Module R6: Interrupt Driven I/O

Spring 2023

1 Introduction

This module is the final step in the MPX project. In this module you will build a complete simulation of MultiProgramming eXecutive by integrating the required project modules that you have previously completed with this final module. This sixth module of MPX deals with the implementation of a simple **interrupt-driven serial port driver**, which will enable efficient communication between your computer and the serial console. This module faces you with the somewhat tricky problem of designing and debugging correct software for handling external interrupts via a device driver. After your device driver is correctly implemented, it will be integrated into the complete MPX.

The basic elements of MPX are the user interface, process management, continuous dispatching, memory management, **interrupt handling**, and **I/O management**. All the basic capabilities of intercepting interrupts have been covered in preceding modules. I/O management has been covered by previous required modules to the extent of implementing a "polling" function to gather keyboard input. In this module we will add to the previous concepts an interrupt driven I/O request handler and scheduler to process I/O requests, using device drivers which you will develop.

In the Final MPX, dispatching is continuous, and the Command Handler itself is a process which can become blocked waiting for the completion of an I/O request. This means that execution of the Command Handler is interleaved with execution of other processes under the continuous control of the CPU scheduler & dispatcher (`sys_call()`). Each process, including the Command Handler, executes until it performs an MPX system request (call to `sys_req()`) or until an I/O interrupt occurs. System requests or interrupts may cause a new process to be scheduled by a context switch. Most system calls will be I/O requests. In the final MPX all requests for keyboard input or terminal (screen) output are intercepted and handled by MPX. I/O interrupts are caused by I/O events generated via the serial port.

2 Key Concepts

2.1 Continuous Dispatch (R4)

(This is primarily R4 information. If your R4 did not work properly, this describes in detail the concept of continuous dispatch. A working R4 is REQUIRED for module R6. If it does not work properly, this module R6 will almost definitely result in a 0%).

Prior to R4, processes which dispatched have done so only when requested by a specific command, and only for a limited period of time. Most of the time MPX was executing the command handler, which was not considered to be a process.

In a realistic multitasking operating system, however, almost all programs (except the operating system resource managers) are viewed as processes. This includes Command Handlers. Execution of the Command Handler should be interleaved with other processes running on behalf of the same user or other users.

MPX is a single-user system, but it is a multiple process system. In its final form the OS must permit a user to initiate many processes and keep them executing even as the Command Handler is running. This is the approach we take in this module; under normal execution the Command Handler is a process, and it competes for processor time and other resources along with any other processes that may be ready.

When dispatching is continuous, it may happen that no process is ready. In order to simplify the dispatcher and avoid dealing with this special case, we introduce an ***Idle*** process which is *always* ready. This process has the lowest possible priority, but it will run when no other process is ready, thus assuring that there is always some process to run. In particular, the Idle process will run while the Command Handler is waiting for a command, if no other processes have been loaded and activated. This process was introduced in R4 and the body of the function can be found in `user/system.c`.

If continuous dispatching is the normal situation, a more complicated initialization procedure must be carried out to activate an initial set of processes. When a multitasking OS is first initiated, it executes an initialization program which is not a process. This program sets up the initial processes and calls the dispatcher for the first time. Similarly, a special procedure may be required at termination.

In the final MPX module, the main program (`kmain()`) begins by calling the usual initialization procedures. Among other responsibilities, these procedures set up the PCBs, queues and initialize the process management system. The main program then activates two processes: one for the Command Handler, and one for the Idle routine. Both processes are set to the ready, not suspended state, but the Command Handler has the highest possible priority and the Idle process has the lowest.

The main program then triggers `sys_call()` (by issuing an interrupt) to begin interleaved execution of these two processes. Command Handler commands may now be used to explicitly load and activate other processes, which will proceed to share the resources with the original two. All processes except the Command Handler and Idle process may be modified and terminated under user control.

The Command Handler terminates when a *shutdown* command is received. Its final action is to remove itself and Idle as active processes. When the dispatcher detects that no processes are active, it returns to the main program. This program performs the final cleanup and exits from MPX.

