A3 Computer Architecture: Interfacing and Control

Dr I. Reid, Michaelmas 2003
[see http://www.robots.ox.ac.uk/~ian/Teaching/ComputerArch]

- Up to now, have considered:
  - Central Processing Unit (architecture, operation)
  - Memory organization
- Now have sufficient knowledge to understand and even consider designing a simple computer
- Without an interface to the outside world it would not be a very interesting one!

Introduction

You are entitled to ask the question "how can a computer be used in the context of an engineering system?"

Answer: "as a general purpose, flexible tool for data collection, analysis and system control"

- **Measurement** – the computer as a tool for data logging and analysis
- **Communications** – the computer as a tool for digital encoding and decoding of communications data
- **Control** – the computer as a general purpose tool for controlling both discrete and continuous devices.

Introduction

In these lectures we will consider how to **interface** I/O devices (**peripherals**) to the CPU.

And we will consider the **hardware** and **software** issues that are raised if we want the computer to **control** the devices.

Lecture 1 Introduction and motivation, Buses and Interfacing devices

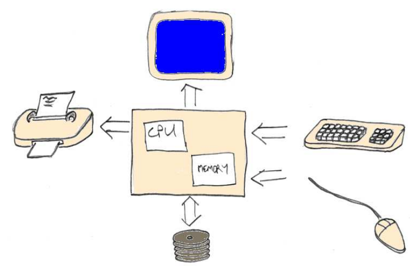
Lecture 2 Input/Output, Polling and Interrupts, Communications

Lecture 3 Timing, scheduling and software

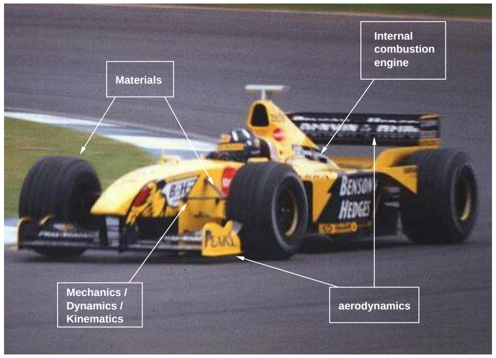
Lecture 4 Fast sampled digital control

Input/Output Devices and Interfacing

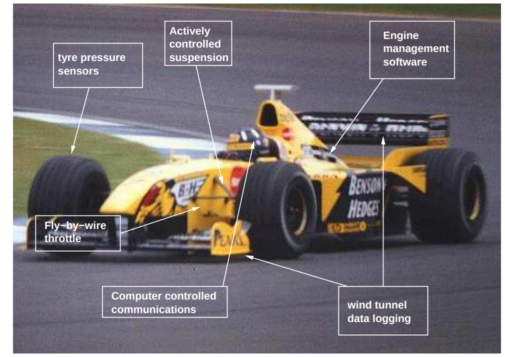
Although many computer architecture texts discuss **peripherals**, they often do so in terms of the PC as a stand-alone tool, and consider "standard" PC devices:



The “traditional” engineering view:



But computer control, data logging and analysis, and communications are becoming huge engineering issues, even in traditional areas:



Matters arising and Course aims

When interfacing a computer to the outside world there are both

- **theoretical** and **practical** issues, involving both
- **hardware** and **software**

Aim of this course is to get to the point where you understand the theoretical and practical issues of both hardware and software pertaining to computers as **components in an engineering system**.

Learning Outcomes

- How to interface a device to a computer/bus
- How to control the device via software
- Timing and software considerations of controlling multiple devices with a single processor
- Theoretical considerations of using a computer to implement a (fast sampled) digital control system

Timing Considerations

- The world "inside" the computer is perfectly orderly, sequential and deterministic; timing is controlled by a regular pulse – a **clock**.
- In contrast, the outside world is non-deterministic; interesting events happen at **unscheduled** moments
- Consequently we need some way of **synchronizing** the computer to these events
- We must consider how the possibly competing demands of devices can be appropriately **scheduled**
- In particular, we must consider issues of how to ensure the system responds **in time** to important events

Real-time systems

> **Real-time Computing**: *Computing for which the correctness of operation depends both on the logical results of the computation and the time at which the results are produced*

Note that this applies equally to **processing** results and **communication** results.

Unlike a stand-alone PC in which the processes have no deadlines or "soft" deadlines, the results of processing and communications in a computer control system have "hard" deadlines.

The Digital/Analogue Interface

- The real world is mostly analogue; quantities tend to take on real-numbered values, and vary **continuously**
- Computers represent the world **digitally**; numbers are represented with finite precision, and we typically must deal with **samples** of continuous signals/measurements

Thus an understanding of

- **sampled data systems**,

and

- **analogue to digital** and **digital to analogue** conversions (**ADC** and **DAC**)

is essential.

Reading

No prescribed text for these four lectures; instead we draw various threads from "standard" texts together.

For material covering various practical issues of computer hardware and **interfacing** (eg **buses**, **input/output**, etc):

- ***Clements**, *The Principles of Computer Hardware*, 3rd ed, OUP, 2000, **Chapter 8** (see also **Ch 12** for ADC/DAC and related topics)
- ***Stallings**, *Computer Organization and Architecture*, 5th ed, Prentice-Hall, 2000, **Chapters 3 and 6**
- Horowitz and Hill, *The Art of Electronics*, 2nd ed, CUP, 1989, **Chapters 10 and 11** (excellent practical treatment)

Reading, ctd

For material covering various software such as **operating system support**, **scheduling**, **real-time processing**, see:

- ***Bennett**, *Real-time Computer Control*, 2nd ed, Prentice-Hall, 1994, **Chapters 1, 6 and 7**

and for theoretical issues involving direct digital control such as **sampling**, **delays** and **difference equations**, see especially

- ***Franklin**, Powell and Workman, *Digital Control of Dynamic Systems*, 3rd ed, Addison-Wesley, 1998, **Chapters 3 and 5**

Reading, ctd

Finally, for a practical though cursory treatment of the whole area,

- Bolton, *Microprocessor Systems*, Longman, 2000, **Chs 2, 3, 10 and 11**

is full of examples.

