LECTURE 3

Timing, scheduling and software considerations

Software

- Up to now we have considered mostly **hardware** for computer control systems (more on this later)
- A computing system is of course useless without **software** to run on the microprocessor; i.e. the sequence of macro-instructions to be executed
- Proving that software does what it purports to (or what we would like) is, even in the simplest cases, difficult (ask any ECS student . . .)
- The situation is *much* more difficult for real-time systems where we have to factor in the impact of timing
- The intimate detail of software design is well beyond the scope of this course; however no sensible discussion of computer control could be complete without considering on the role of software

Processes

- A **program** is an implementation of an algorithm (or algorithms) to accomplish a given objective
  - could be "hand-coded" (eg XY scope display assembly code)
  - may be generated using a higher-level language such as C
- A **process** (or **task**) is an instance of a program that comprising the code, and all data (including volatile register data)
- A simple one-task system (eg an embedded microcontroller system controlling a single feedback loop) could be sensibly implemented in single program
- A more complex system may involve a series of **sequential** tasks (ie one task after another)
  - it makes good design sense to construct these as separate tasks, and to fi t them together within a **finite state machine** architecture
- Many real-world examples have the additional complication of **multiple**, potentially **concurrent** activities

Example: Fly-by-wire aircraft

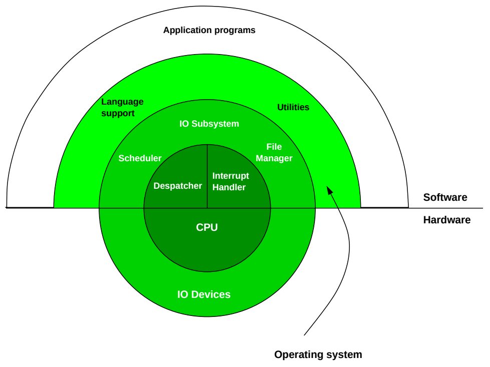
- closed-loop control of flight surfaces
- monitoring sensor data (altitude, speed, GPS, etc)
- managing data display for pilot
- monitoring pilot inputs (eg "joystick")
- switching between control modes (eg taxi vs take-off vs landing vs cruise)

Airbus A320

- Microprocessor is a **sequential** device; it does one thing at a time
- However often it does its "stuff" quickly enough that by rapidly switching between tasks, they can each appear to have exclusive use of the processor.
- It makes sense to design our multi-activity system
  - as a set of processes, each performing a separate task, plus
  - a governing or managing process that (i) resolves competition between processes for resources, and (ii) provides mechanisms for sharing data and synchronization
- For a simple system, even here we might consider writing software ourselves to do this
- More commonly rely on **operating system**

Operating System

- The OS is a process (ie it is a program like any other) that manages a system's resources
- ECS students will have looked into this topic in great detail . . .
- Here, as always, *our* emphasis will be on how OS design affects the function of the computer as a **system component**, and vice-versa

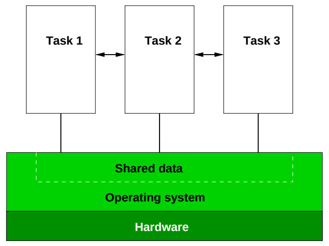


Device Drivers

- Computer hardware is rich with diversity
- Each hardware device requires its own special set of control signals, timing, etc
- One role of the OS is to shield **application programs** from this level of detail by providing a uniform **application program interface** (**API**)
- Does this via software **drivers**
- Driver typically contains code to:
  - initialize the device (*open*)
  - configure the device under software control, or monitor its status (*ioctl*)
  - read or write data (*read*, *write*)
  - service any interrupts that the device might generate

Multi-tasking

- All but the simplest operating systems support **multi-tasking**
- For computer control have a wide variety of tasks:
  - "special occasions" such as reset, alarms, start-up
  - background gathering stats
  - communications (eg with operators, updating status displays)
  - may be controlling several different aspects of a large plant
- Some may be entirely independent, others may need to cooperate by sharing data and/or synchronizing with each other

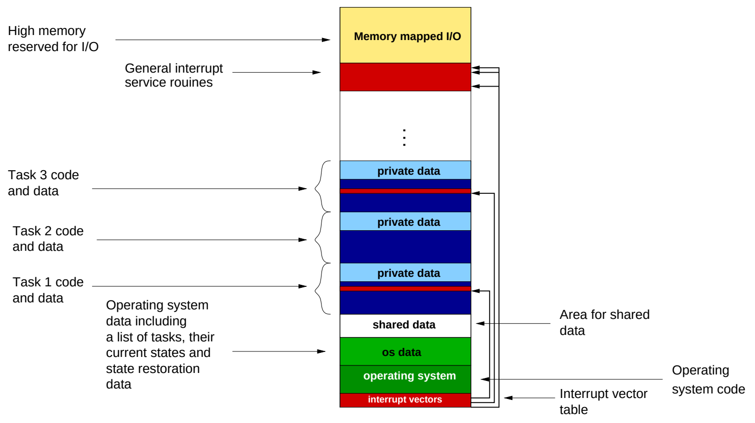


Operating System

In particular, a multi-tasking operating system provides the following:

