Using serial interfaces and the C language for real-time experiments

PHILLIP L. EMERSON

Cleveland State University, Cleveland, Ohio

The use of the serial interface of the IBM PC for timing and stimulus-response signaling is discussed. C language example programs, including two small software utility modules that expedite the programming of real-time experiments using a serial interface, are presented that are compatible with Borland's Turbo C.

The programming language C is ideal for low-level custom programming of serial interfaces, which can be used for the timing and stimulus-response signaling needed for real-time experiments. Serial interfaces are used normally for serial communications among computers, terminals, modems, printers, and special research devices. Those traditional serial applications are not addressed here because they are well-known and software is readily available. It is not as well-known that serial interfaces also can serve the primary functions needed in reaction-time and conditioning experiments.

On the serial interface of the IBM PC, XT, or AT, there are two binary direct current output lines and three binary input lines. In addition, there is a highly accurate crystal oscillator that is programmable for the needed timing. The binary input and output lines can be accessed by simple Microsoft BASIC programs that have the port IN and OUT functions. The timing capabilities, however, are fully exploitable only by the use of hardware-interrupt programming, which cannot be done in BASIC. Turbo C is one implementation of C whose library contains routines for interrupt programming, and it is used in these examples.

Timing

The IBM PC serial interface uses a 1.8432-MHz crystal oscillator for the clock source in serial communications and an International Semiconductor 8250 UART for most of the logic and derived timing operations. The 8250 is programmable to reduce the 1.8432 MHz by different integer divisors, for the production of different baud rates.

Most useful for the purpose of generating periodic interrupts is the 8250 transmitting mechanism, which uses two registers: a holding register and a shift register. To send a serial byte, the IBM PC CPU sends the byte in parallel to the 8250, which accepts the byte into its holding register. If the shift register happens to be empty, the byte is immediately transferred from the holding register to the shift register, which performs the parallel-to-serial

conversion. One bit at a time is sent out from the shift register over the serial signal line. The holding register can be refilled with a new byte from the CPU while the shift register is still busy sending an old one. In that case, there is an automatic transfer of the new byte from the holding register to the shift register as soon as the shift register finishes sending the old byte. The shift register then begins immediately to send the new byte.

The CPU can read a status register of the 8250 to determine whether the holding register is empty (ready to receive a new byte), or whether the shift register is empty, or both. Moreover, the 8250 can be programmed to cause a hardware interrupt each time the holding register becomes empty.

The timing module. The capabilities mentioned above are used in the timing method that is programmed in Listing 1.¹ An interrupt handling routine, `timint()`, is activated each time the 8250 holding register becomes empty. When activated, the interrupt handling routine sends a new byte to the 8250 to initiate the next of the periodic interrupts. It also increments the software counter, `time_count`, and performs other minor chores before returning to the interrupted activity. The module of Listing 1 is intended to be linked with some task module, such as the test program shown in Listing 2.

The elements of the timing module that are accessed by a task module are the routines `time_enable()`, `time_disable()`, `time_count`, and `time_error`. The `control_C()` and `timint()` routines are invoked by interrupts, and not by direct calls. As shown, the timing module implements a process in which `time_count` is incremented periodically 100 times per second. The task program may reset `time_count` to zero for the beginning of the timing of the duration of an event. The value of `time_count` is the elapsed time since the last reset to zero. The maximum positive value of `time_count` is about 10^9 . If allowed to run past the maximum, which would take several days or weeks, the count would cycle through a negative series and back through zero.

The task program may set `time_error` to zero and then test it later to see if there has been some disturbance of the periodic incrementing of `time_count`. A transition of `time_error` to nonzero indicates that an interrupt was not

The author's mailing address is Department of Psychology, Cleveland State University, Cleveland, OH 44115.

dealt with promptly enough and that the value of `time__count` is incorrect by some unknown amount. After such an event, the value of `time__error` will remain nonzero until the task program resets it to zero.