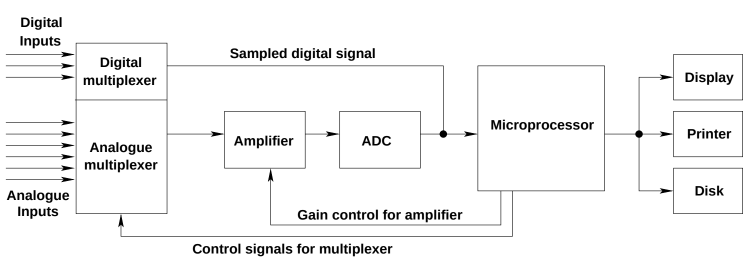
Note that on the previous slides "*"-ed texts are considered **essential** reading. The remainder are recommended but not essential.

The Engineering Library and all College Libraries should have copies of all/most of these.

Finally, some examples . . .

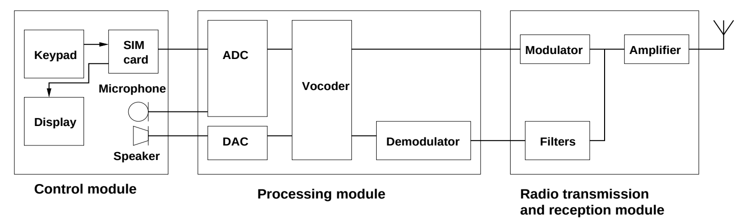
Measurement and Data Logging

- Data acquisition; dedicated system or plug-in board for a PC
- Examples: Numerous – take your pick of any branch of engineering – there is always a need to measure properties of physical systems.



Communications Systems

- Examples: digital TV, mobile phones, internet, satellite comms

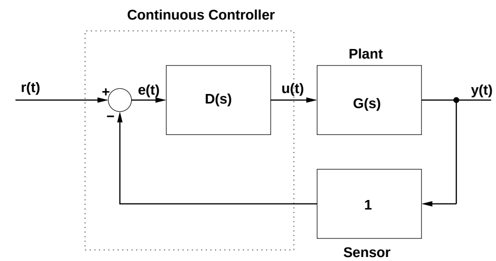


Discrete Computer Control

- Have seen in first year that a **finite state machine** can be built using some D-type latches, or a PROM plus some latches
- and have seen in the previous lectures that microprocessors are just fancy finite state machines
- One major application of microprocessors in computer control is state switching
  - Example: washing machine has various programmes or cycles. CPU controls timing and monitors the state of the system, switching between different stages of the cycle (eg, pre-wash, wash, rinse, spin)
- Although a microprocessor is more complex and possibly more expensive than latches plus PROM, it has the virtue of **flexibility**
  - A particular manufacturer may make a range of machines, each with a different set of legal cycles. But with a microprocessor can have the same control unit for every machine, but running slightly different software tailored to the particular hardware of the specific model.

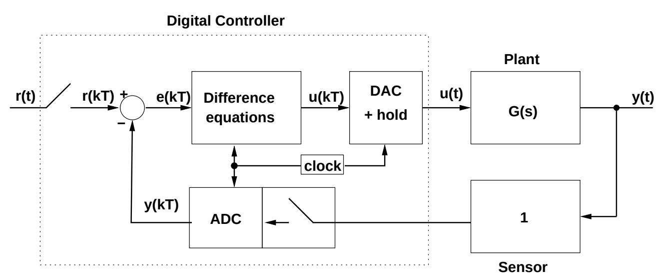
Computer Control

Although you are now "experts" at understanding and designing continuous control systems . . .

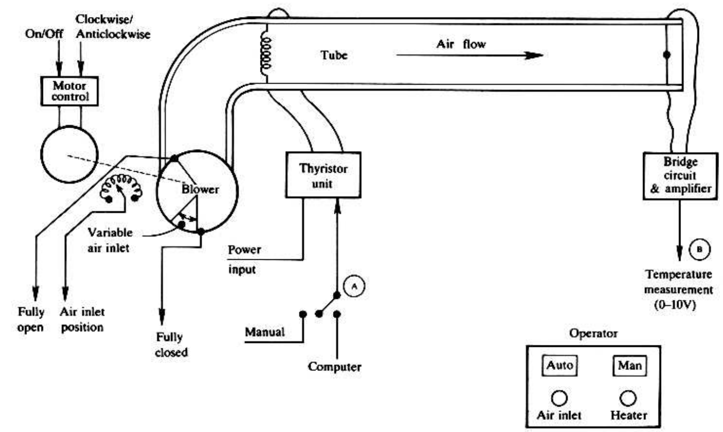


Computer Control

. . . many (even most) modern control systems involve some form of digital computer:

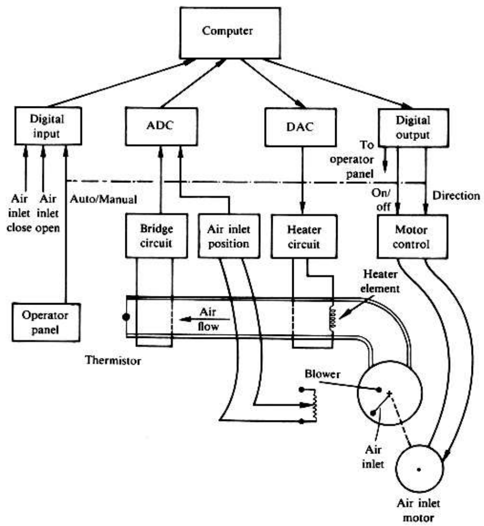


Example:



A hot air blower

Example, ctd

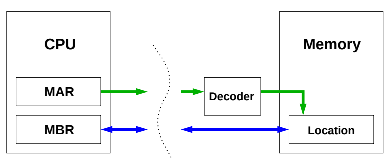


LECTURE 1

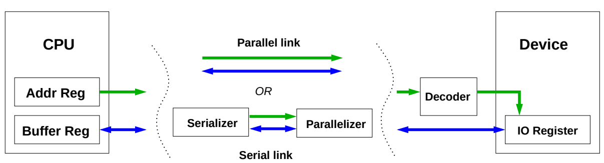
Buses and devices

How does CPU "communicate"?

