# An overview of microcontrollers.

Microcontrollers are often described as single chip computers. They contain a microprocessor core, (often) some memory and various “peripheral” devices such as parallel i/o ports, serial i/o ports, timers, analogue to digital converters (ADC's) and various other special function sub-systems.

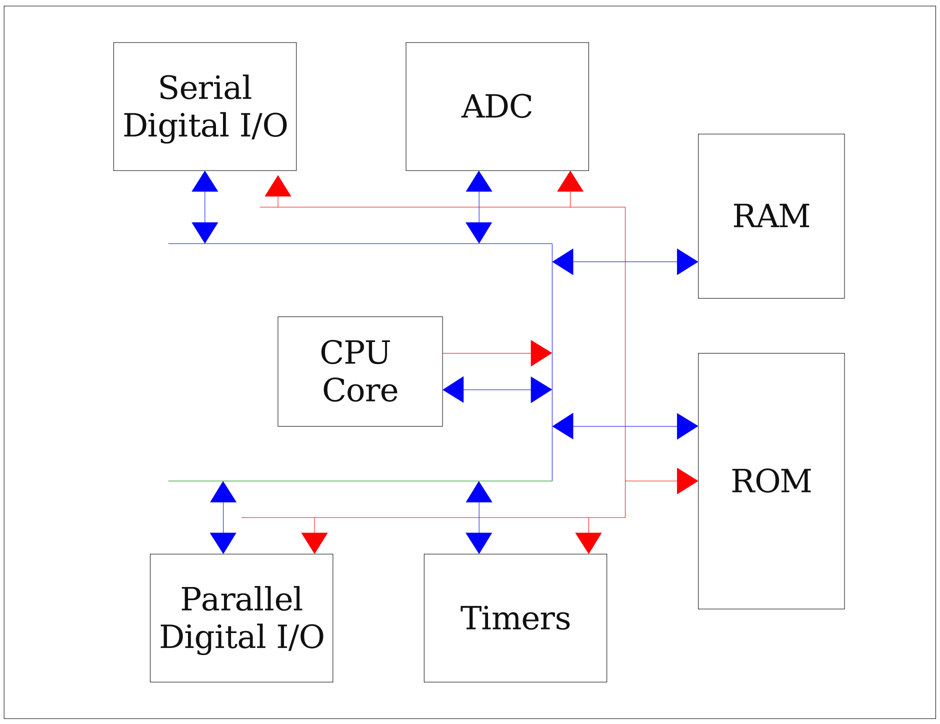

*Figure 1. Microcontroller elements. Red lines represent the address bus, blue lines represent the data bus*

## <u>The CPU</u>

The central processor unit is responsible for executing stored program (in ROM) and managing the peripherals. It fetches numeric instructions from memory (opcodes) one by one, interprets them and carries out some operation as a result. Programs consist of a collection of these opcodes mixed with (numeric) data. The CPU works its way sequentially through these instructions, sometimes jumping from place to place as a result of program design or as a result of operating conditions.

Figure 2 shows the sort of components commonly found in CPU's. Registers are a little like internal memory storage areas. These are useful for interim calculation results (this reduces the number of reads/writes to external memory which is usually slower). Some registers have special functions such keeping track of where the next instruction is supposed to come from in memory.

The arithmetic logic unit is responsible for carrying out calculations. In some CPU's this can be quite simple; perhaps only supporting add, subtract and basic logical operations. In more sophisticated CPU's, there may be several arithmetic units, some/all capable performing advanced floating point operations.

The instruction decoder's job is to translate numeric opcodes into sequences of actions.

The data bus is a collection of wires or tracks that are used to transport numbers into and out of the CPU. A logic '1' on each line of the bus is represented by a particular voltage – commonly 5V or 3.3V. Logic '0' is commonly represented by a 0V signal. Any device that the CPU needs to communicate with is connected (in parallel with others) to the data bus.

The address bus is another collection of wires/tracks. Its purpose is to select which of the external devices (or memory locations) is allowed to use the data bus. Remember, each wire of the data bus is just a simple conduction and is only capable of conveying one bit of data at a time. If for example, two devices attempt to write to the data bus at the same time; their signals will become confused and lost. In simple systems, the CPU alone controls the address bus.

