CT420 REAL-TIME SYSTEMS

DESIGN CONSIDERATIONS REAL-TIME SAFETY-CRITICAL SYSTEMS

Dr. Michael Schukat

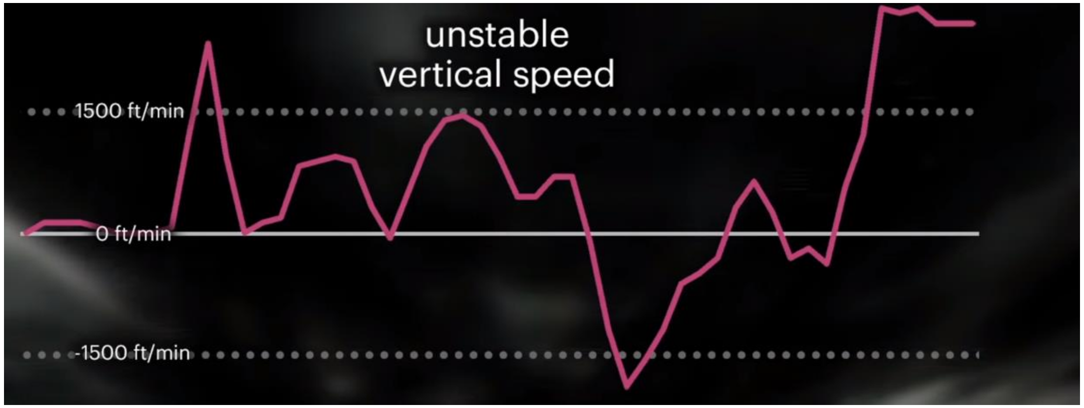
Ollscoil na Gaillimhe
University of Galway

Motivation for and Objectives of this Lecture

- So far, we have addressed the real-time aspects of RTSCS
  - How must software systems be designed to meet RT requirements?
  - What APIs are available (i.e., POSIX)?
- In this lecture we focus on features that increase system safety by increasing its reliability

Recall Case Study: The Boeing 737 Max 8 MCAS Problem

- On 29 October 2018, a just 2-months old Boeing 737 MAX 8 plane crashed into the Java Sea 12 minutes after takeoff, killing all 189 passengers and crew
- The plane's flight recorder showed the following vertical speed pattern before the crash:

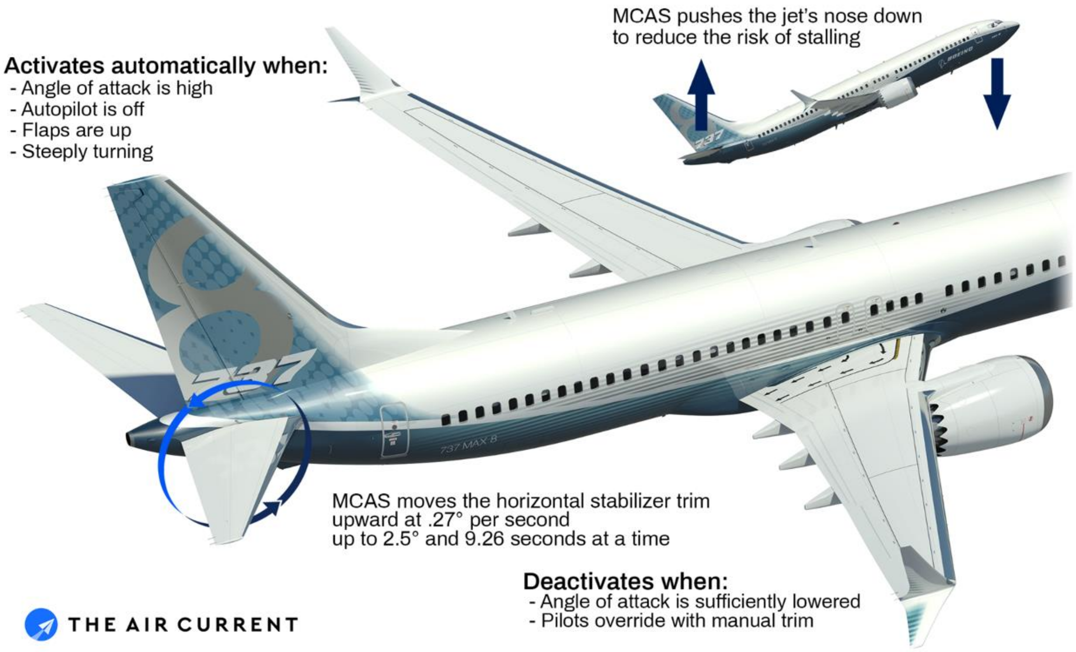


- Then on March 10, 2019, a 4-months old 737 MAX 8 crashed shortly after take-off from Addis Ababa, killing all 149 passengers and 8 crew members on board
  - Evidence retrieved later suggested that again the aircraft's vertical speed after take-off was unstable
- Shortly later, the entire fleet was grounded

The Boeing 737 Max 8

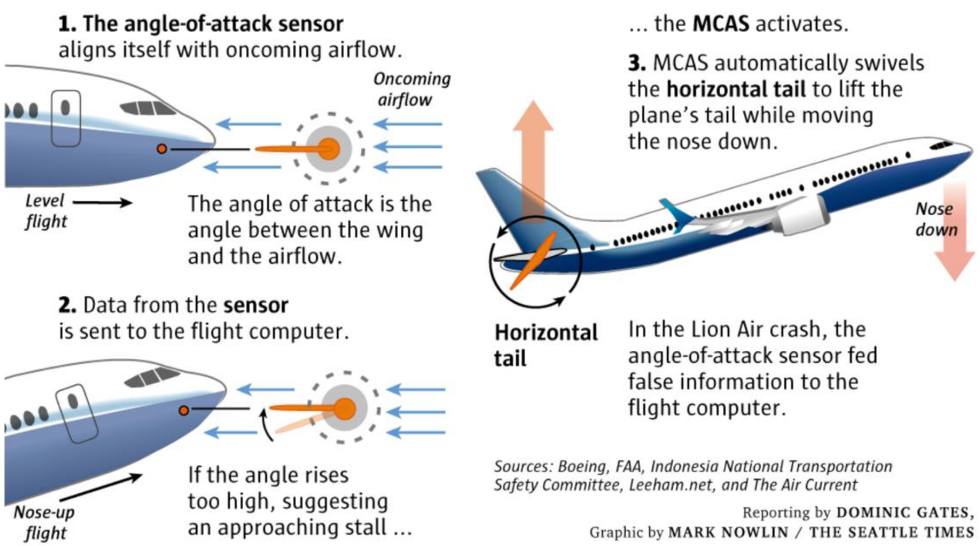
- The latest and most fuel-efficient version of the Boeing 737
  - The 737 series is the highest-selling commercial jetliner in history with more than 10,000 units built since 1967
  - The Max 8 has longer engines than previous models, which sit slightly forward and higher, therefore changing its centre of gravity, making it more likely to pitch upward on take-off

Boeing's Manoeuvring Characteristics Augmentation System (MCAS)



- Source: https://theaircurrent.com/aviation-safety/what-is-the-boeing-737-max-maneuvering-characteristics-augmentation-system-mcas-jt610/

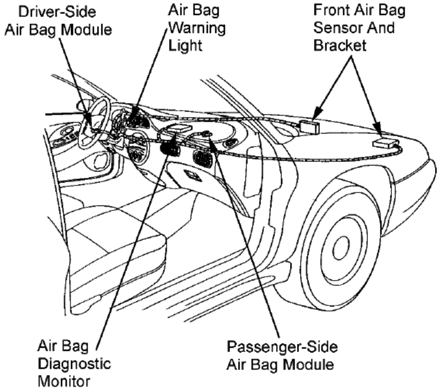
Boeing’s Manoeuvring Characteristics Augmentation System (MCAS)



- Source: https://www.seattletimes.com/business/boeing-aerospace/failed-certification-faa-missed-safety-issues-in-the-737-max-system-implicated-in-the-lion-air-crash/

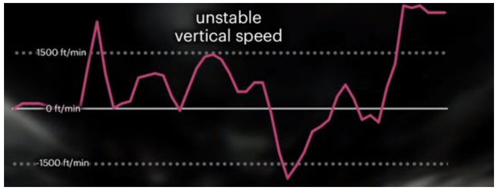
The Angle of Attack (AoA) Sensor

- In both crashes the incorrect airflow angles, reported from only a single (faulty) AoA sensor, were processed
- Using multiple AoA sensors would have allowed to compensate for this
  - Note that the plane had in fact 2 AoA sensors installed
- Compare to Airbag design we discussed before



Other Issues that led to the Crashes

1. MCAS should not be able to repeatedly overwrite pilot decisions



2. Pilots should have been trained how to manually disable the MCAS in-flight
http://www.spiegel.de/wissenschaft/technik/boeing-737-max-abstuerze-welche-rolle-spielten-die-piloten-a-1258835.html (German article)
3. The Lion Air machine did not have dashboard instruments to show both AoA readings, or to alert the pilots about a discrepancy
These were optional extras Lion Air did not want to pay for
http://www.spiegel.de/wissenschaft/technik/boeing-737-max-fehlten-sicherheitsfunktionen-weil-sie-extra-kosten-a-1259117.html (German article)

Recap: Quality Requirements for RTSCS

- RTSCS must be **time responsive**
- RTSCS must be **reliable**
  - The ability to behave in accordance with its specification
- RTSCS must be **safe**
  - Conditions that lead to hazards do not occur
- RTSCS must be **secure**
  - Protect itself against intentional or accidental access, use, modification or destruction
- RTSCS must be **usable**
  - Easy to learn, understand, and use
- RTSCS must be **maintainable**
  - Return swiftly to an operational state after receiving repairs or modification (e.g. plug in-and-forget)