- **Task management:** allocation of memory and processor time to tasks (**scheduling**)
- **Memory management:** control of memory allocation and use (eg making sure processes do not corrupt each other's data)
- **Resource control:** control of all shared resources other than CPU and memory, esp managing I/O devices, including servicing interrupts
- **Intertask communications and synchronization**

Memory map of a multi-tasking system:



Time constraints / Deadlines

In a real-time system, not all tasks need equal priority. In a computer control system we typically will have tasks with both hard and soft **deadlines** both types:

- **Hard** +/- a small percentage
  - important to meet demands of these tasks
  - examples: response to alarm, basic cycle of feed-back control loop
- **Soft** +/- a large percentage
  - deadlines are less crucial
  - perhaps just need to satisfy an *average* response time
  - examples: "book-keeping" tasks such as automatic backup of logged data, buffered tasks such internet communications (streaming video/audio)

In addition tasks for computer control can be classifi ed as

- **Clock-based** (cyclic, periodic, synchronous) *high priority*
  - regular repetitious tasks, (clock generates an interrupt)
  - "jitter" unacceptable
  - examples: sampler, closed-loop control algorithm
- **Event-based** (aperiodic) *high priority*
  - interrupt driven; to be run when "something happens"
  - low-latency, but a small amount of "jitter" acceptable
  - examples: alarm goes, control algorithm needs to switch state in response to external event (washing machine drum full of water)
- **Interactive** (involving external interaction) *medium priority*
  - generally not time-critical, no need for *instantaneous* response
  - examples: airline reservation system, cash-machine
  - not so important in the context of real-time computer control
- **Base level** *low priority*
  - low priority tasks which run when no other tasks "need" the processor
  - soft deadlines or no deadlines

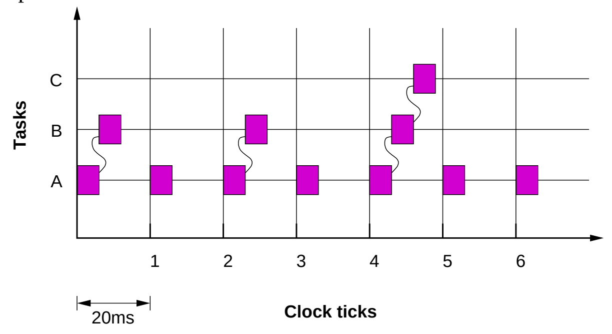
Process scheduling

Return to scheduling . . .

- For a simple system, perhaps just run tasks sequentially until completion
- More generally use **pre-emptive scheduling**:
  - tasks are **time-sliced** and **prioritized**
  - operating system allocates set amount of processing time to each process
  - at the end of each time slice (signalled by a clock interrupt), or after (eg) an I/O interrupt, the scheduler decides whose turn it is to execute again, preferably on the basis of most urgency
- The **granularity** of the time-slicing should be set to ensure that every process is serviced at the appropriate rate

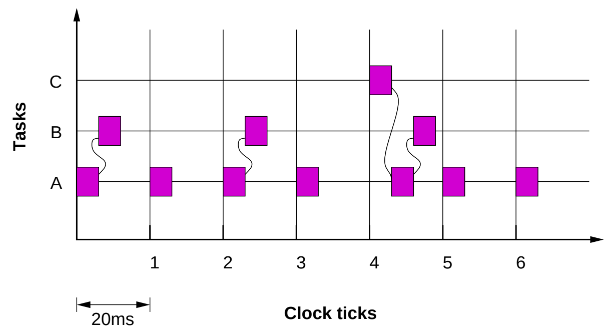
Cyclic scheduling

Suppose we have 3 tasks, A, B and C which must run at 20, 40 and 80 msec intervals respectively. If they have priorities A highest to C lowest, the fi gure shows the CPU usage of each of the processes



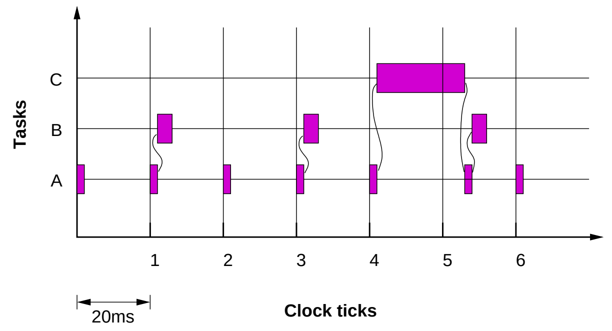
- Note that the CPU usage of the scheduler is not shown although it too is a process (highest priority, but hopefully very short duration)
- In a single processor system, of course not all tasks can start at the clock-tick

If the priority of C is increased above A (ie new order C,A,B from highest to lowest) the task activation diagram appears as follows:



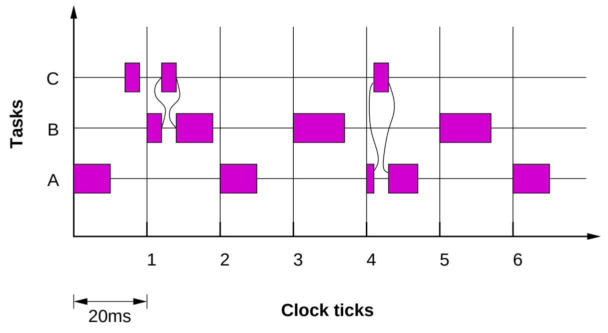
- Note that the regularity of task A has been affected
- Usually the task with the highest repetition rate would have the highest priority