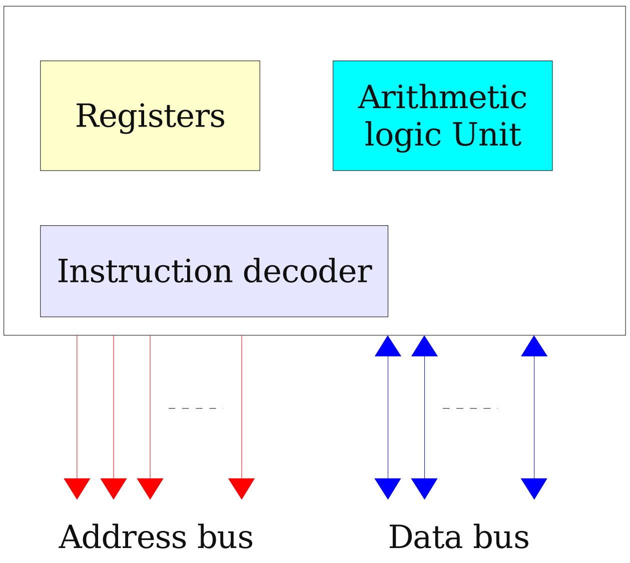

*Figure 2. CPU elements.*

## RAM & ROM

RAM (random access memory) is used for storing values that are liable to change during the course of the execution of a program. RAM contents are (usually) lost each time systems are turned off/on. As a result of this, RAM is not very useful for long term program storage – you don't want to have to reload the control program for your photo copier each time you plug it out. ROM (read only memory) does not lose its contents when power is removed. Thus it is generally used for storing programs (but not variables). There are different technologies used to implement ROM program memory such as EPROM, EEPROM and Flash memory.

Both ROM and RAM consist of arrays of storage locations, often arranged in byte or multiples of bytes in some sort of module (such as a chip). These modules/chips have an address bus which is controlled by some external device (CPU). The address bus selects which of the internal memory locations is connected to the module's data bus.

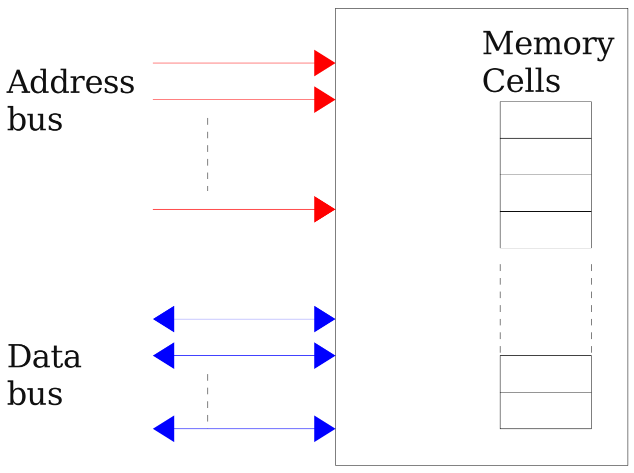

*Figure 3: Memory module structure*

## Parallel Digital I/O

Microprocessors (and microcontrollers – micro's for short) send and receive information to the outside world using ports. There are many different types of port however the simplest is the parallel I/O port. These ports can be thought of as memory cells or registers that are connected to the CPU core using the data bus and also to the outside world via pins on the side of the micro. In the case of parallel ports, each bit in the port register is connected to a pin on the chip. If a given bit is at logic one, then the pin is drive to whatever voltage represents logic '1' for that system (commonly 5V or 3.3V). Similarly bits at logic '0' drive 0V out on the appropriate pin. Figure 4 shows how port I/O takes place in a generic microcontroller system.