- CPU is just a fancy **register transfer** device
- Have previously seen that to transfer data to or from memory, CPU supports **MAR** and **MBR**, and memory can be thought of a bunch of D-type latches

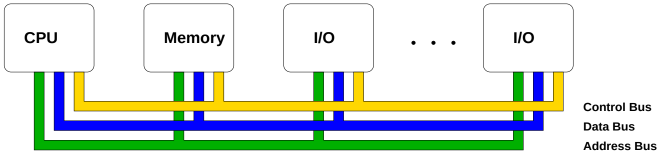


- The same idea holds for a generic device (eg printer, DAC, video frame grabber)



System Buses

In the previous 8 lectures on Computer Architecture you have already encountered the concept of a **bus** as a means for interconnecting computer components:



Modern bus typically consists of 50 to 100 separate lines on a printed circuit board (the "motherboard"), with 3 functional groups:

Data lines Carry data from source to destination (e.g. CPU to memory)

Address lines Used to designate source or destination of the data on the data bus. You have previously seen how these can be decoded to get, eg a particular memory location

Control lines Carry functional information defining and controlling the current bus activity; eg Clock, Memory R/W, IO R/W, Transfer ACK, Bus request (and other arbitration), etc

For example, for a CPU $\rightarrow$ memory transfer or vice-versa, the minimal set of control lines would be a **MemRW** and data strobe **DS**.

Example: PC ISA bus (1983 – 2000)

Buses may be classified as

- Synchronous
- Asynchronous
- Semi-synchronous (synchronous using a WAIT state)

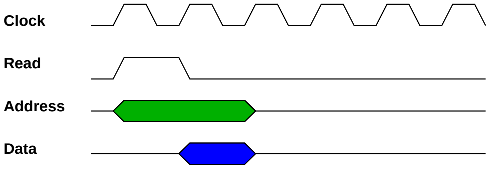
Timing is an issue because of **different device speeds**, but also because of:

- logic/propagation delay — the time spent between input and output changes of a gate (typically 3-6 ns)
- capacitive — around 0.08ns/pF, with delays of 5ns typical
- transit time — $\sqrt{LC}$ where L and C are the inductance and capacitance per unit length. Typical val around 6ns/m.

Synchronous Bus

- Occurrence of events synchronized with a clock
- Sender assumes safe receipt

Advantages	**Disadvantages**
• Fast • Comms protocol is very simple (partial handshaking)	• Does not support mixing of fast and slow devices • Considerable care must be taken with design



A synchronous or "open-loop" read

Asynchronous Bus

- Timing is a property of the devices on the bus, rather than being imposed by the bus itself.
- Data transfer requires acknowledgment
- That is, it is “closed-loop” or **fully handshaked**

  Ready?, Yes, Here’s some data, Got it, OK

Advantages

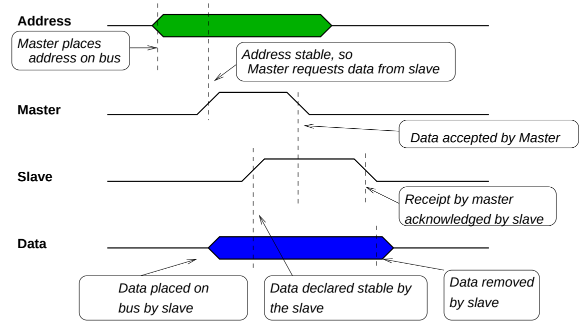
- Mixture of fast and slow devices can share the bus
- Can operate over longer distances

Disadvantages

- Full handshaking means it is (relatively) slow
- Delays more unpredictable

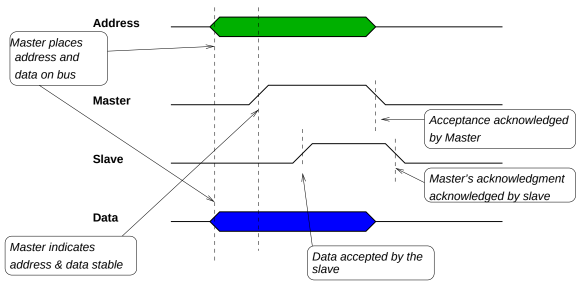
Asynchronous Read

1. Master places address on the bus, then allows some settling time
2. Master indicates that address is active, requesting data from slave
3. Slave places data on the bus, allows some settling time, then indicates data is ready
4. Master reads the data, then acknowledges receipt by going inactive
5. Slave acknowledges Master’s receipt by going inactive, then removes the data.



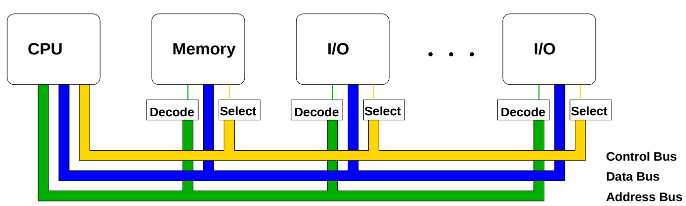
Asynchronous Write

1. Master places address and data on the bus, then allows some settling time
2. Master indicates that address and data are active
3. Slave reads the data, then acknowledges receipt
4. Master reads the data, then acknowledges receipt by going inactive
5. Master goes inactive (effectively acknowledging Slave’s receipt) and removes data and address



Programmed I/O

- Simplest – and most common – method of data transfer on system bus is known as **programmed IO**, and involves the CPU
- Registers for input and output on the device are attached to the data bus



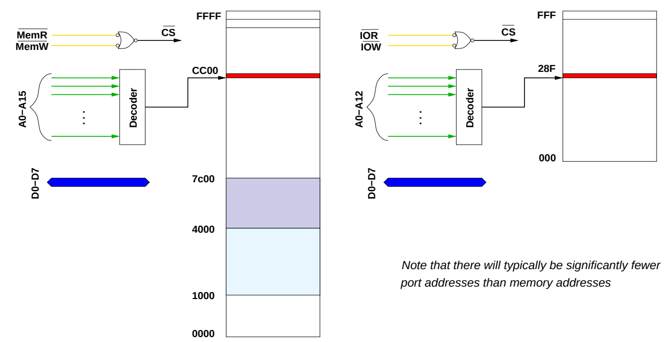
Programmed I/O: Register I/O

- A number of CPUs (eg 80x86 as used in most modern PCs) support `IN` and `OUT` instructions, so-called **register** or **isolated** input/output
- Bus supports use of address lines for either **port address** or **memory address**, as determined by control lines (on older PCs, eg, ($\overline{\mathbf{IOW}}$, $\overline{\mathbf{IOR}}$, $\overline{\mathbf{MEMR}}$, $\overline{\mathbf{MEMW}}$)
- Example:

```
;; Intel 8080 instructions for register IO
MOV AL, 80h ;; get 80 into accumulator
OUT 03C0,AL ;; and send it to port 3C0
```

  – place the address (3C0) of the receiving device on the address bus
  – place the data to be sent (80) on the data bus
  – *strobe* the appropriate bus control line ($\overline{\mathbf{IOW}}$) to indicate an IO write
- With any luck, there will be a device connected to the bus which recognizes (**decodes**) the address and picks up the data.