As a general policy, the task program should call `time__enable()` slightly before it needs to do some timing, and should call `time__disable()` at the beginning of any lengthy phase in which timing is not needed. A programmed exit from the task program should never be done with timing enabled. If that happens, the serial port continues to generate interrupts, even though there is no longer a program in memory to handle them. The consequence in most cases is that all activity stops, and the system responds to nothing short of a hardware power reset. To avoid such mishaps, a call to `time__disable()` should precede any programmed exit from a task program.

Timing parameters. The constants in the seven `#define` directives at the top of Listing 1 can be modified to change two characteristics of the timer. Recompile is needed after such a change. The first four constants are for the selection of the first of the two standard serial interfaces on the IBM PC. For the selection of the second, rather than the first, the proper values of the constants are given in the comments on the corresponding lines.

The last three of the `#define` constants determine the elementary timing unit. The shown value of `COMBITS` makes the elementary counting rate $1/12$ the baud rate. Baud-rate selection is as follows: `LOBAUD` and `HIBAUD` are both 8-bit integers on the range from 0 to 255. The baud rate is given by the ratio $115200/(\text{LOBAUD} + 256\text{HIBAUD})$. Thus, with `LOBAUD` = 96 and `HIBAUD` = 0, the baud rate is 1200 and the rate is 100 counts/sec.

Timing performance. Using the test program shown in Listing 2, this timing method was repeatedly found to give errors of no more than .5 sec in 1 h, and those were probably due mostly to human timing errors in pressing the keys. The timer was found to work at rates up to 1,600 counts/sec, with no `time__error` warnings. As a general policy, though, the timer should be programmed to count no faster than is actually needed. Each of the periodic interrupts consumes a small amount of CPU time, and concurrent processes are slowed more as the counting rate is increased.

For rare cases in which a very high counting rate is needed, note that one code line in the `time__enable()` routine is commented out [`the outp(0x20,PRIORITY);` command]. The removal of those comment delimiters will make this statement an executable part of the subroutine. The effect is to modify the normal MS-DOS arrangement of interrupt priorities, and that effect seems to be harmless for ordinary operations. It places the serial-port interrupt at the highest priority, above those of the time-of-day clock, the keyboard, and disk operations. When that is done, the counting rate can be increased up to 9,600 counts/sec without the occurrence of `time__error` warnings. At such a rate, however, the CPU becomes almost saturated with the timing interrupts. Concurrent output to the screen becomes markedly slow and jerky, and there is danger that the time-of-day clock will lose time.

Binary Input and Output

The port address of the binary output register is `BASAD+4`, where `BASAD` is the primary (lowest) port address for the serial interface. The usable bits are (counting right to left from zero) bits 0 and 1. Bit 0 controls the voltage on the RS232 DB25 connector pin 20, whereas bit 1 controls that of pin 4.

The port address of the binary input register is `BASAD+6`. The usable bits are those numbered 4, 5, and 7. The states of these bits are determined by the voltages at DB25 pins 5, 6, and 8, respectively. This mapping is formulated in the `srpak.c` program module shown in Listing 3. DB25 pin 7 is the ground return line for all these signal lines.

The RS232 voltage levels are specified as +3 to +15 for the "high" state, and -3 to -15 for the "low" state. The actual magnitudes of the open-circuit voltages at the output pins are typically about 8, relative to the signal ground on DB25 pin 7. The available current is sufficient to drive a light-emitting diode in series with a 470- Ω resistor (Campbell, 1984). Most TTL devices can be driven safely by unmodified RS232 signals, but a diode can be installed to prevent the negative voltage from reaching the TTL input.

A bipolar voltage source is needed to drive the RS232 inputs. A pair of 9-V batteries provides an expedient source of power. The pair is connected in series, with the common connection supplying 0 V to DB25 pin 7, leaving the other two battery terminals for +9 V and -9 V sources. With a mechanical single-throw switch, then, a pair of resistors can be used to produce a positive voltage when the switch is closed, and negative when open.

