

# Specifications and Modeling (5) , Data Flow (1)

Section 2.4.2 - of textbook relates

Skip: Sections 2.4.3, 4.4.4

Section 2.5 – Data Flow

L9

## Embedded Systems II

Dr Simon Winberg



Electrical Engineering  
University of Cape Town

# Test study suggestions

## Ch1 problems

- 1.1 Please list possible definitions of the term “embedded system”!
- 1.2 How would you define the term “cyber-physical system (CPS)”? Do you see any difference between the terms “embedded systems” and cyber-physical systems”?
- ~~1.3 What is the “Internet of Things” (IoT)?~~
- 1.4 What is the goal of “Industry 4.0”?
- ~~1.5 In which way does this book cover CPS and IoT design?~~
- ~~1.6 In which application areas do you see opportunities for CPS and IoT systems?~~
- 1.7 Use the sources available to you to demonstrate the importance of embedded systems!
- 1.8 Which challenges must be overcome in order to fully take advantage of these embedded systems opportunities?
- 1.9 What is a hard timing constraint? What is a soft timing constraint?
- 1.10 What is the “Zeno effect”?
- ~~1.11 What is adaptive sampling?~~
- 1.12 Which objectives must be considered during the design of embedded systems?
- 1.13 Why are we interested in energy-aware computing?
- 1.14 What are the main differences between PC-based applications and embedded/CPS applications?
- 1.15 What is a reactive system?
- 1.16 On which Web sites do you find companion material for this book?
- ~~1.17 Compare the curriculum of your ...~~
- ~~1.18 What is flipped classroom teaching?~~
- 1.19 How could we model design flows?
- 1.20 What is the “V-model”?
- 1.21 How could we define the term “synthesis”?



# Test study suggestions

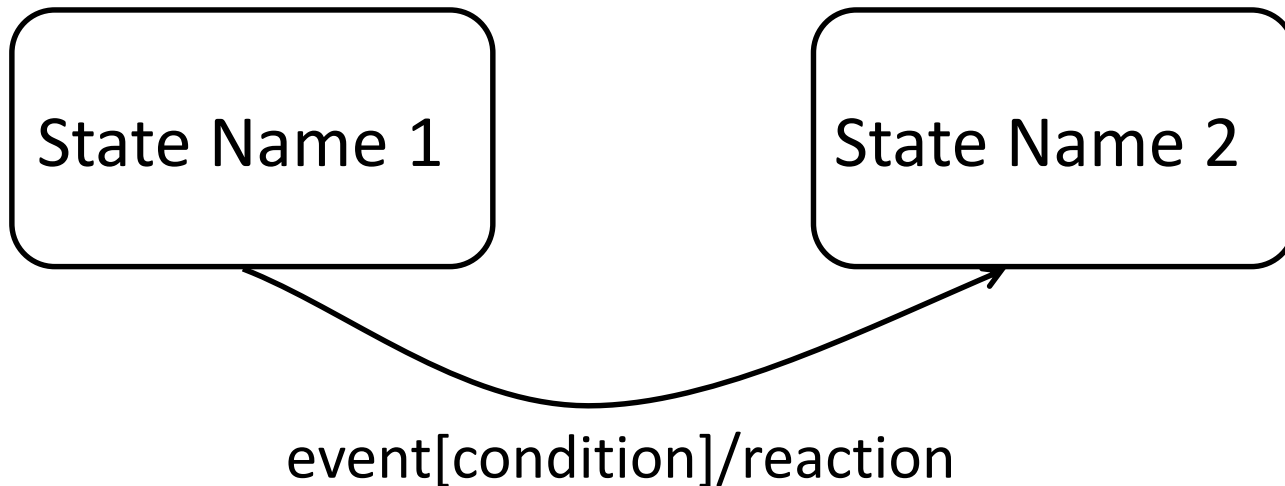
## Ch2 problems

- 2.1: What is a (design) model?
- 2.2: Prepare a list of up to six requirements for specification/modeling languages for embedded systems!
- 2.3: Why could our specification lead to deadlocks?
- 2.4: What is a “model of computation (MoC)”?
- 2.5: What is a “job” and how is it different from “tasks”?
- 2.6: Which are the two key techniques for communication in computers?
- 2.7: Which description techniques can be used for capturing initial ideas about the system to be designed?
- ~~• 2.8: Simulate trains between Paris, Brussels, Amsterdam, and Cologne, using the~~
- ~~• 2.9: Download the OpenModelica simulation software.~~
- 2.10: Modify the answering machine of Example 2.8 such that the owner can intervene at any time during the playing of recorded ...
- 2.11: Model your daily schedule with a timed automaton. Hours are reflected by a variable  $h$ , days by a variable  $d$ .  $d = 1$  means ...
- 2.12: Suppose the StateCharts model in Fig.2.77 (left) model is given...
- 2.13: Are StateCharts determinate models if we follow the StateMate semantics?
- ~~• 2.14: Is SDL a determinate language? Please explain your answer!~~
- 2.15: Let us assume that you have been asked to help modeling the flow of visitors in the hypothetical Museum of Fine Future Information Nuggets (MUFFIN)...

# Outline of Lecture

- Modelling
  - Checking timing constraints of FSMs
  - Data Flow:
    - Kahn Process Networks (KPN)
- Requirements Considerations
  - Interface Control Documents (ICD)

# State Chart transition syntax



While UML State Charts are like Mealy Machines, to be more accurate they consider transitions as more events (on the left before the /) and reactions (on the right after the /). Note the reaction is clearly in addition to changing to whatever state the transition is pointing to.

i.e. it is not necessarily just inputs and outputs that the system models, you could for example have an event as a divide by zero or other exception that is handled.

# Terminology: Determinate

- Defn. A system is **determinate** if it:
  - Always gives the same result for a specific set and timing of inputs (Kahn, 1974)
- This refers to the behavior of systems, i.e. indicates the strength of repeatability
- Not to be confused with e.g.
  - Deterministic state machine or 'deterministic finite automaton' (DFA) = a state machine for which each pair of state and input symbol there is one and only one transition to a next state.