Suppose A runs every 20ms for 1ms, B runs every 40ms and takes 6ms, while C takes 25ms



- If C is allowed to complete then every fourth invocation of A is delayed by 5ms
- Normally those tasks which are guaranteed to complete within the cycle time would be given higher priority, and tasks such as C set to a lower priority and pre-empted by the scheduler at the next clock-tick

Suppose A runs every 40ms and takes 10ms, while B runs every 40ms and takes 15ms. C runs for 5ms in response to an interrupt.



- In this example control returns to the interrupted process

- In a **real-time system** we must ensure that the scheduler behaves in a manner that **guarantees task deadlines**.

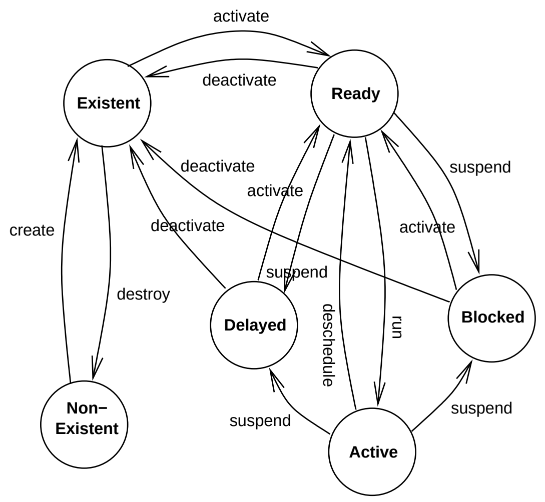
Sadly, the schedulers of the most commonly-used operating systems (Windows, Unix) give no such guarantees.

What is needed is a so-called **real-time operating system** (**RTOS**) which gives the user the ability to set the priority of tasks (examples: QNX, RT-Linux, VxWorks)

In a RTOS tasks classify tasks as:

- **Active** (*running*: the task which has control of the CPU)
- **Ready** (*runnable*: there may be several tasks in this state ready but waiting for the scheduler to grant them control of the CPU)
- **Suspended** (either *delayed* waiting for a certain time to elapse – eg waiting for an ADC to complete – or *blocked*: waiting for an event or system resource)
- **Existent** (*dormant*: processes which have been created so they occupy memory but are using no other resources, and are not runnable)

These task states can be represented in a state transition diagram as follows:



Pre-emptive, priority scheduler

This suggests a **scheduler/dispatcher** with two entry conditions:

1. any real-time clock interrupt (indicating end of time-slice) or any interrupt signalling the completion of an I/O request
2. tasks suspension due to task delaying or requesting I/O

1. In the former, the scheduler searches for the highest priority task which is ready and starts (or restarts) it.
   - If tasks with a high repetition rate are given a high priority they will run as **clock-level tasks**
2. In the latter it searches for the task with the next lowest priority to the task which has just been running
   - There cannot be a higher priority task ready since a higher-priority task becoming ready always pre-empts (via interrupt) a lower prioirty one.

The scheduler above is the simplest and most commonly used method of scheduling in real-time systems

What can we say the behaviour of a system using it? In fact the **only** thing we can say with certainty is that

- the highest priority task always run on-time

The behaviour of the system is dependent on the priorities assigned to the tasks. Furthermore even then it is non-deterministic since the behaviour changes according to the pattern of occurrence of events.

How then should we assign priorities to tasks? Either

- assign priorities according to the importance of the task, or
- assign prioirties according to the cycle of the tasks, with shorter cycle time tasks having higher priority

Often this amounts to the same thing, but there may be situations when this is not the case (eg a task dealing with a safety critical alarm may not run often, but may have to take priority over all other tasks, even clock-based control loops)

In practice designers use a mixture of the two based on common sense.

Cooperation, communication and synchronization

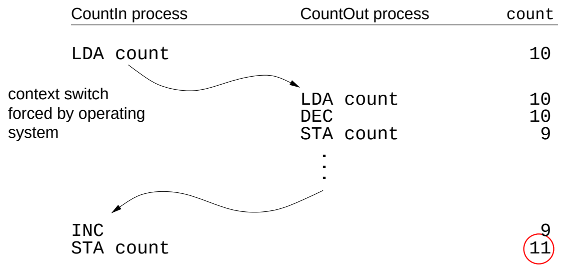
It is inevitable that if the multiple tasks which comprise the system are to cooperate there must be some means for

- competing for and/or sharing system resources
- inter-process synchronization.
- inter-process communication

Mutual exclusion

- Multi-tasking operating system allow sharing of resources (including I/O devices and data) between several concurrently active tasks
- Does not necessarily imply that a resource can be used simultaneously by multiple tasks; the use of some may be (must be) restricted to one process at a time
- Applies especially to I/O devices, but also consider shared data:

Two software modules count pulses from detectors which observe bottles entering and leaving a processing area. They run as independent processes sharing a variable `count`.