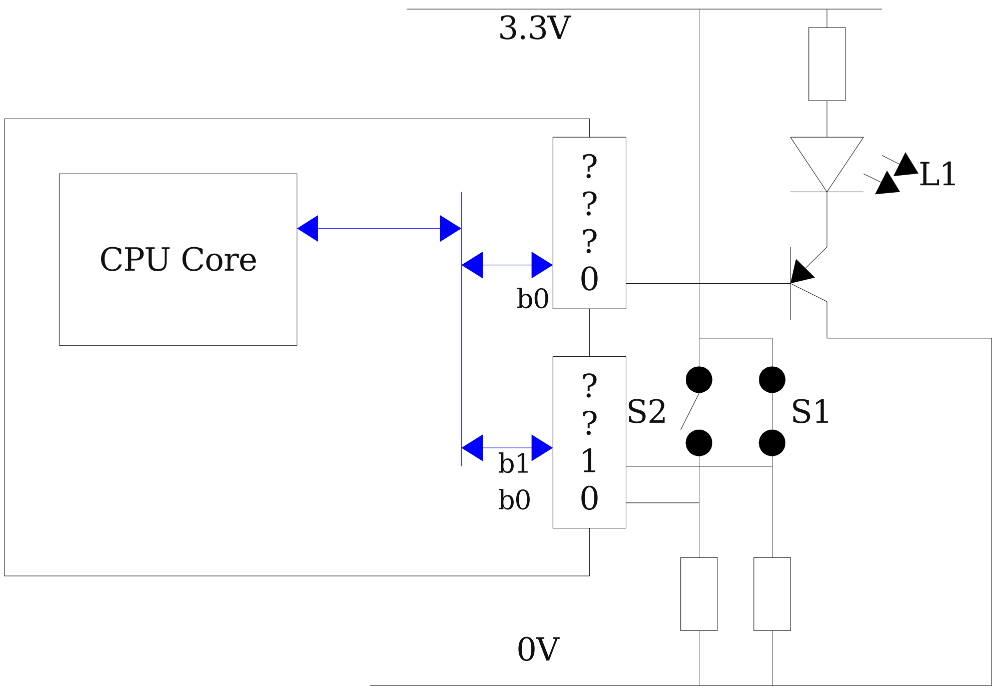

*Figure 4: Parallel port I/O. Shown are two 4 bit ports - one input, one output. To turn the transistor and hence L1 on, its base must be brought to 0V. Bringing it to 3.3V by writing a logic '1' to the appropriate bit (b0) causes L1 to turn off. The closure of S1 causes b1 of the input port to be driven high (logic '1' or 3.3V). S2 is open so the resistor shown pulls b0 low (logic '0' or 0V).*

## Serial I/O

Serial communications requires the sender to send data 1 bit at a time a rate agreed with the intended receiver. Each bit is given a “time slot”, the sets the transmit wire/track to the correct logic level for each bit's time slot. The receiver measures the voltage arriving from the sender at the middle of each bit's time slot. It can thus decide whether a '1' or '0' has been sent for a particular bit. Reliable operation requires precise timing. The receiver must look at the correct times for each bit. There are two common ways in which this is achieved : the sender and receiver each have very accurate clocks that are periodically re-aligned – this is called Asynchronous transmission. The other method requires both parties to share a single clock source – Synchronous transmission. Transmitting serial data is cheaper than parallel transmission as there are fewer conductors involved. Mice, keyboard, USB and Ethernet are all examples of serial transmission.