2.2 Device Control

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

2.2.1 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 **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 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.

2.2.2 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 x86 architecture. The machine instructions in and out are used to transfer either 8-bit bytes, 16-bit words, or 32-bit double words between a specific CPU register (AL, AX, or EAX, respectively) and a designated I/O port. I/O ports have addresses which may range from 0 to 65,535.

The MPX provides two macros to access the I/O ports of the PC, which may be found in `include/mpx/io.h`. These macros are `inb()` and `outb()`, allowing 8-bit data transfer at a time. These macros are described below in the discussion of support software.

For 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.

2.2.3 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.

2.3 Handling I/O Interrupts

Module R3 has provided some experience in writing interrupt handlers. 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 x86 architecture provides machine instructions to enable and disable interrupts. These can be invoked by passing the proper byte sequence through the desired register via `outb()`.

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 12 bytes onto the stack. No matter when the interrupt occurs, the stack pointer must be pointing to a valid stack with sufficient room for these values. If you ever set it to an invalid stack, however briefly, you must first disable interrupts.

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 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 calls to `sys_req()`, `serial_out()`, or `klogv()` 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 `inb()` and `outb()` macros 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.

2.4 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
- 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 later.

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 returned to the requesting program (by placing it in EAX during I/O completion).

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

- Current location in the buffer
- Total requested number of characters or buffer size
- Number of characters already transferred

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 implementation and management of a **ring buffer** is used 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. Essentially, characters are written and read in circular fashion. Writing occurs in the next sequential position; which will overwrite the oldest character in the event the buffer is "full". Reading occurs in the same sequential pattern, reading the oldest character of the buffer.

Another principal data structures to be added for this module are the IOCBs which record information about the current transfer and any other requested transfers for each device. While it is possible to maintain a combined waiting queue, we recommend a separate queue for each device.

Each active or pending I/O request must be represented by some type of descriptor. This descriptor must identify the process, the device, and the operation. In addition it should indicate the location and size of the transfer buffer. Note that, except for the process ID, this is the information passed as parameters during a system call.

It is possible to define a special I/O descriptor record type to hold this information. Alternately, space for an I/O descriptor may be allocated within each PCB. Note that, in MPX, a process may have only one I/O request at a time.

The IOCB must make it possible to locate the current active request for a given device, and to select a new one from the waiting queue when necessary. The IOCB must also provide the event flag to be used to control I/O for each specific device.

2.5 I/O Management

In the previous section, you have developed a device handler for a device. This handler provides low-level procedures to open and close devices and to transfer blocks of data to or from the device. This transfer could be performed without interrupts, but to maximize concurrency we have implemented interrupt-driven device control and included interrupt handlers which are activated after each character is transferred, in order to begin transfer of the next character, or to terminate the operation if the entire block has been processed.

A set of drivers such as you have implemented forms an important part of a complete I/O management system, but only a part. In your initial implementation these drivers were called directly by test programs. In a complete multitasking operating system they must be called in an orderly way in response to requests that are generated by running processes. This requires an I/O management and scheduling strategy, as described in the *Device Management* chapter of your textbook.

A key element of an I/O management system is an I/O Control Block (IOCB) for each device or channel. This data structure has a role complementary to that of the Device Control Block (DCB) which you have embedded in the driver itself, but it is not the same structure. The IOCB contains information allowing higher-level software to access the device driver.

Some of the information in the IOCB describes permanent characteristics of the device: name, channel number, etc. This information also specifies the interrupt IDs and interrupt vectors associated with the device, and the address of each procedure in the device driver. This latter information must be kept current in case the driver is loaded as needed into transient areas of memory.

Additional information in the IOCB identifies the current operation, if any, that is underway for this device. The ID of the process which requested the transfer is recorded in the IOCB. Finally, the IOCB includes an event flag, as introduced previously, that can be used by an interrupt handler to report back to the system about the current status of the I/O transfer.

