Chapter R5

Device Drivers

R5.1 INTRODUCTION

The fifth required module of MPX deals with the implementation of a simple interrupt-driven, serial port driver, which will enable communication between your computer and another attached computer or video terminal.

In this module you will *not* make use of the command handler software developed in Modules R1 through R4. Instead, special test programs are provided to test and demonstrate your driver. This allows you to face the somewhat tricky problem of designing and debugging correct software for handling external interrupts without complex interaction with the larger MPX environment. After your device driver is correctly implemented, it will be integrated into the complete MPX in a later module.

R5.2 KEY CONCEPTS

Device Control

This module is concerned with *low-level* control of input and output devices. A more general discussion of device management is contained in the text. This section summarizes the information most pertinent to the device drivers to be developed in this module.

Device Types

Devices may be classified in many ways. A fundamental distinction is made between so-called **character devices**, which transfer one byte or word at a time, and **block devices**, which transfer large blocks of data at high speed. Character devices transfer each data unit via a CPU register using an explicit machine instruction. Because of the high speed required for block transfers, the CPU only starts the transfer; the data is then moved directly between the device and main memory, without further involvement by the CPU.

An important distinction can also be made in the way character devices are physically connected to the CPU. A **parallel** device is connected using a separate wire for each bit in the data unit. All the bits of a byte or word can therefore be transferred at once. This method is often used for printers. A **serial** device is physically connected to the computer using only a single data wire, although additional control and status wires may sometimes exist. The bits of a byte or word must be transferred one at a time. This is slower and more complex than parallel data transfer, but less expensive. All communication devices and most connections to remote computers and terminals use serial connections. This is the type of connection considered in the second part of this module.
Whether the connection is parallel or serial, another distinction can be made based on the strategy for controlling the flow of data to or from a character device. Typically the CPU can process data at a higher speed than the device, so a method is needed for controlling the transfer rate. The **synchronous** method is based on strict timing; the device is assumed to be ready after a fixed interval of time. A synchronous device driver can be fairly complex. This method is used mainly by high-speed communication devices. In contrast, **asynchronous** control relies on a signal from the device to the CPU, which indicates when the device is ready to transfer another data unit. This is the method used for the device driver in this module.

Device Registers

Because of the wide range of device types, it is not possible for the CPU to have a built-in understanding of the characteristics and requirements of each device. Instead, a standard, general model must be used to represent all devices to the CPU. In this model, all devices are represented by a set of **device registers**. The CPU and the device communicate primarily by reading and writing information using these registers.

Device registers can be classified in three general categories: *data*, *status*, and *control*. Most devices require at least one of each of these registers. Complex devices may require many of each.

Data registers are used to transfer the actual data that is being input from or output to the device. This includes, in particular, character codes to be read from a keyboard, displayed on a screen, or printed by a printer. Depending on the nature of the device, these registers may be either read or written by both the device and the CPU.

Status registers are used to inform the CPU of particular status conditions that exist within the device. They may indicate, for example, whether a printer is out of paper, or whether a character is available from a keyboard. Status registers are usually written only by the device and read only by the CPU.

Control registers are used by the CPU to perform control operations on a device. Examples include initializing a printer, or setting the speed for a communication device. These registers are written by the CPU and read by the device.

The interpretation of the control and status register bits, of course, is wholly dependent on the specific device. There is no consistency even in the use of ones and zeros. In a particular case either value may be the "significant" one.

The discussion so far has implied that devices are connected directly to the CPU. In fact, as the text explains, most devices are connected through an intermediate hardware unit called an **interface** or **controller**. The controller is the only element of the system that understands both the CPU and the device. This approach makes it much simpler to connect many different kinds of devices to many different kinds of CPUs. The device registers are usually physically located within the controller.

