link to *the **Technology Interface***
the Electronic Journal for Engineering Technology

*the **Technology Interface** /Fall 96*

Use of a PC Printer Port for Control and Data Acquisition

by

Peter H. Anderson
pha@eng.morgan.edu
Department of Electrical Engineering
Morgan State University

Abstract: *A PC printer port is an inexpensive and yet powerful platform for implementing projects dealing with the control of real world peripherals. The printer port provides eight TTL outputs, five inputs and four bidirectional leads and it provides a very simple means to use the PC interrupt structure.*

This article discusses how to use program the printer port. A larger manual which deals with such topics as driver circuits, optoisolators, control of DC and stepping motors, infrared and radio remote control, digital and analog multiplexing, D/A and A/D is avaialable. See

http://www.access.digex.net/~pha

A special thanks to Morgan State University students Towanda Malone, Christine Samuels and H. Paul Roach for their technical contributions and to New Mexico State University student Kyle Quinnell for preparing the html file.

I. Printer Port Basics

A. Port Assignments

Each printer port consists of three port addresses; data, status and control port. These addresses are in sequential order. That is, if the data port is at address 0x0378, the corresponding status port is at 0x0379 and the control port is at 0x037a.

The following is typical.

Printer	Data Port	Status	Control
LPT1	0x03bc	0x03bd	0x03be
LPT2	0x0378	0x0379	0x037a
LPT3	0x0278	0x0279	0x027a

My experience has been that machines are assigned a base address for LPT1 of either 0x0378 or 0x03bc.

To definitively identify the assignments for a particular machine, use the DOS debug program to display memory locations 0040:0008. For example:

```
>debug
-d 0040:0008 L8
0040:0008       78 03 78 02 00 00 00 00
```

Note in the example that LPT1 is at 0x0378, LPT2 at 0x0278 and LPT3 and LPT4 are not assigned.

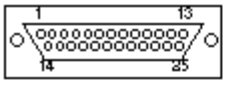
Thus, for this hypothetical machine;

Printer	Data Port	Status	Control
LPT1	0x0378	0x0379	0x037a
LPT2	0x0278	0x0279	0x027a
LPT3	NONE		
LPT4	NONE		

An alternate technique is to run Microsoft Diagnostics (MSD.EXE) and review the LPT assignments.

B. Outputs

Please refer to the figures titled Figure #1 - Pin Assignments and Figure #2 - Port Assignments. These two figures illustrate the pin assignments on the 25 pin connector and the bit assignments on the three ports.



View is looking at
Connector side of
DB-25 Male Connector.

Pin	Description	
1	Strobe	PC Output
2	Data 0	PC Output
3	Data 1	PC Output
4	Data 2	PC Output
5	Data 3	PC Output
6	Data 4	PC Output
7	Data 5	PC Output
8	Data 6	PC Output
9	Data 7	PC Output
10	ACK	PC Input
11	Busy	PC Input
12	Paper Empty	PC Input
13	Select	PC Input
14	Auto Feed	PC Output
15	Error	PC Input
16	Initialize Printer	PC Output
17	Select Input	PC Output

Pin Assignments

Note: 8 Data Outputs
4 Misc Other Outputs

5 Data Inputs

Note: Pins 18-25 are
Ground

Fig 1. Pin Assignments

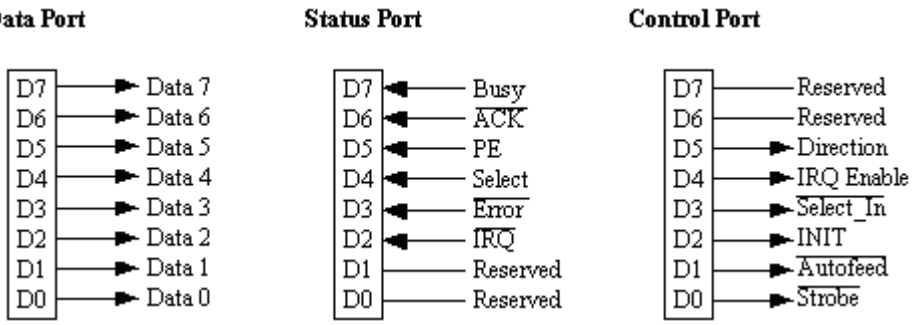


Fig 2. Port Assignments

Note that there are eight outputs on the Data Port (Data 7(msb) - Data 0) and four additional outputs on the low nibble of the Control Port. /SELECT_IN, INIT, /AUTO FEED and /STROBE.

[Note that with /SELECT_IN, the "in" refers to the printer. For normal printer operation, the PC exerts a logic zero to indicate to the printer it is selected. The original function of INIT was to initialize the printer, AUTO FEED to advance the paper. In normal printing, STROBE is high. The character to be printed is output on the Data Port and STROBE is momentarily brought low.]

