-and-stay-resident (TSR) program.
Compiling:	MASM BADSCOP; LINK BADSCOP; EXE2BIN BADSCOP.EXE BADSCOP.COM DEL BADSCOP.EXE
Parameters:	Installed by entering *BADSCOP*; no parameters for installation. The TSR is controlled by passing parameter data in registers with an Interrupt 60H call. The registers can have the following values:

AH:	Command:
00H	STOP
01H	FLUSH AND START

(more)

02H	RESUME TRACE
03H	RETURN TRACE BUFFER ADDRESS
DX:	COM port:
00H	COM1
01H	COM2

Interrupt 60H returns the following in response to function 3:

CX	Buffer count in bytes
DX	Segment address of buffer
BX	Offset address of buffer

Name:	UPPERCAS.C
Figure:	18-13
Status:	Incorrect — do not use.
Purpose:	Converts a fixed string to uppercase and prints it.
Compiling:	MSC /Zi UPPERCAS; LINK UPPERCAS /CO;
Parameters:	No parameters.

Name:	UPPERCAS.C
Figure:	18-14
Status:	Correct version of Figure 18-13.
Purpose:	Converts a fixed string to uppercase and prints it.
Compiling:	MSC /Zi UPPERCAS; LINK UPPERCAS /CO;
Parameters:	No parameters.

Name:	ASCTBL.C
Figure:	18-16
Status:	Incorrect — do not use.
Purpose:	Displays a table of all displayable characters.
Compiling:	MSC /Zi ASCTBL; LINK ASCTBL /CO;
Parameters:	No parameters.

Name:	ASCTBL.C
Figure:	18-17
Status:	Correct version of Figure 18-16.
Purpose:	Displays a table of all displayable characters.
Compiling:	MSC /Zi ASCTBL; LINK ASCTBL /CO;
Parameters:	No parameters.