Accident, Risk and Hazard

- **Accident** is a loss of some kind, such as injury, death, or equipment damage
- **Risk** is a combination of the likelihood of an accident p(a) and its severity s(a):
  - Often numerical models are used with p(a) being based on a probability distribution (i.e. $0 <= p(a) <= 1$), and s being or normed value (i.e. $0 <= s(a) <= 1$): risk = p(a) * s(a)
- **Hazard** is a set of conditions and/or *events* that leads to an accident

Faults and Hazards

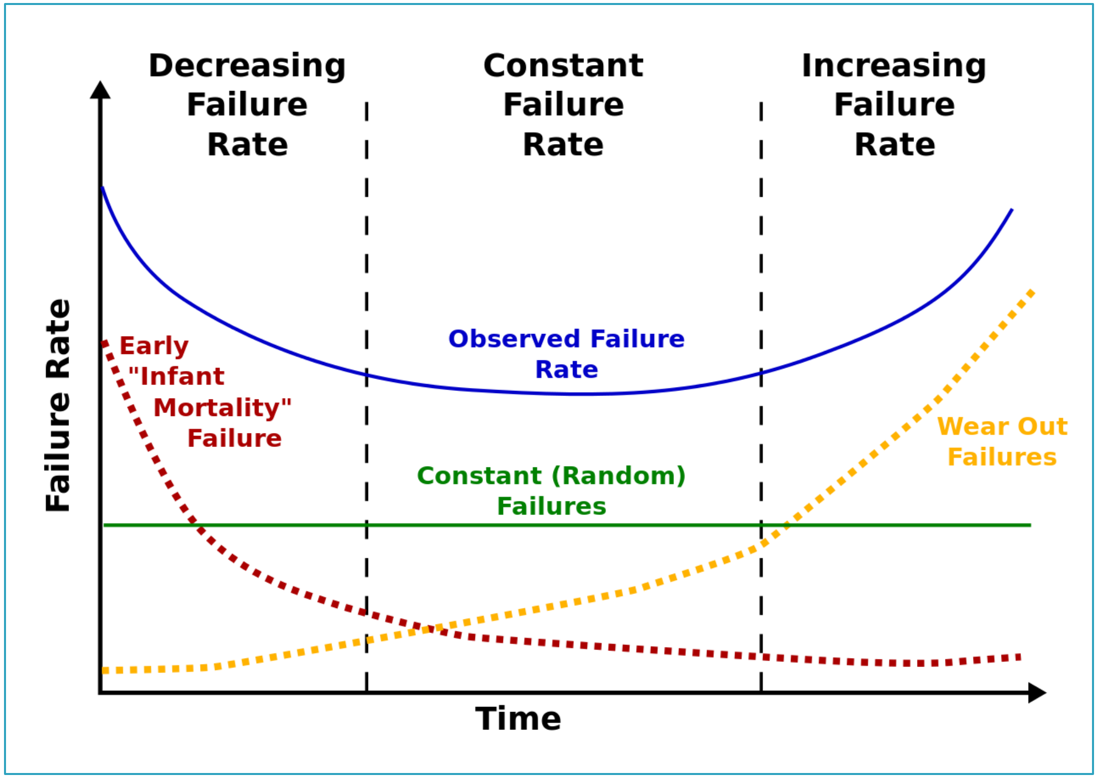
- Faults lead to hazards, which lead to accidents
- Faults are the manifestation of a:
  - Failure
    - Random non-performance of a component (e.g. wear and tear)
  - Error
    - Systematic; i.e. design fault or software fault (bug)
- Faults can be permanent, intermittent, or transient

Determining Failure Probability

- Assume a driver airbag system with N (= 5) independent components, each with a fault probability of 5% over 10 years of operation (i.e. 95% probability that component will function ok after 10 years)
- Overall system failure probability after 10 years (assuming statistical independence):
  - 1 – (Probability that ALL components ok)
  - $1 - (1 - 0.05)^5 = 1 - 0.773 = 0.227 = \sim 22.7\%$
- Components may include
  - Sensors, actuators, controllers
  - Their components, e.g. CPU, RAM, storage

The Bathtub Curve

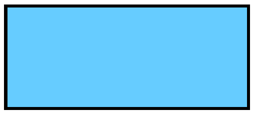
- The bathtub curve is a particular shape of a failure rate graph
  - Failure rate is the frequency with which an engineered system or component fails over time
- Component failures can be pre-empted via
  - Component redundancy
  - Scheduled component replacement



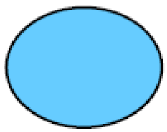
Fault Tree Analysis (FTA)

- FTA is a top-down (from hazard / event to basic fault type), deductive failure analysis in which an undesired state of a system is analysed using Boolean logic to combine a series of lower-level events
- This analysis method is mainly used in safety engineering and reliability engineering to understand how systems can fail, to identify the best ways to reduce risk and to determine (or get a feeling for) event rates of a safety accident or a particular system level (functional) failure

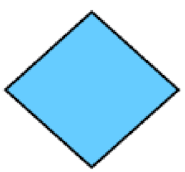
Fault Tree Analysis Symbology



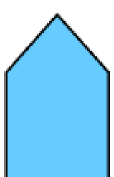
An event that results from a combination of events through a logic gate



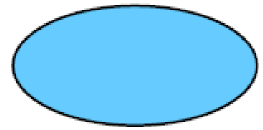
A basic fault event that requires no further development



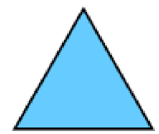
A fault event because the event is inconsequential or the necesary information is not available



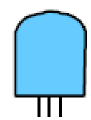
An event that is expected to occur normally



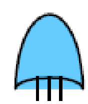
A condition that must be present to produce the output of a gate