All outputs on the Data Port are true logic. That is, writing a logic one to a bit causes the corresponding output to go high. However, the /SELECT_IN, /AUTOFEED and /STROBE outputs on the Control Port have inverted logic. That is, outputting a logic one to a bit causes a logic zero on the corresponding output. This adds some complexity in using the printer port, but the fix is to simply invert those bits using the exclusive OR function prior to outputting.

[One might ask why the designers of the printer port designed the port in this manner. Assume you have a printer with no cable attached. An open usually is read as a logic one. Thus, if a logic one on the SELECT_IN, AUTOFEED and STROBE leads meant to take the appropriate action, an unconnected printer would assume it was selected, go into the autofeed mode and assume there was data on the outputs associated with the Data Port. The printer would be going crazy when in fact it wasn't even connected. Thus, the designers used inverted logic. A zero forces the appropriate action.]

Returning to the discussion of the Control Port, assume you have a value val1 which is to be output on the Data port and a value val2 on the Control port:

```
#define DATA 0x03bc
#define STATUS DATA+1
#define CONTROL DATA+2
...
int val1, val2;
...
val1 = 0x81;    /* 1000 0001 */         /* Data bits 7 and 0 at one */
   outportb(DATA, val1);
val2 = 0x08;    /* 0000 1000 */
   outportb(CONTROL, VAL2^0x0b);
/* SELECT_IN = 1, INIT = 0, /AUTO_FEED = 0, /STROBE = 0 */
```

Note that only the lower nibble of val2 is significant. Note that in the last line of code, /SELECT_IN, /AUTO_FEED and /STROBE are output in inverted form by using the exclusive-or function so as to compensate for the hardware inversion.

For example; if I intended to output 1 0 0 0 on the lower nibble and did not do the inversion, the

hardware would invert bit 3, leave bit 2 as true and invert bits 1 and 0. The result, appearing on the output would then be 0 0 1 1 which is about as far from what was desired as one could get. By using the exclusive-or function, 1 0 0 0 is actually sent to the port as 0 0 1 1. The hardware then inverts bits 3, 1 and 0 and the output is then the desired 1 0 0 0.

C. Inputs

Note that in the diagram showing the Status Port there are five status leads from the printer. (BSY, /ACK, PE (paper empty), SELECT, /ERROR).

[The original intent in the naming of most of these is intuitive. A high on SELECT indicates the printer is on line. A high on BSY or PE indicates to the PC that the printer is busy or out of paper. A low wink on /ACK indicates the printer received something. A low on ERROR indicates the printer is in an error condition.]

These inputs are fetched by reading the five most significant bits of the status port.

However, the original designers of the printer interface circuitry, inverted the bit associated with the BSY using hardware. That is, when a zero is present on input BSY, the bit will actually be read as a logic one. Normally, you will want to use "true" logic, and thus you will want to invert this bit.

The following fragment illustrates the reading the five most significant bits in "true" logic.

```
#define DATA 0x03bc
#define STATUS DATA+1
...
unsigned int in_val;
...
in_val = ((inportb(STATUS)^0x80) >> 3);
```

Note that the Status Port is read and the most significant bit, corresponding to the BSY lead is inverted using the exclusive-or function. The result is then shifted such that the upper five bits are in the lower five bit positions.

```
0 0 0 BUSY /ACK PE SELECT /ERROR
```

Another input, IRQ on the Status Port is not brought to a terminal on the DB-25 printer port connector. I have yet to figure out how to use this bit.

At this point, you should see that, at a minimum, there are 12 outputs; eight on the Data Port and four on the lower nibble of the Control Port. There are five inputs, on the highest five bits of the Status Port. Three output bits on the Control Port and one input on the Status Port are inverted by the hardware, but this is easily handled by using the exclusive-or function to selectively invert bits.

D. Simple Example

Refer to the figure titled Figure #3 - Typical Application showing a normally open push button switch being read on the BUSY input (Status Port, Bit 7) and an LED which is controlled by Bit 0 on the Data Port. A C language program which causes the LED to flash when the push-button is depressed appears below. Note that an output logic zero causes the LED to light.

```
/* File LED_FLSH.C
**
** Illustrates simple use of printer port.  When switch is
** depressed LED flashes.  When switch is not depressed, LED is
** turned off.
**
** P.H. Anderson, Dec 25, '95
*/

#include <stdio.h>
#include <dos.h>   /* required for delay function */

#define DATA 0x03bc
#define STATUS DATA+1
#define CONTROL DATA+2

void main(void)
{
   int in;
   while(1)
   {
      in = inportb(STATUS);
      if (((in^0x80)&0x80)==0)
      /* if BUSY bit is at 0 (sw closed) */
      {
         outportb(DATA,0x00);   /* turn LED on */
         delay(100);
         outportb(DATA, 0x01);   /* turn it off */
         delay(100);
      }
      else
      {
         outportb(DATA,0x01);
         /* if PB not depressed, turn LED off */
      }
   }
}
```