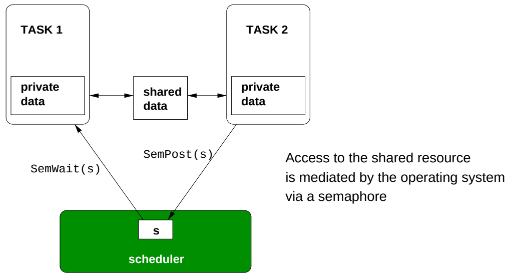
```
CountIn process       CountOut process       count

LDA count                                      10

                      LDA count                10
                      DEC                      10
                      STA count                 9
                        :
                        :

INC                                             9
STA count                                      11
```

Semaphores

- Most widely used solution to mutual exclusion problem is the **semaphore**
- A semaphore `s` “protecting” a resource is initialized to a positive integer representing the maximum number of concurrent usesr of the resource (typically this will be one for exclusive use, `SemInit(s,1)`)
- When a process wants to use a resource it calls system function `SemWait(s)`. If the value of the semaphore is nonzero it is immediately decremented (atomically) and the process proceeds. If it is zero then the process is suspended until the semphore has nonzero value again
- When a process is ready to release a resource it calls a `SemPost(s)` operation which increments the semaphore
- Both `SemPost` and a blocking `SemWait` must result in the scheduler being run
- Sections of code which need to be protected by mutual exclusion are called **critical**. They should be as short as possible to minimize delays to other processes which may need the same resources.

Semaphores

```
TASK 1
  :
general code      (remainder 1)
SemWait(s)        (pre−protocol)
access resource   (critical section)
SemPost(s)        (post−protocol)
general code      (remainder 2)
  :
```



Example: Task 1 obtains PID parameters (say from a user) and Task 2 performs servo-loop. The shared data are the parameters K, T_d and T_i.

Task 1 must ensure that all three values are written to the shared area in a protected section of code, otherwise servo process may (eg) see a new value for K but old ones for T_d and T_i.

Synchronization: Messages and Signals

- Frequently one task needs to inform another task of an event, or a task needs to wait for a particular event.
- Require synchronisation to avoid race conditions
- Simplest form of synchronization achieved via a **signal**
- Implemented as a binary sempahore initialized to 0:
  - Signal receiver issues a `Wait(s)` call and is suspended until signal value is one. As soon as the value of `s` becomes one, the system sets it to zero (atomically) and makes the receiving process runnable (possibly resulting in it becoming active depending on its priority).
  - If more than one process is waiting for the same signal the one with the highest priority will be "picked", the rest remaining suspended
  - Signal sender issues a `Send(s)`, atomically setting the value of the signal to one.

Synchronization: Messages and Signals

- An alternative to shared data supported by some systems (and essential if the multiple processes are mapped to multiple communicating processors) is the idea of a **channel**
- **Asynchronous channel**: sender (producer) places data in a buffer (FIFO queue) and asynchronously the data is read by a receiver (consumer).
  - In a single system the buffer is probably in shared memory, but if the tasks are mapped to physically different processors there may be a buffer at either end of a *physical* channel.
  - A consumer trying to read from an empty buffer may either (i) block or (ii) carry on and periodically poll the channel
  - A producer trying to write to a full buffer will either (i) block or (ii) carry on waiting periodically polling the channel to see if the buffer has space
- **Synchronous channel**: both producer and consumer must simultaneously want to communicate
  - If consumer is not ready, producer either (i) blocks, (ii) periodically polls channel
  - If producer is not ready, consumer either (i) blocks, (ii) periodically polls channel

  (ECS students may recognize this as the method of inter-process communication in CSP)

Common real-time program pitfalls

- **Deadlock:** two or more processes have arrived in a state where it is impossible for either to proceed
  - Example 1: Task 1 obtains excuslive use of Resource 1 (eg via a semaphore), while Task 2 similarly obtains Resource 2. If Task 1 then requests Resource 2 and Task 2 requests Resource 1 (without fi rst releasing their other semaphores), both will indefi nitely block.
  - Example 2: Task 1 tries to send Task 2 a message over a synchronous channel but Task 2 is not ready (so Task 1 blocks). Task 2 then tries to send Task 1 a synchronous message.
- **Livelock:** two or more processes are continuing to operate but cannot achieve a goal
  - Example: both a producer and consumer of a message periodically poll a synchronous channel to see if the other is ready, but always "miss" each other. Solution is to have at least one of the tasks block so it is guaranteed to be waiting when the other is polls
- **Indefinite postponement:** a task is permanently locked out of a particular resource
  - Example: A (badly-designed) system has a high-priority task which takes 12ms to complete but wants to run every 10ms. All other processes are denied processor time by the pre-emptive priority scheduler

Further words of warning

- Despite the best intentions of designers, virtually all complex software can be made to behave in an unexpected fashion under some conditions.
- This is especially true of systems with real-time elements since verifying real-time software is incredibly difficult.
- Do not confuse reliability with safety

  A system may operate reliably for long periods, but when reliability fails it must still be **safe**
- Corollary: a system which relies totally on software for safe operation is badly designed.