# State Chart Execution

- It may be fairly obvious how state charts are evaluated... However, certain **assumptions can creep in** at this stage of development which may cause a **specification mismatch** or lead to **non-determinate** behaviour (either at the execution stage or in how a human understanding what the model is showing).
- For this reason, and because state charts and state machines are so commonly used in embedded systems, it is important for us to consider **formal procedure** for generating an implementation from these models



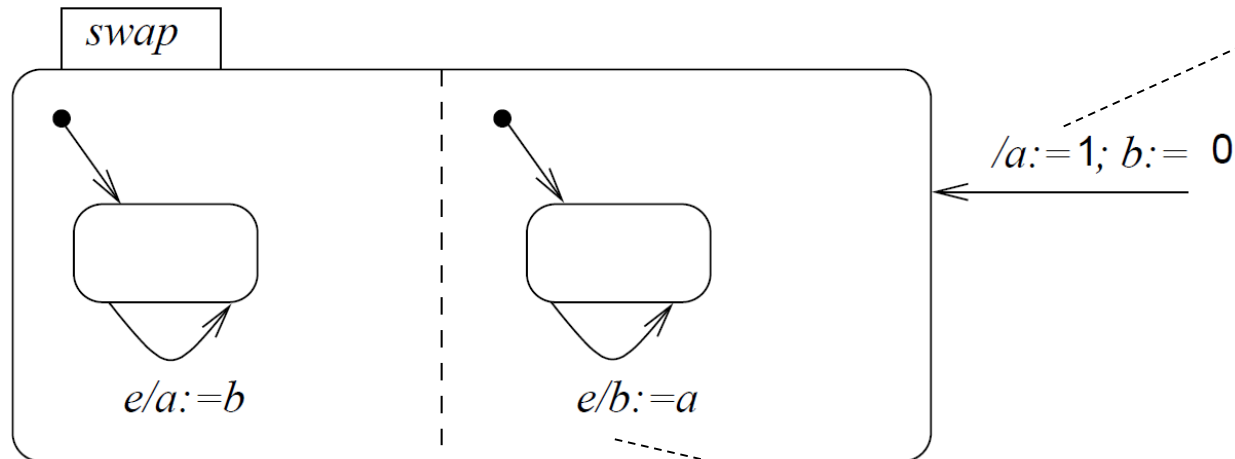
# Semantics of StateCharts – execution phases (based on StateMate execution semantics)

- How are edge labels **evaluated**?
- Three phases:
  1. Effect of external changes on **events and conditions** is **evaluated**,
  2. The **set of transitions to be made** in the current step\* **and** right hand sides of **assignments** are computed,
  3. The **transitions are effected**, variables obtain new values.
- Separation into phases 2 and 3 enables a resulting unique (“determinate”) behavior.

\* Remember State Charts can express concurrent state machines



# Example



Clearly this transition is simply saying force a change to  $a$  and to  $b$ . You wouldn't really do this in a useful system.

- In phase 2, variables  $a$  and  $b$  are assigned to temporary variables:

$a' := b, b' := a;$

In phase 3, these are assigned to  $a$  and  $b$ .

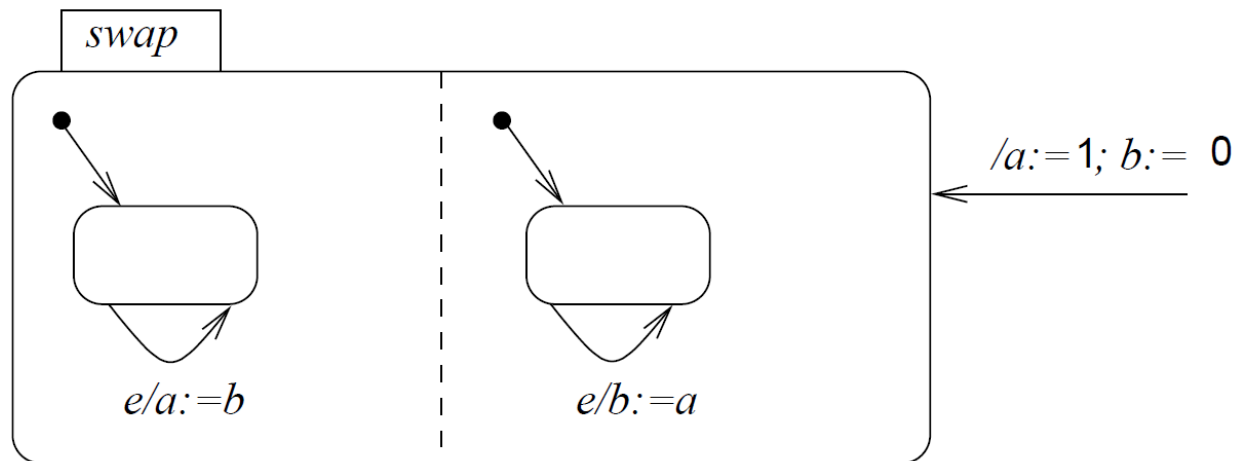
$a := a', b := b';$

As a result, variables  $a$  and  $b$  are swapped.

( $e$  = shortcut for any event received by the system)

On any event just assign  $b$  to  $a$

## Example (2)



- In a single phase environment, executing the left state first would assign the old value of  $b$  ( $=0$ ) to  $a$  and  $b$ :

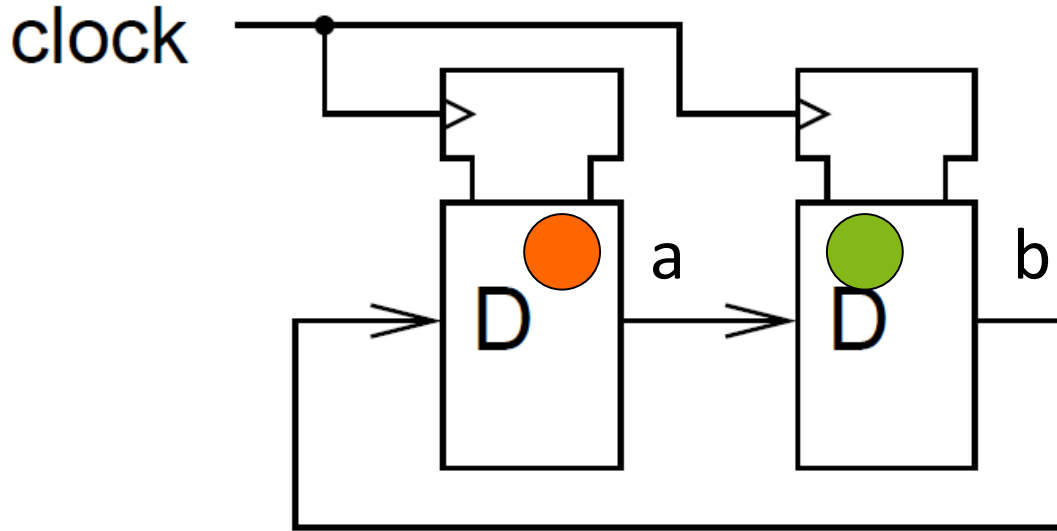
$a := 0, b := 0;$

Executing the right state first would assign the old value of  $a$  ( $=1$ ) to  $a$  and  $b$ .

$b := 1, a := 1;$

The result would depend on the execution order.