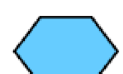
Transfer



AND gate

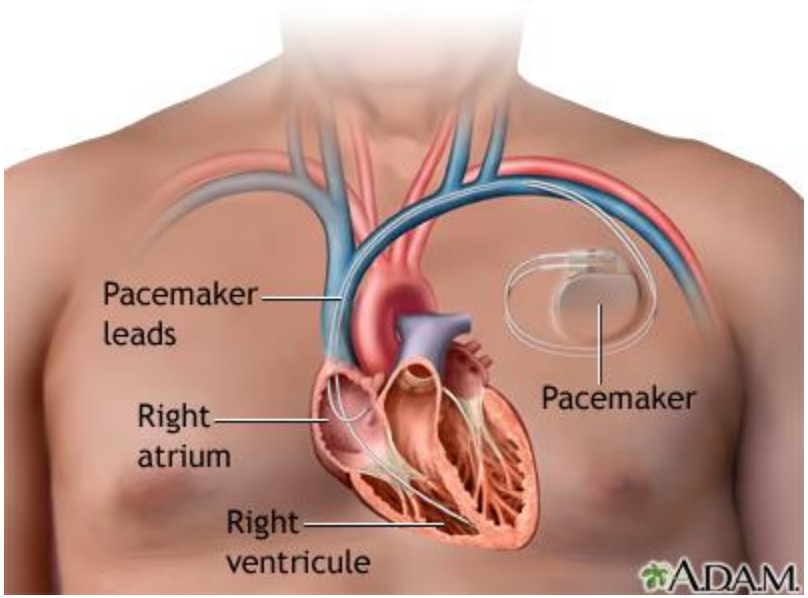


OR Gate

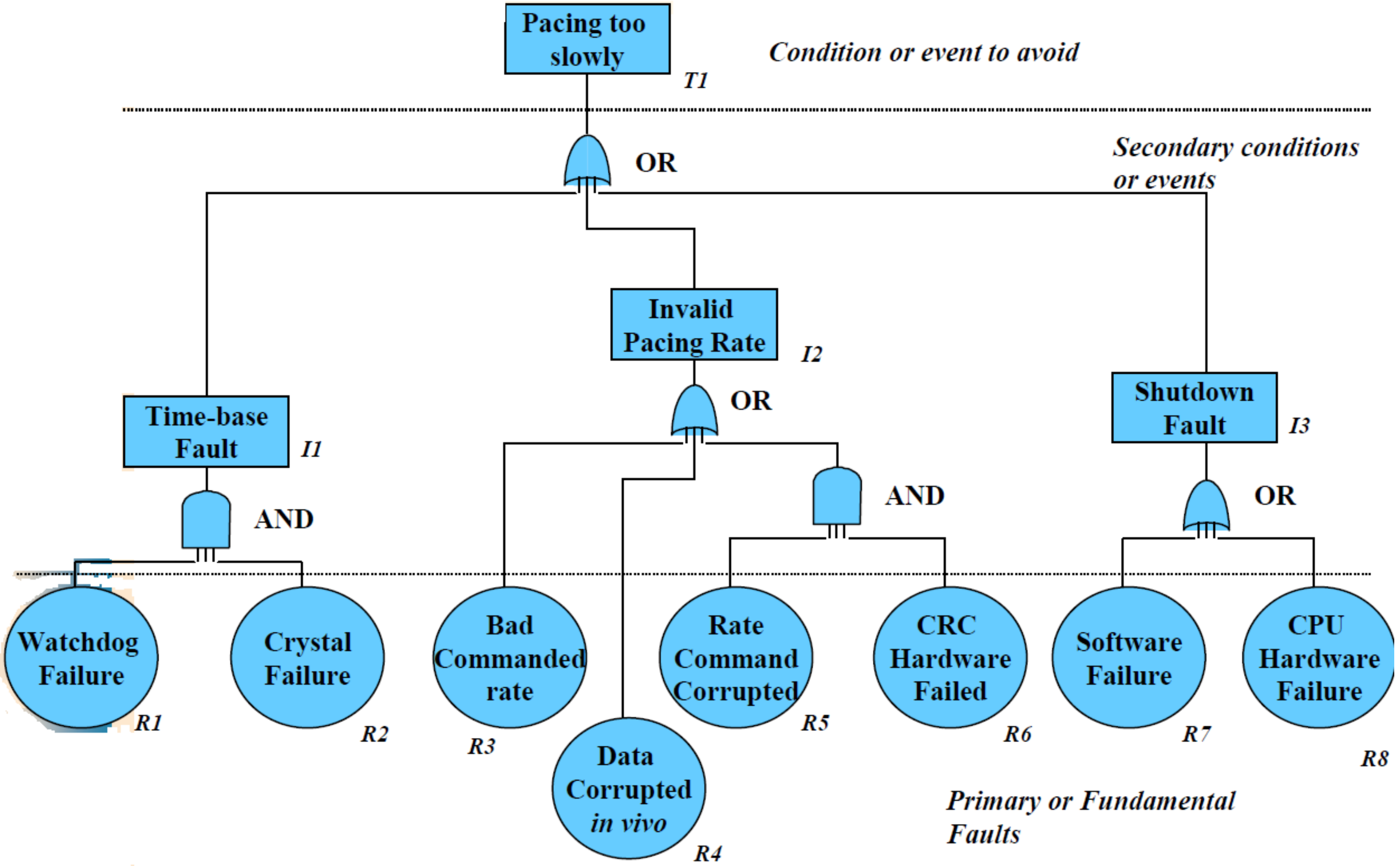


NOT Gate

Example Pacemaker



Example Pacemaker (Subset) Fault Analysis

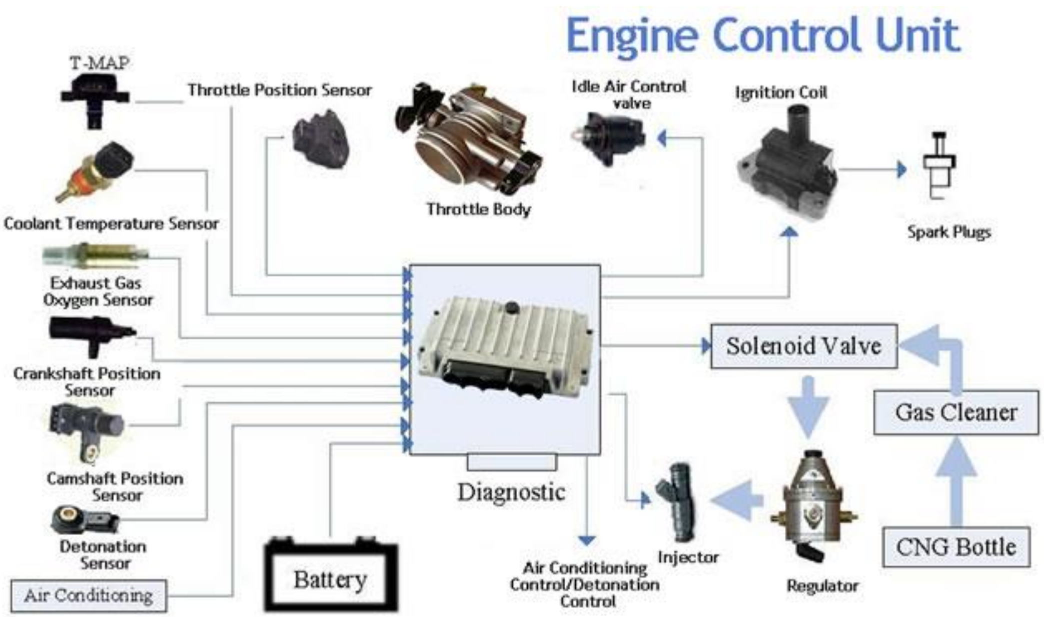


Making RTSCS safe: Fail-Safe

- Fail-safe describes a device or feature which, in the event of failure, responds in a way that will cause no harm or at least a minimum of harm to other devices or danger to personnel
- Examples:
  - Air brakes on railway trains and air brakes on trucks
  - Luggage carts in airports
  - Lawnmowers

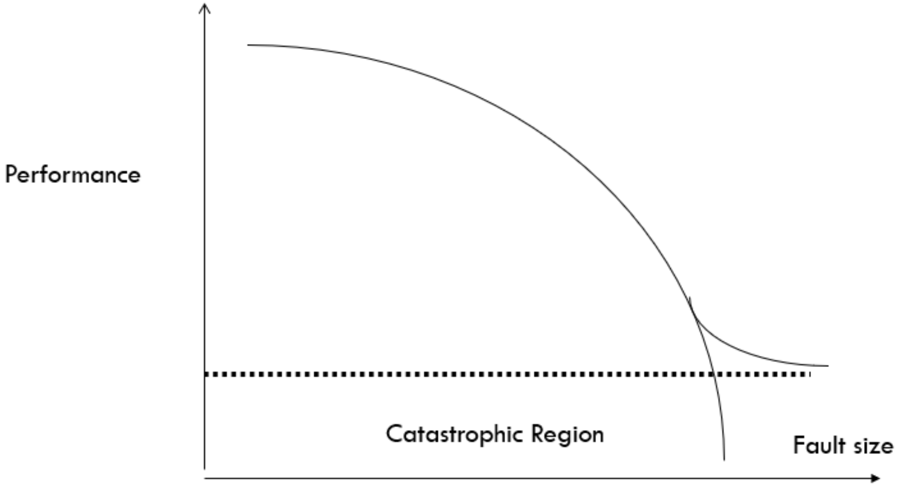
Making RTSCS safe: Fail-Soft

- Pertaining to or noting facilities built into a system, as in an automobile or a computer, for continuing operations on an <u>interim basis</u> and probably with reduced efficiency, if parts of the system fail
- Example: Fail-Soft of ECU via “Limp Mode”



Making RTSCS safe: Graceful Degradation

- As size of faulty set increases, system **must** not suddenly collapse, but must gracefully degrade
- Failures will eventually impact on (RTS) operation
- System perhaps operates with reduced functionality
- Avoid catastrophic failure



Example of graceful Degradation: The Citroen CX

- Common hydraulic system for steering, brakes and suspension
- What goes first, second and last when hydraulic pressure drops?

M

Fault Types

- Permanent
  - Easiest to detect
  - Hardware failure or software design/code fault
- Intermittent
  - Fault appears from time to time
  - Loose wire, poor contacts, certain sequence of events
- Transient
  - Fault appears but dies away with time
  - α particle impact: non-destructive to memory

Transient Soft Errors due to Neutron Strikes or α Particle Impacts

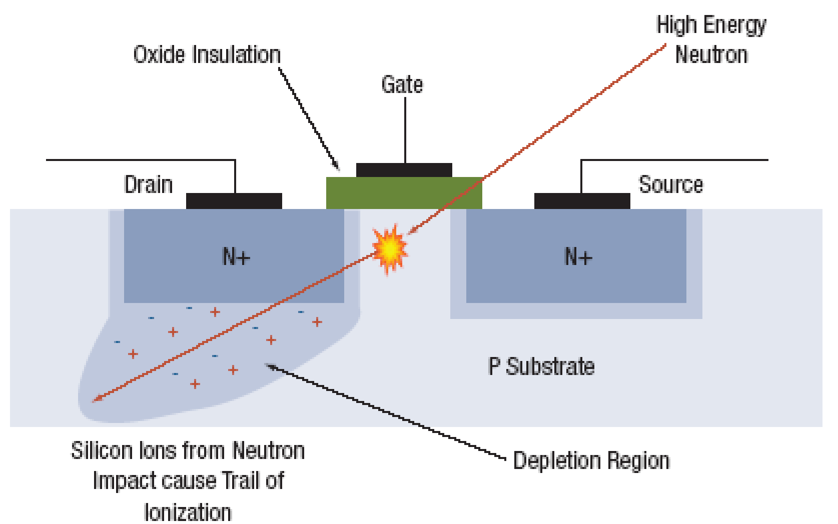
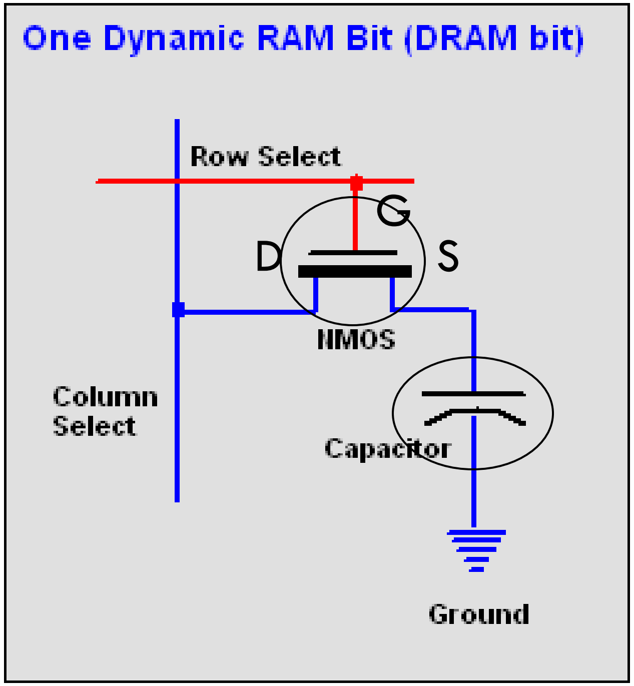


Figure 1

Designers of space systems must consider immunity to single-event upsets (SEUs) caused by neutron strikes. When neutrons strike silicon atoms, heavy ions are ejected, which cause momentary current pulses in CMOS ICs. Although the ICs' memory cells and flip-flops are not damaged, the data in them can change, causing "soft" bit errors.