A typical I/O device can process only one transfer at a time, but additional requests may be received before the current transfer is completed. For this reason there must be a waiting queue for each device which may contain pending I/O requests. Each entry in this queue describes an I/O request by identifying the process making the request, the buffer location, and the type of transfer to perform. Additional information is required in the case of an addressable device such as a disk; these cases are not included in MPX.

Requests for I/O transfers are initiated by processes using system calls, and passed to an *I/O scheduler*. This routine examines the IOCB to see if the device is busy; if so the request is placed in the waiting queue. This queue is normally organized in a simple first-come, first-served order.

If the device is not busy, the request may be passed to the *read* or *write* procedure in the appropriate device driver. This procedure initializes the buffer, enables the appropriate interrupt, and (if the transfer is a write) outputs the first byte to the device's output port.

In either case, the process is moved from the *ready/running* state to the *blocked* state (unless non-blocking I/O was requested). A context switch is performed, and another ready process is dispatched.

As the transfer of each byte is completed, an interrupt occurs, generated by the device's hardware interface. This type of interrupt-driven I/O was developed in the previous sections. The interrupt handler in the device driver checks to see if more data is to be transferred. If so, it processes the next byte and returns to the interrupted process. Note that in general this is not the process that requested the data transfer.

When the interrupt handler detects that the final byte has been transferred, it must set the event flag in the IOCB. This flag is a signal to the perform an I/O completion sequence. This typically involves returning the requesting process to the ready state and setting an event flag to notify that process that the transfer is complete. The waiting queue for the device must also be checked, and if there are requests waiting, the next transfer must be initiated by a new call to the device driver. Finally, either the interrupted process may be resumed, or a new process (possibly the one which had requested the I/O) may be scheduled instead.

3 Detailed Description & Common Issues

3.1 General Structure

A typical data transfer operation begins with a request such as `READ` or `WRITE` issued from within the normal program code of an application process. This request specifies the transfer of a block of data (typically a line of text) from a program buffer to a specified device, or vice versa. In MPX such a request is made using *int 0x60* which is invoked by the support routine `sys_req()`. This in turn invokes the system call handler provided by your program.

The call handler must process `READ` and `WRITE` requests in the same manner that it has previously processed `IDLE` and `EXIT` operations. In principle, each request invokes a call to the appropriate routine in one of the I/O drivers. But it is not quite this simple. There are multiple processes, and we must be sure that another process is not currently accessing the same device. Although devices are not allocated to processes on a long-term basis, we must ensure that the transfer of each block (i.e., line of characters) is completed before a new transfer begins.

To resolve competition for devices, the system call handler passes each request to the I/O scheduler. This scheduler maintains a record (in the form of a set of queues) of which processes are currently using, or waiting to use, each device. If the requested device is busy, the request is placed in the appropriate waiting queue.

If the device is not currently in use, the request can be honored immediately. In this case a call is made to the appropriate device driver procedure (e.g., `serial_write()`, `serial_read()`). This procedure sets up the transfer information in the IOCB, enables the appropriate interrupts, and begins transfer of the first byte.

In either case the process is placed in the blocked state. The dispatcher is called, and a new process is dispatched.

As each byte is transferred, an interrupt occurs which invokes the interrupt handler in the device driver. This handler determines whether there is more data to be transferred. If so, it processes the next byte and returns to the point of interruption. If not, it sets the event flag, signaling that the device has completed a transfer. This flag will be detected during the next system call, and cause activation of the I/O completion sequence.

The completion sequence has several responsibilities. It must return the process which was performing the I/O to the *ready* state, remove it from any waiting queue, and reinstall it in the ready queue. In addition, the handler must determine if there are requests pending in the waiting queue for the device. If so, it must setup the transfer and call the device driver as described above.

This cycle continues until MPX is terminated by an explicit command. The remainder of this section describes these components and operations in more detail.