# Reflects model of clocked hardware

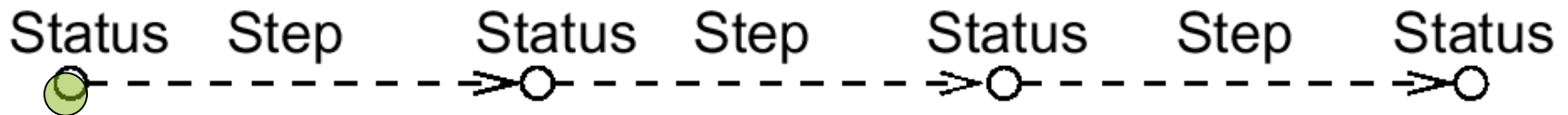


- In an actual clocked (synchronous) hardware system, both registers would be swapped as well.

Same separation into phases found in other languages as well, especially those that are intended to model hardware.

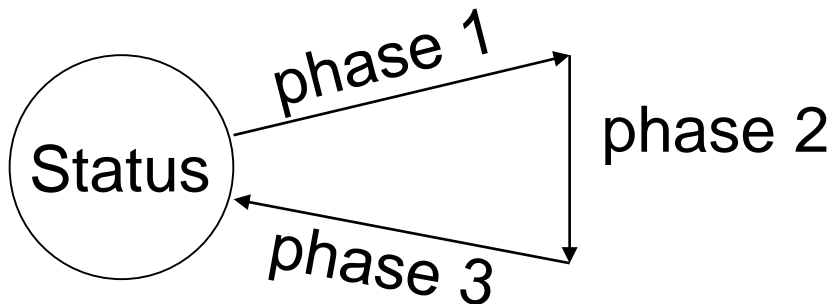
# State Chart Execution Steps

Execution of a State Chart model consists of a sequence of (status, step) pairs



Status = values of all variables + set of events + current time

Step = execution of the three phases (e.g. **StateMate** semantics)



(not examined)

Other implementations of StateCharts do not have these 3 phases (and hence could lead to different results)!



# Evaluation of StateCharts (1)

## Pros (👍):

- Hierarchy allows arbitrary nesting of AND- and OR-super states.
- (StateMate-) Semantics defined in a follow-up paper to original paper.
- Large number of commercial simulation tools available (StateMate, StateFlow, BetterState, ...)
- Available “back-ends” translate StateCharts into SW or HW languages, thus enabling software or hardware implementations.

# Evaluation of StateCharts (2)

## Cons (👎):

- Not useful for distributed applications,
- no program constructs,
- no description of non-functional behavior,
- no object-orientation,
- no description of structural hierarchy,
- generated programs may be inefficient.

## Extensions:

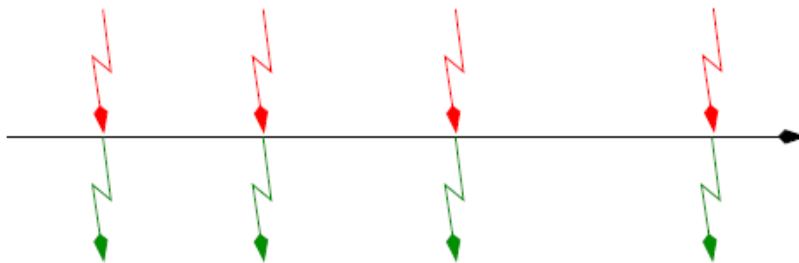
Module charts for description of structural hierarchy.

(we won't really use this in the course, more relevant to big projects)

# Practical Constraints and Compositionality

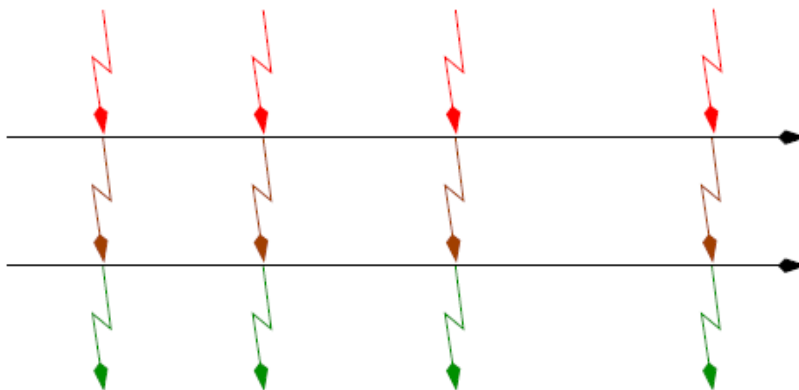
## Abstract synchronous behavior

sequence of reactions to input events, to which all processes take part:



At the abstract level,  
a single FSM reacts  
*immediately*

Composition of behaviors:

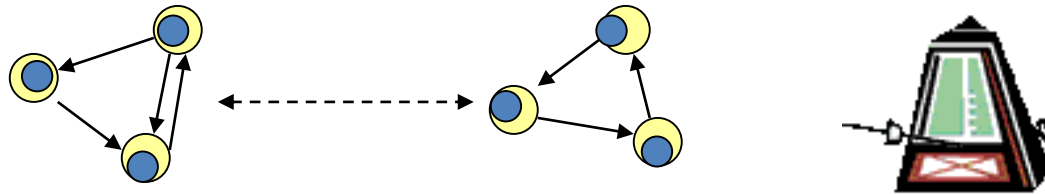


At the abstract level,  
reaction of **connected**  
automata is likewise  
*immediate*

*Main Point:* The physical implementation will have certain physical execution delays

# Synchronous vs. asynchronous languages or state machine descriptions

*Clocked synchronous behavior – a means to abstract away the physical delays*



- Synchronous design (or representation languages) implicitly assume the presence of a (global) clock.
- Each clock tick, all inputs are considered, new outputs and states are calculated and then the transitions are made.

**Why would you want to use such a model / design construct?**

*Reasons:* It makes the design easier to understand, more easily grab a system-wide snapshot of the system. Helps debugging. In an asynchronous system each state might operate at different speeds, expecting input, generating results at different times.



# Abstraction of delays

- Let
  - $f(x)$ : some function computed from input  $x$ ,
  - $\Delta(f(x))$ : the delay for this computation
  - $\delta$ : some abstraction of the real delay
- Consider compositionality:  $f(x)=g(h(x))$

Then, the sum of the delays of  $g$  and  $h$  would be a safe upper bound on the delay of  $f$ .

This is necessary to consider when calculating the anticipated performance of a state machine model.

