CODING DOS INTERRUPT HANDLERS IN C

David L. Doss and Bill E. Swafford
Applied Computer Science
Illinois State University
Normal, Illinois 61790-5150

ABSTRACT

This paper presents the technical details involved with writing specific DOS interrupt handlers in C and discusses related example student assignments. The specific interrupts discussed are the clock-tick and serial interrupts.

INTRODUCTION

Interrupt handlers have historically been viewed as very complex programs that must be written in assembly language by machine code gurus and must be kept private so that less experienced programmers will not cause problems by attempting to modify them. This has changed in recent years because of the personal computer and the availability of good C compilers that support the development of interrupt handlers. The advent of the single user PC has led to a better understanding of the interrupt mechanism by the average programmer by stimulating more experimentation. After all, there is minimum inconvenience if a single user computer is disabled if an experiment fails. Good C compilers have aided in this experimentation by allowing options that generate interrupt service routines, so that very low level services can be coded in a high level language.

Interrupt service routines can be utilized to enhance existing programming exercises and to create new opportunities to stimulate student enthusiasm for learning. Technical details, and in some cases entire subsystems, must be provided by the instructor. This paper discusses two very different applications for interrupts that can be included in student programming projects. The first application uses the clock-tick interrupt (1C) to set up a timed wait that can be terminated early by a single keystroke. The second application uses an interrupt handler to buffer input data from one of the serial ports.

The interruptable timed wait can be used in programming situations that require that a specific amount of time elapses before a program continues from the current state to the next state. It might, for example, be used in displaying a message or graphic at program initiation that will remain on the screen until either a fixed time interval has expired or a key is pressed. It can also be used to synchronize the CPU with the completion of a unit of work assigned to a peripheral device such as a scrolling display. Basic timing applications such as a digital clock can be easily developed from the ideas associated with the timed wait.

Serial interrupts can be used to develop a terminal emulator or a real time data acquisition system. Either of these applications provide students with a very real example of the producer-consumer model. The interrupt handler is the producer and the consumer is the terminal emulator or data acquisition program. Even if the interrupt handler and related service routines for initialization and retrieving data from the buffer must be given to the student, important ideas relating to mutual exclusion and process synchronization can be examined.

There are several good C compilers currently available for DOS platforms, including those from Borland and Microsoft, that support interrupt service routines. Details provided in this paper will reflect the Borland C compilers beginning with Turbo C version 2.0.

INTERRUPT SERVICE ROUTINES

Interrupt service routines must be constructed differently than regular C functions because of the circumstances in which they are activated. When an interrupt occurs, the currently executing instruction is allowed to complete and the flags, Code Segment (CS), and Program Counter(PC) registers are pushed onto the stack. Then the CS and PC registers are loaded from the interrupt vector table position that corresponds to the identity of the element that raised the interrupt. Consequently, the first instruction in the interrupt handler is the next instruction to execute. It is the responsibility of the interrupt handler to preserve the environment of the interrupted program so that it can be restarted with no ill effect from having been interrupted.

C compilers support interrupt handlers by generating additional code during program initiation and termination that save and restore essentially all CPU registers. They also generate a return from interrupt instruction instead of a normal return so that the flags register that was pushed by the interrupt mechanism will be restored. Interrupt routines coded in C must still be crafted carefully, but they can utilize powerful high level control structures.

THE CLOCK-TICK INTERRUPT

IBM's design of the original PC specified that interrupt 1C would be activated 18.2 times per second from the timer interrupt to provide timing services to application programs. Consequently, when DOS is initialized, the interrupt vector for this interrupt (1C) is set to point at a return from interrupt instruction. In order to utilize this timing feature, the programmer must reset the interrupt vector table entry corresponding to vector 1C to point at their own interrupt handler. After the required timing services are obtained, and before program termination, the vector table entry should be reset to its original value. If the vector is not reset to point at the return from interrupt instruction, disastrous results can occur, because whatever resides at the memory location vacated by the interrupt handler will be activated 18.2 times per second.

```
static int ticks, counter, tock;
void wait(int seconds)
/*
  Usage:  wait(10) - waits ten seconds
*/
{
  void interrupt (*oldtick)(void);
  counter = 0;
  ticks = 18;
  tock = 0;
  oldtick = getvect(0x1C); /* save old int vector */
  setvect(0x1C,mytick);    /* set new int vector */
  while( bioskey(1) == 0 && seconds > 0 )
    if( tock == 1 ) {
        seconds--;
        tock = 0;
```

```
    }
  setvect(0x1C,oldtick);   /* restore old vector */
}
void interrupt mytick(void)
/*
  Activated by interrupt mechanism
*/
{
  if( ++counter >= ticks ) {
    tock = 1;
    counter = 0;
  }
}
```

Figure 1.