Programming a Reaction-Time Experiment

The timing module of Listing 1 and the stimulus-response module of Listing 3 are separate files, and each can be used without the other, depending on the needs of the experiment to be programmed. Both are used in the programming of the following simple reaction-time experiment.

The subject starts each trial voluntarily when he/she feels ready, by pressing the first of two keys, `KEY(ONE)`. As soon as `KEY(ONE)` is pressed, a warning signal, `WARNING__SIGNAL`, goes on. Then nothing happens until a certain time, `FOREPERIOD`, elapses. When the `FOREPERIOD` has elapsed, another signal, `TRIGGER__SIGNAL`, goes on to indicate that the subject should now press the second key, `KEY(TWO)`, as quickly as possible. However, subjects sometimes anticipate and press `KEY(TWO)` while the `FOREPERIOD` is still in progress before `TRIGGER__SIGNAL` comes on. When such an anticipation happens, no reaction time is recorded, and the trial is restarted immediately.

If there is no anticipation, the trial proceeds with the timer reset to zero as `TRIGGER__SIGNAL` goes on. The time from the onset of `TRIGGER__SIGNAL` until `KEY(TWO)` is pressed is the recorded reaction time for that trial. However, it can happen, under some circumstances, that `KEY(TWO)` is never pressed within a

reasonable time. Therefore, a time limit, MAX__TIME, is specified as part of the design. If the elapsed time reaches MAX__TIME before KEY(TWO) is pressed, the trial is terminated and MAX__TIME is recorded in place of the actual reaction time.

The timpak.c module provides for the timing, and srpak.c for the signals. A less crucial tool is introduced in Listing 4. The rtjargon.h file is used to translate some of the reaction-time terminology used above to identifiers expected by the more abstract formulations of timpak.c, srpak.c, and the C language itself. A different jargon file can be written for a different class of experiments.

The program in Listing 5 implements the reaction-time experiment. It assumes that the time unit is .01 sec and that 20 reaction-time trials are to be conducted according to the pattern just described. There is output to the console screen on each trial so that the experimenter can monitor the progress. At the end of the 20 trials, summary output goes to the screen. With MS-DOS command-line options, however, the summary output can be redirected to a disk file or printer, or even piped directly to some analysis program such as those in Perlman and Horan's (1986) statistical package. The use of fprintf(stderr,...) ensures that the screen is the destination of the trialwise output, even if redirection or piping is used for the final summary output.

Conclusion

If an available IBM PC or compatible clone has no asynchronous serial interface, one can be purchased for

little more than the price of an RS232 cable. The greatest expense in implementing this method of real-time control and data acquisition is the Turbo C compiler, which is available for little more than the price of two or three RS232 cables.

The timpak.c and srpak.c modules harness many of the useful capabilities of a serial interface for nonserial applications in real-time experiments. In most cases, such experiments could be programmed with accesses to the facilities of these modules, with no need for other low-level port accesses.

REFERENCES

- CAMPBELL, J. (1984). *The RS232 solution*. Alameda, CA: Sybex.
 KERNIGHAN, B. W., & RITCHIE, D. M. (1978). *The C programming language*. Englewood Cliffs, NJ: Prentice-Hall.
 PERLMAN, G., & HORAN, F. L. (1986). Report on UNIX/STAT release 5.1: Data analysis programs for UNIX and MSDOS. *Behavior Research Methods, Instruments, & Computers*, 18, 168-176.
 SARGENT, M., III, & SHOEMAKER, R. L. (1986). *The IBM PC from inside out*. Reading, MA: Addison Wesley.

NOTE

1. The most citable sources for the details are (1) the Turbo C documentation, (2) Kernighan and Ritchie (1978), (3) Sargent and Shoemaker (1986), (4) the Datalight C documentation (Version 3.01), and (5) Campbell (1984). Other important sources of information are various technical reference manuals and data sheets on MS-DOS, the IBM PC and AT, the International Semiconductor 8250 UART, and the Intel 8259 interrupt controller. In addition, I used informative tips from reviewers of an earlier manuscript, out of which this paper developed.