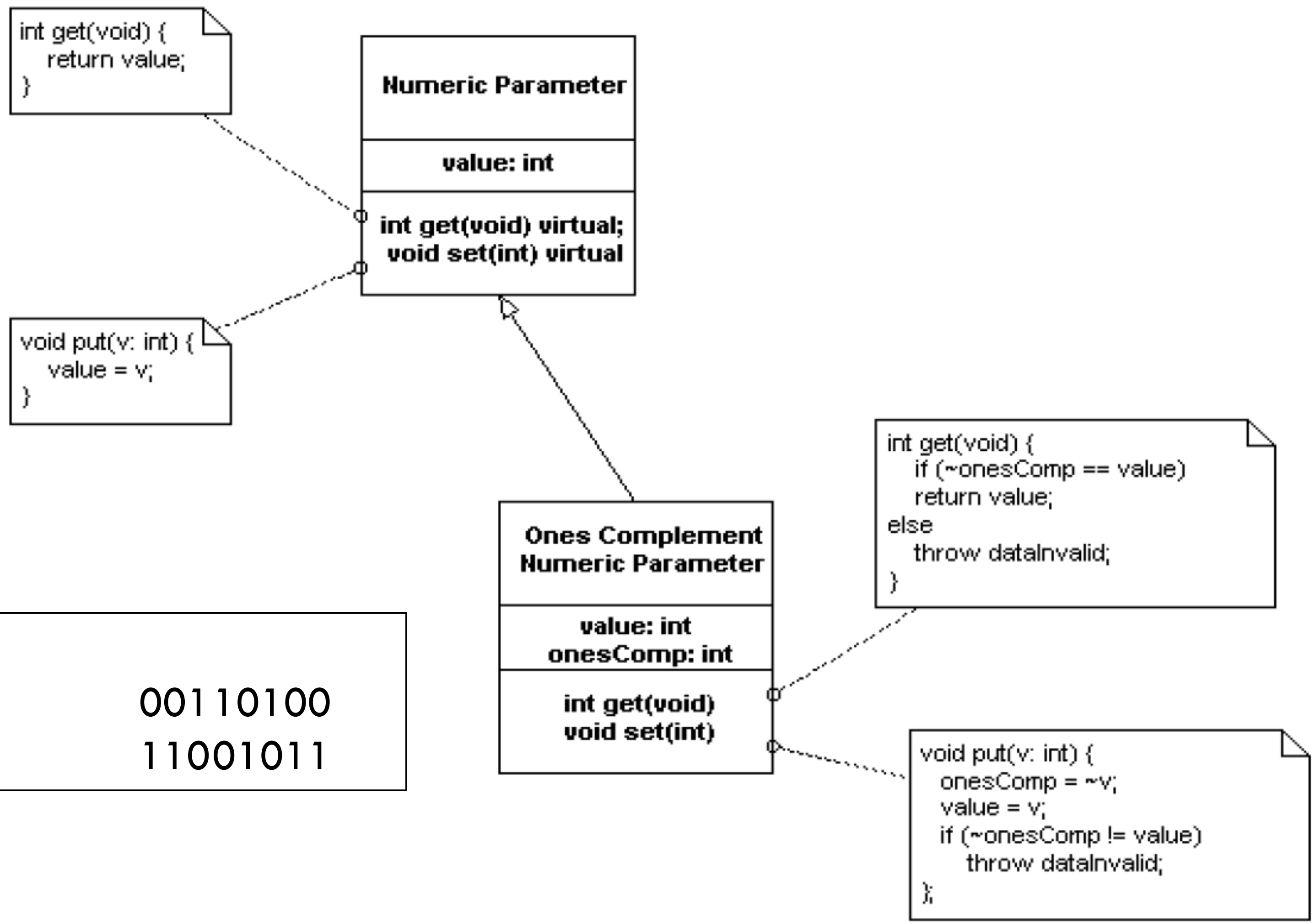


- The capacitor of a DRAM cells stores a single bit (0 = no charge, 1 = charged), while high DRAM integration density result in very small capacitors
- An α particle penetrating a DRAM cell may have sufficient kinetic energy to distort the capacitor charge and changes its state (chip-level soft error)
- α particles can be emitted from radioactively contaminated packaging
- As a result, high-end servers use DRAM with error detection/correction capabilities (→ **information redundancy**)

Information Redundancy

- Data Validity Checks of data at rest (i.e., in RAM, or in secondary storage) can be achieved via:
  - Cyclic Redundancy Check (CRC)
    - Here blocks of data get a 16-bit or 32-bit check value attached, based on the remainder of a polynomial division of their contents
    - This identifies
      - all single or dual bit errors
      - a high percentage of multiple bit errors
    - Widely used in data communication (e.g, TCP/IP)
  - One's complement
  - Error correcting codes
  - RAID
- Redundancy should be set every write access
- Data should be checked every read access

One’s Complement Data Validity Check



Example:	
value:	00110100
onesComp:	11001011

Error Correcting Codes (ECC)

- ECC is redundant information added to the data / message
- This allows detecting and correcting a limited number of errors that may occur anywhere in the message
- The American mathematician Richard Hamming pioneered this field in the 1940s and invented the first error-correcting code in 1950: the Hamming (7,4) code

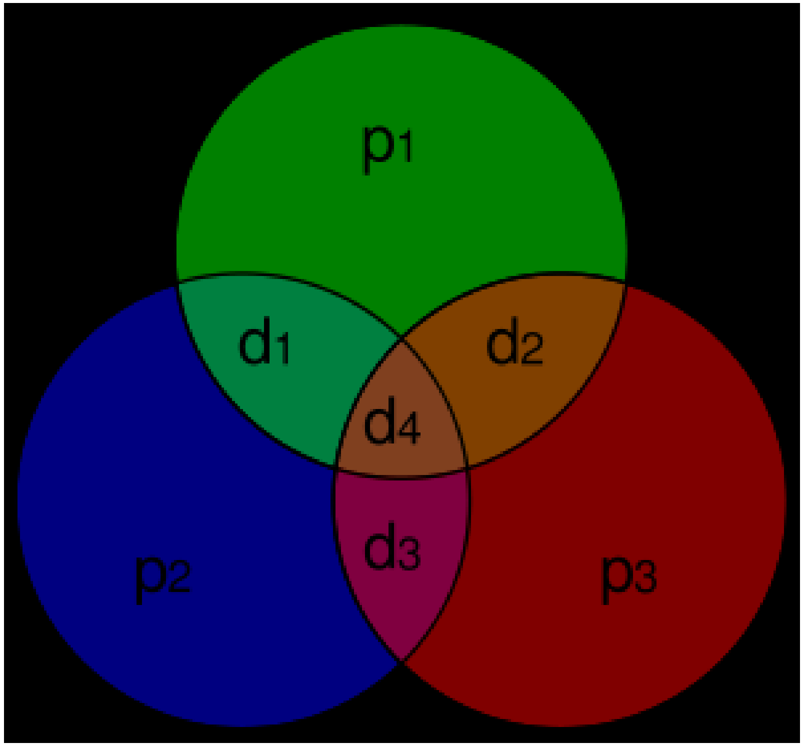
Hamming Distance

- Minimum number of bits to be toggled to convert one codeword into another
  - HD (ASCII) = 1
  - HD (ASCII + Parity Bit) = 2
    - even parity: Total number of "1"s is even number; odd parity: ditto
- A code with a given HD x may be able to
  - detect (x – 1) bit errors
  - correct (1 <= y < x / 2) bit errors
- Examples:
  - ASCII
    - "@"=64_{10} = 01000000
    - "A"=65_{10} = 01000001
    - Hamming Distance 1
  - ASCII + even parity
    - "@"=64_{10} = 01000000 1
    - "A"=65_{10} = 01000001 0
    - Hamming Distance 2

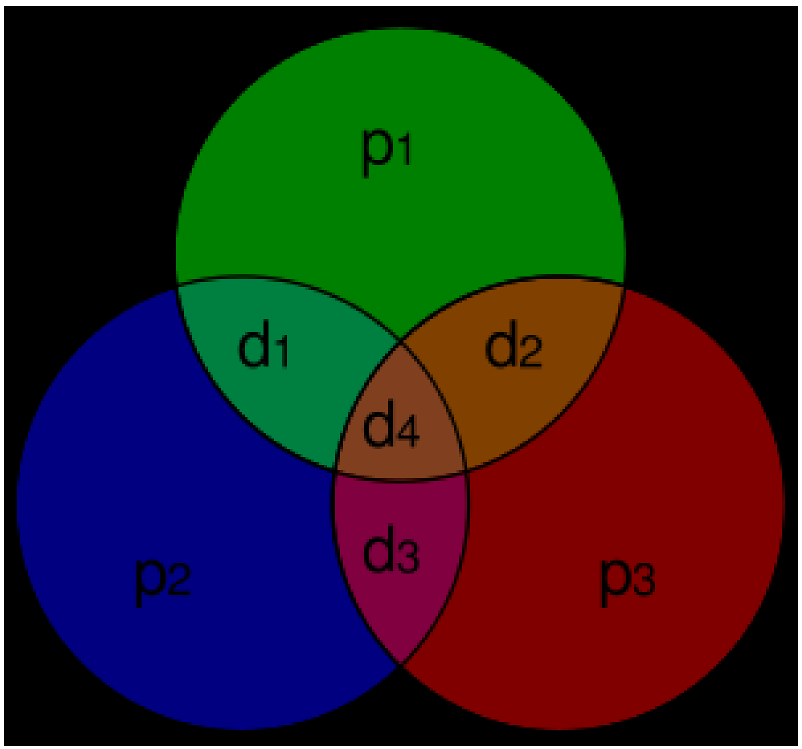
Hamming (7,4) Code

- Hamming (7,4) is an error-correcting code that encodes four bits of data into seven bits by adding three (even or odd) parity bits

Bit #	1	2	3	4	5	6	7
Transmitted bit	p_1	p_2	d_1	p_3	d_2	d_3	d_4
p_1	Yes	No	Yes	No	Yes	No	Yes
p_2	No	Yes	Yes	No	No	Yes	Yes
p_3	No	No	No	Yes	Yes	Yes	Yes