In the example above (Figure 1), the function wait first initializes the variables required to count by seconds. It then saves the old clock-tick vector and sets vector 1C to point at the address of mytick. The functions getvect, setvect, and bioskey are provided by Borland. The function bioskey(1) returns a non zero value if a key has been pressed. Notice that the keyword interrupt is used to identify interrupt service routines.

After installing its own handler for interrupt 1C, wait loops until the required number of seconds expires or a key is pressed. The body of the loop recognizes the seconds that are counted by the interrupt handler mytick.

Setting and resetting interrupt vectors will need some discussion with students, as will the concept of interrupt 1C. However, once these details are discussed, good students are quite capable of producing simple timing related algorithms like the one expressed above. Whether timing algorithms are developed by students or provided by the instructor, exciting applications can then be developed.

SERIAL INTERRUPTS

The original IBM PC provided for an adapter board called the Asynchronous Communications Adapter that supported serial communications. This board was based on the INS 8250 UART, an already well understood chip, and was completely documented in technical reference material. Clone manufacturers utilized the same chip for their serial support facilities. Consequently, all PC clones, with rare and forgotten exceptions, can run the same serial communication software. The National Semiconductor 16550 UART is currently used in place of the much older 8250 for many high speed applications. The 16550 contains a FIFO buffer to reduce data loss for high speed transmissions. However, the newer chip accepts the same programming as the 8250.

The original design of the PC incorporated an Intel 8259 programmable interrupt controller to initially receive hardware interrupts. Based on prior programming, the chip arbitrates which interrupts are serviced by the CPU, and in what priority order. This device is attached to the I/O buss by two ports, the control port at 0x20, and the mask port at 0x21. The eight bit value in the mask port determines which interrupts are passed to the CPU. Any bit value of 1 in the mask causes the corresponding interrupt to be suppressed, while a 0 bit causes the interrupt to be raised. In order to activate interrupt driven serial capability without disturbing other programming, first read the current mask port value, then logically AND it with 0xEF for COM1 or 0xF7 for COM2, and rewrite it to the mask port. The only other

detail required for the 8259 is that at the completion of each interrupt, the value 0x20 must be written to the control port to signal end of interrupt so that the next interrupt will be accepted.

The 8250 is attached to the I/O buss at port addresses 0x3F8 through 0x3FF for COM1, and at port addresses 0x2F8 through 0x2FF for COM2. The complex connection is necessary in order to allow numerous details to be set and observed. Before a serial port can be used, it must be programmed. The program must initialize the chip for a specific bits-per-second (bps) rate, the number of data bits, the number of stop bits, and the parity scheme to use. Also, modem control settings must be specified, as well as under what conditions to generate interrupts. An interrupt can be generated when a data character is received or when it is transmitted. Our application will request that an interrupt be generated only when a character is received from the modem.

The following function (Figure 2) initializes the 8259 to allow interrupts generated by COM1, and initializes the 8250 for 9600 bps, eight data bits, even parity, and one stop bit. The program also requests the 8250 to assert the signals data terminal ready (DTR), request to send (RTS), and OUT2. The OUT2 signal is a technical detail required so that interrupts can be generated. Finally, the code specifies that an interrupt is to be generated when a byte of data is received from the modem. Complete details for all of the possible settings can be found in the literature identified in the bibliography.

```
void interrupt (*oldcomm)(void);
int whichcomm=12;
int highbyte=0;
int highbyte=6;
int port=0x3F8;

void intson(void)
/*
  This function initializes the communication port, installs
  the interrupt handler, and turns input interrupts on.

  Usage:  intson()
*/
{
  char mask;
  oldcomm = getvect(whichcom);/* save old comm vector */
  setvect(whichcom,newcomm); /* set new comm int handler */
  enable();                  /* enable interrupts */
  mask = inportb(0x21);      /* read 8259 mask register */
  mask &= 0xEF;              /* enable IRQ4 for COMM1 */
  outportb(0x21,mask);       /* write new mask */
  outportb(port+3,0x80);     /* request divisor latch */
  outportb(port,lowbyte);    /* set low byte of bps rate */
  outportb(port+1,highbyte); /* set high byte of bps rate */
  outportb(port+3,7);        /* set 8 bits & 1 stop */
  outportb(port+4,11);       /* set DTR, RTS, and OUT2 */
  outportb(port+1,1);        /* enable receive interrupts */
}
```