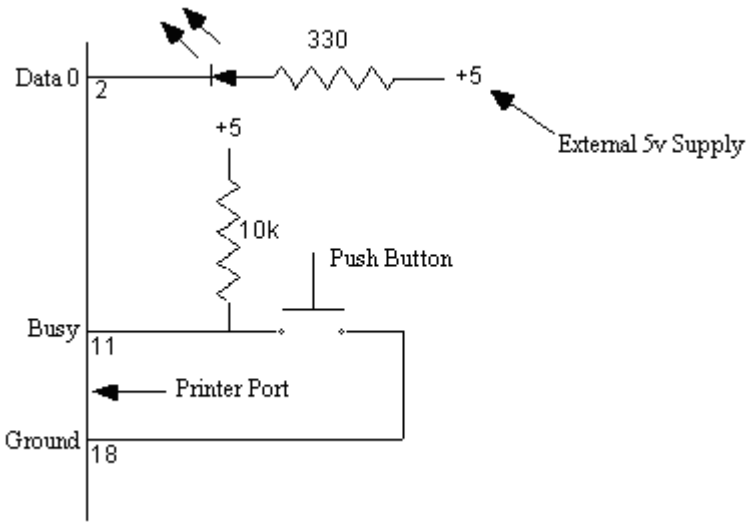


Fig 3 Printer Port - Typical Application

Circuit Description: Logic 1 on output DATA 0 (Data Port - Bit 0) causes LED to be off. Logic 0 causes LED to turn on.

Normally open push-button causes +5V (logic 1) to appear on input BUSY (STATUS PORT - Bit

7). When depressed, push-button closes and ground (logic 0) is applied to input Busy.

Note external source of 5V.

Program Description: When idle, push-button is open and LED is off. On depressing push-button, LED blinks on and off at nominally 5 pulses per second.

E. Test Circuitry

Refer to the figure titled Figure #4 - Printer Port Test Circuitry. This illustrates a very simple test fixture to allow you to figure out what inversions are taking place in the hardware associated with the printer port. Program test_prt.c sequentially turns each of the 12 LED's on and then off and then continually loops to display the settings on the five input switches.

