Chapter IX
I/O controllers and I/O devices

I/O devices are the pieces of hardware, often outside the computer cabinet, which perform input and output. Examples of I/O devices (typical interface formats in brackets) are: modems and other serial devices (RS232 serial interface), printers (parallel port interface), disk drives (IDE, SCSI), video monitors (analogue video signal), and local area network cables and transceivers (high frequency serial signal).

I/O controllers are the pieces of digital electronics, inside the computer cabinet, which transform commands from the processor into the interface signals understood by the I/O device.

IX.1 Example I/O controller – UART

The I/O controller used to connect a serial device to a computer is variously known as a UART (Universal Asynchronous Receiver Transmitter). The I/O device is connected to this I/O controller via a serial port interface, which consists of a minimum of three wires: **Tx Data**, **Rx Data** and a **Ground** wire. Data is transmitted one character (7 or 8 bits) at a time, and each character is transmitted serially along the **Tx Data** wire (to the device) or the **Rx Data** wire (from the device). When no character is on the wire, the wire is at the logic **1** voltage. Characters are transmitted as (typically – there are variants) 10 or 11 bits of data, starting with a *start bit*, value `0`, then seven or eight bits of data for the character (least significant bit first), followed by a *parity bit*, followed by a *stop bit*, value **1**. The parity bit is chosen so there is always an odd (or as a settable option, even) number of **1** bits in the data bits and parity bit together. So the receiver can detect a transmission error which flips the logic value of a single bit within the data or parity bits.

The UART I/O controller translates between this serial data format, and the format in which a processor handles characters, i.e. as a byte, transferred in parallel down a set of eight wires. *Shift registers* are employed to convert between the serial and parallel data representations. An abstract block diagram of a UART may look like figure IX.1.

Received characters may be accessed by the CPU from the Rx data register, and to transmit a character, the CPU loads it into the Tx data register. The receiver status signal indicates that a character has arrived, and the transmitter status signal that the character previously loaded into the Tx data register has

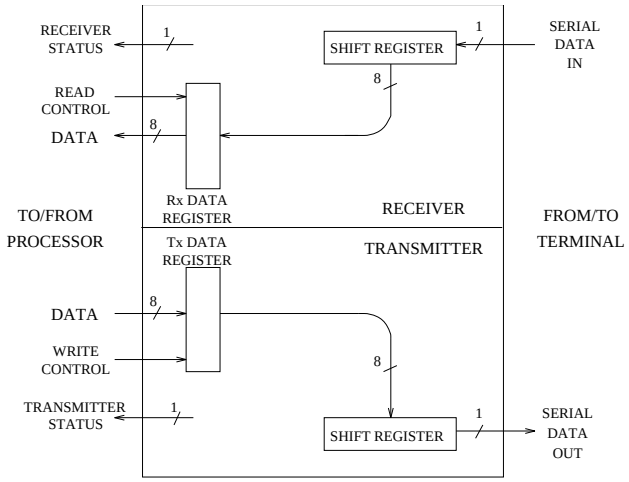


Figure IX.1: UART block diagram.

been completely transmitted.

IX.2 Interconnection of CPU and I/O controllers

It would be possible to build the Tx and Rx data registers of the UART into the CPU itself. Data could then be moved directly between these registers and the status wires could be tested by inventing new conditional branch instructions. This is the way the earliest computers performed I/O transfers, but it is obviously not suitable for a computer with a range of I/O controllers, especially if the number and type of I/O controllers is to be flexible.

An improvement is therefore to keep the I/O controller registers outside the CPU, and connect the two via an *I/O bus*, which is very similar to the CPU - memory bus. The I/O bus has three parts: the data bus (typically eight bits wide), the control bus (READ and WRITE signals) and an address bus, which indicates which register in which of the I/O controllers the CPU is accessing. Each I/O controller register is allocated a unique address. I/O transfers are then achieved using special I/O load and store instructions which move a data byte between a general purpose CPU register and an I/O controller register.