Example: Hamming (7, 4) Code with even Parity



d1	d2	d3	d4	p1	p2	p3
1	1	0	1	1	0	0

Start

d1	d2	d3	d4	p1	p2	p3
0	1	0	1	1	0	0

Correction of single data bit error

d1	d2	d3	d4	p1	p2	p3
1	1	0	1	1	0	1

Correction of single parity bit error

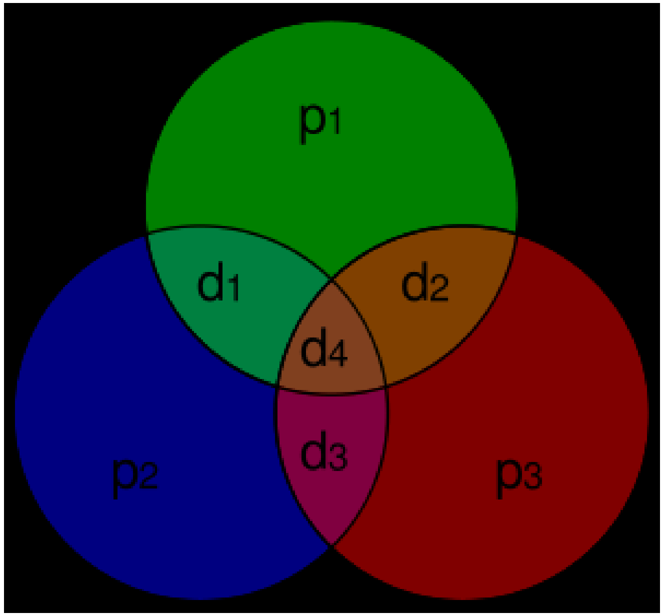
d1	d2	d3	d4	p1	p2	p3
0	1	0	0	1	0	**0**

Detection of double data bit error

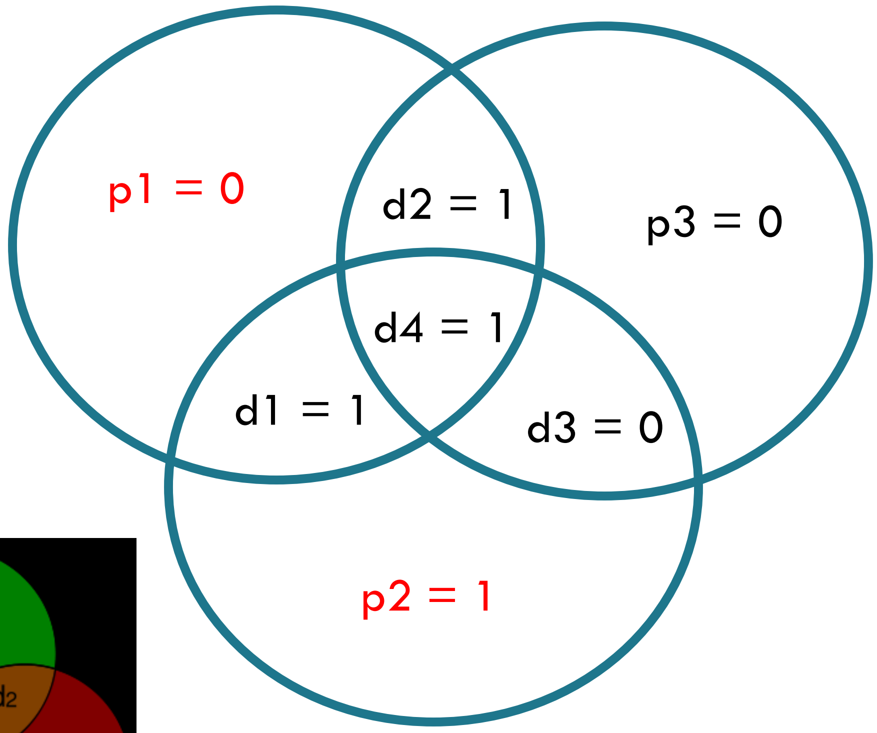
Hamming(7,4) has a HD of 3!

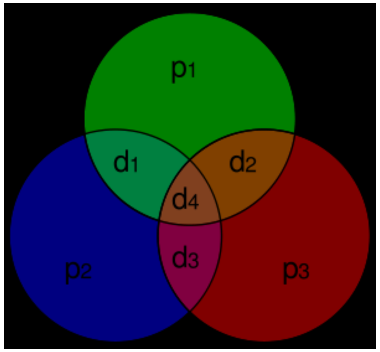
Problem

- Consider a Hamming (7, 4) code with even parity
- Does the Bitvector “1101: 010” (d1 - d4 : p1 - p3) indicate a bit-error? If yes, which bit got flipped?
- Draw a Venn Diagram to figure it out…



Hamming(7, 4) Problem Solution





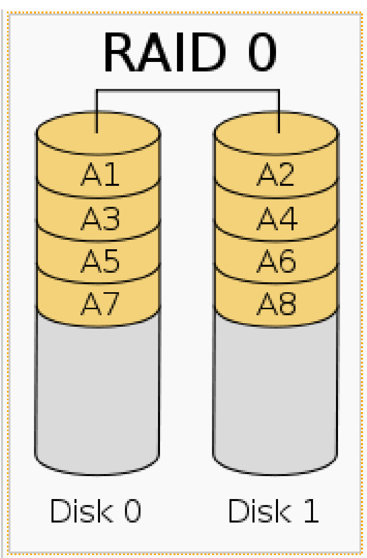
Mass-Storage Redundancy via RAID

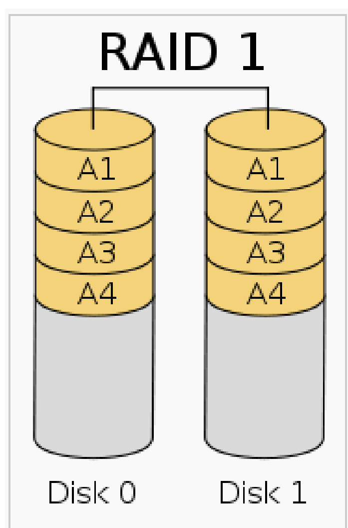
- Redundant Array of Independent Disks (RAID) is a data storage virtualisation technology that combines multiple physical disk drive components into one or more logical units for the purposes of **data redundancy** and / or **performance improvement**

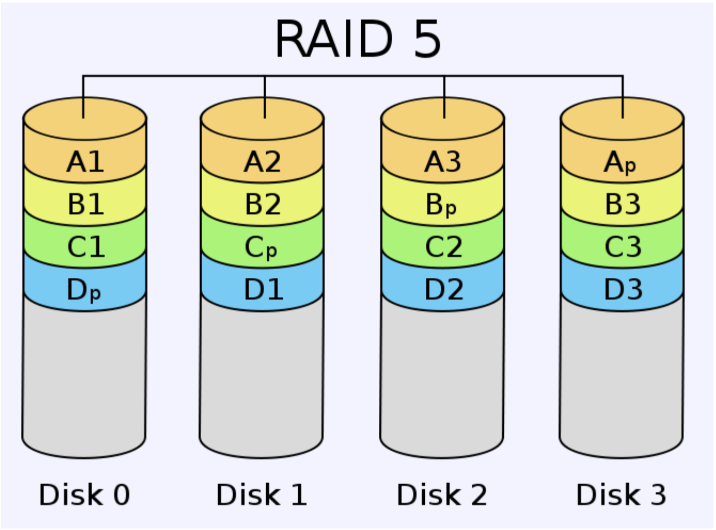
- **Data blocks** are distributed across the drives in one of severa as RAID levels, depending on the required level of redundancy and performance
- Many RAID levels use a parity-based error protection scheme (see RAID-4), example (with 12 bit / block):
  - Block 1: 010001101001
  - Block 2: 110011011010
  - Block 3: 000100100101
  - P Block: 100111010110 (bitwise EXOR, equivalent to even parity)

Mass-Storage Redundancy via RAID

- RAID storage systems require a dedicated RAID controller, that supports the required RAID level
  - See also the diagram on the next slide
  - Normally such controllers are not shown in RAID diagrams

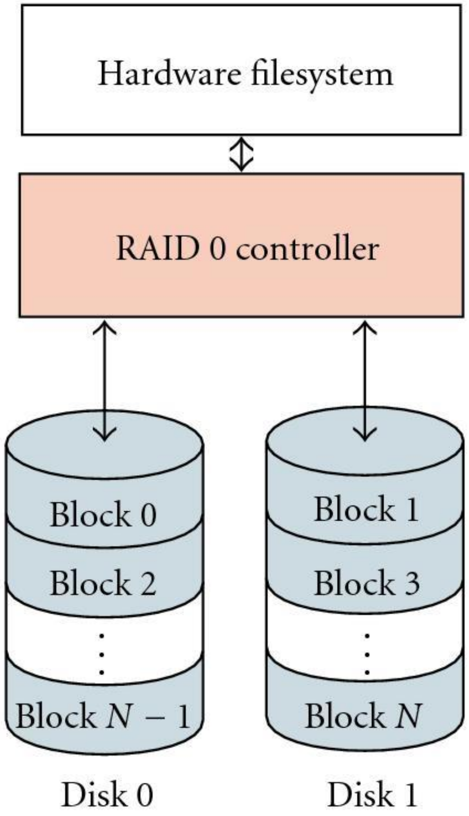






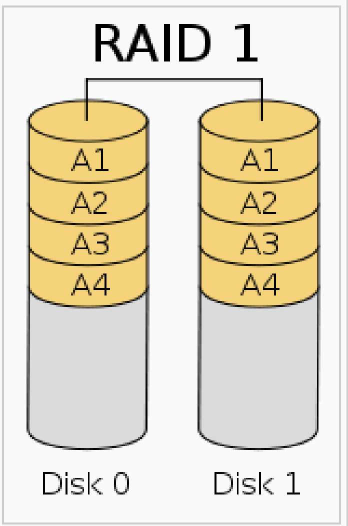
RAID 0

- Block-level striping without parity or mirroring
  - **data striping** is the technique of segmenting logically sequential data, such as a file, so that consecutive segments are stored on different physical storage devices
- 2 or more drives (n) required
- No redundancy, but up to n-times R/W performance increase