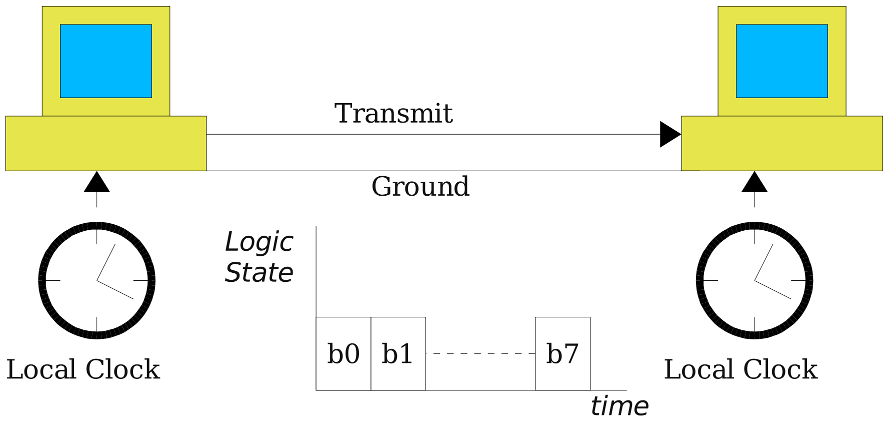

*Figure 5a: Asynchronous serial transmission*

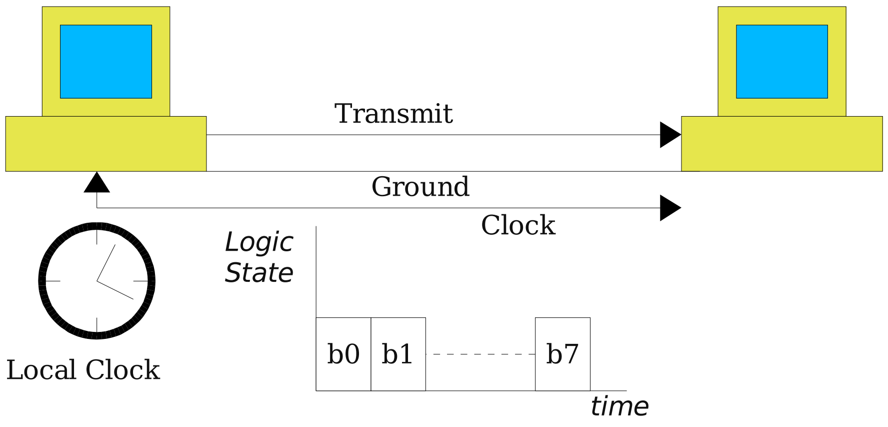

*Figure 5b: Synchronous serial transmission*

## Timers

Timers are typically constructed using a clock source and a counter. Counters count clock periods that are input to them. Some counters count up, some down. Counters have a limited "width" or number of bits and as a result can only count up or down so far without "rolling over". An 8 bit up counter for example can only count up to 255 decimal; one more period input will cause it to overflow and revert to zero. A 16 bit down-counter will decrement down to zero, and given one more input it will roll around to 65535 (0xffff). When this roll over takes place, most counters emit some kind of signal.

Counter/timer application 1: Divide down frequency by N. Figure 6 shows a down-counter and it associated “reload” register. When the down-counter attempts to roll below zero, it takes whatever value is in its reload register and resumes its downward count from there. This arrangement can be used to “divide” a frequency by an arbitrary value – 0x100 (or 256) in this case. A microprocessor could use this sort of arrangement to divide down a high speed system clock down to a 1Hz tick – useful for keeping track of time of day.

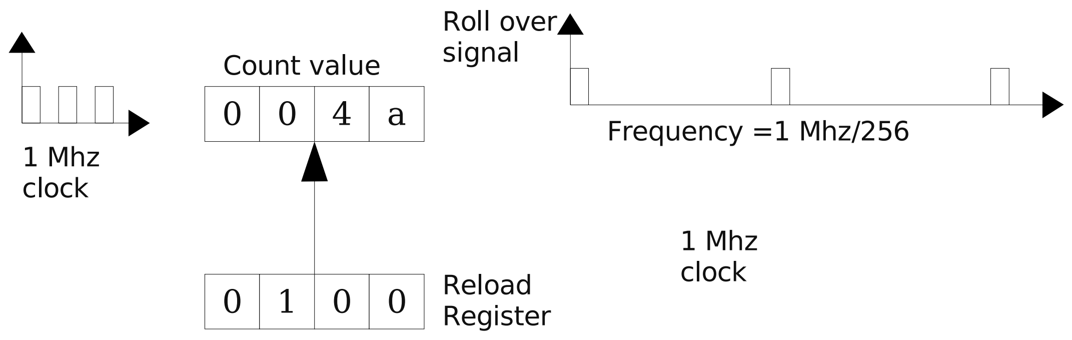

*Figure 6. Dividing down a clock signal*

Figure 7 shows how a counter/timer can be used to measure time periods. Suppose a microprocessor is required to measure the duration of some external event. If it raises the “Enable” signal shown to logic 1 AND, the external event signal is high then the gate output will switch in accordance with the clock. When the event signal goes low, the gate is effectively closed. The counter value remains static and the number of pulses (microseconds) can be read from the counter.