3.2 System Call Handler

The system call handler for the Final MPX should have the same structure as in previous modules but expanded functionality. It must now handle `READ` and `WRITE` operations in addition to `IDLE` and `EXIT`. These operations will be passed to the system call handler by `sys_req()`.

Each time `sys_call()` is invoked, it should check to see if any device event flags are set. If they are, it should perform the required I/O completion sequences as described below.

Each I/O system request should be passed in turn by the system call handler to the I/O Scheduler. This procedure will check the state of the device and determine whether to initiate I/O or to place the request in a waiting queue. The request parameters (which are still on the stack), and the identity of the calling process, must be provided to the I/O Scheduler. The Scheduler should check the validity of the parameters, and return an error code if the request is invalid.

3.3 I/O Scheduler

The I/O Scheduler processes input and output requests. Its first task is to examine the system call parameters and ensure that the request is valid. In particular, the operation must be `READ` or `WRITE`; the device must be a recognized one, and the operation must be legal for the specified device. If these conditions are not met, an error code should be returned.

The next task is to check the status of the requested device by accessing its IOCB. If there is no current process using the device, then the request can be processed immediately. In this case the requesting process is made the active one by installing a pointer to its PCB in the IOCB. The buffer address and length must also be placed in the IOCB. The appropriate driver procedure is then called.

If the device was busy, the request is installed on the waiting queue. The information in each queue element must include the PCB pointer, device ID, and operation code.

Finally, the I/O Scheduler returns to the system call handler, which in turn will invoke the dispatcher to dispatch the next process.

3.4 I/O Processing

I/O processing for the duration of the block transfer is managed by the routines of the device driver, primarily the interrupt handler. Note that while the transfer is continuing under interrupt control, processes other than the requesting process will be executing.

When the interrupt handler detects that the entire block has been transferred, it sets the event flag to request the I/O completion procedure.

3.5 I/O Completion

Each time the system call handler is invoked, one of its responsibilities is to examine the data structures that represent active I/O transfers, to determine if any of their event flags are set. For each flag that is set, the appropriate completion sequence must be performed.

First, the active process must be switched from the `blocked` state to the `ready` state. This may require both setting of state variables and adjustment of the appropriate queues.

Second, the number of bytes transferred must be provided as the return value seen by the process that requested the action. This is accomplished by setting EAX in that process's context to the number of bytes transferred indicated in the IOCB.

Finally, the active data structure must be cleared, signaling that no request is currently active for this device.

The routine then searches the waiting queue for another process waiting to use the now-available device. If such a process is found, the I/O scheduler is called to start the I/O. This sequence is repeated for all devices with a just-completed I/O request.

3.6 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 had 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. One of these (interrupt 2) is historically connected to a second PIC, allowing for a total of 15 possible hardware interrupts (since interrupt 2 is used for multiplexing, it is not a valid interrupt itself). 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.

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 0x21. 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 disables the interrupt associated with bit 8. Similar steps will be used to disable other interrupts.

```
cli();
int mask = inb(0x21);
mask |= (1<<7);          // Mask (disable) hardware IRQ 8
outb(0x21, mask);
sti();
```

Note that the `cli()` ... `sti()` 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 0x20. 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.

3.7 Passing Parameters

Some of the parameters which are passed to I/O driver routines are pointers to buffers located within the calling program. These pointers may be used by interrupt handlers to access the requestor's data areas. However, in a multiprocessing environment, the process that was active at the time of the interrupt will generally not be the process which contains these data areas.

4 The Serial Port Driver

(This is the bulk of the work for this module. The following sections read similar to a "cook-book" on how your interrupt interface should be implemented.)

4.1 General Discussion

Module R6 deals with the construction of a device driver for a PC serial port. Using Qemu, we assume a terminal emulator is directly connected to the serial port.

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 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 discussion in this section will assume that a ring buffer is to be used.

4.2 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, or even billions with modern Universal Serial Bus (USB) devices. 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 is almost certainly 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 19,200 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. Traditional hardware supports up to 4 serial ports, which Qemu emulates. The base addresses are listed in the table below.