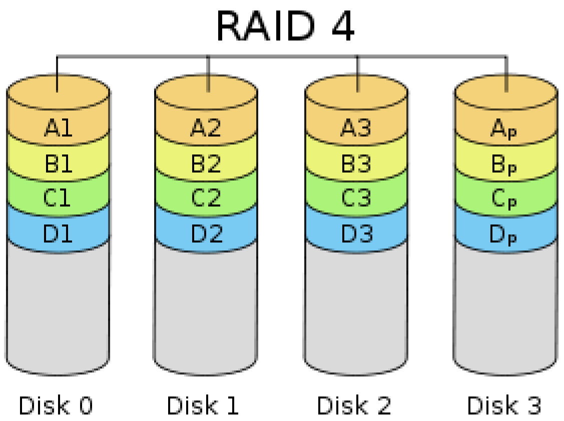


RAID 1

- Block-level mirroring without parity or striping
- 2 or more drives (n) required
- (n – 1) drive failures can be compensated; here each disk can
  - diagnose catastrophic failures (e.g. head crash)
  - detect (but **not** correct) sector-wise bit errors on platters
- No increase in R/W performance



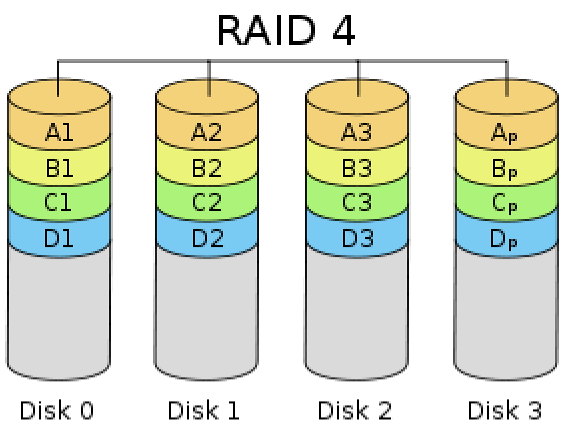
RAID 4



- Block-level striping with single parity disk
- Single catastrophic drive failure can be compensated (<u>any</u> drive can fail)
- RAID 4 provides good performance of random reads, while the performance of random writes is low due to the need to write all parity data to a single disk (Disk 3 in the diagram above)
- Minimum of 3 drives required

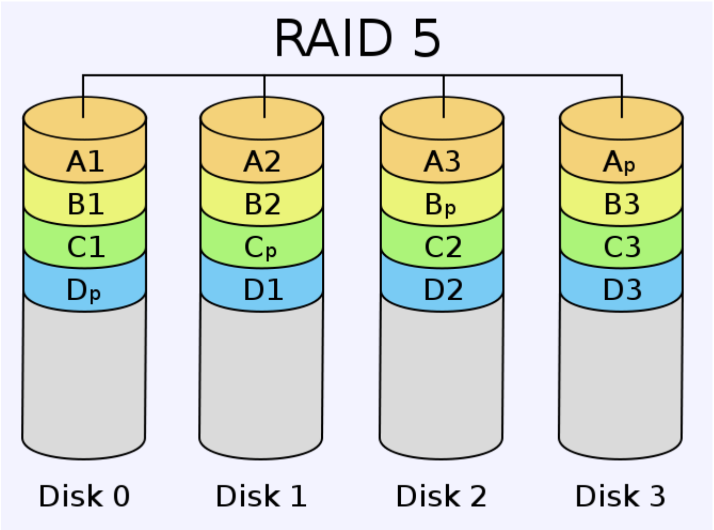
Drive Hot-Swapping in RAID

- In RAID a defect drive will be (ASAP)
  - manually swapped for a new drive (hot-swap), or
  - replaced by an idle drive (hot-spare) already in the system
- The new drive’s content is rebuild by the RAID controller while the disk set is still operational
- Example RAID 4 with Disk 0 swapped:
  - $A1 = A2 \text{ EXOR } A3 \text{ EXOR } A_P$
  - $B1 = B2 \text{ EXOR } B3 \text{ EXOR } B_P$
  - $C1 = C2 \text{ EXOR } C3 \text{ EXOR } C_P$
  - $D1 = D2 \text{ EXOR } D3 \text{ EXOR } D_P$



RAID 5

- Similar to RAID 4, but:
  - Block-level striping with distributed parity
  - Distributed parity evens out the stress of a dedicated parity disk among all RAID members
  - Write performance is increased since all RAID members participate in the serving of write requests
- Minimum of 3 drives required

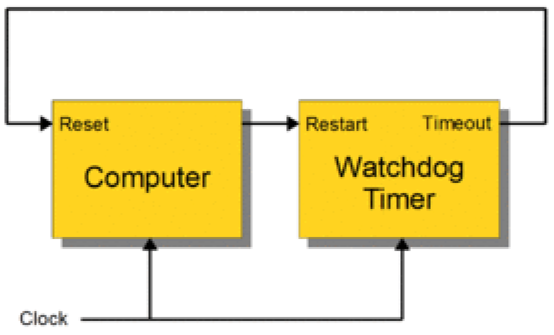


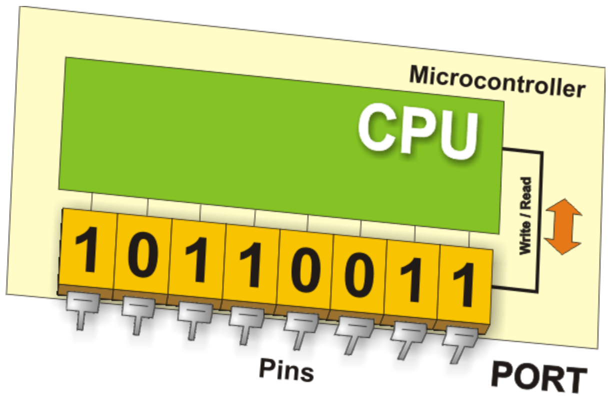
Increasing Hardware Reliability

- Information redundancy protects against memory-related faults
- However, it assumes that the underlying computer system works satisfactorily
- Therefore, we need to determine ways for a system's
  - Fault detection
    - → Watchdog
  - Fault recovery
    - → Failover
    - → Redundancy

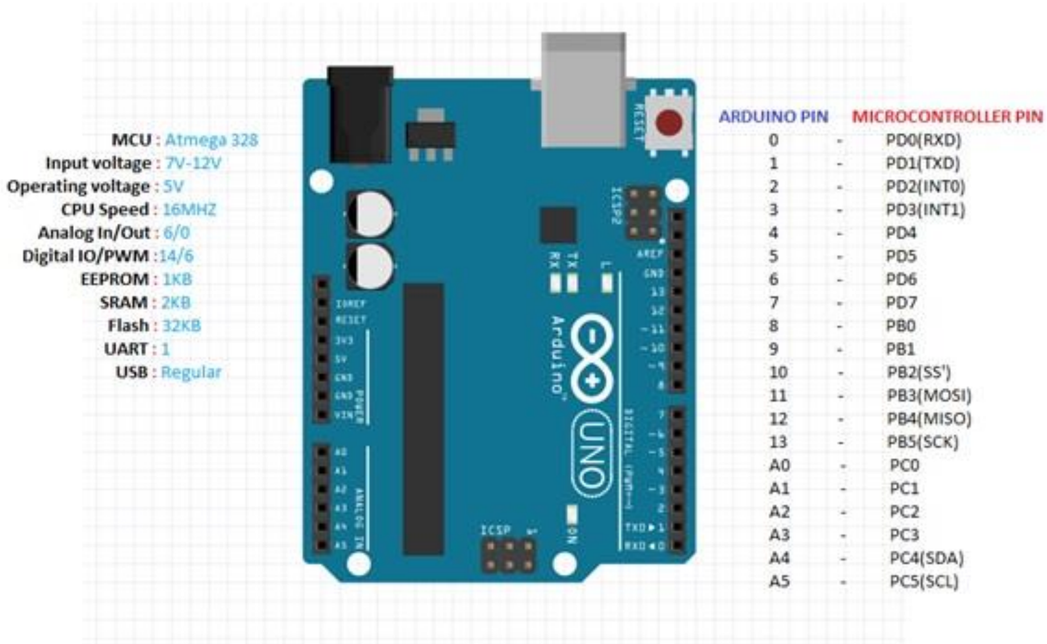
Watchdog

- Idea: Restart can be a fail-safe state!
- Here the computer / CPU is reset by the watchdog, unless it is regularly (typically every 1ms to every 10s) triggered by the CPU, for example using a PIO
- Requires validation that the system is in safe state during reset
- Watchdog can be internal or external (as shown) component





Example for internal Watchdog (Arduino Uno)



#define wdt_reset(); //resets WDT

#define wdt_disable(); //disables WDT

#define wdt_enable(timeout); //sets the watchdog pre-scaler, using one of the constants below:

#define WDTO_15MS 0

#define WDTO_30MS 1

#define WDTO_60MS 2

#define WDTO_120MS 3

#define WDTO_250MS 4

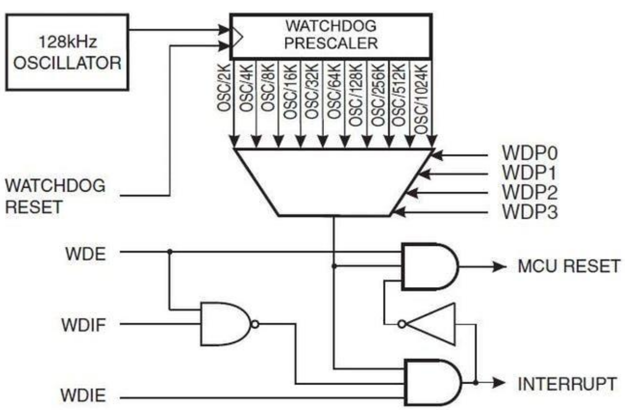
#define WDTO_500MS 5

#define WDTO_1S 6

#define WDTO_2S 7

#define WDTO_4S 8

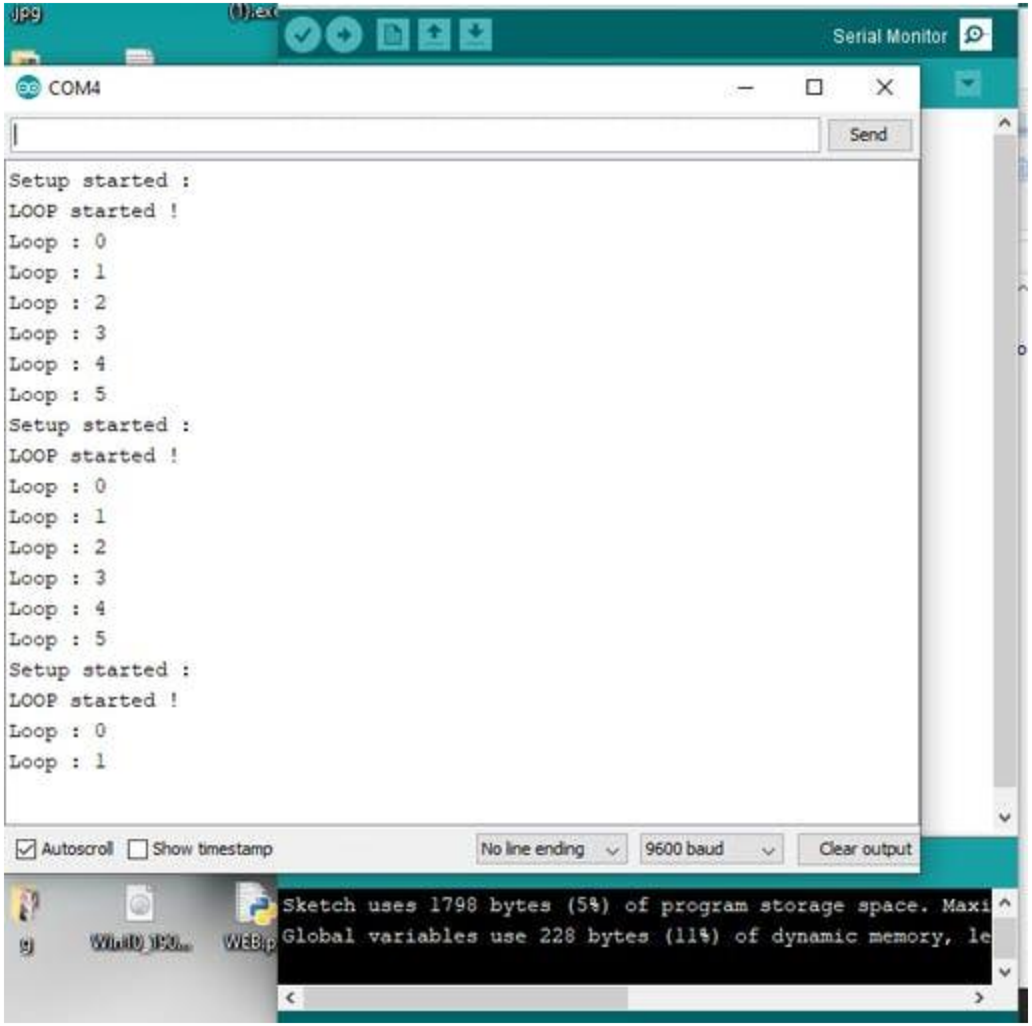
#define WDTO_8S 9



Example for internal Watchdog (Arduino Uno)

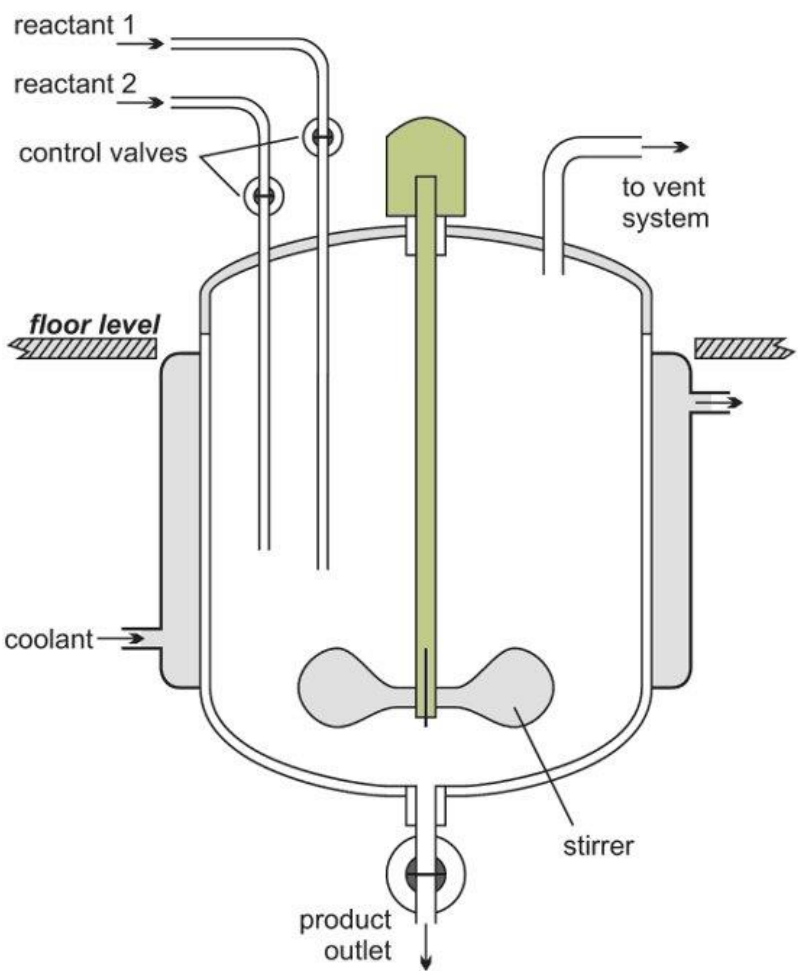

```
#include <avr/wdt.h>
void setup(){
    Serial.begin(9600);
    Serial.println("Setup started :");
    delay(2000);
    wdt_enable(WDTO_4S);
}
void loop(){
    Serial.println("LOOP started ! ");
    for(int i=0; i<=5; i++){
        Serial.print("Loop : ");
        Serial.print(i);
        Serial.println();
        delay(1000); }
        wdt_reset();

//infinity loop to hang MCU
while(1){}
}
```



Watchdog Coding Challenge

- Consider the Arduino controls a chemical reactor in a factory, where it is continuously operating. A watchdog monitors the Arduino
  - The control code is executed in loop(){}
- Every 3 months the reactor is turned off and serviced
- Here technicians need to establish, how many watchdog resets occurred since the last service
  - A watchdog reset occurs if the code within a single loop iteration is not completed in time
- This value is stored in some non-volatile (Flash) memory
- This memory can store 32 integer values, and can be accessed via
  - int flash_read(int address) // address is between 0 and 31
  - flash_write(int address, int value) // ditto
- The Flash memory is reset (all 0s) when the system is deployed first and after each service
- Use as much as needed for your solution
- Hint: Use both the setup() and loop() routine



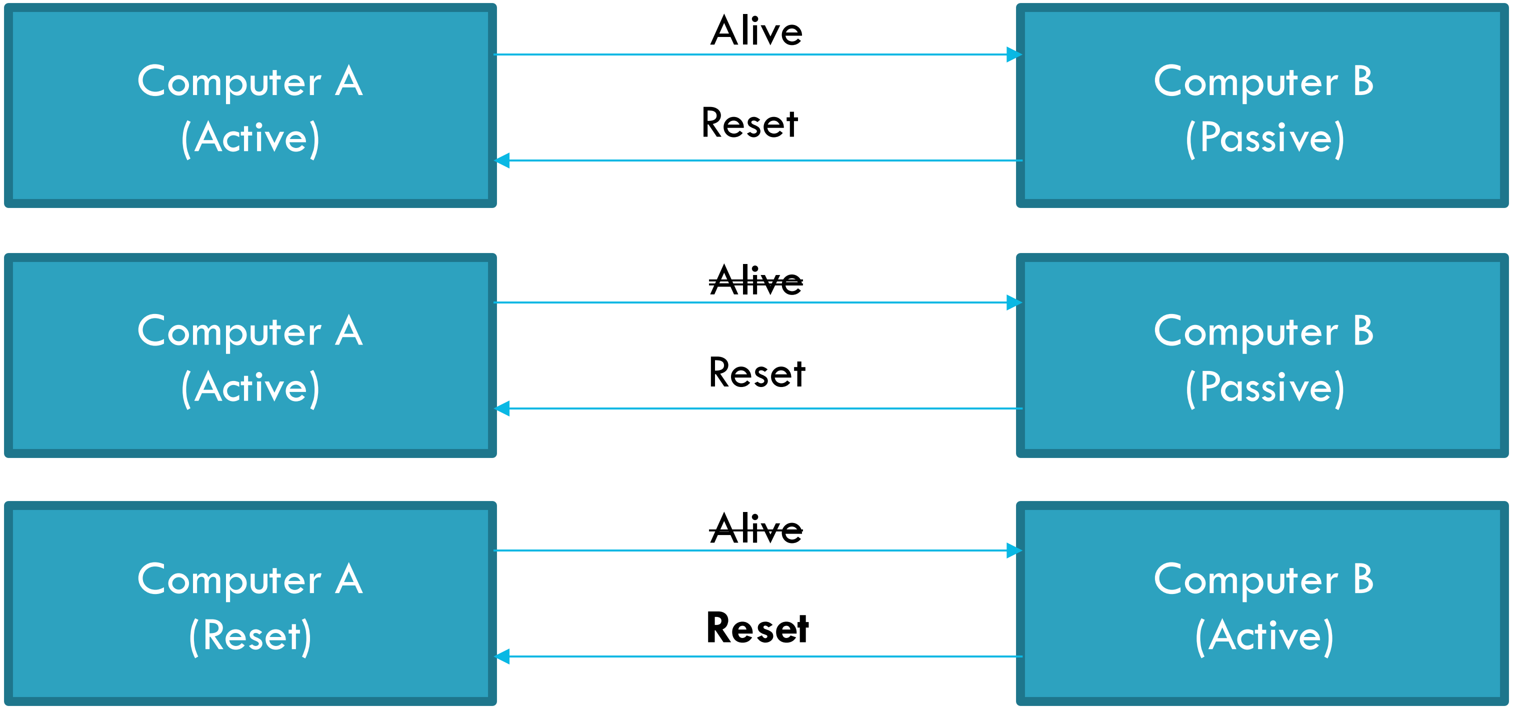
My Solution


```
loop() {
  flash_write(0, 1); // write a "1" to Flash cell #0
  // Execute loop code
  ...
  flash_write(0, 0); // write a "0" to Flash cell #0
  wdt_reset();
}

setup() {
  if (flash_read(0) == 1) {// WDT reset occurred
    flash_write(1, flash_read(1) + 1); // increment counter
    flash_write(0, 0) // Reset flag
   ...
```