Memory and port address space



However many CPUs do not support explicit IO instructions. How, then, do they “talk” to the outside world??

Programmed I/O: Memory-mapped I/O

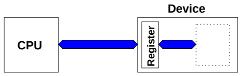
- Devices sit in the **memory address space**
- Each device (and we can include memory as a fast storage “device”) occupies part of the address space and decodes the address lines appropriately
- A peripheral such as an ADC therefore appears to the CPU like any other memory location, and
- IO is effected in the CPU via a standard **memory transfer** (eg using MAR and MBR registers)
- Example:

```
;; Intel 8080 instructions for Memory Mapped IO
MOV AL,80h ;; get 80 into accumulator
MOV CC00,AL ;; and send it to location CC00
```

Most devices are at least partially memory mapped.

Programmed I/O: Design considerations

- The designer of a peripheral must ensure that the address range decoded by the device does not conflict with other devices
  – Older plug-in cards have jumpers or DIP switches that can be set before installation
  – This has gradually been replaced with programmable decoding; the address to decode is stored in non-volatile ram on the device and can be changed if necessary
  – Modern cards try to do this “intelligently” to hide the details, resulting in so-called **plug-and-play**
- Since the CPU may be reading/writing to the device registers at the same as the device, the registers must be **dual-ported**

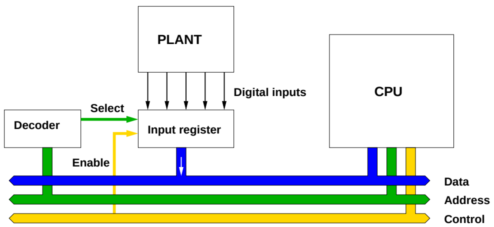


- For devices on a synchronous bus we must of course ensure that IO register transfers look just like a memory transfers from the point of view of timing
  – Slow devices will often use a **fifo buffer** (ie first-in, first-out) which can accept (or supply) a chunk of data rapidly.

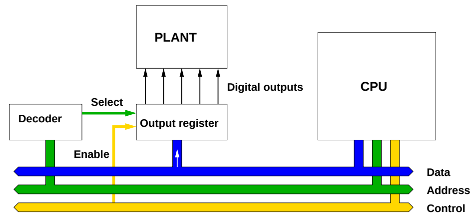
Devices for Real-time Computer Control

- Many texts you read will discuss in some detail devices such as keyboards, printers, disks, etc.
- The purpose of this course, however, is to explore the computer as part of a general engineering system
- What sort of IO devices are need in this context?

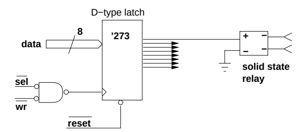
Simple Digital Input: (e.g. a valve open/closed switch)



and **Digital Output:**



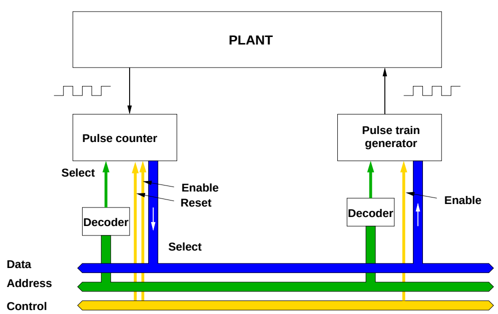
Example:



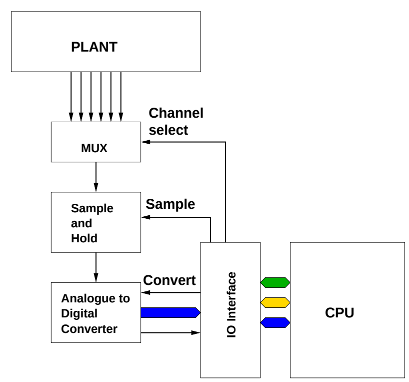
Relay control for switching a large current

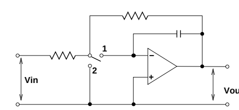
Pulse reading/writing:

- **Reading** shaft encoders
- **Writing** pulse width modulation for electric motors

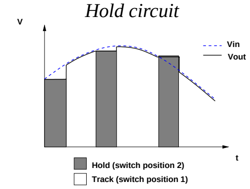


Analogue input: eg any analogue sensor such as a thermistor, strain gauge, etc

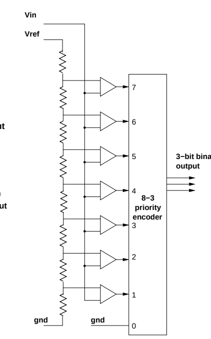




Hold circuit

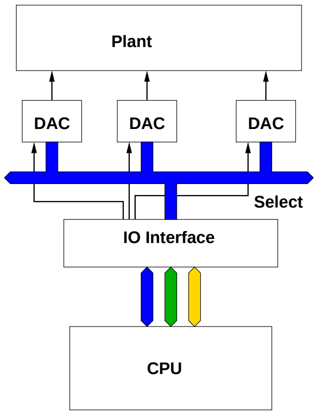


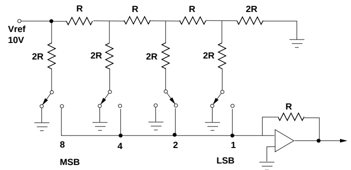
Hold circuit behaviour



A/D flash converter

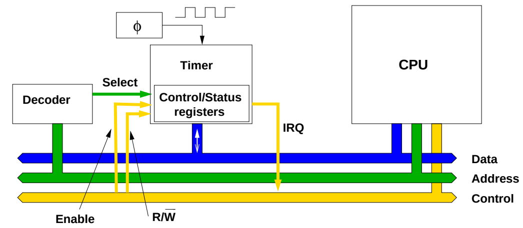
Analogue output: eg continuous voltage or current control