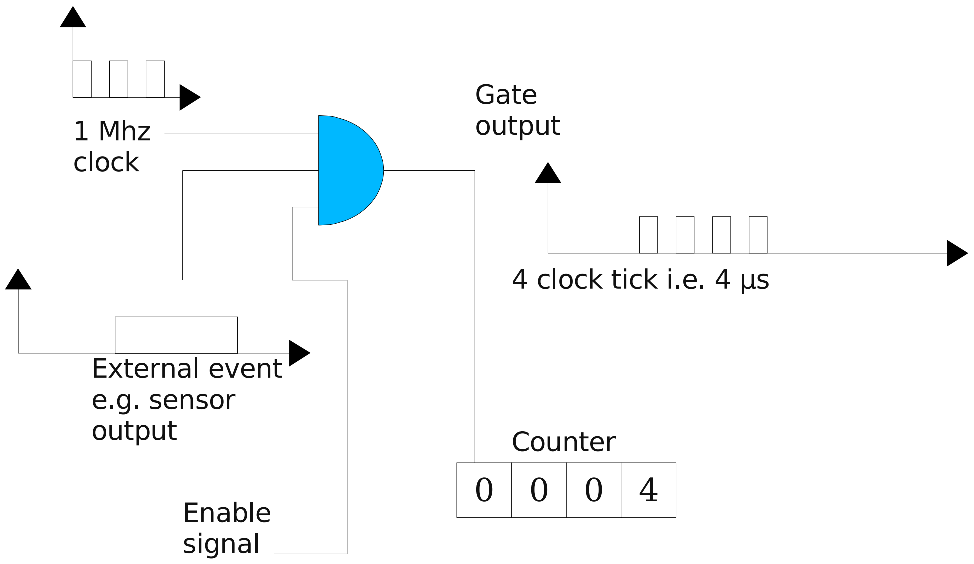

*Figure 7. Period measurement using a timer/counter.*

Apart from the above two applications, counters can simply be used as event counters allowing the microprocessor to perform other tasks.

**ADC's (and DAC's)**

Frequently microprocessors are required to process non-digital i.e. analogue (or continuous) signals. In order for them to do this, a special device called an Analogue to Digital Converter is required. This device accepts an analogue input and translates this to a digital number whose size is proportional to the magnitude of the analogue signal. Each Analogue to Digital Converter has an associated input voltage range and an output numeric range.

Example: The ADS7891 ADC from Texas Instruments is a 14 bit converter that typically operates at 5V. Its input signal range is given as 0 to 2.5V (assuming it operates with a 5V power supply). Its conversion time is quoted as around 250ns. The device has an 14 bit parallel output (many ADC's use serial outputs). Figure 8 shows how such an ADC could be used in conjunction with a microcontroller. The microcontroller instructs the ADC to “convert” the analogue input to a digital signal. Internally the ADC takes a snapshot (sample and hold) of the analogue input and, in this case uses an successive approximation to map this signal to a proportional digital number. While this is going on, the microcontroller must wait. It monitors the status of the “conversion finished” (sometimes called “End of conversion”) signal. When the conversion is done, the 14 converted result can be read using 14 input port bits.