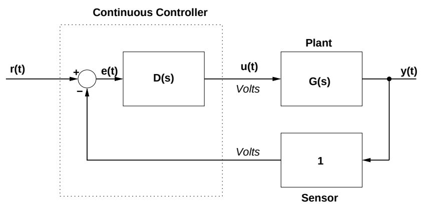
  Hardware interlocks can prevent disaster even if there are software bugs

LECTURE 4

Fast sampled digital control

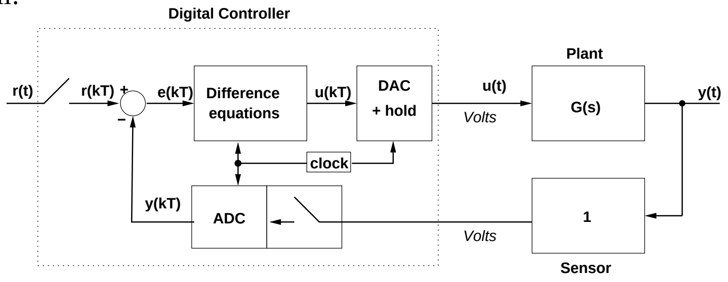
Digital control

- Continuous controller built using analogue electronics; resistors, capacitors, op-amps



- Most modern control systems use digital computers (microprocessors or micro-controllers):
  - ease of reprogramming, hence **flexibility**
- In B4 course look in some detail at **design** of digital controllers, esp. using the so-called **z-transform**, the discrete equivalent of the Laplace transform
- Our aim here is to consider how you would go about replacing a continuous controller with a digital one and to explore the main issues this raises

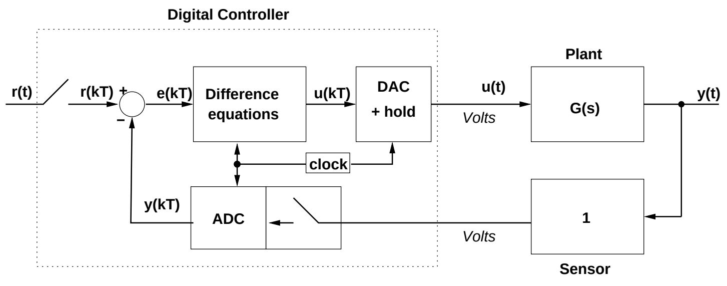
Recall:



- Operates on **discrete samples** of the input demand $r(t)$, and the sensed output $y(t)$ (often obtained by A/D conversion).
- Samples at **regular intervals** T apart (governed by clock), giving **sample rate** of $\frac{1}{T}$Hz (or $\frac{2\pi}{T}$ rad/sec)
- Typically require sample rate 30-40 $\times$ the closed-loop bandwidth of the continuous system; see why later

Components of a digital controller

- **Clock:** regimenting a regular **control cycle**
- **Sampler:** sampling the output of the plant; ie sampling the sensor data (looked at this earlier)
  - ADC, or
  - pulse counter
- **Microprocessor:** running software to implement the **control law**
- **Output:** typically a DAC + hold device



Controller implementation

- Digital controller implemented using **difference equations**
- Difference equation is discrete approximation to differential equation

$$\text{Example: instead of } \dot{x}(t) = Ax(t)$$
$$\text{have } x((k+1)T) = Fx(kT)$$

- **Finite differences** provide a simple (first order) approximation (Euler's method)

$$\dot{x}(k) \approx \frac{x(k) - x(k-1)}{T} \quad \text{backward difference}$$
$$\dot{x}(k) \approx \frac{x(k+1) - x(k)}{T} \quad \text{forward difference}$$

where T is the sample interval.

Example: PID control

Recall: Transfer function of PID controller from error signal $E(s)$ to control action $U(s)$ has the form

$$D(s) = \frac{U(s)}{E(s)} = K(1 + \frac{1}{sT_i} + sT_d)$$

To convert this to the time domain:

- Write $U(s) = D(s)E(s)$
- Multiply through by s

$$sU(s) = K(s + \frac{1}{T_i} + s^2T_d)E(s)$$

- Now take inverse Laplace Transforms

$$\dot{u}(t) = K\left(\dot{e}(t) + \frac{1}{T_i}e(t) + T_d\ddot{e}(t)\right)$$

Now to obtain the corresponding difference equation from the differential equation

$$\dot{u}(t) = K\left(\dot{e}(t) + \frac{1}{T_i}e(t) + T_d\ddot{e}(t)\right)$$

- Substitute the finite differences for differentials

$$\dot{u}(k) \approx \frac{u(k) - u(k-1)}{T} \quad \text{and} \quad \dot{e}(k) \approx \frac{e(k) - e(k-1)}{T}$$

and (applying Euler's method twice)

$$\ddot{e}(k) \approx \frac{\dot{e}(k) - \dot{e}(k-1)}{T} \approx \frac{e(k) - 2e(k-1) + e(k-2)}{T^2}$$

- Rearrange to obtain

$$u(k) = u(k-1) + K\left[(1 + \frac{T}{T_i} + \frac{T_d}{T})e(k) - (1 + 2\frac{T_d}{T})e(k-1) + \frac{T_d}{T}e(k-2)\right]$$