Figure 2.

The variables highbyte and lowbyte hold bps rate divisors that are used to determine the rate at which data is clocked through the 8250. A more complete table of possible bps rate divisors is given below (Figure 3). The variable whichcomm identifies interrupt vector 12, which is correct for COM1. The variable port identifies the base I/O address for COM1. Other necessary addresses are obtained by adding to this value. The functions enable, inportb, and outportb are all provided by Borland.

```
char divisors[]={1,128,   /*  300   bps */
                 0,192,   /* 600   bps */
                 0,96,    /* 1200  bps */
                 0,64,    /* 1800  bps */
                 0,58,    /* 2000  bps */
                 0,48,    /* 2400  bps */
                 0,32,    /* 3600  bps */
                 0,24,    /* 4800  bps */
                 0,16,    /* 7200  bps */
                 0,12,    /* 9600  bps */
                 0,6};    /* 19200 bps */
```

Figure 3.

After the communication is complete, it is necessary to tell the 8259 not to pass on any more interrupts and to reestablish the old interrupt vector. This can be accomplished by function given below (Figure 4).

```
void intsoff(void)
/*
  Usage:  intsoff()
*/
{
  char mask;
  disable();
  mask = inportb(0x21);      /* get 8259 mask */
  mask &= 0xE7;              /* disable COMM ints */
  outportb(0x21,mask);
  outportb(port+1,0);        /* disable 8250 ints */
  setvect(whichcom,oldcomm); /* restore old handler */
  enable();
}
```

Figure 4.

THE PRODUCER

Before the interrupt handler below (Figure 5) can be described, some discussion of the buffering technique should be given. A circular 4K queue or ring buffer will be employed that is considered empty when the front and rear pointers are equal.

```
char buffer[4096];   /* interrupt buffer */
int front, rear;     /* front and rear of queue */
int qlength=4096;    /* size of buffer */
int overs;           /* buffer overflow count */
```

```
int bcount;          /* buffer character count*/
int xonoff=FALSE;    /* flag for XON & XOFF logic */

void interrupt newcomm(void)
/*
  Activated by interrupt mechanism
*/
{
  char data;
  data = inportb(port);            /* get data byte */
  if( rear >= qlength-1 )          /* end of queue? */
    rear = 0;
  else
    rear++;
  if( rear == front ) {
    overs++;
       if( rear == 0 )
              rear = qlength -1;
       else
              rear--;
  }
  else {
    buffer[rear] = data;
    bcount++;
    if( bcount > qlength-256 ) {  /* almost full? */
              send(XOFF);            /* stop sender */
              xonoff = TRUE;         /* tell queout() */
    }
  }
 outportb(0x20,0x20);              /* satisfy 8259 */
}
```

Figure 5.

We use XON/XOFF flow control logic to guard against buffer overflow. When the buffer is within 256 bytes of full, an XOFF is sent out the port to ask the sending entity to wait for an XON before sending more data. The corresponding XON will be sent by the function that removes data from the buffer. The variable overs is used to count the number of characters that are received for which the buffer is full. It can be monitored while testing to see if the buffer is large enough.

The function send used by the interrupt handler to send the XOFF signal uses polling to transmit data through the serial port. It is given below (Figure 6).

```
#define xmit_buf_empty   ( inportb(port2+5) & 0x20 )
#define send_data(value) ( outportb(port2,value) )

void send(char c)
/*
  This function uses polling to send a
  character out the communication port.

  usage: send(c);
```

```
*/
{
  while( !xmit_buf_empty )     ; /* wait till it is */
  send_data(c);                  /* send next char */
}
```

Figure 6.