The CPU actually accesses device registers in an indirect manner, by reading and writing *I/O ports*. Depending on the CPU architecture, ports may be accessed by special input and output machine instructions, or by reading and writing special memory locations. The former method is used in the 8086 architecture. Section I2.3 provided an introduction to the 8086 approach. The machine instructions *in* and *out* are used to transfer either 8-bit bytes or 16-bit words between a specific CPU register (AX or AL) and a designated I/O port. I/O ports have addresses which may range from 0 to 65,535.

Turbo C provides the four functions `inport, inportb, outport,` and `outportb` to access the I/O ports of the PC. In MPX all data transfers are 8-bit, so only `inportb` and `outportb` will be used. These functions are described below in the discussion of support software.

For a some simple devices such printer there is a one-to-one correspondence between I/O ports and device registers. This may not be true for more complex devices or controllers. In some cases, the same port may be used to access multiple device registers; the particular register is selected using bits in a different (control) register. This is the case for some registers of the serial device to be programmed in this module. In other designs, a *sequence* of transfers using a single port may actually access a series of device registers in a fixed order.

Interrupts

Device registers are one of the two mechanisms by which I/O devices communicate with the CPU. The second method is **interrupts**. A device (controller) may be designed to generate an interrupt when certain status conditions occur, especially when the transfer of a data unit has been completed. Various error conditions may also cause interrupts. Generally, control register bits may be used to enable or disable each type of interrupt which a device may produce.
The driver in this module will use interrupts. I/O interrupts raise some difficulties that have not been encountered using the explicit program-generated interrupts of previous modules. These problems are discussed in the next section.

Handling I/O Interrupts

Module R3 has provided some experience in writing interrupt handlers. Section R3.2 of this manual contains a discussion of some of the important issues. It would be a good idea to reread this discussion now.

I/O interrupts can raise problems not encountered in the handling of programmed interrupts. The reason is that I/O interrupts are caused by external events, and they may happen at any time. We do not have the luxury of deciding on a "good" time to trigger an interrupt. If there exist any "bad" times, you may be sure, that is when the interrupts will occur.

To avoid critical problems, most I/O interrupts may be disabled when they would cause too much trouble. The 8086 provides machine instructions to enable and disable interrupts. These can be invoked using the Turbo C functions `enable` and `disable`, which are described below.

A few critical interrupts cannot be disabled. Even so, I/O interrupts should be completely disabled only for extremely short periods. This disabling affects the clock interrupt, which must occur about 18 times per second. It is possible to disable *selected* I/O interrupts using the *Programmable Interrupt Controller*, as described below.

One problem with I/O interrupts is caused by the fact that an interrupt *cannot* avoid having some effect on the current stack. When an interrupt occurs, the hardware immediately pushes six bytes onto the stack. If the handler is a TURBO C interrupt handler, the software then pushes 18 additional bytes. No matter when the interrupt occurs, the stack pointer `SS:SP` *must* be pointing to a valid stack with sufficient room for these values. If you ever set `SS:SP` to an invalid stack, however briefly, you must first disable interrupts. In particular, recall the discussion about the problem of modifying

`SS` and `SP` in Chapter I2. If you change both `SS` and `SP`, it is not possible to change them simultaneously. Therefore, interrupts should be disabled during this sequence.

If the interrupt handler in turn calls other routines, or makes additional use of the stack, there must be additional stack space to support this use. For this reason it is a good plan for the handler to switch the stack pointer to a private stack with sufficient room.

Another problem occurs whenever an interrupt handler calls a subprogram which could also be called by the interrupted programs. This could lead to an attempt to re-execute the subprogram before a previous execution is finished. Trouble will occur unless the design of the subprogram allows this type of reuse. A program which can be re-entered in this fashion is called *reentrant.* Any program which accesses global (static) data is *not* reentrant. If two instances of such a program are active, they will both try to manipulate the same global data structure.