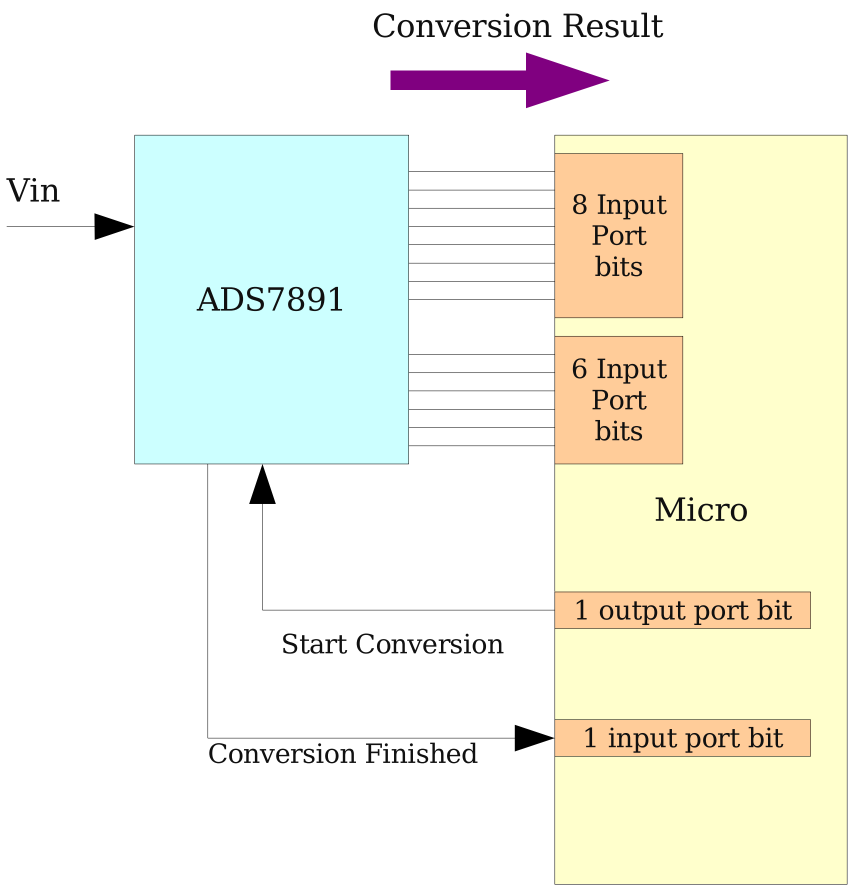

*Figure 8. Sample (approximate) interfacing of an ADC with a microcontoller.*

Some ADC's have multiplexed inputs which allow them to convert several input signals in sequence. Others have a address/data bus style of interface to the microprocessor allowing them to be mapped into the memory address space of a system. Many microcontrollers have integrated multichannel ADC's.

Digital to Analogue converters perform the inverse of ADC's. They accept a digital input signal and output a proportional analogue signal. Historically, DAC's have rarely been found integrated into microcontrollers for fabrication reasons. If an analogue output is required for a microcontroller without an internal DAC then an external IC and support circuitry must be added. Alternatively, it may be possible to generate a pseudo analogue output by controlling the duty cycle of an output port pin. For example, if a digital output pin spends half of its time at 5V and the rest of its time at 0V its average voltage is 2.5V. By varying the percentage "high time" the average output voltage may be controlled. This is know as Pulse Width Modulation (PWM). The pulsed output can be filtered to create a continuous analogue signal using a simple filter as shown in Figure 9.