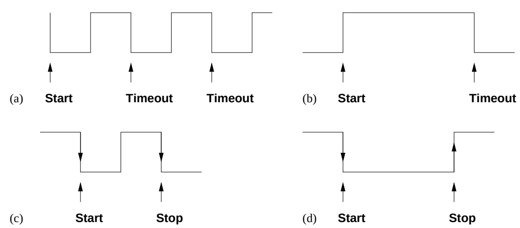
R-2R ladder for D/A conversion

Timer / Real-time Clock: vital axillary device, used for synchronizing events to a timer, controlling sampling intervals, time/date stamping sensor data, etc



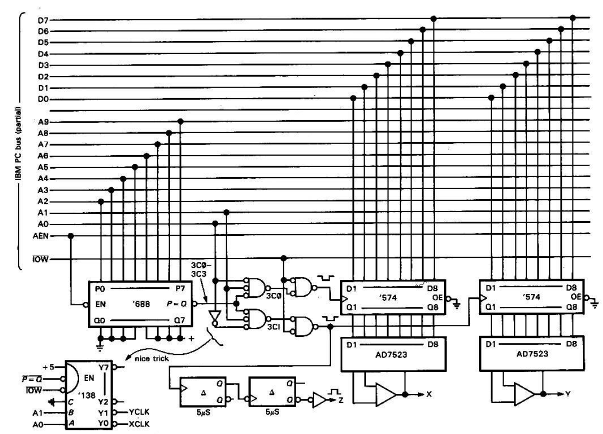
Timer / Real-time Clock:

- Fixed frequency oscillator feed a counter
- Typically **programmable** with one of a number of modes of operation:
  - *(a) Continuous mode:* decrement pre-loaded value until zero, generate event (interrupt), then reset and start counting again
  - *(b) Single pulse:* as previous but count down just once
  - *(c) Frequency comparison:* count the period of a signal applied to an input gate pin; ie between two falling edges
  - *(d) Pulse width:* count time between falling and rising edge of a pulse



Example: XY scope display

- Control the X and Y coordinates of an oscilloscope beam based on PC ISA bus (H & H, ch 10, p686)
- Consists of decoding circuitry, and two latches attached to a pair DACs



XY scope display: Notes

- The two ’574 latches are the device’s **input registers** (from the point of view of the CPU they are output registers)
- To control the device the CPU must **output** data to the appropriate address
- The device does not have any output registers; there is no way for the CPU to determine its state (other than remembering what it last told it to do)
- Circuit **fully decodes** the address lines; could in fact ignore A1 altogether and use 2-input NANDs instead. What would be the effect?
- In reality the pair of latches and pair of DACs would probably be replaced by DACs with built-in latches

XY scope display: Code

8080 assembly code to drive the device:

```
;; 8080 assembly code to drive XY scope display
INIT:   MOV     SI,xpoint       ;init address of x array
        MOV     DI,ypoint       ;init address of y array
        MOV     CX,npoint       ;init counter

PLOT:   MOV     AL,[SI]         ;get x byte
        OUT     03C0h,AL        ;output it
        MOV     AL,[DI]         ;get y byte
        OUT     03C1h,AL        ;output it
        INC     SI              ;advance x address
        INC     DI              ;advance y address
        DEC     CX              ;decrement counter
        JNZ     PLOT            ;not done, plot next point
        JMP     INIT            ;done, so start again
```


Micro-controllers

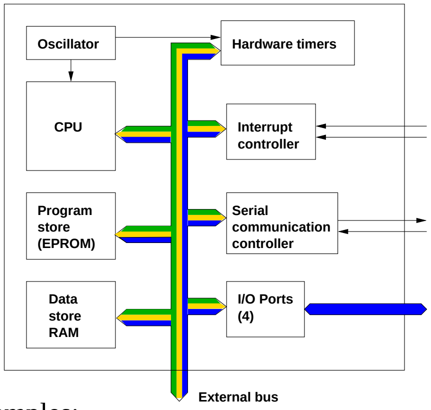
In a world in which there seems to be a VLSI chip for just about any task we might like to think about, it will be no surprise to learn that there are chips which integrate:

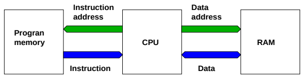
- CPU
- Memory (ROM, RAM and (E)EPROM)
- IO Ports (digital and A/D, pulse, serial/parallel comms, etc)
- IO control and status registers
- Timers
- Internal bus to connect the components

onto a single chip.

These devices are known as **micro-controllers** and widely used in embedded systems for control-oriented activities

Micro-controllers





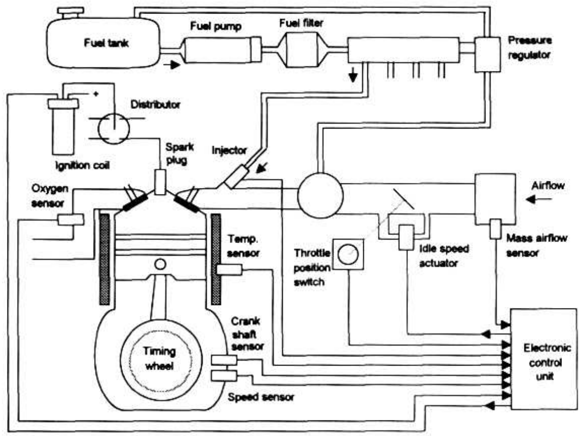
The “Hardvard” architecture, as used in the PIC family of micro-controllers

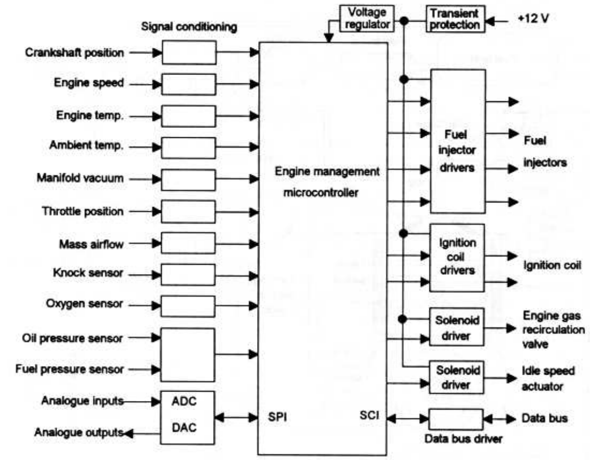
Examples:

- Motorola 68HC11, PIC16C6x/7x family, 8051 (8/16-bit micro-controllers, typically £5 per unit)

Microcontroller: example application

Engine management:



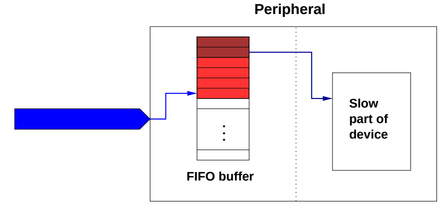


LECTURE 2

Input/Output, Interrupts and Communications

Buffered I/O

- Although I/O is driven at the speed of bus transfers, the physical device may not be able to keep up if a constant stream of data is being supplied.
- One solution is to **buffer** data by having some fast memory on the device



- What happens if the buffer fills up?