LISTING 1
The Timing Module, Using the First Serial Interface on an IBM PC

```

/* timpak.c */
#include <dos.h>
#define BASAD 0x3f8
#define INTNUM 0x0c
#define MASKIRQ 0xef
#define PRIORITY 0xc3
#define LOBAUD 96
#define HIBAUD 0
#define COMBITS 31
volatile long time_count ;
volatile int time_error ;
static int enabled = 0 ;
static int intmask ;
static void interrupt (*old_irqx)() ;
static void interrupt (*old_CTRLC)() ;

/* Turbo header for low-level IO <3> */
/* 1st serial port ; 0x2f8 for 2nd <1> */
/* irq4 vector; 0x0b for irq3 <1> */
/* irq4 mask ; 0xf7 for irq3 <1> */
/* irq4 to top ; 0xc2 for irq3 <1> */
/* 1200 baud, low byte <2> */
/* " , high byte <2> */
/* 8 data, 2 stop, 1 parity <2> */
/* Declare volatile counter. <3> */
/* Declare volatile error flag. <3> */
/* Keep track, enabled or not? */
/* Old 8259 interrupt mask. <4> */
/* Old irqx vector. <3> */
/* Old CTRL-C vector. <3> */

void time_disable() /* Routine to disable timing. */
{ if( !enabled ) return ; /* Do nothing if not enabled. */
  setvect(0x23,old_CTRLC) ; /* Restore old control-c vector. <3> */
  outportb(BASAD+1,0) ; /* Disable 8250 interrupts. <2> */
  setvect(INTNUM,old_irqx) ; /* Restore old serial vector. <3> */
  disable() ; /* Disable all interrupts. <3> */

```

LISTING 1 (Continued)

```

    outportb(0x21,intmask)      ;      /* Restore old 8259 int mask.      <4> */
    outportb(0x20,0xc7)        ;      /* Restore standard priorities.  <4> */
    enable()                    ;      /* Enable interrupts.           <3> */
    enabled = 0                  ;      /* Timing now disabled.         */
}

static void interrupt control_C() /* Replacement CTRL-C handler.    <3> */
{ time_disable() ; }            /* 2 CTRL-Cs needed to exit.    */

static void interrupt timint()    /* Replacement irqx handler.      <3> */
{ if( inportb(BASAD+2) & 2 )      /* Holding-register interrupt?    <2> */
    { time_error |= (inportb(BASAD+5) & 0x40) ; /* Shift-reg empty? <2> */
      outportb(BASAD,0)      ;      /* Send new byte.                 <2> */
      outportb(0x20,0x20)    ;      /* Issue non-specific EOI.        <4> */
      time_count++           ;      /* Increment counter.             */
    }
}

void time_enable()                /* Routine to enable timing.      */
{ if( enabled ) return           ; /* Do nothing if already enabled. */
  enabled = 1                    ; /* Timing will now be enabled.    */
  outportb(BASAD+1,0)            ; /* Disable 8250 interrupts.       <2> */
  outportb(BASAD+3,128)          ; /* Access baud rate divisors.     <2> */
  outportb(BASAD,LOBAUD)         ; /* Set baud rate, low byte.       <2> */
  outportb(BASAD+1,HIBAUD)       ; /*      "      high byte.         <2> */
  outportb(BASAD+3,COMBITS)      ; /* data, stop, parity.           <2> */
  old_CTRL_C = getvect(0x23)     ; /* Save old CTRL-C vector.        <3> */
  setvect(0x23,control_C)        ; /* Reroute ctrl-C.                <3> */
  old_irqx = getvect(INTNUM)     ; /* Save old irqx vector.          <3> */
  setvect(INTNUM,timint)         ; /* Reroute irqx.                  <3> */
  intmask = inp(0x21)            ; /* Save old 8259 mask.            <4> */
  disable()                      ; /* Disable all interrupts.        <3> */
  outportb(0x21,intmask & MASKIRQ) ; /* Unmask irqx.                   <4> */
/* outportb(0x20,PRIORITY) ; */ /* Rotate irqx to highest priority.<4> */
  enable()                      ; /* Enable interrupts.             <3> */
  outportb(BASAD+4,8)            ; /* Set out2 bit in modem-con. reg. <2> */
  outportb(BASAD+1,2)            ; /* Enable 8250 THRE interrupt.    <2> */
}