The subprograms most likely to be called by both the regular program and the interrupt handler are system routines. Unfortunately, many MS-DOS system routines are *not* reentrant. Therefore, your interrupt handlers should make *no calls* to system routines. Calls to C library functions which could in turn access system resources must also be avoided. In particular, your interrupt handler should perform *no input or output,* and this means

*** ***no*** *`printf` statements can be inserted into an I/O interrupt handler for debugging!* ***

It should not be necessary to call any system routines from the interrupt handlers to be developed in this module. Note that the Turbo C calls `inportb, outportb, disable`, and `enable` translate directly to machine instructions; they do *not* call the operating system.

Similar problems can occur if an interrupt handler can be interrupted in such a way that the same handler can be invoked again. We avoid this problem by ensuring that interrupts remain disabled until the interrupt handler returns.

I/O Data Structures

A number of data structures play an important role in the low-level management of I/O devices. Some of the principal ones are discussed briefly in this section.

A record known as a *Control Block* is used to maintain information about the properties and current status of many types of OS resources. You have learned in Module R2 about the Process Control Block. Similarly, each device driver maintains a **Device Control Block (DCB)**. DCBs are discussed more generally in the text. Some of the key information to be stored in a DCB for character devices such as those considered in this module include:

- Allocation status (available, in use by process X, etc.)
- Current operation (read, write, etc.)
- Event flag identifier
- User's buffer descriptor
- Internal buffer and descriptor

Event flags and buffers are discussed below.

Another data structure which was important in process management is the **queue**. Queues are also used in device management; however, their primary use is in connection with I/O scheduling. This usage of queues will be explored in a later module.

An **event flag** is a binary value used for communication between two processes or other concurrent program units. When interrupt-driven I/O is performed on behalf of a program, the program will continue with other activities until the I/O completes. An event flag can be used to tell the program when the I/O has completed successfully. In this case the program will periodically examine the flag, and the device interrupt handler will set the flag when the transfer is complete. This type of communication will be used for the two drivers of this module.

The final data structure types we will consider here are **I/O buffers**. When a character device driver receives a read or write request, a buffer must be specified by the requesting program. This buffer will normally be described by an address and a size. The size is the number of characters to transfer, for an output request; or the maximum space available to receive data, for an input request.

Often the data is transferred one character at a time directly between the requestor's buffer and the device, with no intermediate storage. The DCB must maintain a record of the current position in the buffer and the amount of space (or characters) remaining. It is also important to keep track of the number of characters that have *been* transferred, since the final number may not always equal the requested number, especially if an error occurs. When the transfer is complete, this value should be copied to a specified variable in the requesting program.

To meet these requirements, the DCB should keep track of at least four items of information:

- Current location in the buffer
- Total requested number of characters or buffer size
- Number of characters already transferred
- Address of requestor's count variable

In some cases, internal buffers are also desirable in the driver itself. This is true especially for input devices such as keyboards or communication channels, which may deliver characters before a program is ready to receive them. The text provides a detailed description of the management of a **ring buffer** to serve this purpose. This type of buffer is a fixed-size array managed as a circular list. Pointers are maintained to the current input and output position, along with a count of the number of characters currently in the buffer. Use of a ring buffer is an optional but strongly recommended element of the serial port driver in this module.

R5.3 COMMON ISSUES

The Programmable Interrupt Controller

I/O interrupts generated by device controllers in the PC are not sent directly to the CPU. Instead, they are mediated by a hardware chip known as the Intel 8259 **Programmable Interrupt Controller,** or **PIC**. The PIC is necessary because the 8086 CPU has only one interrupt input signal wire, but there are various sources of I/O interrupts. Up to eight different interrupts are connected to the PIC, which in turn is connected to the single interrupt line of the 8086. When the PIC receives an interrupt request, it forwards the signal to the CPU. The CPU in turn generates a signal to request the identity of the device performing the interrupt. This information is used to select the appropriate interrupt vector.

Some PC and PS/2 models support more than 8 interrupt types. In this case PIC inputs can be shared, or a second PIC may be used, with its output connected to one of the inputs of the main PIC.

The word *programmable* is part of the name of the PIC for good reason. The operation of the PIC can be controlled by software in various ways, and the PIC is associated with several status and control registers. In this project we will be concerned with the *mask register* and the *command register*.