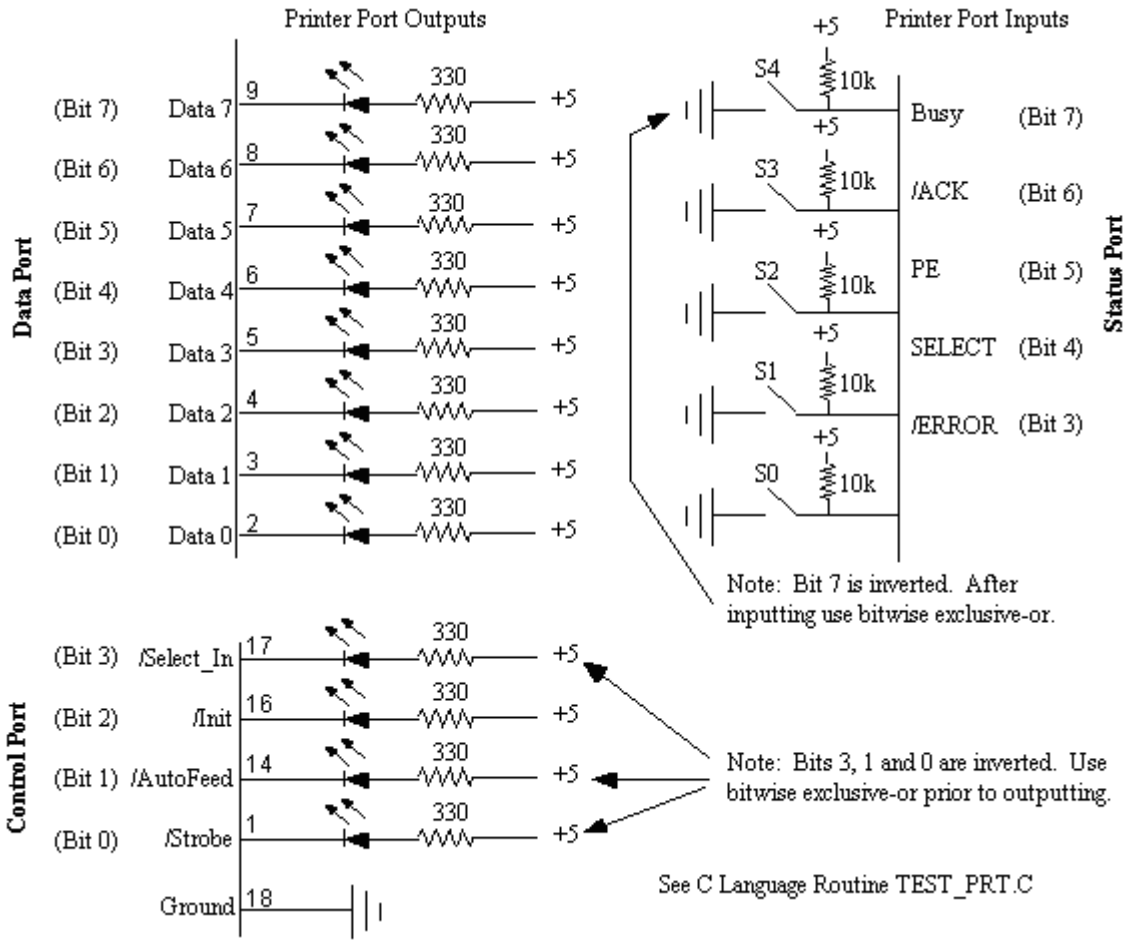


Fig 4. Printer Port Test Circuitry

```
/* File TEST_PRT.C
**
** Program to exercise 12 outputs and five inputs.
**
** Program sequentially turns off LEDs on Bits 7, 6, 5, ... 0 on the
** Data Port, and then Bits #, 2, 1 and 0 on the Control Port.  Each
** LED is held off for nominally 1 second.  Note that an LED is turned
** off with a logic one.  This process is executed once.
**
** Program then loops, scanning the highest five bits on the Status Port
** and continuously displays the content in hexadecimal.
**
```

```
** P.H. Anderson, Dec 25, '95
*/

#include <stdio.h>
#include <dos.h>        /* required for delay function */

#define DATA 0x03bc     /* for the PC I used */
#define STATUS DATA+1
#define CONTROL DATA+2

void main(void)
{
   int in, n;

   outportb(DATA,0x00); /* turn on all LEDs on Data Port */
   outportb(CONTROL, 0x00^0x0b); /* same with Control Port */

   /* now turn off each LED on Data Port in turn by positioning a logic
   one in each bit position and outputing.
   */
   for (n=7; n>=0; n++)
   {
      outportb(DATA, 0x01 << n);
      delay(1000);
   }
   outportb(DATA, 0x00);

   /* now turnoff each LED on control port in turn
   ** note exclusive-or to compensate for hardware inversions
   */

   outportb(CONTROL, 0x08^0x0b);        /* bit 3 */
   delay(1000);
   outportb(CONTROL, 0x04^0x0b);        /* bit 2 */
   delay(1000);
   outportb(CONTROL, 0x02^0x0b);        /* bit 1 */
   delay(1000);
   outportb(CONTROL, 0x01^0x0b);        /* bit 0 */
   delay(1000);

   outportb(CONTROL, 0x00);

   /* Continuously scan switches and print result in hexadecimal */

   while(1)
   {
      in = (inportb(STATUS)^0x80)&0xf8;
      /* Note that BUSY (msbit) is inverted and only the
      ** five most significant bits on the Status Port are displayed.
      */
      printf("%x\n", in);
   }
}
```

F. Interrupts

Again refer to Figure #2-Port assignments. Note that bit 4 on the Control Bit is identified as IRQ Enable. Normally, this bit is set to zero.

However, there are times when interrupts are of great value. An interrupt is nothing more than a hardware event which causes your program to momentarily stop what it is doing, and jump to a function to do what you desire. When this is complete, your program returns to where it left off.

In using the printer port, if the IRQ Enable is set to a logic one, an interrupt occurs when input ACK next goes from a logic one to logic zero. For example, you might use input ACK for an intrusion alarm. You might have a program running which is continually monitoring temperature. But, when input ACK goes low, it interrupts your temperature monitoring and goes to some other code you have written to handle an alarm. Perhaps, to fetch the time and date off the system clock and write this to an Intrusion File. When done, your program continues monitoring temperature.

Interrupts are treated in detail elsewhere in this manual.

G. Changing Only Selected Bits

Frequently, when outputting, the programmer is interested in only a portion of a byte and it is a burden to remember what all the other bits are. Please review the following where bit 2 is brought high for 100ms and then brought low.

```
int data;
...
data=data | 0x04;/* bring bit 2 high */
outportb(DATA,data);
delay(100);
data=data & 0xfb; /* bring bit 2 low */
outportb(DATA,data);
```

Note that variable data keeps track of the current state on the output port. Each bit manipulation is performed on data and variable data is then output.

To bring a specific bit to a logic one, use the OR function to OR that bit with a logic one (and all others with a logic zero.)

To bring a specific bit to a logic zero, AND that bit with a zero (and all others with a logic one.) Calculating this value can be tedious. Consider this alternative:

```
data=data & 0xfb;       /* hard to calculate */
data=data & (~0x04);    /* the same but a lot easier */
```

This really isn't very difficult. Assume, you currently have; XXXX XXXX and desire XX01 X100.

```
data=data & (~0x20) | 0x10 | 0x04 & (~0x02) & (~0x01);
```

H. Ports on Newer PC's

A few words about the DIRECTION bit on the Control Port. I have seen PC's where this bit may be set to a logic "one" which turns around the Data Port such that all of the Data leads are inputs. I have also seen PC's where this worked for only the lower nibble of the Data Port and other PC's where it did nothing. It is probably best not to use this feature. Rather, leave the DIRECTION bit set to logic "zero".