C implementation

```
void pidcontrol(float K, float Ti, float Td, // controller params
                float T)                     // sampling interval
{
  float e[3] = {0.0, 0.0, 0.0},
        u[2] = {0.0, 0.0},
        sample;

  while (1) {
    // shuffle error and control histories
    e[2]=e[1]; e[1]=e[0]; u[1]=u[0];

    wait();                 // wait for clock interrupt
    sample = getADC();      // sample system output from A/D
    e[0] = demand - sample; // compute current error

    // compute next control action and do it
    u[0] = u[1] + K*((1.0 + T/Ti + Td/T)*e[0]
                - (1.0 + 2*Td/T)*e[1] + Td/T * e[2];

    putDAC(u[0]); // send control action to D/A
  }
}
```

Effect of sampling

- We are replacing a **continuous** system with a **sampled** one
- The difference equations implementing controller provide a set of values $u(kT)$ at each clock tick, but we need to apply the action (voltage, current, whatever) continuously
- Thus, we need to reconstruct a continuous signal from a set of samples
- You already know all the theory from your A1 and A2 courses . . .
- The **sampling theorem** tells us that the theoretical lower limit for the sampling rate which enables reconstruction of the signal from the samples is **twice** the highest frequency present in the original signal
- It also tells us constructively how to do so . . .

Sampling (revision)

- Let $u(t)$ be the original signal (in our case, the continuous control action we'd like to reconstruct)
- The sampling operation can be represented mathematically by convolving the signal with the infinite train of impulses

$$\sum_{k=-\infty}^{\infty} \delta(t-kT) = \frac{1}{T}\sum_{n=-\infty}^{\infty} e^{j\omega_s nt}$$

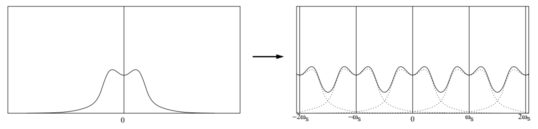
(where ω_s is the sampling frequency, and the equality has been derived by writing the impulse train as a Fourier series)

- Thus the Fourier Transform of the sampled signal can be written

$$\begin{aligned}\mathcal{F}\left(\sum u(t)\delta(t-kT)\right) = U^*(\omega) &= \int_{-\infty}^{\infty} e^{-jwt}u(t)\sum_{n=-\infty}^{\infty} e^{j\omega_s nt}dt \\ &= \frac{1}{T}\sum\int_{-\infty}^{\infty} u(t)e^{-j(\omega-\omega_s n)t}dt \\ &= \frac{1}{T}\sum U(\omega-\omega_s n)\end{aligned}$$

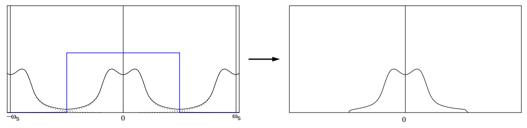
Sampling (revision)

- So the Fourier Transform of the sampled signal is the Fourier Transform of the original, but with an infinite series of copies (**sidebands**) spaced ω_s apart.



$U(j\omega)$ $U^*(j\omega)$

- If the sidebands overlap then we will see the effects of **aliasing**; faster sampling pushes the sidebands "further out"
- If the original signal $u(t)$ is strictly band-limited, then we can recover it from the samples by applying a low-pass filter to the samples to cut out the sidebands



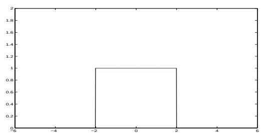
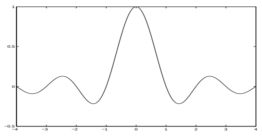
$L(jw)U^*(j\omega)$ $U(j\omega)$

Data interpolation (revision)

- The ideal low-pass filter with sharp cutoff frequency π/T is

Frequency domain *Time domain*

$$L(j\omega) = \begin{cases} T & \frac{-\pi}{T} \le \omega \le \frac{\pi}{T} \\ 0 & \text{elsewhere} \end{cases} \qquad l(t) = \frac{1}{2\pi}\int_{-\pi/T}^{\pi/T} T e^{j\omega t} d\omega = \text{sinc}\frac{\pi t}{T}$$

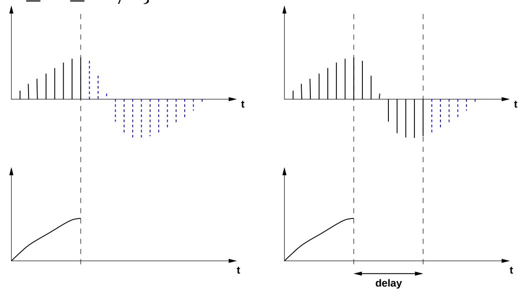
- Thus in the time domain application of the filter is a convolution with sinc functions:

$$u(t) = \sum_{k=-\infty}^{\infty} u(kT)\text{sinc}\frac{\pi(t-kT)}{T}$$

- The sinc functions **interpolate** between the sample gaps

Data interpolation (revision)