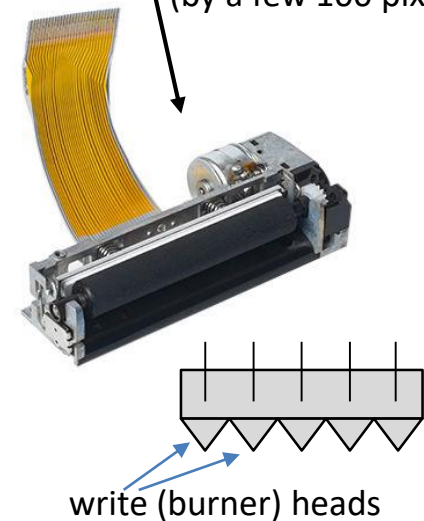
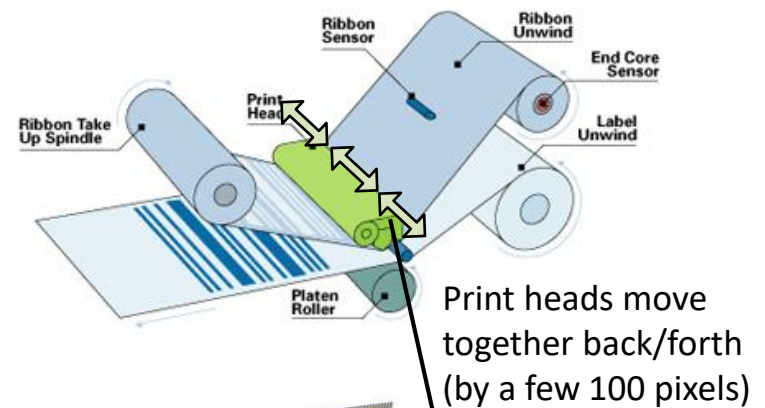
This can then be used in calculating if the timing constraints of a system are satisfied (e.g. hard deadlines always be met).

not examined

# Accurate Modelling

- The **abstraction of synchronous languages** (or state machine models) is valid if the real delays (e.g. combination and latencies of functions) are always shorter than the clock period.
- Proving a synchronous FSM satisfied timing constraints is thus easier, but you need to
  1. Calculate longest computation that a state needs to do
  2. Check this time is  $<$  clock period
  3. Check the time for the sequence of states ( $n$  x clock period) to handle a deadline is sufficiently small

# Brief Example



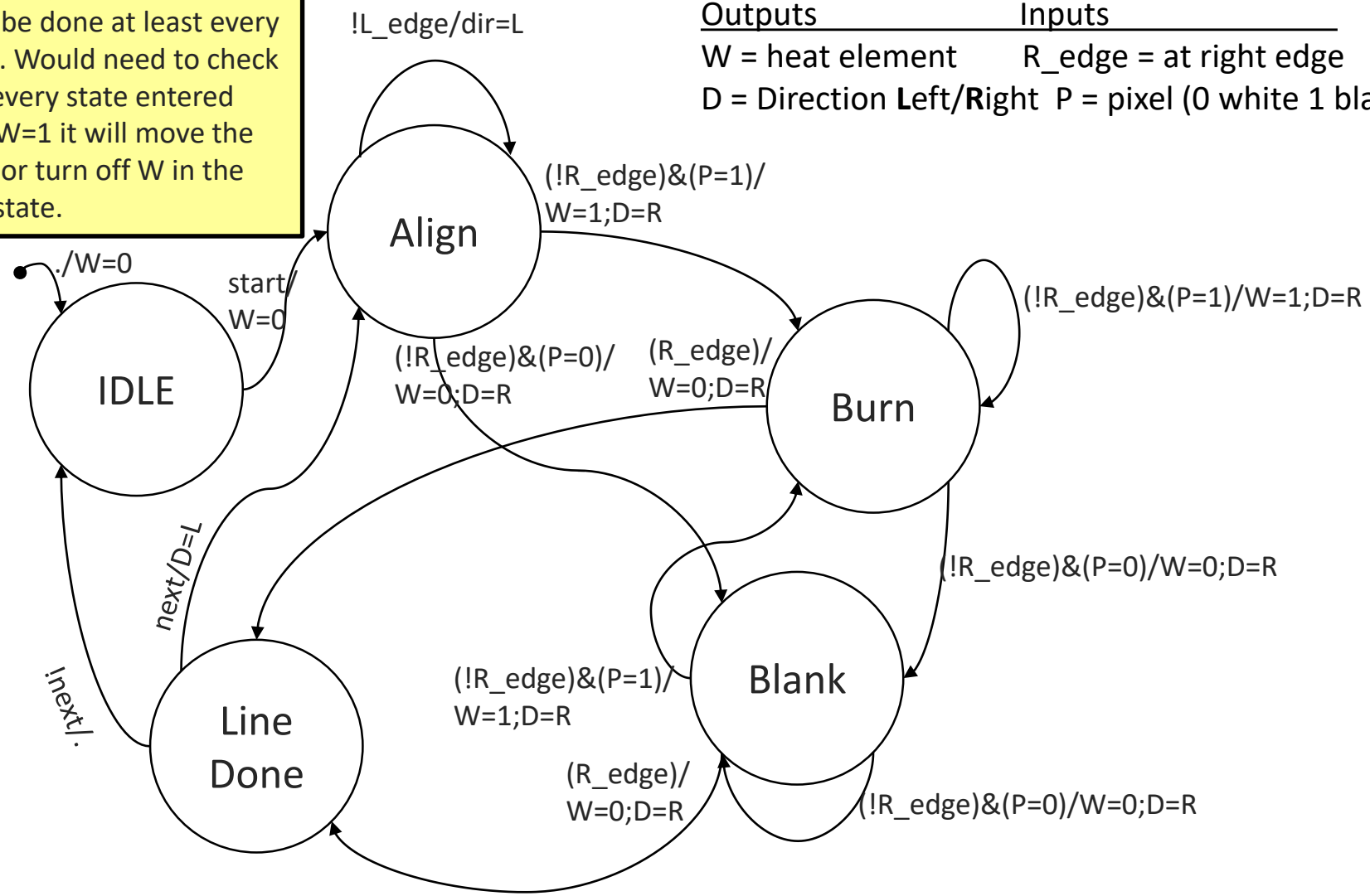
Checking a state chart satisfies the timing constraint for the (simplified) operation of a thermal printer.

**Timing constraint:** heating element never on for more than 20ms in one place. If this statechart is synchronous it would be quite easy, each transition must be done at least every 20ms. Would need to check that every state entered with W=1 it will move the head or turn off W in the next state.