The PIC mask register is accessed at port number 21h. This is a control/status register which can be both read and written. It contains a bit for each of the PIC inputs, indicating whether that input is enabled or disabled. A 0 value represents an enabled interrupt, and a 1 represents an interrupt which is disabled (masked). For the device driver in this module it will be necessary to modify a particular bit in the PIC mask register *without* changing any other bits. The following code is an example of how to set a specific bit in the PIC mask register. This code enables the interrupt associated with bit 7. Similar steps will be used to enable other interrupts.

```
#define PIC_MASK 0x21

  ...
  int mask;
  ...
  disable();
  mask = inportb(PIC_MASK);
  mask = mask & ~0x80;
  outportb(PIC_MASK, mask);
  enable();
```

Note that the `disable ... enable` sequence is necessary to ensure that interrupts will not cause trouble while the mask register is being modified.

The PIC mask bit numbers are also referred to as *levels*, since they define *priorities* for the connected interrupts. Level 0 represents the highest priority, and level 7 the lowest. When an interrupt at a specific level is being serviced, no *lower* priority interrupts can occur.

The other PIC register to be used is the command register, located at port 20h. After each PIC interrupt it is necessary to send an end-of-interrupt (EOI) code to this register. The value of this code is 0x20, which by coincidence is the same as the port number. The purpose of the EOI is to tell the PIC to turn off the current interrupt. If this is not done, the same interrupt will be processed again as soon as interrupts are enabled, and lower-priority interrupts will continue to be blocked.

Passing Parameters

Some of the parameters which are passed to I/O driver routines are pointers to buffers or variables located within the calling program. These pointers may be used by interrupt handlers to access the requestor's data areas. However, in a multiprogrammed environment, the process that was active at the time of the interrupt will generally not be the process which contains these data areas. In general there is no guarantee that the `DS`

value in use by the interrupt handler is the correct one for the requestor's data areas. For this reason, far pointers should be used for address parameters whenever the data areas will be accessed asynchronously after the original call completes. Since we are using the large model, however, these pointers will have the far attribute by default.

In Module R5 you will not be concerned with *calling* driver routines, since they will be called only by the supplied test programs. For this reason the issue of constructing far pointers will be deferred to a later module.

R5 THE SERIAL PORT DRIVER

General Discussion

Module R5 deals with the construction of a device driver for a PC serial port. If you have a PC with a physical serial port, we assume that a standalone video terminal or another PC is directly connected to the chosen port. If the connected device is another PC, it must be connected by a cable which provides a "crossover" of the request and ready signals (This is sometimes known as a *null modem*.). The remote PC should be running a terminal emulation (communications) program. This module can also be completed on machines (primarily laptops) which do not have a physical serial port using a virtual serial port emulator, provided by your TA.

The serial driver must support both input and output via the chosen serial port, though not necessarily both at the same time. It will consist of four control procedures, an interrupt handler, and a set of data structures. The control procedures perform standard operations: *open* (initialize) the port, *close* the port, and *read* and *write* a block of characters. These procedures are described in detail below.

The simplified serial port driver developed for MPX-PC will assume *error-free* communication. In reality, many types of errors may occur to disrupt communication, especially when a modem or other communication device is employed.
The use of a ring buffer to enable typeahead is an optional but strongly recommended element of this project. The discussion in this section will assume that a ring buffer is to be used.

Device Registers

A general-purpose serial port includes a great deal of complexity, a typical port is capable of being set to a variety of different transmission *speeds*, expressed in *baud* (roughly

equivalent to bits-per-second). Typical baud rates range from a few hundred baud to tens of thousands. In addition, the option of using the high-order bit of a character either as data or as an error-checking (parity) bit is provided. Other options are concerned with *framing bits* which come before and after each character. Still others concern *flow control*, the mechanism by which a receiving device may tell a sending device when it is unable to accept more data. The most essential requirement that must be met using all of these options is that the rules of the sending device and those of the receiving device be matched.