This deals with the data transfers; what about the status signals? It is not feasible to have separate status wires from many I/O controllers all feeding directly into the CPU for testing, so the status signals are fed into a *status register* within the I/O controller itself. That register can be accessed by the CPU in the same way as the data registers in the I/O controller and is allocated its own address

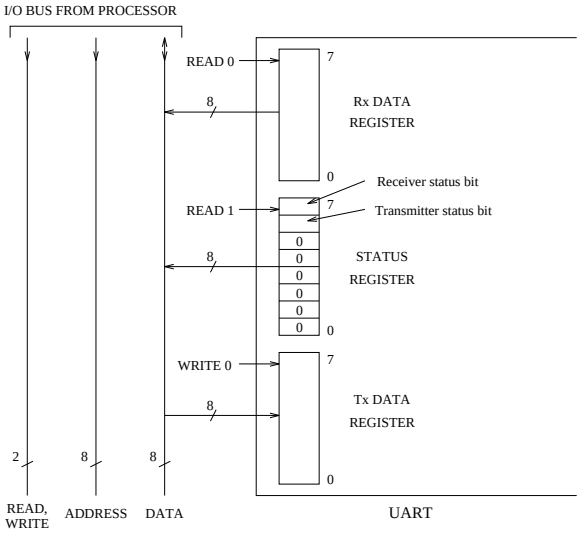


Figure IX.2: UART with status register, connected to an I/O bus.

on the I/O bus.

IX.2.1 Memory-Mapped I/O

The I/O bus and the memory bus of a CPU are very similar. Almost all modern CPU designs dispense with the I/O bus, and use the memory bus to access the I/O controller registers. This is done by allocating a portion of the memory address space to I/O controllers (and obviously putting no memory in that portion).

As before, each register of each I/O controller will be allocated a unique address. However, it is no longer necessary to have special I/O load and store instructions, as the I/O controller registers look, to the CPU, exactly like memory locations, so can be accessed using the ordinary memory load and store instructions.

IX.2.2 Polling and interrupts

The mechanisms described above enable the CPU to access data in registers in I/O controllers and, by reading an I/O controller status register, the CPU can find out when the I/O controller is ready for more data to be transferred. In practice it is not feasible for the CPU in a general purpose computer to keep scanning the status registers of all the I/O controllers to see if they wish to have data transferred — it would require every user program to call the operating system at regular short intervals to do the scan. This regular scanning of sta-

tus registers is called *polling*, and it is normally only found in single-function computers in embedded control applications inside cars, appliances etc.

In general purpose computers, an I/O controller generates an *interrupt* when it is ready for data transfer, via an interrupt request signal from the I/O controller to the CPU. When the CPU sees the interrupt request signal, the operating system exception handler is called, and can identify which I/O controller interrupted. It can then look in that I/O controller's status register to ascertain the exact cause of the interrupt, and can deal with the requested data transfer.

IX.3 Disks and Disk Controllers

Floppy and hard disk surfaces are divided into concentric *tracks* (hundreds per surface on a hard disk), and each track is divided into *sectors* (typically a few tens per track). A sector holds a fixed number of bytes, variously 512, 1024 etc. for different disk formats.

The *access time* of a disk is the time it takes the head to get into position above the required sector, and is made up of two parts, the track *seek* time (the time it takes to move the head to the required track), plus the rotational latency to get to the required sector. Data transfer rates are largely determined by how closely the bits can be packed on the disk surface (the *recording density*).

All disk drives require controllers, to interface the disk head movement and data read/write electronics to something the CPU can understand. The two common standards for CPU communication with these controllers are *EIDE*, and *SCSI*. An EIDE bus is very similar in structure to the processor's own memory bus so very little hardware is required to interface the processor to the EIDE bus. In the case of SCSI, a SCSI bus controller is needed, through which the CPU operates the SCSI bus.