I. Differences In Printer Ports

The material discussed above is believed to be pretty generic; that is, common to all

manufacturers.

You may well ask, "why would they be different. After all, programs such as WordPerfect must work on all machines." The answer is that the programmers who write such programs as WordPerfect do not get down to this low level of hardware detail. Rather, they write to interface with the PC's BIOS.

The BIOS (Basic Input-Output System) is a ROM built in to the PC which makes all PC's appear the same. This is a pretty nice way for each vendor to implement their design with a degree of flexibility.

An example is the port assignments discussed above. This data is read from the BIOS ROM when your PC boots up and written to memory locations beginning at 0040:0008. Thus, the designers of WordPerfect don't worry about the port assignments. Rather, they read the appropriate memory location.

In the same way, they interface with the BIOS for printing. For example, if the designers want to print a character, the AH register is set to zero, the character to be printed is loaded into AL, the port (LPT1, LPT2, etc) is loaded into the DX register. They then execute a BIOS INT 17h. Program control is then passed to the BIOS and which performs at the low level of hardware design which we are trying to work. The BIOS varies from one hardware design to another; it's purpose is to work with the hardware. If inversions are necessary, it is done in the BIOS. When the BIOS has completed whatever bit sequencing is required to write the character to the printer, control is passed back to the program with status information in the AH register.

J. Summary

In summary, the printer port affords a very simple technique for interfacing with external circuitry. Twelve output bits are available, eight on the Data Port and four on the lower nibble of the Control Port. Inversions are necessary on three of the bits on the Control Port. Five inputs are available on the Status Port. One software inversion is necessary when reading these bits.

II. Forcing an Interrupt on the Printer Port.

A. Introduction

This section describes how to use hardware interrupts using the printer port. The discussion closely follows programs prnt_int.c and time_int.c

A hardware interrupt is a capability where a hardware event causes the software to stop whatever it is doing and to be redirected to a function to handle the interrupt. When done, the program picks up where it left off. Aside from loosing time in executing the interrupt service routine, the operation of the main program remains unaffected by the interrupt.

This is quite powerful and although at first, the whole process may appear difficult to grasp, it is in fact quite simple.

Although this discussion focuses on using the interrupt associated with the printer port, the same technique may be adapted to exerting interrupts directly on the ISA bus.

B. Interrupt Handler Table

When an interrupt occurs, the PC must know where to go to handle the interrupt.

The original 8088 PC design provided for up to 256 interrupts (0x00 - 0xff). This includes both hardware and software interrupts. Each of these 256 interrupt types has four bytes in a table beginning at memory location 0x00000. Thus, INT 0 uses memory locations 0x00000, 0x00001, 0x00002, 0x00003, INT 1 uses the next four bytes in the table, etc. Note that INT 8 then uses the four bytes at 0x0020, INT 9 begins at 0x0024, etc. This 1024 bytes (256x4) is termed the interrupt vector table.

These four bytes contain the address of where the PC is to go to when an interrupt occurs. Most of the table is loaded when you boot up the machine. The table may be added to or entries modified when you run various applications.

IBM reserved eight hardware interrupts beginning at INT 0x08 for interrupt expansion. These are commonly known as IRQ0 - IRQ7, the IRQ corresponding to the lead designations associated with the Intel 8259 which was used to control these interrupts. Thus, IRQ 0 corresponds to INT 8, IRQ 1 corresponds to INT 9, etc.

Exercise. Use debug to examine the interrupt vector table which is assigned to IRQ 0 through IRQ 7.

```
-d 00000:0020 20

        B3 10 3B 0B 73 2C 3B 0B-57 00 70 03 8B 3B 3B 0B
        ED 3B 3B 0B AC 3A 3B 0B-B7 00 70 03 F4 06 70 00
```

(Recall that the table allocation for INT 8 begins at 0x0020).

From this I can see that the address for the interrupt service routine associated with IRQ 0 is 0B3B:10B3. For IRQ 7, 0070:06F4. You should be able to see the algorithm I used to obtain this.

Thus, when an IRQ 7 interrupt occurs, we know this corresponds to INT 0x0f and the address of the interrupt service routine is located at 0070:06F4.

Exercise. Use the debugger to examine the interrupt vector table. Then use Microsoft Diagnostics (MSD) and examine the IRQ addresses and compare the two.

C. Modifying the Interrupt Handler Table

Assume, you are going to use IRQ 7. Assume that when an IRQ 7 interrupt occurs, you desire your program to proceed to function irq7_int_serv, a function which you wrote. In order to do so, you must first modify the interrupt handler table. Of course, you may wish to carefully take what is already there in the table and save it somewhere and then when you leave your program, put the old value back.