The communication mechanisms required for a serial port are often implemented by a special type of chip called a *Universal Asynchronous Receiver-Transmitter (UART)*. The PC's version of a UART is called the INS8250 **Asynchronous Communications Controller,** or **ACC**. The many options available in the ACC lead to a large number of device registers. These include:

- An 8-bit data register for data input, and a separate 8-bit data register for data output. These are called by IBM the *Receiver Buffer* and the *Transmitter Holding Register*, respectively. We will refer to them simply as the *input register* and the *output register*.
- A 16-bit control register used to specify communication speed, called the *Baud Rate Divisor* register. This register is actually accessed as two separate 8-bit parts, the Most Significant Byte (MSB) and the Least Significant Byte (LSB).
- Three additional 8-bit control registers. The *Interrupt Enable* register is used to enable or disable each of the interrupt types associated with the port. The *Line Control* register is used primarily to set options such as parity and the number of data bits. Lastly, the *Modem Control* register controls options that are needed if the port is connected to a remote communication device, and also provides a universal enable/disable for serial port interrupts.
- Three 8-bit status registers. The *Interrupt ID* register indicates if an interrupt has occurred and what its type is. The *Line Status* register indicates the ready status of the port for input and output, along with several error conditions. Finally, the *Modem Status* register indicates various status values associated with a communication device.

The Baud Rate Divisor register must be loaded with a special value which indirectly specifies the desired baud rate. This value is the integer by which the serial port clock speed must be divided to produce the target rate. The serial port clock speed is fixed at 1.8432 MHz, or 1,843,200 "ticks" per second; note that this does *not* depend on the CPU cycle speed, which may be much faster. This value is actually divided by the baud rate times 16. The following statement may be used to compute the divisor:

```
baud_rate_div = 115200 / (long) baud_rate
```

Here `baud_rate` is the desired rate, and `baud_rate_div` is the resulting divisor. It is sufficient to declare each of these variables as type `int`. The baud rates supported by the ACC are: 110, 150, 300, 600, 1200, 2400, 4800, 9600, and 19,200. For this project we recommend a rate of 1200 baud. The same speed is always used for both input and output.
Most of the bits of the remaining control and status registers have a fairly straightforward interpretation. We will identify specific bits as they are needed for the project.

The ten 8-bit registers of the ACC are associated with *seven* (!) I/O ports. These ports are located at seven consecutive addresses. Some PC and PS/2 models support up to three serial ports; we will assume you are using serial port 1 (COM1) or serial port 2 (COM2). The base address of the seven-port sequence (for IBM products) is 0x3F8 for COM1, or 0x2F8 for COM2. *Remember, for other brands these addresses may vary. You must check the configuration of your own system!*

Five registers are associated with the first two port addresses in a peculiar way. Usually, the base address is the input register when read, or the output register when written. Also, the base+1 I/O port is used to access the Interrupt Enable register. However, if bit 7 of the *Line Control* register is set to 1, these assignments change. In that case, the base and base+1 I/O ports are attached to the LSB and MSB, respectively, of the Baud Rate Divisor register!

The remaining assignments are permanent and straightforward. They are:

base+2: Interrupt ID register
base+3: Line Control register
base+4: Modem Control register
base+5: Line Status register
base+6: Modem Status register

Along with the various device registers, the ACC provides four different interrupt types. All of these interrupts are associated with a single interrupt vector and a single PIC level. The assignments for serial ports 1 and 2 are:

	Interrupt ID	*Vector Addr.*	*PIC level*
COM1	0Ch	0030h	4
COM2	0Bh	002Ch	3

Two interrupt types are associated with device ready conditions: The *Receiver Data Available* interrupt occurs when a new input character is available, and the *Transmitter Holding Register Empty* interrupt occurs when the output register has become free. We will refer to these interrupts as *input ready* and *output ready*. Two other types are associated with auxiliary status and error conditions: the *Line Status* interrupt occurs when a data error is detected during transmission, and the *Modem Status* interrupt is caused by certain events associated with a communication device.