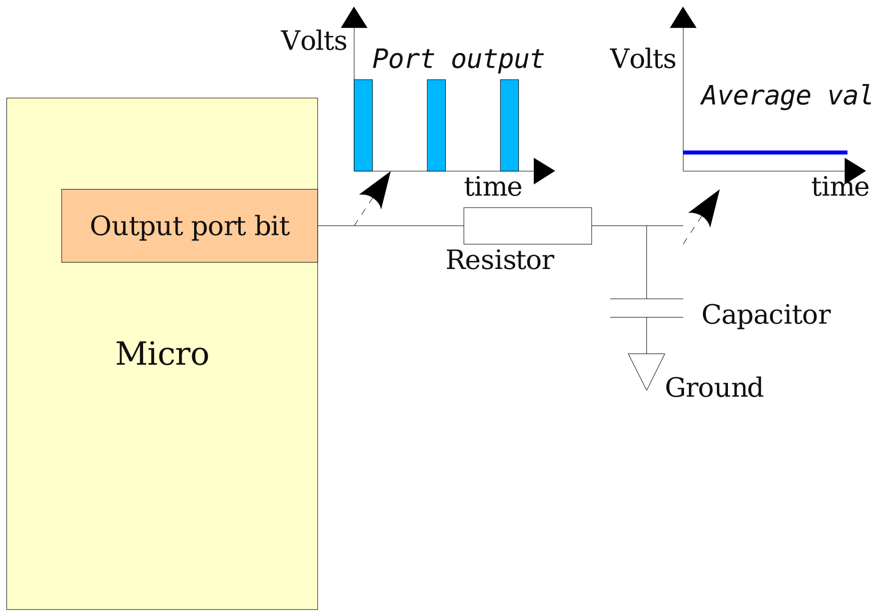

*Figure 9: Analogue output by PWM*

**Interrupts.**

Very simple microprocessor systems are often built which execute only one task. For example, a burglar alarm system may spend all of its time reading the status of various sensors checking for an alarm condition. Similarly, a microprocessor in a musical greeting card may spend its time generating analogue outputs. More complex microprocessor systems are often required to monitor and control man inputs and outputs. In such circumstances, the allocation of CPU time for driving the outputs and monitoring the inputs must be carefully balanced. One way of doing this is to scatter calls to the various i/o routines throughout the program code. Consider for example a modern PC running a gaming application. The CPU must "run around" the mouse and keyboard drivers to check for new inputs while at the same time update the video display. As systems become more complex, this "running around" becomes more convoluted and ultimately impractical. Interrupts offer a solution. Figures 10a and 10b illustrate how a gaming console could be implemented using interrupt driven I/O.

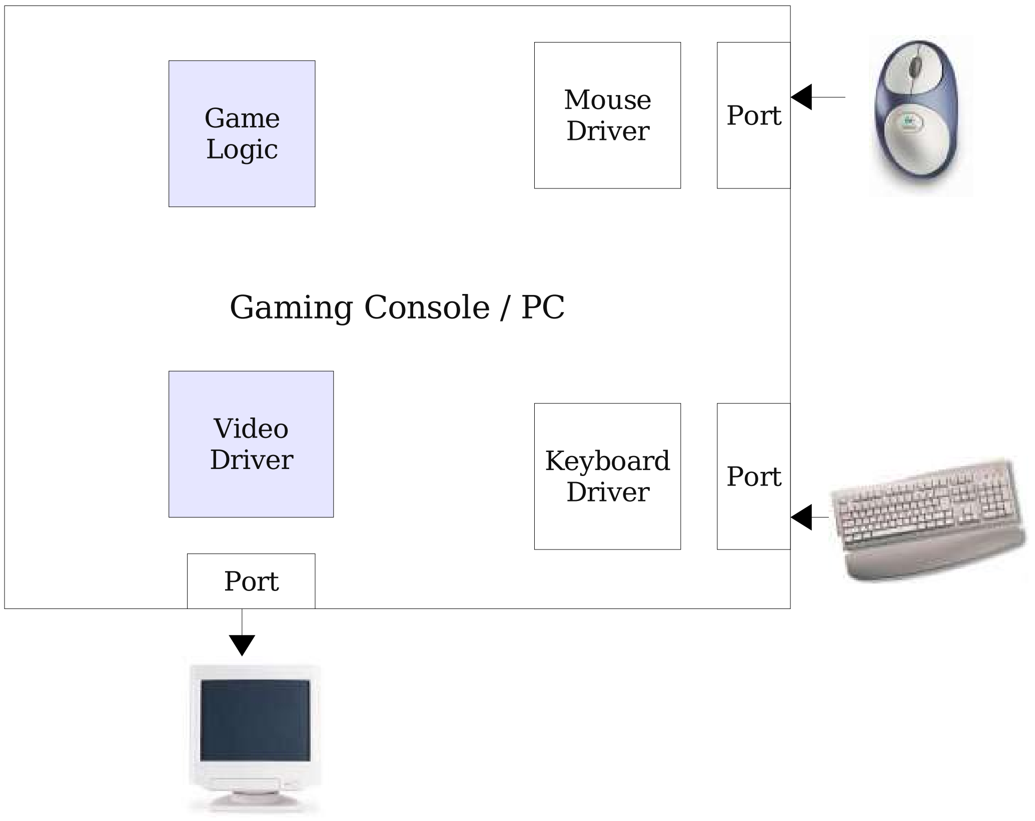

*Figure 10a. Most of the time, the CPU is busy processing game logic and updating the video display.*