THE CONSUMER

The consumer of the items produced by the interrupt handler might be a program that in a loop alternately checks the keyboard and the buffer for data. If a key has been pressed, it is read and then transmitted out the serial port, and if there is data in the buffer, a character is retrieved from the buffer and displayed on the console. A very simple version of this logic is given below (Figure 7). Because getchar is used to obtain keyboard data, this data will be merged with data coming in from the serial port. In this simple example, if CTRL-Z is entered at the keyboard, the consumer terminates. This necessitates the call to the function intsoff so that communication interrupts are disabled and the old interrupt handler address is reinstalled in the interrupt vector table.

```
void main()
{
  char x;
  while( 1 ) {
    if( !empty() )
      putchar(queout());
    if ( bioskey(1) == 0 )
  continue;
    if( (x=getchar()) == EOF )
      break;
    send(x);
  }
  intsoff();
}
```

Figure 7.

The functions empty and queout are the usual queue maintenance routines that their names suggest. The function queout also checks to see if the buffer is more than half empty. If it is, XON is sent out the serial port to request the sending entity to continue transmission. Please note that queout should not be called if the buffer is empty. Both functions are given below (Figure 8).

```
int empty(void)
/*
  Usage:  if( !empty() ) c = queout()
*/
{
  return (front == rear ? 1 : 0);
}
int queout(void)
```

```
/*
  Usage:  c = queout()
*/
{
  int data;
  disable();
  if( front >= qlength-1 )
    front = 0;
  else
front++;
  data = buffer[front];
  bcount--;
  if( xonoff )
    if( bcount < qlength/2 ) {
                send(XON);
                xonoff = FALSE;
        }
        enable();
        return data;
}
```

Figure 8.

MUTUAL EXCLUSION AND SYNCHRONIZATION

The important topic of mutual exclusion is clearly exposed in the above example, particularly in the consumer function queout where interrupts are disabled while the buffer is manipulated. The corresponding mutual exclusion in the producer interrupt handler is not immediately clear. However, since it is the interrupt handler, the consumer agent queout cannot interrupt it.

Synchronization of the producer and consumer processes is achieved with the function empty. The consumer removes data from the buffer only when the buffer is not empty. The model is not completely perfect because of the possibility that incoming data might be lost if the buffer is full. The XON-XOFF flow control logic improves the chances that data will not be lost, and the variable overs can be used to ascertain if data has been lost. The size of the buffer can be modified in an effort to eliminate lost data. Also, changing the bps rate might solve a data loss problem.

CONCLUSION

Even though interrupt handlers are complex, difficult to develop, and almost impossible to debug, students find them stimulating. They are quite willing to work harder than usual in order to be able to learn how to use interrupt handlers to enhance otherwise ordinary computing applications. The clock-tick interrupt is relatively simple to access and is safe to use in experiments. It can be used to introduce sophisticated timing into applications and can even be used to add a digital clock to a display. Serial interrupts are much more difficult to understand because there are many more variables that must be set correctly in order to achieve success. However, the magnitude of the applications that can be accomplished using serial interrupts entices students to make the extra effort and learn proportionally more. Coding interrupt handlers in C makes them easily accessible to students.

BIBLIOGRAPHY

Campbell, J. (1986). Crafting C Tools for the IBM PC. Englewood Cliffs: Prentice-Hall.

Campbell, J. (1987). C Programmer's Guide to Serial Communications. Indianapolis: Howard Sams.

Doss, D. & Swafford, B. (1990, October). Birth of an Emulator. DG Review, pp. 41-44.

Duncan, R. (1988). Advanced MS DOS Programming (2nd ed.). Redmond: Microsoft.

Hunt, W. (1985). The C Toolbox. Reading: Reading.

IBM Corporation. (1983). Technical Reference for the IBM Personal Computer. White Plains: IBM.

Swafford, B. (1988 November). PC-Based Communication Using Interrupts, Micro/Systems Journal, pp. 50-54.

Young, M. (1988). Systems Programming in Turbo C. Almedia: SYBEX.