# Example – Synchronous

## Thermal Printer Example

Outputs	Inputs
W = heat element	R_edge = at right edge
D = Direction Left/Right	P = pixel (0 white 1 black)



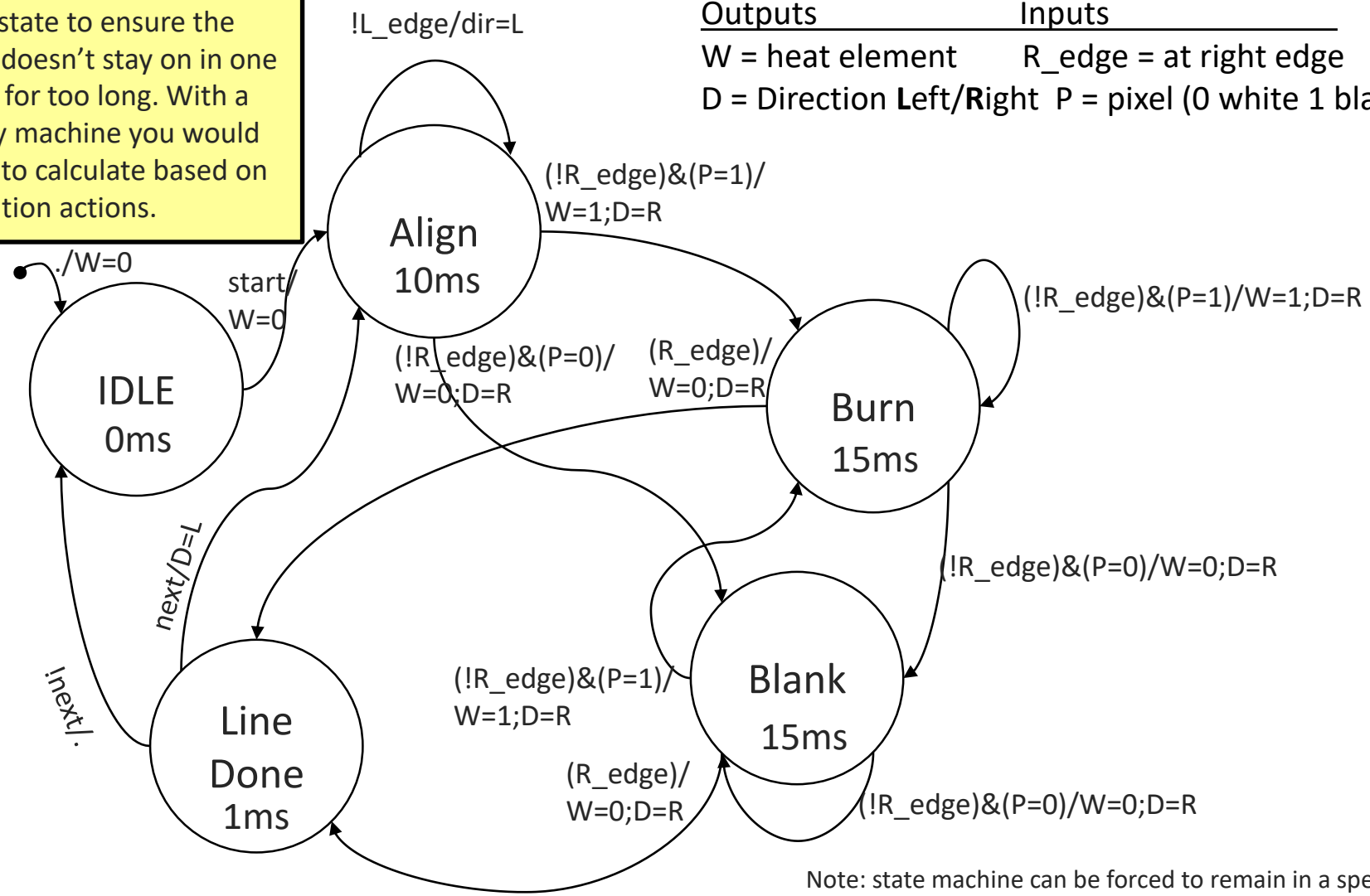


**Timing constraint:** heating element never on for more than 20ms in one place. If asynchronous it is more difficult to check, would have to check the delays for each state to ensure the head doesn't stay on in one place for too long. With a mealy machine you would need to calculate based on transition actions.

# Example – Asynchronous

## Thermal Printer Example

Outputs	Inputs
W = heat element	R_edge = at right edge
D = Direction Left/Right	P = pixel (0 white 1 black)



Note: state machine can be forced to remain in a specific state for a specific time before transitioning

# Data Flow

# Models of computation in this course

Communication/ local computations	Shared memory	Message passing	
		Synchronous	Asynchronous
Undefined components	Plain text ✓, use cases ✓   Sequence Charts ✓, ICD		
Communicating finite state machines ✓	StateCharts ✓		SDL
Data flow	Scoreboarding + Tomasulo Algorithm (☞ Comp.Archict.)		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, ...	
Discrete event (DE) model	VHDL*, Verilog*, SystemC*, ...	Only experimental systems, e.g. distributed DE in Ptolemy (Ptolemy only discussed briefly)	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA	

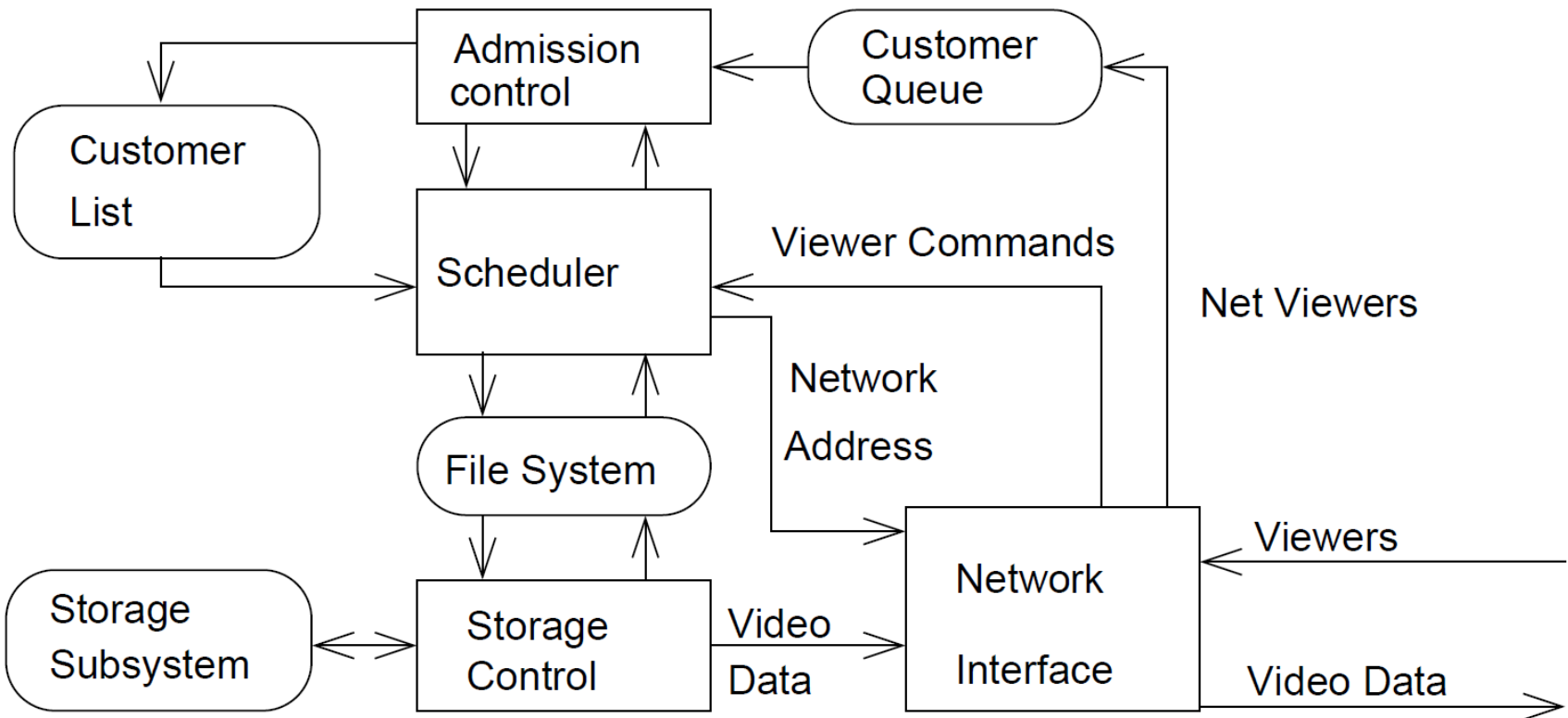
\* Classification based on implementation with centralized data structures