Borland's Turbo C provides functions to do this.

```
int intlev=0x0f;

oldfunc = getvect (intlev);
```

```
        /* The content of 0x0f is fetched and saved for future
        **use. */

setvect (intlev, irq7_int_serv);
        /* New address is placed in table. */
        /* irq7_int_serv is the name of routine and is of type
        ** interrupt far*/
```

This may look bad, but, in fact it isn't. Simply get the vector now associated with INT 0x0f and save it in a variable named oldfunc. Then set the entry associated with INT 0x0f to be the address of your interrupt service routine.

Good programming dictates that once you are done with your program, you would restore the entry to what it was;

```
setvect(intlev, oldfunc);
```

After all, what would you think of WordPerfect, if after running it, you couldn't use your modem without rebooting.

D. Masking

The programmer can mask interrupts. If an interrupt is masked you are saying to the PC, "for the moment ignore any IRQ 7 type interrupts". Normally, we don't do this. Rather we desire to set the interrupt mask such that IRQ 7 is enabled.

Port 0x21 is associated with the interrupt mask. To enable a particular IRQ, write a zero to that bit location. However, you don't want to disturb any of the other bits.

```
mask=inportb(0x21) & ~0x80;
        /* Get current mask.  Set bit 7 to 0.  Leave other bits
        ** undisturbed. */
outportb(0x21, mask);
```

The user is now ready for IRQ 7 interrupts. Note that each time there is an IRQ 7 interrupt, the program is unconditionally redirected to the function irq7_int_serv. The user is free to do whatever they like but must tell the PC that the interrupt has been processed;

```
outportb(0x20, 0x20);
```

Prior to exiting the program, the user should return the system to its original state; setting bit 7 of the interrupt mask to logic one and restoring the interrupt vector.

```
mask=inportb(0x21) | 0x80;
outportb(0x21, mask);

setvect(intlev, oldfunc);
```

E. Interrupt Service Routine

In theory, you should be able to do anything in your interrupt service routine (ISR). For example, an interrupt might be forced by external hardware detecting an intrusion. The ISR might fetch the time off the system clock, open a file and write the time and other information to the file, close the

file and then return to the main program.

In fact, I have not had good luck in doing this and you will note that my interrupt service routines are limited;

- disable any further interrupts.
- set a variable such that in returning to the main program there is an indication that an interrupt occurred.
- indicate to the PC that the interrupt was processed; outportb(0x20,0x20);
- enable interrupts.

I think that my problem is that interrupts are turned off during the entire ISR which may well preclude a C function which may use interrupts. For example, in opening a file, I assume interrupts are used by Turbo C to interface with the disk drive. Unlike the IRQ we are discussing, the actual implementation of how C handles these interrupts necessary to implement a C function will not be "heard" by the PC and the program will appear to bomb.

My suggestion is that you initially use the technique I have used in writing your interrupt service routine. That is, very simple; either setting or incrementing a variable. However, recognize that this is barely scratching the surface.

Then you might try a more complex ISR of the following form. At the time of this writing I have not tried this.

- disable all interrupts.
- set mask to disable IRQ 7 interrupts.
- outportb(0x20,0x20).
- enable all interrupts.
- .. do whatever needs to be done ..
- disable all interrupts.
- set mask to enable IRQ 7 interrupts.
- enable interrupts.

Note the difference from the previous. Any further IRQ 7 interrupts are blocked while in the ISR, but in the middle of the ISR, all other interrupts are enabled. This should permit all C functions to work.

F. IRQ Enable Bit

Recall that there are three ports associated with the control of a printer port; Data, Status and Control. Bit 4 of the Control Port is a PC output; IRQ Enable. Note that Bit 2 of the Status Port is a PC input; /IRQ. Neither of these bits are associated with the DB-25 connector. Rather, they control logic on the printer card or PC motherboard.

If the IRQ Enable output is at logic one, an interrupt occurs on a negative going transition on the /ACK input. (I have yet to figure out what the IRQ input does).

Thus, in addition to setting the mask to entertain interrupts from IRQ 7 as discussed above, you must also set IRQ Enable to a logic one.

```
mask=inportb(0x21) & ~0x80;
outportb(0x21,mask);  /* as discussed above */
```

```
outportb(CONTROL, inportb(CONTROL) | 0x10);
```

Note that in this implementation, all bits other than Bit 4 on the Control Port are left as they were.