The CPU interface of a floppy or EIDE disk controller is the usual bank of registers mapped into the CPU I/O bus address space, or memory-mapped. The individual functions carried out by the disk controller for the CPU are quite complex, and one register, written by the CPU, is usually designated the *command register* — the value written to this register determines which complex function the controller carries out next.

Typical commands might be:

Seek n Move the head to track n.

Read Sector m Wait for sector m to rotate under head, then read the data from it.

Write Sector m Wait for sector m, then write new data into it.

Format Track Initialise the track, writing the sector marks and identifying information.

IX.3.1 Using a disk controller

If the CPU wishes to read the data from a particular sector (the directory system provides the mapping from parts of files to physical disk sectors), it must first issue a *seek* command to the required track. Tens or hundreds of ms later, the seek will complete, and the disk controller will interrupt the CPU. Next the CPU issues a *read sector* command for the required sector. Again there may be a long delay for the rotational latency, so the controller will interrupt again when the head is above the requested sector. Now the data must be transferred, typically one 16-bit word at a time, and at hard disk rates (i.e. 20Mbit/sec or more).

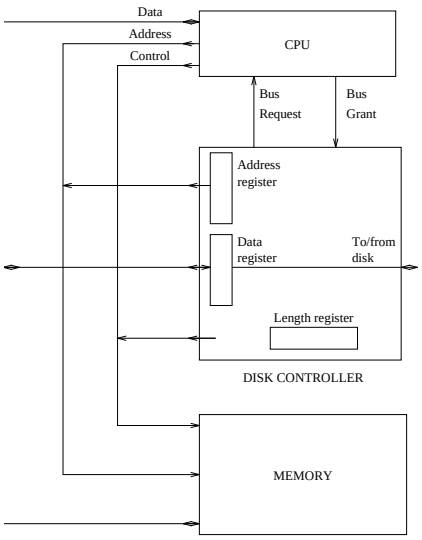
Dealing with an interrupt takes a considerable amount of time making it impossible to synchronise the disk data transfer by an interrupt for each word to be transferred. An alternative is for the OS to sit in a loop, polling the disk controller status register to find out when each word is ready. However, that would require the computer to do nothing else while the operating system transfers a complete sector from disk.

Ideally, what we want is to relieve the CPU entirely of the task of moving the data between memory and the disk controller, and this can be done if we arrange for the disk controller to be able to access the memory directly itself.

IX.4 Direct Memory Access

The disk controller needs to be able to transfer a block of words to or from memory (for read sector and write sector commands respectively). For this, additional hardware is needed, called a *direct memory access* (DMA) controller.

The DMA controller includes an address register, holding the address in memory of the next word to be read/written, a data register, through which data is transferred from/to memory, and a length register holding the number of words still to be transferred. To read a word from memory, the DMA controller outputs the contents of the address register onto the memory address bus, and asserts the *read* signal in the memory control bus. The memory responds with the data word, which the DMA controller copies into the data register, to be sent as a serial data stream to the disk. The address register is then incremented, ready for the next word, and the length register decremented. The DMA operation is complete when the length register reaches zero. Writing to memory is similar, except that the DMA controller *outputs* the data words onto the memory data bus.



To read or write a sector, the CPU must first issue the command to seek to the correct track, as before, then write initial values into the DMA controller address register (i.e. a pointer to the data buffer in memory) and the length register, and then issue the disk command *read sector m* or *write sector m*. The disk operation now proceeds without further CPU intervention — once the head is above the required sector, the data is transferred between the disk and the memory buffer by the DMA controller, and the CPU is interrupted only when the operation is complete.

We now have two devices, the CPU and the disk DMA controller, both controlling the memory bus — how do we ensure that they don't both try to use the bus at the same time? This is achieved with a circuit in the CPU called the *bus arbiter*, which arbitrates between requests by the CPU to use the memory bus, and requests by the DMA controller. The DMA controller is connected to the arbiter by two wires, a signal to the arbiter called *bus request* and an acknowledgment signal from the arbiter called *bus grant*.