SystemC will not be delved into detail. Only brief flavour of VHDL and Verilog given

# Data flow as a “natural” model of applications

*Example Data Flow (DF) model*

## Example: Video on demand system



*It is not necessary to show a starting point, it is generally assumed to be a continuously running system, data flows from one component to another.*



# Data flow modeling

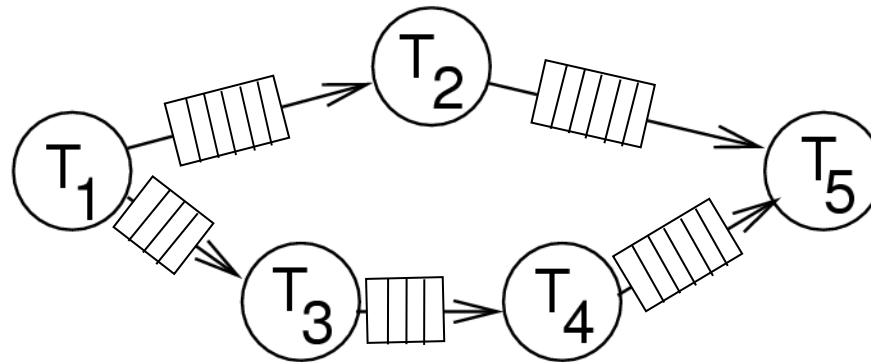
**Definition:** Data flow modeling is ... *“the process of identifying, modeling and documenting how data moves around an information system.*

*Data flow modeling examines*

- processes (activities that transform data from one form to another),*
- data stores (the holding areas for data),*
- external entities (what sends data into a system or receives data from a system, and*
- data flows (routes by which data can flow)”.*

# Kahn Process Networks (KPN)

- Each **component** is modeled as a program/task/process, (underlying each component could be a FSM with many states)
- Communication is by **FIFOs**; no overflow considered
  - 👉 writes never have to wait,
  - 👉 reads wait if FIFO is empty.

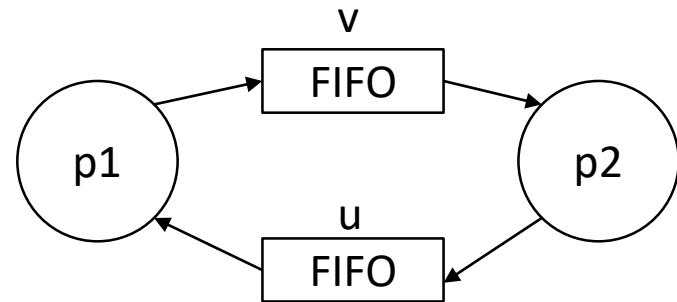


- Only one sender and one receiver per FIFO
  - 👉 no SDL-like conflicts at FIFOs

# Example code and equivalent KPN Diagram

```
process p1 (in int u, out int v){  
  int i;  
  i = 0;  
  for (;;) {      // i.e. same as while (1) loop (just more cryptic)  
    send (i,v);    // sent i to channel v  
    i = wait (u);  // read i from channel u  
    i = i - 1;    }  
}
```

```
process p2 (in int v, out int u){  
  int i;  
  for (;;) {  
    i = wait (v);  // read i from channel v  
    i = i + 1;  
    send (i,u);   } // sent i to channel u  
}
```

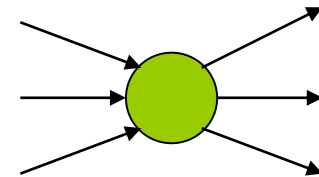


Equivalent KPN Diagram

Notes: this isn't real C++ code, it's just for illustration – but you could conceivably define in and out as classes, although the parameter declarations would probably look more like `buff_in<int> v, buff_out<int> u`

# Properties of Kahn process networks

- Communication is only via channels (no shared variables)
- Mapping from  $\geq 1$  input channel to  $\geq 1$  output channel possible;
- Channels transmit information within an unpredictable but finite amount of time;
- In general, execution times are assumed unknown.

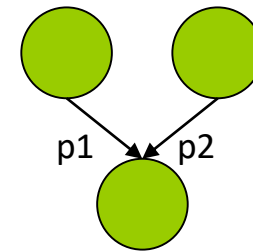




# Key beauty of KPNs (1)

- A process cannot check for available data before attempting a read (wait).

~~if nonempty(p1) then read(p1)  
else if nonempty(p2) then read(p2);~~



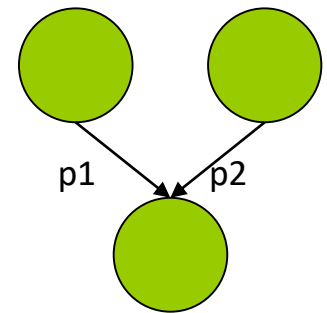
- A process cannot wait for data for >1 port at a time.

~~read(p1|p2);~~

- ☞ Processes have to commit to wait for data **from a particular port**;

# Key beauty of KPNs (2)

- ➡ Therefore, the order of reads does not depend on the arrival time (but may depend on data).
- ➡ Therefore, Kahn process networks are **determinate** (!); for a given input, the result will always be the same, regardless of the speed of the nodes.
- ➡ Many applications in embedded system design: Any combination of fast and slow simulation & hardware prototypes always gives the same result.