Scheduling I/O

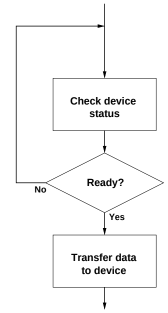
- A device needs some way of indicating to the sender (typically CPU) that it is ready or not; minimally it must have a bit in a **status register** indicating READY (=1) or NOT READY (=0)

Three main ways of controlling the timing of I/O:

- **Polling**
- **Interrupts**
- **Direct memory access (DMA)**

Polling

- **Polling** is a **software** solution to determine is a device is ready
- CPU constantly (or periodically) checks the status of the device




```
      ;; Pseudo 68000 assembly language polling loop
      MOVE  #128,D1
      LEA   1000,A0  ;Load addr reg A0 with loc of data
      LEA   8000,A1  ;Load addr reg A1 with loc of output port
      LEA   8002,A2  ;Load addr reg A2 with loc of status byte
LOOP  MOVE  (A0)+,D0 ;Get byte from table into ac, and inc A0
WAIT  MOVE  (A2),D2  ;Read status byte into D2
      AND   #1,D2    ;Logical AND with LSB: sets the Zflag
      BEQ   WAIT     ;If Z=0 jump to wait
      MOVE  D0,(A1)
      SUB   #1,D1
      BNE   LOOP     ;If Z not set then goto LOOP.
```

68000 assembly code to transfer a block of data to a device

- Processor may spend a *lot* of time waiting
- Example: Fast ink-jet printer
  - printing speed approx 100 characters (bytes) per second
  - typical modern bus (eg PCI) transfers at in excess of 40 Mb/sec
  - typical modern processor cycles at 1Gz.

  Four instructions at the core of the polling loop get executed over and over

```
LOOP  MOVE  (A0)+,D0 ;Get byte from table into ac, and inc A0
WAIT  MOVE  (A2),D2  ;Read status byte into D2
      AND   #1,D2    ;Logical AND with LSB: sets the Zflag
      BEQ   WAIT     ;If Z=0 jump to wait
```

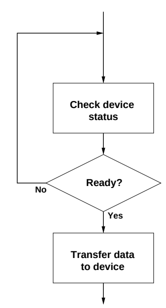
A 1GHz processor, with 4 cycles per instruction will take

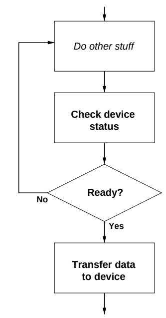
$$4 * 4/10^9 = 1.6 * 10^{-8}\text{secs}$$

so the polling loop will execute approx

$$(1 - 0.01)/1.6 * 10^{-8} \approx 63\text{million}$$

times while waiting for the printer to be ready again . . .





- Loop on left is ineffi cient, but this may not matter for small, single task, dedicated machine

  Example: cash-machine (ATM); waits for a card to be inserted, then waits for keys to be pressed, etc
- Loop on right avoids lots of wasted cycles, but relies on program to make sure the device is checked suffi ciently often

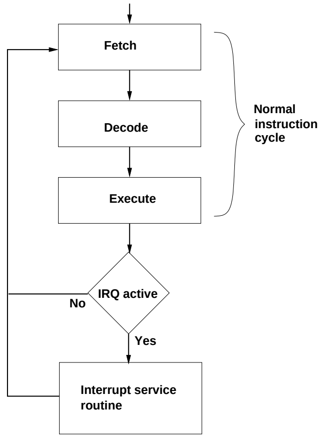
Interrupts

- With only a few devices needing quick response (eg fast disks or latency sensitive I/O) a polling machine would rapidly become bogged down checking status flags
- Better to have devices **signal** the CPU that they are ready

> **Interrupt:** *a spontaneous hardware request for attention by a peripheral, producing a program jump to a dedicated handler routine (usually resulting in some programmed I/O) followed by a return to the code that was interrupted*

Interrupt driven I/O

- Most CPUs have **hardware support** for **interrupting** the CPU during normal program execution
- A device **requests** an interrupt by lowering a dedicated line on the control bus, an **interrupt request line**
- **Micro-instructions** in the CPU test for presence/absence of interrupt request (ie polling at the microinstruction level)
- Somewhere in memory we have an **interrupt service routine**, a sub-routine to do whatever is necessary (eg read a character from a keypad, or power down a plant that has signalled an alarm)



Interrupt servicing

What does the CPU do on receiving an interrupt?

- Finish the current macro instruction
- “Recognize” the interrupt (i.e. determine which service routine is needed)
- Save the Program Counter, Registers and Status Word
- Jump to the routine
- Run the service routine
- Resume normal service: restore PC, Regs and Status Word and continue normal execution

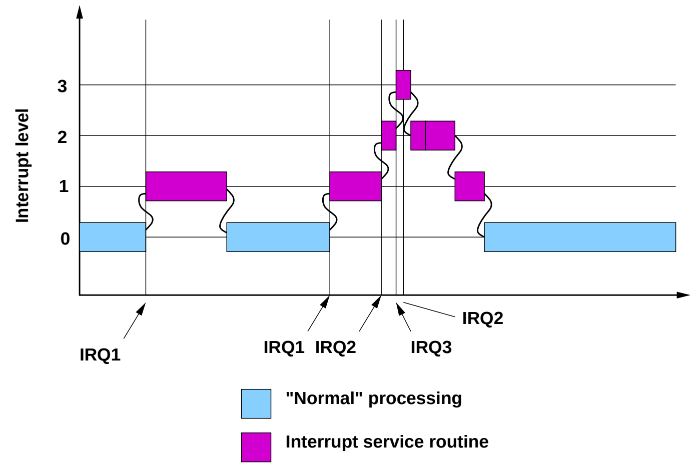
Some considerations:

- Multiple devices may be connected, so we need hardware support for the following:
  - A way of identifying which device interrupted (so the appropriate service routine can be called)
  - A way of prioritizing devices to reflect the fact that some will require more urgent attention that others
  - Some way to prevent interrupt service routines constantly being interrupted themselves
- From the software point of view we also need:
  - Service routines which are short and to the point!
  - Routines which do not corrupt the current state of the machine (this is why the registers and status word are saved, so that the CPU state appears exactly the same after the interrupt as before)
  - Why must the status word be saved??