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.

Device	Base I/O Address	Hardware IRQ (PIC Level)	Interrupt Vector
COM1	0x3F8	4	0x24
COM2	0x2F8	3	0x23
COM3	0x3E8	4	0x24
COM4	0x2E8	3	0x23

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

4.3 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;
- The event flag – 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

4.4 `serial_open(device dev, int speed);`

The `serial_open()` function is called to initialize the serial port. This function has two parameters. The first is a the device to open (e.g. `COM1`). The second is the desired 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 `serial_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

Please see `serial_init()` in `kernel/serial.c` for a **sample** on serial initialization. The `serial_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, setting the event flag to 0, and setting the initial device status to idle. In addition, the ring buffer parameters must be initialized.

3. 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.

4.5 `serial_close(device dev);`

The `serial_close()` function will be called at the end of a session of serial port use.

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 `serial_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.

4.6 `serial_read(device dev, char *buf, size_t len);`

The `serial_read()` function obtains input characters and loads them into the requestor's buffer. The use of the read-ahead 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 `dev` parameter indicates which device to read from. The `buf` parameter is a pointer to the starting address of the buffer to receive the input characters, and `len` is an integer count value indicating the number of characters to be read.

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 `serial_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 new-line (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 `serial_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.

4.7 `serial_write(device dev, char *buf, size_t len);`

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

The `dev` parameter is the destination device, `buf` is a pointer to the starting address of the buffer containing the block of characters to be written, and `len` is an integer count value indicating the number of characters to be 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 `serial_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.

4.8 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 a new-line, 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 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.

5 Support Software

No new support procedures are required for Module R6 from us. Everything you need should already exist in the MPX, or will be created by yourself.

Two low level C macros are used in this module to control interrupts and access I/O ports. These macros are `inb()` and `outb()`. For convenience, their specifications are summarized here. Definitions for these macros are contained in the MPX header file `<mpx/io.h>`. This file should be included by your device drivers.

5.1 `char inb(uint16_t port);`

Reads a byte from a specified I/O port. The parameter `port` identifies the port.

5.2 `void outb(uint16_t port, char data);`

Writes a byte to a specified I/O port. The parameter `port` identifies the port, and the parameter `data` gives the byte to be written.

6 Documentation

You must finalize your documentation in this module. Your *User's Manual* must contain documentation on all commands and error messages. The *Programmer's Manual* must include descriptions of all of *your* procedures and data structures. If you have been adding to this manual in each module (as required), you will have only a limited number of additions.

7 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.
- Create two instances of Command Handler, one attached to COM1 and the other attached to COM2. They should both be able to run simultaneously and independently.

8 Hints and Suggestions

The strongest suggestion that can be given for device driver programming is: *read and check carefully!* Remember that `serial_out()` or `klogv()` functions cannot be used within interrupt handlers. For the same reason, the normal debugger may behave unpredictably.

A problem in which the system "hangs up" is very probably due to a failure to set or clear interrupts or interrupt vectors properly.

Remember that the stack of whatever process happens to be running at the time an interrupt occurs is the one that gets used while processing that interrupt. In the (common) case of that process being Idle, there should be plenty of stack to use. But be judicious with your stack usage nevertheless. Your entire ISR call stack shouldn't require more than a handful of variables, and certainly no arrays.

The software for this module is more complex than any other, and a great deal of concurrent activity must be managed. However, a thoughtful approach will keep you from being overwhelmed. The actual amount of new software needed for this module is small. Do not make changes to previous components unless absolutely necessary. If this rule is followed, you can be sure that any difficulties are localized to a small number of routines.

When the basic processes are operating, choose new processes to load carefully, to exercise one new feature at a time.

Consider very carefully the situations in which interrupts may occur, and the possibly conflicting actions which could be taken in response to those interrupts. If an interrupt occurs during queue manipulation or memory allocation, and the interrupt handler attempts to invoke the same procedures, problems will occur. It may be necessary to disable or defer interrupts during certain critical operations.