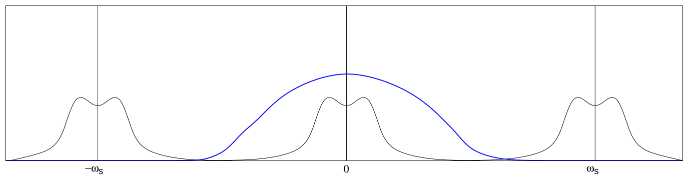
- **Bad news:** Ideal low-pass filter is **non-causal** because $l(t)$ is non-zero for $t < 0$, thus we need the **entire time history** $\{u(kT), -\infty \le k \le \infty\}$ of $u^*(t)$ before we can perform a reconstruction
- If adding some delay were not important (eg many communications applications) in practice we could probably get away with adding a phase lag $e^{-j\omega\lambda}$ to the low-pass filter and use $\{u(kT), -\infty \le k \le 2\pi/\lambda\}$



- Adding delays like this in a control system could lead disastrously to **instability**

Data extrapolation (revision)

- The solution is to increase the sample rate, consequently pushing the sidebands further away

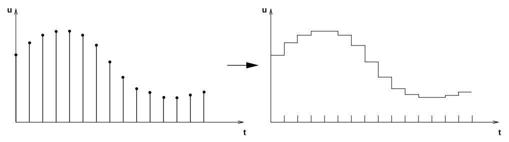


- This has two significant benefits
  - we reduce aliasing effects
  - we can use a "simpler" filter
  - we can use a filter that introduces less phase lag
- To minimize delay what we really want, rather than a **data interpolator** (which requires both past and future values before it produces a signal), is a **data extrapolator**

Output: Zero-order Hold

Finally we return to the components of the digital controller, and now consider the output part. . .

- A general technique of **data extrapolation** from samples is to use a polynomial to fit past samples
- If the polynomial used is the constant, zero-order polynomial, then the extrapolator is called a **zero-order hold**.

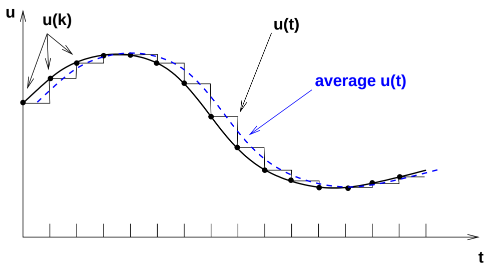


- Most digital controllers use ZOH since it has the tremendous virtue of being almost trivial to implement: digital hold (ie a latch) followed by a digital to analogue conversion (see, eg, XY scope display presented earlier).
- ZOH still exhibits some delay (**phase lag**) which we will now quantify

ZOH

The **single most important** impact of implementing a control system digitally is the **delay** associated with the hold. Delays of course lead to loss of damping and degrade system stability.

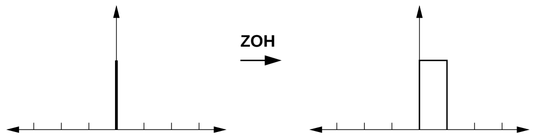
- Each value $u(k)$ is held constant until the next value is available
- Continuous value of $u(t)$ therefore lags on average by $T/2$



ZOH

More formally: we can incorporate the effect of the hold delay in a continuous analysis of the system.

- Recall that the transfer function of a system is given by its impulse response
- The impulse response of ZOH is a unit pulse of width T

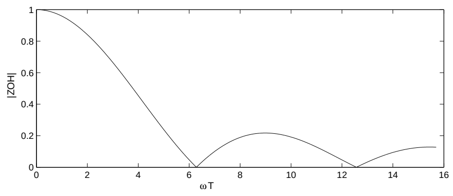


- Thus the transfer function of zero-order hold, ZOH(s), is given by

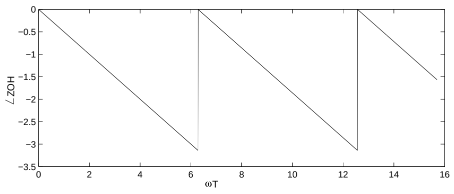
$$\text{ZOH}(s) = \frac{1 - e^{-sT}}{s}$$

ie a unit step followed one unit of time later by a step in the opposite direction

ZOH



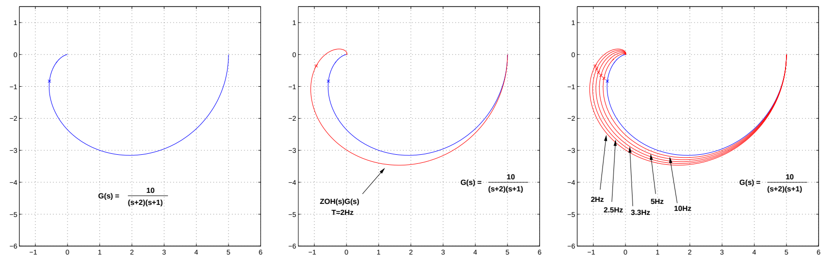
Magnitude of ZOH versus ωT



Phase of ZOH versus ωT

- Note that the phase of ZOH is $\frac{-\omega T}{2}$; ie it introduces a **phase lag**