Interrupt Priorities and Masking

- Some devices are more important or impatient than others (eg clock, alarms, time critical devices such as controllers)
- Many CPUs allow for a number of different interrupt **priorities**
- Implemented by separate control lines on the system bus (ie one line per level)
- Processor can deliberately ignore, or **mask** interrupts at a given priority or lower
  Achieved by having bits in the processor's status word indicating the current mask. Each subsequent interrupt is first compared with the mask before being allowed to proceed (or being blocked)
- Masking does not cancel an interrupt; just prevents it being serviced temporarily
- To ensure low priority devices do not interrupt high priority ones the CPU will usually block lower priority interrupt requests during the interrupt service routine
  When an interrupt at level n occurs, subsequent interrupts at that level and lower are masked off until the service routine completes. However higher priority interrupts $> n$ may interrupt the service routines of lower priority ones.
- The most urgent interrupt is **non-maskable**; usually associated with system functions affecting the machine's well-being (eg reset button)

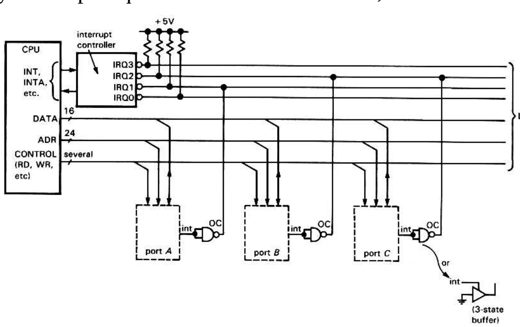
Multilevel interrupts: Example



Task activation in an interrupting system

Sharing interrupt lines

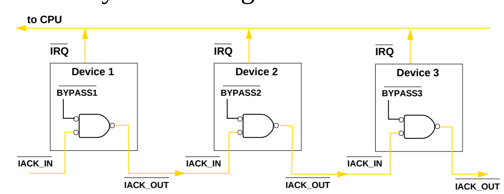
- In simple machines such as old PCs (pre-early 1990s) only one device may be attached to an interrupt request line (IRQs in old PCs were **edge-driven**)
- More commonly interrupt request lines are **level-driven**, and can be shared:



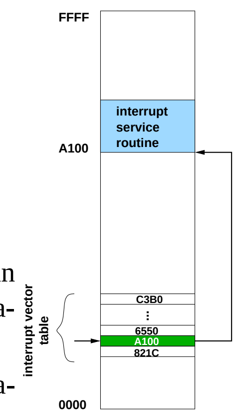
- How, though, does the CPU determine which device has interrupted? And what does it do when it has this information?

Recognizing interrupts

- Devices daisy-chained together



Daisy-chained devices

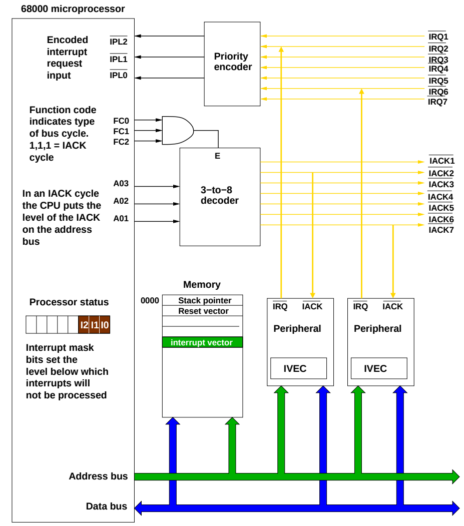


- Once the CPU knows which device interrupted it, it can look up the **interrupt vector table**, a list of the locations (or **vectors**) of the various service routines
- It then forces a jump (ie change in the PC) to the location specified in the table

Example: Vectored Interrupts in the 68000

You will explore this further in the Computer Systems Lab . . .

- 68000 has 7 interrupt levels; 0 (no interrupt), 1 (least urgent), . . . 7 (highest priority)
- Devices can share interrupt lines using the daisy-chaining mechanism outlined above

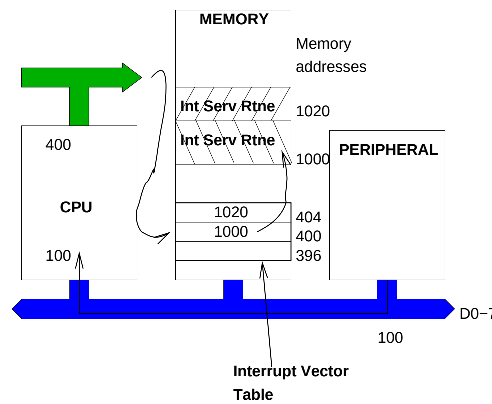


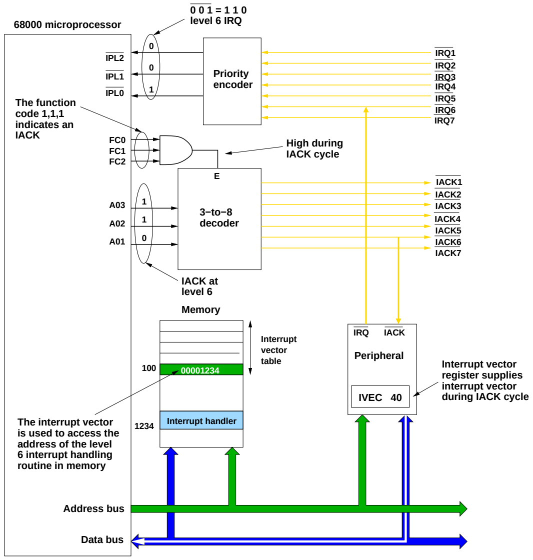
Recognizing interrupts

- Interrupting device pulls its IRQ low
- **Priority encoder** outputs (in 3 bit encoded form) the highest interrupt being requested
- Encoded request is then masked by the interrupt mask (part of the processor's **status word**). If the mask is at the same level or higher, the interrupt cannot proceed.
- Once past the mask, processor interruption is inevitable. CPU finishes current (macro) instruction, then sets the **bus condition flags** FC2,1,0 = 1, 1, 1. This indicates to devices on the bus that the processor is acknowledging an interrupt.
- The CPU then places the level of the interrupt being acknowledged onto the lowest three address lines A3, A2, A1. These are decoded to give 7 IACK lines
- IACK lines are **daisy-chained** in and out of devices on the bus.
- $\overline{\text{BYPASS}}$ is asserted by all devices not requesting an interrupt
- Thus the first interrupting device in the chain to "capture" the IACK signal does not pass it on down the chain.
- Capturing device places an 8-bit **interrupt vector number** onto the data bus (D0-D7)

Example, ctd.