# Computational power and analyzability

- It is a challenge to schedule KPNs without accumulating tokens
- KPNs are Turing-complete (anything which can be computed can be computed by a KPN)
- KPNs are computationally powerful, but difficult to analyze (e.g. what's the maximum buffer size?)
- Number of processes is static (cannot change)

# More information about KPNs

*Optional additional reading concerning KPNs if you like these models*

- <http://ls12-www.cs.tu-dortmund.de/teaching/download/levi/index.html>: Animation
- [http://en.wikipedia.org/wiki/Kahn\\_process\\_networks](http://en.wikipedia.org/wiki/Kahn_process_networks)
- See also S. Edwards: <http://www.cs.columbia.edu/~sedwards/classes/2001/w4995-02/presentations/dataflow.ppt>

not examined



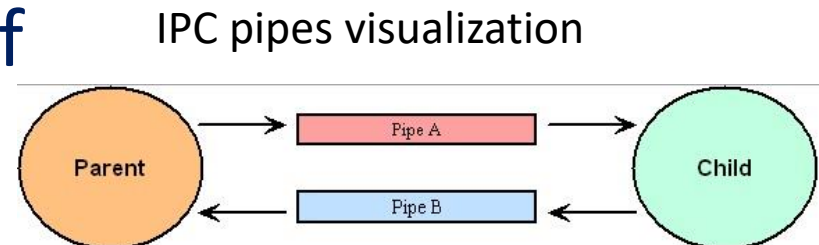
# IPC Pipes – de facto KPN implementation

- The implementation of a KPN essentially need a mechanism to send data between processes... you could do this in various ways e.g.
  - Write your own thread application and FIFOs in shared memory (bit of work)
  - Use TPC or UDP sockets (but much work)
  - Use IPC pipes – quick and easy!  
(if everything running on one CPU)

Let's have a super quick look at using IPC pipes

# IPC Pipes

- IPC pipes are natively provided in the Linux kernel, they are also included with Win32/64
- Linux supports two types
  - Half Duplex: pipe that can send or receive data in only one direction at a time.
  - Full Duplex: pipe that can both send and receiving data at the same time.
- There are a small set of easy functions to use...



Functions worth remembering!

# Linux IPC Pipe Functions

Function	Purpose: Example	Parameters	Result Type
<b>pipe</b>	If successful, creates a new pipe and returns a value of 0. The read and write file descriptors are written into the array argument. If not successful, returns -1.  <code>pipe (filedes);</code>	<code>int filedes[2]</code>	<code>int</code>
<b>dup2</b>	If successful, duplicates the file descriptor <code>oldfiledes</code> into the file descriptor <code>newfiledes</code> and returns <code>newfiledes</code> . If not successful, returns -1.  <code>dup2 (oldfiledes, newfiledes);</code>	<code>int oldfiledes</code> <code>int newfiledes</code>	<code>int</code>
<b>sleep</b>	If successful, pauses the program execution for the specified number of seconds and returns 0. If unsuccessful, returns the number of seconds remaining to sleep.  <code>sleep (seconds);</code>	<code>unsigned int seconds</code>	<code>unsigned int</code>
<b>close</b>	Closes the designated file. Returns 0 if successful; -1 if unsuccessful.  <code>close (oldfiledes)</code>	<code>int oldfiledes</code>	<code>int</code>
<b>read</b>	Reads <code>numbytes</code> bytes from file <code>oldfiledes</code> into array <code>buffer</code> . Returns the number of bytes read if successful; returns -1 if unsuccessful.  <code>read (oldfiledes, buffer, numbytes)</code>	<code>int oldfiledes</code> <code>void* buffer</code> <code>size_t numbytes</code>	<code>size_t</code>
<b>write</b>	Writes <code>numbytes</code> bytes to file <code>newfiledes</code> from array <code>buffer</code> . Returns the number of bytes read if successful; returns -1 if unsuccessful.  <code>write (newfiledes, buffer, numbytes)</code>	<code>int newfiledes</code> <code>void* buffer</code> <code>size_t numbytes</code>	<code>size_t</code>

# Example Program

```
.... #include <unistd.h>

/* Main function - creates a pipe, opens it and forks into two processes, one to send data into the pipe and the other to receive the data and write it to
the screen. */
int main(void)
{ // define some local variables
  int  fd[2]; // file descriptors
  int  bz;    // size in types
  pid_t childpid; // process ID of the child process
  char readbuffer[80]; // buffer that the child process will store input data into
  char send_message[] = "Hello world!\n"; // the data to be sent
  // tell the OS to create a pipe
  pipe(fd);
  // fork the process, creates a child process that will have a copy of the parent's registers and stack to this point, and will set childpid to a positive value
  in the parent
  // and to zero in the child - if the fork fails the result is negative.
  childpid = fork();
  // check result of the fork
  if (childpid < 0) {perror("Error: fork failed!"); return 1; }
  if (childpid == 0) {
    close(fd[0]); // Start of child process: closes the input side of pipe
    printf("CHILD: sending message\n");
    write(fd[1], send_message, (strlen(send_message)+1)); // Send the message through the output side of pipe to the parent
  } else {
    // Parent process closes up output side of pipe
    close(fd[1]);
    bz = read(fd[0], readbuffer, sizeof(readbuffer)); // Read in a string from the pipe
    printf("PARENT: received: %s\n", readbuffer);
  }
  return(0); // exit the program, returning 0 to report no errors
}
```



pipe\_ex.c

makefile

This program works with gcc on Linux / Cygwin (presumably should work on Mac too), need separate library for Windows



# The Interface Control Document (ICD)



(just the main concept examinable not the specifics on document structure)

# Why bore you with an ICD??

I'm putting this in mainly as it is a best practice.

But also because...

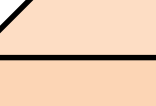
You might well work on projects where there are several interacting systems.

... and you might remember 'this is where that ICD stuff would be rather valuable so everyone knows how to connect up to our respective systems!'

# CMAASM: Continuous Miner Automation-Assist and Safety Monitor

Head Control System

ICD



Master Control

The diagram shows a yellow box labeled "Master Control" with four white arrows pointing towards it from the top, bottom, left, and right. To the right of the box is a stack of three white documents with black lines representing text. The top document has the letters "ICI" in large, bold, black font.