Prior to exiting from your program, it is good practice to leave things tidy. That is, set Bit 4 back to a zero.

```
outportb(CONTROL, inportb(CONTROL) & ~0x10);
```

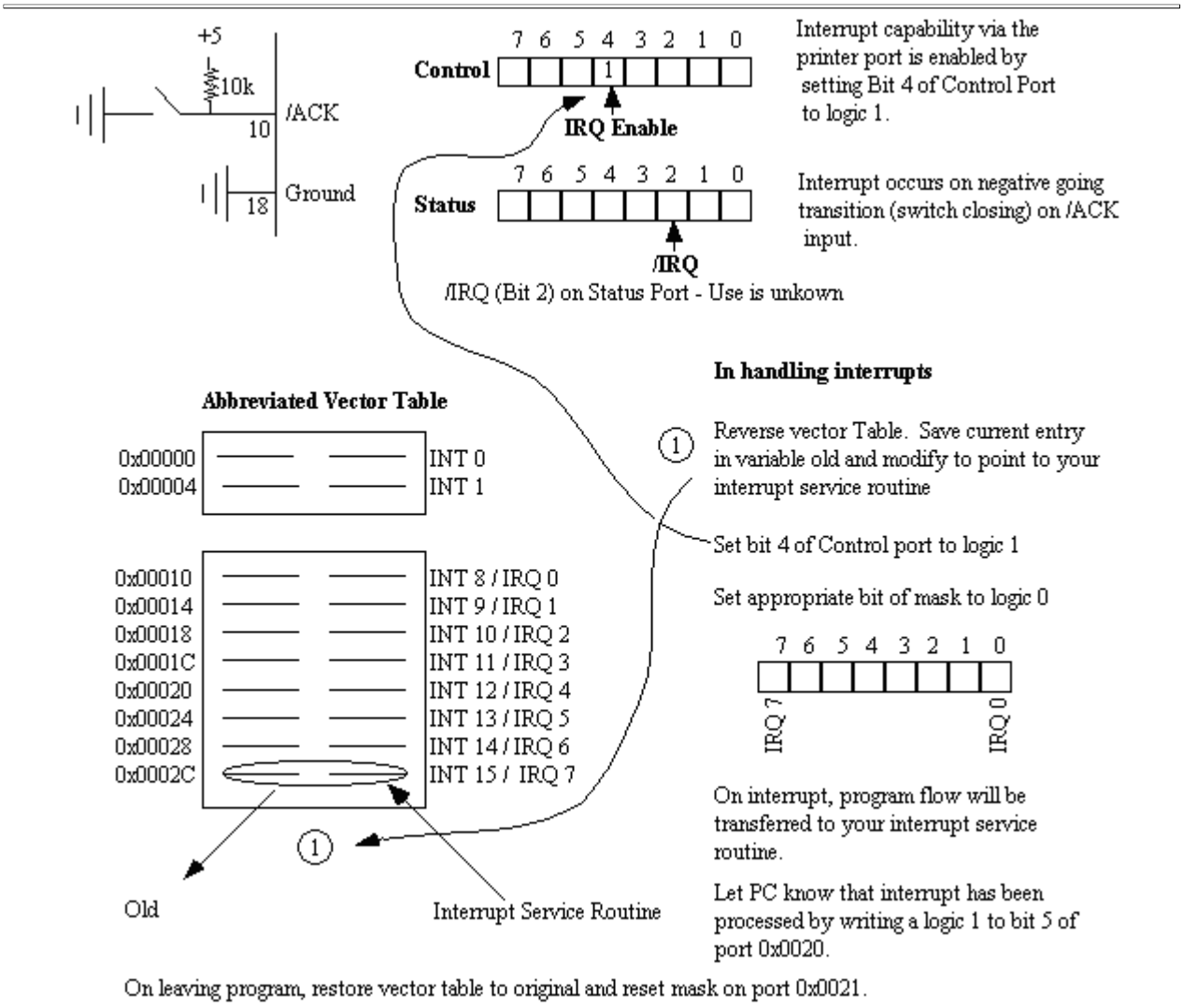


Fig 5. Use of Parallel Printer Port For Interrupts

G. Programs

Program PRNT_INT.C simply causes a screen message to indicate an interrupt has occurred. Note that global variable "int_occurred" is set to false in the declaration. On interrupt, this is set to true. Thus, the code in main within the if(int_occurred) is only executed if a hardware interrupt did indeed occur.