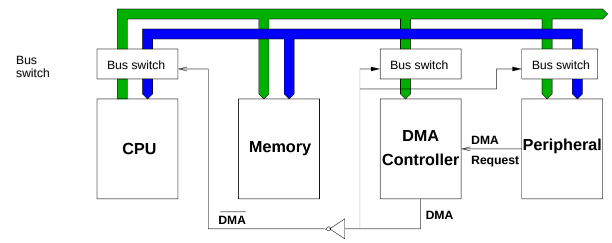
- The CPU multiplies the interrupt vector number by 4 to generate a memory address. This address is one of a table of such addresses. The contents of this address is the address of the interrupt service routine to which the processor should jump.





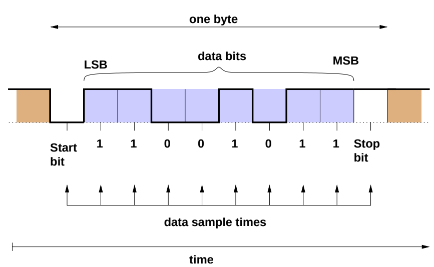
I/O using Direct Memory Access

- If large quantities of data need to be shipped from one place to another (eg memory to a disk), it is wasteful of CPU time for all the data to have to pass through the CPU
- **DMA** is much more efficient, but also requires special hardware support
- A **DMA controller** orchestrates the data transfer, leaving the CPU to get on with its processing



Communications

- System buses, esp synchronous ones, operate over short distances; usually internal to computer
- Parallel data transmission requires multiple data paths, leading to expensive cabling; usually restricted to short distances
- For communications over longer distances, say between two separate computers, **serial communication** is much more common



Asynchronous serial communication of one 8-bit word. Start bit tells receiver when to start looking for data. Assume that sender and receiver have separate clocks whose speed is matched to within 10%.

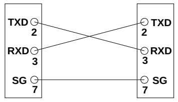
RS-232

The most commonly used serial interface is **RS-232**. Provides point to point bidirectional asynchronous communications.

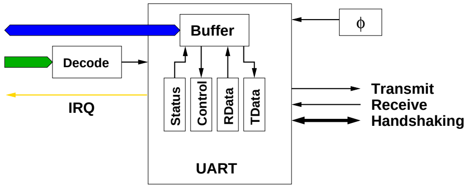
- Old (1962)
- Slow: 115200 baud (bits per second) or slower
- *Very* common . . . examples: modem, integrated servo motor/controller
- Real-time (ie send/receive time may be slow, but is *guaranteed*)
- Consists of 3 categories of signals
  - *Data*: two lines, send and receive
  - *Handshaking*: handshaking signals to control data flow
  - *Timing*: for synchronous operation, clock signals
- Signals are +12V (logic 1) and -12V (logic 0)

RS-232

- Simplest implementation connects only data lines plus ground



- Parallel data from microprocessor is converted into a serial bit-stream by a UART (Universal Asynchronous Receiver/Transmitter), which has buffers, control and status registers and is interfaced to the CPU on the system bus



Serial communications

- Most micro-controllers have an integrated UART
  - One main use is for down-loading code, up-loading data and other communications with a "host" computer
- Modern PCs retain RS-232 ports, but also provide higher spec serial lines
  - USB (Universal Serial Bus), 10 Mbit/sec
    [there are microcontroller interfaces for USB as well]
  - IEEE-1394 (Firewire, I-Link), 400 Mbit/sec
    [no micro-controller interfaces here yet, but maybe soon . . .]

Ethernet

While RS232 provides point-to-point (or "peer-to-peer") two-way communication, when multiple computers are locally connected they are more commonly **networked** using using a "bus-like" architecture

- **Network communications** between computers most commonly done by **ethernet**
- Modern ethernet yields up to 1Gbit/sec
- Nodes listen to a single transmit/receive cable
- Sender waits for "silence" then sends a **packet** of data
- There is a possibility (because of the propagation delays between nodes and in each node's ethernet hardware) that two nodes try to send at once resulting in a **collision**; the packets are garbled and must be re-sent
- For this reason one should be aware that for ethernet or similar technologies (eg Bluetooth) send/receive times cannot be *guaranteed*; this may be important in the context of a real-time system.

Communications Bandwidth

1. Beware that a quoted bandwidth of a communications medium may refer to the *average* bandwidth over time, not a guaranteed rate
   - Not a problem if the fluctuations up and down are fairly frequent and we can afford the delays associated with **buffering**
   - Streaming internet audio/video (eg RealPlayer or MediaPlayer) rely on an average bandwidth of the internet and use a buffer to smooth out the irregular arrival times of data packets.
   - Problematic if we need to avoid delays, eg closed-loop control – maybe tele-surgery over the internet is not such a great idea . . .
2. Beware that the bandwidth does not necessarily tell us about the **transmission time**
   - It would not matter how high the data rate to/from the Mars rover "Sojourner", real-time closed-loop control was impossible because the round trip journey time is 20 minutes.

A general design solution, and which applied in an extreme form to the Mars rover is to use an inner loop (local, fast, synchronous), responding to demands communicated (possibly asynchronously) from a slower outer loop.

- The Mars rover was equipped with simple visual and laser-range-finding sensors. These sesnors were used by a local controller on the vehicle responsible for achieving a higher level goal set by the NASA boffins ("traverse this path") while (i) avoiding obstacles and (ii) monitoring the system state for dangerous or extreme conditions.
- A more down-to-earth (literally!) and very common scenario is to have a local micro-controller dedicated to performing a fast servo-loop, but communicating via an RS232 link with another processor.