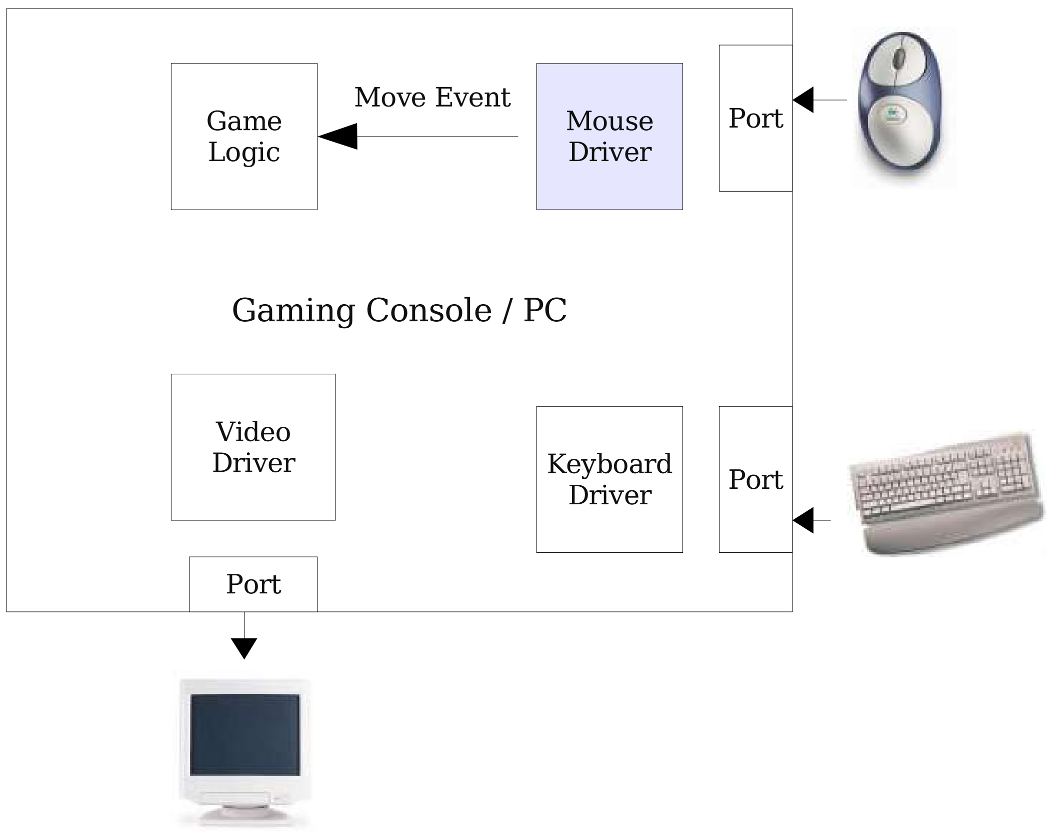

*Figure 10b: When a player moves the mouse, an interrupt signal is generated. This interrupt signal causes the CPU to suspend its processing of the game logic/video display and to switch to code that handles signals from the mouse (mouse driver). The mouse driver will generate a message that informs the game logic code of the player's movement. The handling of this event would typically take place in less than 1 millisecond – so short a time that the user is unaware that it is taking place.*

So what then are interrupts? Interrupts are signals that cause the CPU to suspend its current activity and perform some other task. Frequently interrupt signals are produced by hardware devices that require urgent attention (e.g. a network card signals that it has received an incoming block of data). A given computer system may need to support several hardware subsystems each of which will occasionally require urgent attention. CPU's are commonly designed to handle interrupts from various sources. Each interrupt signal is associated with a particular memory location which contains the address of the subroutine that should be executed on

receipt of this interrupt.  System programmers can populate these special memory locations with the addresses of interrupt handling code.
**Definition**: Interrupt vector : A (special) memory location which contains the address of an interrupt service routine.
**Definition**: Interrupt service routine: A piece of code that is executed on receipt of an interrupt signal.
**Definition**: Interrupt vector table : A collection of interrupt vectors.