Because there are multiple interrupt types but only one vector, a serial port interrupt handler must have a first-level, second-level structure. The first-level handler reads the Interrupt ID register to determine the specific interrupt type. This code is used to select the appropriate second-level handler.

Each second-level handler must perform a characteristic action to *clear* the specific interrupt. These actions are as follows:

Interrupt	*Action to Clear*
Input Ready	Read the Receiver Buffer
Output Ready	None; already cleared by reading the Interrupt ID register
Line Status	Read the Line Status register
Modem Status	Read the Modem Status register

Data Structures

The principal data structure required by the serial port driver is a Device Control Block. The information required in the DCB includes the following:

- A flag indicating whether the port is open;
- A pointer to the associated event flag, this is a far pointer to an integer event flag. This flag is set to 0 at the beginning of an operation, and set to 1 to indicate when the operation is complete;
- A status code, with possible values *idle, reading* and *writing*;
- Addresses and counters associated with the current input buffer.
- Addresses and counters associated with the current output buffer.
- An array to be used as the input *ring buffer*, with associated input index, output index, and counter.

`com_open`

The `com_open` function is called to initialize the serial port. This function has two parameters. The first is a pointer to an integer event flag within the calling program. The second is an integer value representing the desired baud rate.
The prototype for `com_open` is:

```
int com_open (int *eflag_p, int baud_rate);
```

The responsibilities of this routine are: to initialize the DCB; to set the new interrupt handler address into the interrupt vector; to compute and store the baud rate divisor; to set the other necessary line characteristics; and to enable all of the necessary interrupts. **Note** that *as soon as* the device is opened, characters will begin to be accepted in the ring buffer. It should not be necessary to wait until `com_read` has been called.

If there is no error, the value returned should be zero. Otherwise, one of the following error codes should be returned:

-101	invalid (null) event flag pointer
-102	invalid baud rate divisor
-103	port already open

The `com_open` routine should perform the following steps:

1. Ensure that the parameters are valid, and that the device is not currently open.
2. Initialize the DCB. In particular, this should include indicating that the device is open, saving a copy of the event flag pointer, and setting the initial device status to *idle*. In addition, the ring buffer parameters must be initialized.
3. Save the address of the current interrupt handler, and install the new handler in the interrupt vector.
4. Compute the required baud rate divisor.
5. Store the value 0x80 in the Line Control Register. This allows the first two port addresses to access the Baud Rate Divisor register.
6. Store the high order and low order bytes of the baud rate divisor into the MSB and LSB registers, respectively.
7. Store the value 0x03 in the Line Control Register. This sets the line characteristics to 8 data bits, 1 stop bit, and no parity. It also restores normal functioning of the first two ports.
8. Enable the appropriate level in the PIC mask register.
9. Enable overall serial port interrupts by storing the value 0x08 in the Modem Control register.
10. Enable input ready interrupts only by storing the value 0x01 in the Interrupt Enable register.

com_close

The com_close function will be called at the end of a session of serial port use. Its prototype is

```
int com_close (void);
```

If there is no error, the value returned should be zero. Otherwise, the following error code should be returned:

-201	serial port not open

The com_close routine should perform the following steps:

1. Ensure that the port is currently open.
2. Clear the open indicator in the DCB.
3. Disable the appropriate level in the PIC mask register.
4. Disable all interrupts in the ACC by loading zero values to the Modem Status register and the Interrupt Enable register.
5. Restore the original saved interrupt vector.

com_read

The com_read function obtains input characters and loads them into the requestor's buffer. The use of the readahead ring buffer introduces some complexity to this function. Input characters must first be obtained from the ring buffer, if any are pending. If the ring buffer contains enough characters to satisfy the request, then the read terminates immediately. Otherwise, the device status is then changed to reading to notify the read interrupt handler that it should now begin placing characters directly into the requestor's buffer, rather than into the ring buffer.