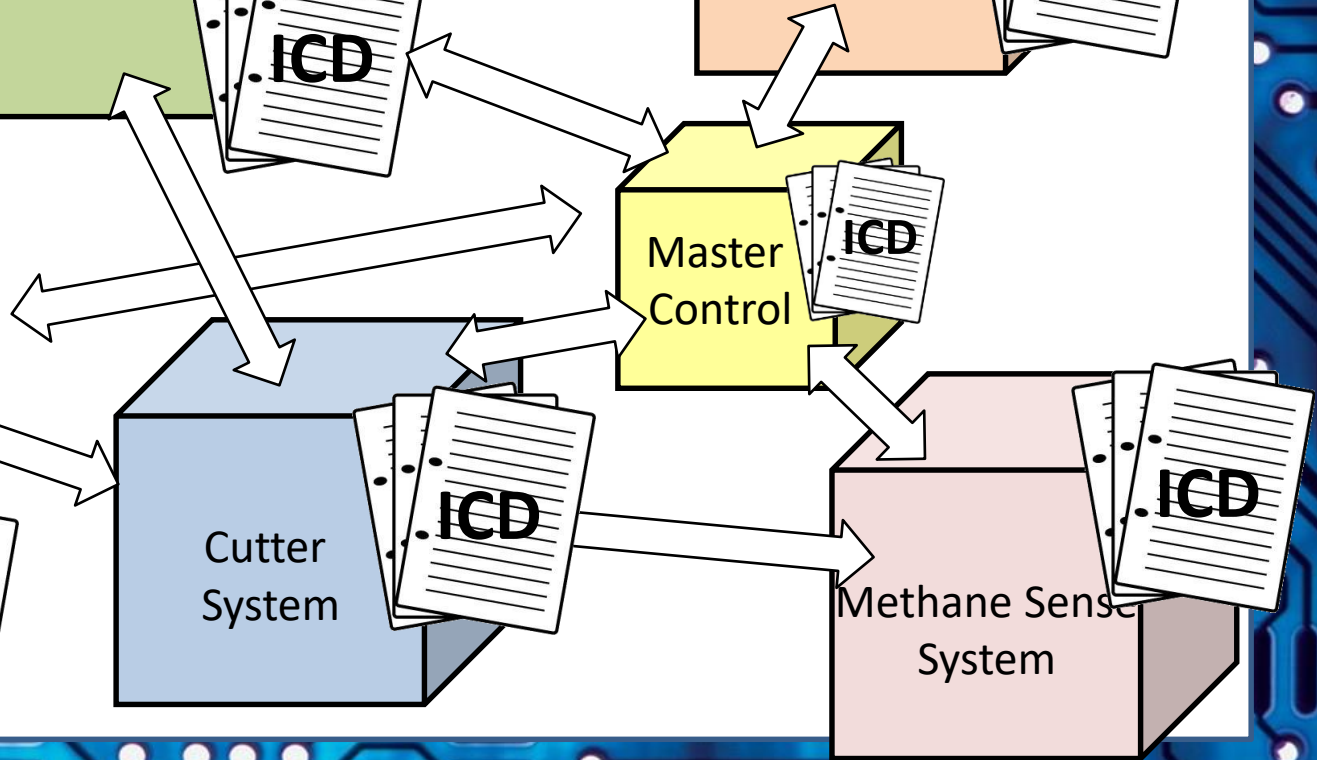
Conveyor System

ICD

A diagram of a 'Cutter System' represented as a blue 3D box. Two white arrows on the top surface point towards each other, forming a loop. On the right side, a stack of papers is shown, with the top paper labeled 'IC' and featuring a list of bullet points. A white arrow points from the stack of papers back into the left side of the box, completing a feedback loop.

Methane Sensor System

ICD




# The Interface Control Document (ICD)

- A document which describes the interfaces of a system
  - Between subsystems within a system, or
  - Between the system and an external system
- For example:
  - Description of the USB communication protocol between a scanner and a PC
  - Description of I2S data transferred between a microprocessor and a DAC



# The Interface Control Document

- The ICD is a textual document
-  Knowing the protocol is useless if you don't know *where* it is used.
- The ICD is often provided with a product
- Allow users to connect the product with other systems
- Allow the product to interface freely with other products which extend it's functionality
- These are **commonly found for systems needing certification or formal conformance**. e.g.:
  - safety-critical systems (e.g. hospital equipment)
  - automotive and aviation
  - military applications

# The Focus of the ICD

- The ICD is not a user manual, service manual, or complete description of the system.
- The ICD has a single focus:
  - To provide a brief, clear and complete specification of the interface(s) used by a system
  - The ICD is not intended for general users
  - It is intended for engineers
  - It should be written for the audience it is intended for

# The Format of the ICD

ICDs can be broken into two basic formats...

- ICDs describing the connection between the system and another specific system
  - E.g. Interface between control tower and flight control system
- ICDs describing the interface a system presents to the outside world
  - E.g. The ATAPI interface supported by a DVD drive

# The Format of the ICD

- There is no widely accepted ICD standard
- Several formats are widely used
- Some companies use different formats for internal use and release
- The format might depend on the content
- But aim for consistency



# Sample ICD Format

- Title Page
- Header
- Overview
- Interface Definitions
- Glossary
- References
- Appendices

# The ICD Header

- Scope
  - Explains what the document covers (and not)
  - Explains which product, which subsystems and which interfaces are described
- System Overview
  - A brief summary describing the systems
  - Table Of Contents

# The ICD Overview

- Introduction
  - Short explanation of the document's layout
- Block Diagram
  - Systems and subsystems involved
  - Label interfaces between systems
- List Protocols
  - Name, type (RF, serial, etc.)
  - Cross references to relevant standards

# The ICD Interface Definition

- Multiple interface definitions segments will be present, with suitable headings
- Structure depends on whether a hardware or software interface is being described
- All software interfaces depend on hardware interfaces
  - The relevant hardware interface must be referenced
  - Many systems use a layered model (remember the OSI model? Is useful here.)



# The ICD Interface Definition 2

- Hardware Interface Definition
  - Voltage, current and SNR limits
  - Type of signaling used (eg LVDS)
  - Transmission speeds and timing
  - Bit representation (eg positive edge trigger)
  - Physical ports (how many conductors, etc)
  - Other Details
    - Maximum distances, etc.
    - Encoding (EFM, Manchester coding, etc)

# The ICD Interface Definition 2

- Software Interface Definition
  - Packet or data stream structure
  - Source coding method
  - Error correction method (eg CRC codes)
  - Packet control method
  - Many other details
  - Read IETF RFCs for examples

# Good ICD Practices

- Don't recreate the wheel explain existing well documented standard protocols (e.g. RS232)
- Use diagrams (e.g. timing diagrams) where appropriate
- Refrain from using complex diagrams, where multiple diagrams or textual description would do
- Use examples to clarify important concepts
  - Avoid excessive examples

# Good ICD Practices II

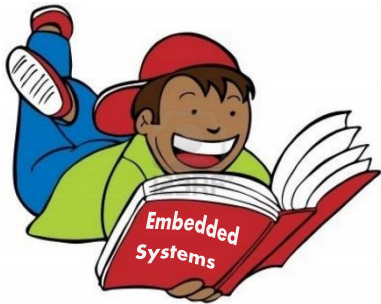
- Where existing standards are used, cite references
  - e.g. The IETF standards for internet protocols
- Indicate where further information can be found in related documents
- Make consistent use of units
  - e.g. Don't switch between cm and mm
- Clearly indicate timing information
- Provide a terminology/glossary to explain acronyms and unusual terms



# The Next Episode...

## Lecture L10

L10: Dataflow (part 2),  
Synchronous (or "static") data flow (SDF)



**Reminder:** Read section 2.5