Effects on closed-loop control



- Recall that for a second-order system (one with a dominant pole pair)

$$\zeta \approx \frac{\text{PM}^o}{100}$$

- The effect of the phase-lag is to reduce the damping
- Thus to find the approximate loss of phase margin we find the frequency at the phase margin circle ω_{PM}, or the *gain cross-over frequency* (this would have been our design frequency when compensating the continuous system), and subtract $\frac{\omega_{\text{PM}} T}{2}$

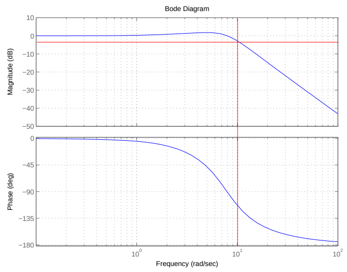
Example

Design a digital controller to implement the lead compensator

$$D(s) = 70\frac{s+2}{s+10}$$

to control the plant

$$G(s) = \frac{1}{s(s+1)}$$



Closed-loop bandwidth is about 10 rad/sec ≈ 1.6Hz, so we might expect to need a sampling frequency in excess of 50Hz.

Derive finite difference approximation to controller:

Control action $U(s)$ is

$$U(s) = D(s)E(s) = 70\frac{s+2}{s+10}E(s)$$
$$\Rightarrow U(s)(s+10) = 70(s+2)E(s)$$

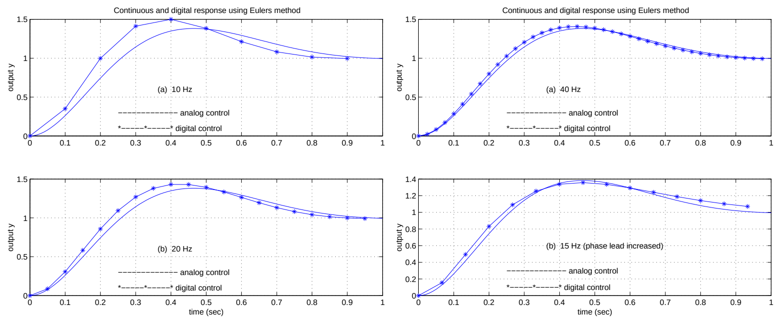
So writing time domain representation yields

$$\dot{u}(t) + 10u(t) = 70\dot{e}(t) + 140e(t)$$

Difference equations using forward differences:

$$\frac{u(k+1)-u(k)}{T} + 10u(k) = 70\frac{e(k+1)-e(k)}{T} + 140e(k)$$

$$\Rightarrow u(k+1) = (1-10T)u(k) + 70(2T-1)e(k) + 70e(k+1)$$

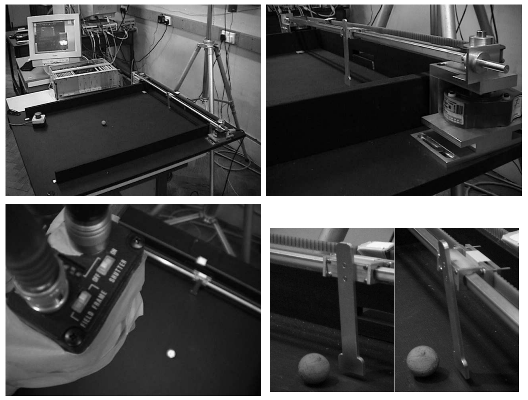


- As we would expect, lower sampling rates result in lower damping (more overshoot)
- With a sampling rate of 40Hz the digital approximation has very similar performance to the continuous controller
- A little thought suggests that by increasing the phase-lead in the controller we may be able to get away with a lower sampling rate. Plot (d) shows the response of a digital approximation to $D(s) = 70\frac{s+2}{s+12}$ using only a 15Hz (ie 10× bandwidth) sample rate.
- In general however, if we can't sample at 30-40× the system bandwidth we may be better of with a more rigorous digital design technique (see B4 option)

Example

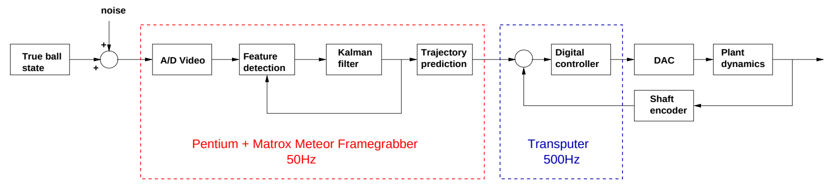
Final example: **A robotic table football goalkeeper**

- Aim: to control a paddle to intercept a ball by observing the ball's position with a camera
- 4th year project, 2000/2001



Example

- Based on a 200Mhz Pentium PC equipped with a Matrox Meteor frame-grabber (A/D Video)
- Visual processing at 50Hz (analogue video fi eld rate)
- Servo-control (digital PD controller) running at 500Hz on a T800 transputer
- Communications between PC and Transputer via a 2Mbit serial link


noise
True ball state
+
+
A/D Video
Feature detection
Kalman filter
Trajectory prediction
Digital controller
DAC
Plant dynamics
Shaft encoder
Pentium + Matrox Meteor Framegrabber 50Hz
Transputer 500Hz