The prototype for com_read is

```
int com_read (char *buf_p,
int *count_p);
```

buf_p is a far pointer to the starting address of the buffer to receive the input characters, and count_p is the address of an integer count value indicating the number of

characters to be read. At the end of the block transfer, the count value will be modified to show the number of characters that were actually transferred.

If there is no error, the value returned should be zero. Otherwise, one of the following error codes should be returned:

-301	port not open
-302	invalid buffer address
-303	invalid count address or count value
-304	device busy

The `com_read` routine should perform the following steps:

1. Validate the supplied parameters.
2. Ensure that the port is open, and the status is *idle*.
3. Initialize the input buffer variables (not the ring buffer!) and set the status to *reading*.
4. Clear the caller's event flag.
5. Copy characters from the ring buffer to the requestor's buffer, until the ring buffer is emptied, the requested count has been reached, or a `CR (ENTER)` code has been found. The copied characters should, of course, be removed from the ring buffer. *Either input interrupts or all interrupts should be disabled during the copying.*
6. If more characters are needed, return. If the block is complete, continue with step 7.
7. Reset the DCB status to *idle*, set the event flag, and return the actual count to the requestor's variable.

Notice that it is not necessary for `com_read` to enable or disable input interrupts, *except* while the ring buffer is being accessed. These are *always* enabled while the port is open. However, we must not allow the process of removing characters from the ring buffer to be interrupted by an attempt to put a new character in.

com_write

The `com_write` function is used to initiate the transfer of a block of data to the serial port.

The prototype for `com_write` is

```
int com_write (char *buf_p,
int *count_p);
```

`buf_p` is a pointer to the starting address of the buffer containing the block of characters to be written, and `count_p` is the address of an integer count value indicating the number of characters to be transferred. At the end of the block transfer, the count value will be modified to show the number of characters that were actually transferred.

If there is no error, the value returned should be zero. Otherwise, one of the following error codes should be returned:

-401	serial port not open
-402	invalid buffer address
-403	invalid count address or count value
-404	device busy

The `com_write` routine should perform the following steps:

1. Ensure that the input parameters are valid.
2. Ensure that the port is currently open and idle.
3. Install the buffer pointer and counters in the DCB, and set the current status to *writing*.
4. Clear the caller's event flag.
5. Get the first character from the requestor's buffer and store it in the output register.
6. Enable write interrupts by setting bit 1 of the Interrupt Enable register. This must be done by setting the register to the logical *or* of its previous contents and 0x02.

The Interrupt Handler

The serial port interrupt handler has a hierarchical structure. The interrupt vector transfers initially to the first-level handler, which is responsible for determining the exact cause of the interrupt and performing some general processing. This handler in turn selects and calls the specific second-level handler appropriate for the specific interrupt.

The specific steps to be carried out by the first-level interrupt handler are as follows:

1. If the port is not open, clear the interrupt and return.

2. Read the Interrupt ID register to determine the exact cause of the interrupt. Bit 0 must be a 0 if the interrupt was actually caused by the serial port. In this case, bits 2 and 1 indicate the specific interrupt type as follows:

Bit 2	Bit 1	Interrupt Type
0	0	Modem Status Interrupt
0	1	Output Interrupt
1	0	Input Interrupt
1	1	Line Status Interrupt

3. Call the appropriate second-level handler.
4. Clear the interrupt by sending EOI to the PIC command register.

The second-level handler for the *input interrupt* should perform the following actions:

1. Read a character from the input register.
2. If the current status is not *reading*, store the character in the *ring buffer*. If the buffer is full, discard the character. In either case return to the first-level handler. Do not signal completion.
3. Otherwise, the current status is *reading*. Store the character in the *requestor's input* buffer.
4. If the count is not completed and the character is not CR, return. Do not signal completion.
5. Otherwise, the transfer has completed. Set the status to *idle*. Set the event flag and return the requestor's count value.