```

Note—A CTRL-C exit while timing is enabled requires that two CTRL-Cs be typed. There are two callable subroutines, `time_enable()` and `time_disable()`, and two accessible variables, `time_count` and `time_error`. The numeral in angle brackets in the comment indicates the primary source of information underlying the main operation of the program line, as follows: <1> IBM PC and MS-DOS technical reference manuals, <2> International Semiconductor 8250 data sheets, <3> Borland Turbo C documentation, and <4> Intel 8259 technical data sheets.

LISTING 2

Program for Testing the Operation of the Timer of Listing 1

```

/* ttimer.c */
#include <stdio.h>

extern volatile long time_count ;          /* Timpak identifiers */
extern volatile int time_error ;
extern void time_enable(void) ;
extern void time_disable(void) ;

```

LISTING 2 (Continued)

```

main()
{ time_enable() ;                               /* Enable timing.    */
  while(1)                                       /* Loop forever.    */
  { printf("type a key to reset timer\n") ;      /* Prompt to screen.*/
    while( !kbhit() ) ; getch() ;              /* Wait for keypress.*/
    time_count = 0 ;                            /* Reset count.     */
    time_error = 0 ;                            /* Reset error flag. */
    printf("type a key to read timer\n") ;       /* Prompt to screen.*/
    while( !kbhit() ) ; getch() ;              /* Wait for keypress.*/
    printf("time = %ld\n",time_count) ;          /* Report time.     */
    if( time_error )                            /* Test for error.   */
      { printf("timing error\007\n") ;           /* Message & beep.  */
      }
  }
}

```

Note—This program repeatedly measures the time between two presses of a key on the keyboard, while testing for a transition of time_error from zero to nonzero.

LISTING 3
The srpak Module

```

/* srpak.c */
#define BASAD 0x3f8                               /* Port address.    */
#include <dos.h>                                   /* Low-level I/O.   */

void set_output(pin,level) int pin,level ;         /* Output routine.  */
{ static int out_pat = 8 ;                         /* Always leave Out2 set. <2> */
  if(pin == 4)                                     /* Which stimulus?  */
    out_pat = (out_pat & 0xfd) | ((level & 1) << 1) ; /* DB25, pin 4 <2> */
  else if(pin == 20)                               /* or else          */
    out_pat = (out_pat & 0xfe) | (level & 1) ;       /* DB25, pin 20 <2> */
  outportb(BASAD+4,out_pat) ;                      /* Put out new      */
}                                                    /* out_pat. <2> */

int get_input(pin) int pin ;                      /* Input routine.   */
{ int level ;                                       /*                  */
  if(pin == 5) level = inportb(BASAD+6) & 0x10 ;   /* DB25, pin 5 <2> */
  else if(pin == 6) level = inportb(BASAD+6) & 0x20 ; /* DB25, pin 6 <2> */
  else if(pin == 8) level = inportb(BASAD+6) & 0x80 ; /* DB25, pin 8 <2> */
  if( level ) return(1) ;                          /* 1 for +volts    */
  return(0) ;                                       /* 0 for -volts    */
}

```

Note—The srpak module contains routines for binary input and output, accessing RS232 DB25 pins 4 and 20 for output, and pins 5, 6, and 8 for input. The *level* variable is 1 for positive voltage and 0 for negative.