Program TIME_INT.C is the same except for main. When the first interrupt occurs, the time is fetched off the system clock. Otherwise the new time is fetched and the difference is calculated and displayed.

```
/*
** Program PRNT_INT.C
**

Uses interrupt service routine to note interrupt from printer port.
The interrupt is caused by a negative on /ACK input on Printer Port.
This might be adapted to an intrusion detector and temperature logger.

Note that on my machine the printer port is located at 0x0378 -
0x037a and is associated with IRQ 7.  You should run Microsoft
Diagnostics (MSD) to ascertain assignments on your PC.

**      Name Address in Table
**
**      IRQ2 0x0a
**      IRQ4 0x0c
**      IRQ5 0x0d
**      IRQ7 0x0f
**

** P.H. Anderson, MSU, 12 May 91; 26 July 95
*/

#include <stdio.h>
#include <bios.h>
#include <dos.h>

#define DATA 0x0378
#define STATUS DATA+1
#define CONTROL DATA+2

#define TRUE 1
#define FALSE 0

void open_intserv(void);
void close_intserv(void);
void int_processed(void);
void interrupt far intserv(void);

int intlev=0x0f;   /* interrupt level associated with IRQ7 */
void interrupt far (*oldfunc)();
int int_occurred = FALSE;   /* Note global definitions */

int main(void)
{
   open_intserv();
   outportb(CONTROL, inportb(CONTROL) | 0x10);
   /* set bit 4 on control port to logic one */
   while(1)
   {
      if (int_occurred)
      {
         printf("Interrupt Occurred\n");
         int_occurred=FALSE;
      }
   }
   close_intserv();
   return(0);
}

void interrupt far intserv(void)
/* This is written by the user.  Note that the source of the interrupt
** must be cleared and then the PC 8259 cleared (int_processed).
** must be included in this function.
*/
{
```

```
   disable();
   int_processed();
   int_occurred=TRUE;
   enable();
}

void open_intserv(void)
/* enables IRQ7 interrupt.  On interrupt (low on /ACK) jumps to intserv.
** all interrupts disabled during this function; enabled on exit.
*/
{
   int int_mask;
   disable();  /* disable all ints */
   oldfunc=getvect(intlev);  /* save any old vector */
   setvect (intlev, intserv);  /* set up for new int serv */
   int_mask=inportb(0x21);  /* 1101 1111 */
   outportb(0x21, int_mask & ~0x80);  /* set bit 7 to zero */
                                      /* -leave others alone */
   enable();
}

void close_intserv(void)
/* disables IRQ7 interrupt */
{
   int int_mask;
   disable();
   setvect(intlev, oldfunc);
   int_mask=inportb (0x21) | 0x80;  /* bit 7 to one */
   outportb(0x21, int_mask);
   enable();
}

void int_processed(void)
/* signals 8259 in PC that interrupt has been processed */
{
   outportb(0x20,0x20);
}
```

```
/*
* Program TIME_INT.C
*

Uses interrupt service routine to note interrupt from printer port.
The interrupt is caused by a negative on /ACK input on Printer Port.

Calculates time and displays the time in ms between interrupts.

* P.H. Anderson, MSU, 10 Jan, '96
*/

#include <stdio.h>
#include <bios.h>
#include <dos.h>
#include <sys\timeb.h>

#define DATA 0x0378
#define STATUS DATA+1
#define CONTROL DATA+2

#define TRUE 1
#define FALSE 0

void open_intserv(void);
void close_intserv(void);
void int_processed(void);
void interrupt far intserv(void);

int intlev=0x0f;  /* interrupt level associated with IRQ7 */
```

```
void interrupt far (*oldfunc)();
int int_occured=FALSE;  /* Note global definitions */

int main(void)
{
   int first=FALSE;
   int secs, msecs;
   struct timeb t1, t2;
   open_intserv();
   outportb(CONTROL, inportb(CONTROL) | 0x10);
   /* set bit 4 on control port (irq enable) to logic one */
   while(1)
   {
      if (int_occurred)
      {
         int_occurred=FALSE;
         if (first==FALSE)
         /* if this is the first interrupt, just fetch the time */
         {
            ftime(&t2);
            first=TRUE;
         }
         else
         {
            t1=t2;  /* otherwise, save old time, fetch new */
            ftime(&t2);  /* and compute difference */
            secs=t2.time - t1.time;
            msecs=t2.millitm - t1.millitm;
            if (msecs<0)
            {
              --secs;
              msecs=msecs+1000;
           }
           printf("Elapsed time is %d\n",1000*secs+msecs);
           }
       }
    }
    close_intserv();
    return(0);
}

void interrupt far intserv(void)
/* This is written by the user.  Note that the source of the interrupt
/* must be cleared and then the PC 8259 cleared (int_processed).
/* must be included in this function.
/*******/
{
   disable();
   int_processed();
   int_occurred=TRUE;
   enable();
}

void open_intserv(void)
/* enables IRQ7 interrupt.  On interrupt (low on /ACK) jumps to intserv.
/* all interrupts disabled during this function; enabled on exit.
/*******/
{
   int int_mask;
   disable();  /* disable all ints */
   oldfunc=getvect(intlev);  /* save any old vector */
   setvect(intlev, intserv);  /* set up for new int serv */
   int_mask=inportb(0x21);    /* 1101 1111 */
   outportb(0x21, int_mask & ~0x80);  /* set bit 7 to zero */
                              /* -leave others alone */
   enable();
}

void close_intserv(void)
```

```
/* disables IRQ7 interrupt */
{
   int int_mask;
   disable();
   setvect(intlev, oldfunc);
   int_mask=inportb(0x21) | 0x80;  /* bit 7 to one */
   outportb(0x21,int_mask);
   enable();
}

void int_processed(void)
/* signals 8259 in PC that interrupt has been processed */
{
   outportb(0x20, 0x20);
}
```