The second-level handler for the *output interrupt* should perform the following actions:

1. If the current status is not *writing*, ignore the interrupt and return.
2. Otherwise, if the count has not been exhausted, get the next character from the requestor's output buffer and store it in the output register. Return without signaling completion.
3. Otherwise, all characters have been transferred. Reset the status to idle. Set the event flag and return the count value. Disable write interrupts by clearing bit 1 in the interrupt enable register.

In MPX-PC the other two interrupt types should not occur. In case they do, however, handlers should be provided. If a *line status* interrupt is received, just read a value from the Line Status register, and return to the first level. Similarly, if a *modem status* interrupt is received, read the Modem Status register.

R5.6 SUPPORT SOFTWARE

No new support procedures are required for Module R5. Several test programs are provided to exercise your drivers. These are described in the next section.

Six low-level Turbo C functions are used in this module to control interrupts and access I/O ports. These functions are `inportb, outportb, enable, disable, getvect,` and `setvect`. They are fully described in the Turbo C Reference Manual. For convenience, their specifications are summarized here. Definitions for these functions are contained in the Turbo C header file `DOS.H`. This file should be included by your device drivers.

inportb

Reads a byte from a specified I/O port. The parameter `port_id` identifies the port. The prototype is:

```
unsigned char inportb (int port_id);
```

outportb

Writes a byte to a specified I/O port. The parameter `port_id` identifies the port, and the parameter `value` gives the byte to be written. The prototype is:

```
void outportb (int port_id, unsigned char value);
```

enable

Enable all maskable external interrupts. The prototype is:

```
void enable(void);
```

disable

Disable all maskable external interrupts. The prototype is:

```
void disable(void);
```

getvect

Get the value presently stored at a specific interrupt vector, which should be the address of the current handler for that interrupt. It is necessary to use this function to avoid any possibility that the vector will change while being accessed. The prototype is:

```
void interrupt(*getvect(int int_ID))(void);
```

This rather complex-looking prototype indicates that the function returns a pointer to a function, and the function is of type `void interrupt`. An example of usage would be:

```
void interrupt (*oldfunc) (void);
...
oldfunc = getvect(COM_INT_ID);
```

Note that the parameter is the *interrupt ID*, not the interrupt vector address.

setvect

Set a specific interrupt vector to point to a designated handler. The prototype is:

```
void setvect(int int_ID,void interrupt(*handler)(void));
```

An example of usage is:

```
setvect(COM_INT_ID, &new_handler)
```

where `new_handler` is the address of an interrupt handler declared as

```
void interrupt new_handler(void);
```

R5.7 TESTING AND DEMONSTRATION

Two test programs are provided for the serial port driver. Each should be used in a similar manner. `TESTCOMW.C` tests the serial port output by displaying a series of

strings to the “terminal”. `TESTCOMR.C` exercises the output capability and also prompts for and reads input lines.

R5.8 DOCUMENTATION

It is not necessary to update your *User's Manual* for Module R5. However, the structure of your device drivers should be described with extra care in your *Programmer's Manual*..

R5.9 OPTIONAL FEATURES

A number of optional extensions are possible to the basic drivers. Some possible ideas include:

- Provide for a combined status (reading and writing) in the serial port driver, to allow input and output to proceed at the same time.
- Add an *echoing* capability to the serial driver, so that each input character will be sent back to the output as it is received.

R5.10 HINTS AND SUGGESTIONS

The strongest suggestion that can be given for device driver programming is: *read and check carefully!* Remember that `printf` statements *cannot* be used within interrupt handlers (although they are okay in the regular driver functions). For the same reason, the normal Turbo C source debugger may behave unpredictably. If available, the extended Turbo Debugger sold separately or packaged with Borland C++ can be used more reliably to debug interrupt handlers.

A problem in which the system "hangs up" is very probably due to a failure to set or clear interrupts or interrupt vectors properly. The stack pointer is *not* manipulated in this module, so you may focus on solving problems that arise from other causes.