LISTING 4
Jargon Header File for Reaction-Time Experiments

```

/* rtjargon.h */
#define TURN set_output                          /* Imperative, to turn on or off. */
#define KEY get_input                            /* Query, for state of an input.   */
#define WARNING_SIGNAL 4                       /* Selection of output pin 4.      */
#define TRIGGER_SIGNAL 20                      /* Selection of output pin 20.     */
#define ONE 5                                  /* Selection of input pin 5.       */
#define TWO 6                                  /* Selection of input pin 6.       */

```

LISTING 4 (Continued)

```

#define THREE 8           /* Selection of input pin 8.      */
#define OFF 0             /* Specification of output -volts. */
#define ON 1              /* Specification of output +volts. */
#define UP 0              /* Value of input -volts.          */
#define DOWN 1            /* Value of input +volts.          */
#define AND &&            /* Logical and.                    */
#define OR ||             /* Logical or.                     */
#define IS ==             /* Relational equals.              */
#define WAIT kbhit()      /* Do nothing, but respond if CTRL-C. */
#define RESTART continue  /* Jump to head of for block.      */

```

Note—Such a file of substitutions can be used to make the source program for an experiment read somewhat like a flowchart, with a pattern that is almost understandable to a nonprogrammer.

LISTING 5
Program for a Reaction-Time Experiment

```

/* rtexp.c */
#include <stdio.h>
#include "rtjargon.h"
extern volatile long time_count ;           /* Timpak      */
extern volatile int time_error ;           /* identifiers. */
extern void time_enable(void) ;
extern void time_disable(void) ;
extern void set_output(int n, int m) ;      /* Srpak      */
extern int get_input(int n) ;              /* identifiers. */

#define FOREPERIOD 200                     /* 2 seconds.  */
#define MAX_TIME 300                      /* 3 seconds.  */
#define N 20                              /* Nr of trials. */

int rt[N] ;                               /* Reaction times. */
int trial ;                               /* Trial number.  */

main()
{ time_enable() ;                          /* Enable timing. */
  for(trial=0 ; trial < N ;                /* Do N trials.  */
      { TURN( WARNING_SIGNAL,OFF ) ;       /* Extinguish all */
        TURN( TRIGGER_SIGNAL,OFF ) ;       /* signals.      */
        while( KEY(ONE) IS DOWN OR KEY(TWO) IS DOWN ) WAIT ; /* Wait, all clear? */
        while( KEY(ONE) IS UP ) WAIT ;     /* Wait, S ready? */
        TURN( WARNING_SIGNAL,ON ) ;        /* Signal on.    */
        time_count = 0 ;                    /* Reset time.   */
        while( time_count < FOREPERIOD AND /* Wait, FP     */
              KEY(TWO) IS UP ) WAIT ;      /* elapsed?     */
        if( time_count < FOREPERIOD )      /* Anticipation? */
        { fprintf(stderr,                  /* Message to    */
            "Anticipation, repeating trial %d\n",trial) ; /* screen.      */
          RESTART ;                        /* Restart trial. */
        }
        time_count = 0 ;                    /* Reset timer.  */
        TURN( TRIGGER_SIGNAL,ON ) ;        /* Signal on.    */
        while( time_count < MAX_TIME AND   /* Wait for      */
              KEY(TWO) IS UP ) WAIT ;      /* response.     */
      }
}

```

LISTING 5 (Continued)

```

    rt[trial] = time_count ;                /* Store datum.  */
    fprintf(stderr,                        /* Report to    */
        "trial %d RT = %d\n",trial,rt[trial]) ; /* screen. */
    trial++ ;                             /* Next trial.  */
}
TURN( WARNING_SIGNAL,OFF ) ;              /* Extinguish all */
TURN( TRIGGER_SIGNAL,OFF ) ;              /* signals. */
for(trial=0; trial < N ; trial++)         /* Put out data. */
    printf("%4d\n",rt[trial]) ;
time_disable() ;                          /* Disable timing. */
}

```

Note—The subject operates two telegraph keys, KEY(ONE) and KEY(TWO), while monitoring two stimulus signal sources, WARNING_SIGNAL and TRIGGER_SIGNAL.

(Manuscript received November 3, 1987;
revision accepted for publication January 4, 1988.)