Flash Memory Cell #0	Flash Memory Cell #1
1 == WDT reset; 0 == Ok	#of WDT resets

Master-Slave Fail Over

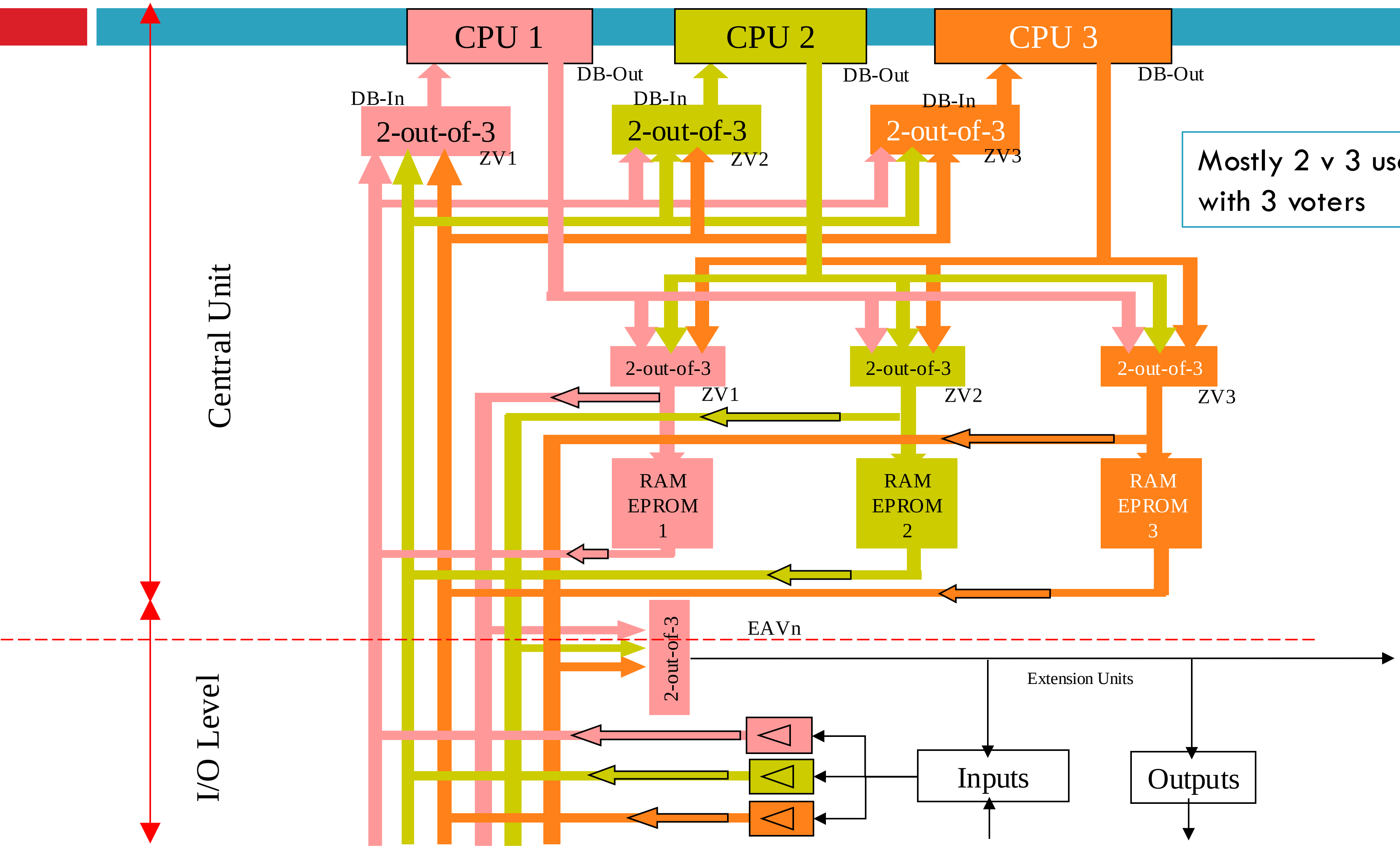


- A watchdog reset may disable a single controller for too long, therefore a second controller in stand-by mode may take over instead
- Here the equivalent of a watchdog reset pulse (the Alive signal) send by the active computer A is monitored by the passive computer B
- If computer A fails to provide this pulse in time, a timeout will occur, causing computer B to take over, while keeping computer A in a reset state

Redundancy via Synchronously Operating and Clocked independent Computers

- Example Moneypoint's Burner Management System Siemens AS220
  - Complex control system that manages power plant
- Triple-redundant hardware
  - RAM, ROM, CPU
  - CPUs run in synchronously
- 2-out-of-3 voters are used to deal with single faulty component

AS220 EHF Mode Of Operation

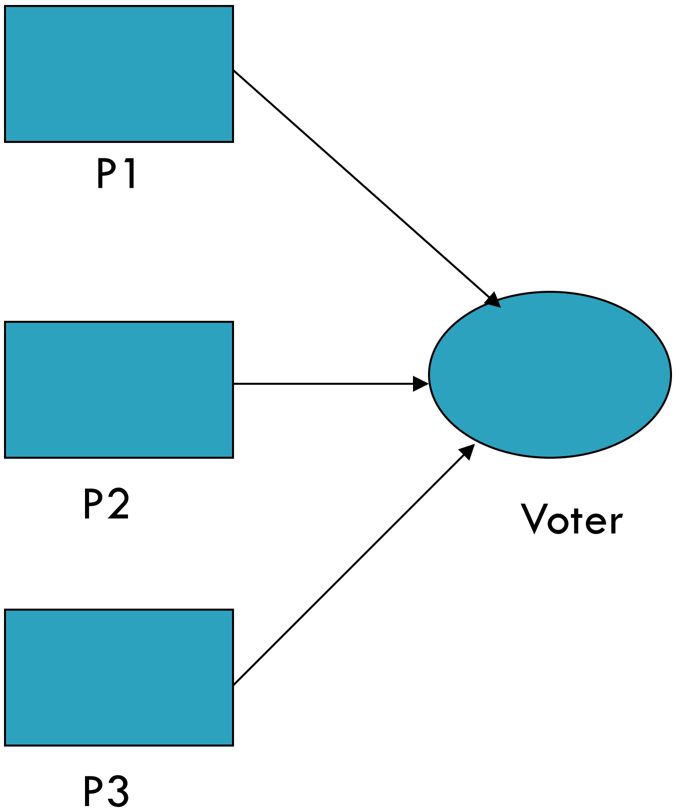


Mostly 2 v 3 used with 3 voters

Increasing Sensor Reliability

- The MCAS example showed that malfunctioning system sensors can lead to hazards and accidents
- Therefore, multiple redundant sensors must be put in place
- The (MCAS) computer detects the sensor reading discrepancy
- But…
  - What degree of difference indicates a faulty unit?
    - This has very much to be decided on a case-by-case basis
  - How can one identify the faulty unit in the first place?

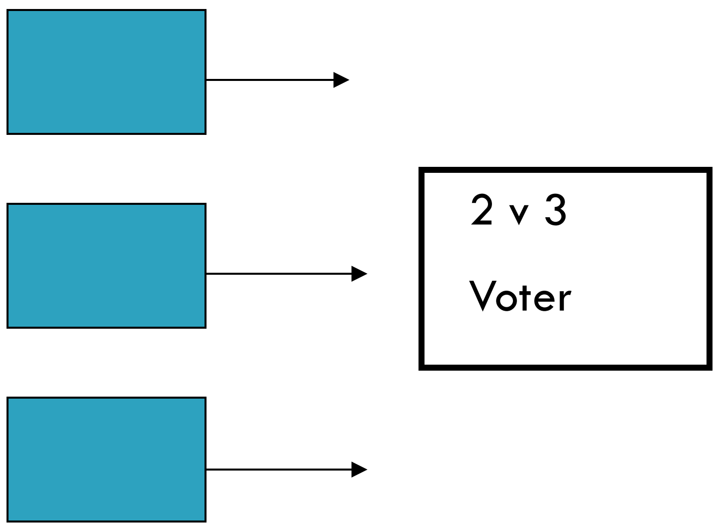
NMR: N-Modular Redundancy (Single Voter)



- Simple Design based on identical sensors P1, P2 and P3
- The voter determines if a sensor is faulty, i.e. returns incorrect readings
- Good for dealing with random faults only
- Only feasible if voter has a much lower failure probability than P1, P2 and P3

Example NMR (Single Voter)

- N identical independent components, each failure probability of 5%
- 2v3 System → 2 units need to fail for a complete system failure
- Overall system reliability: $1-(0.05)^2 = 99.75\%$
- Compared to 95% system reliability without redundancy
- This of course assumes that the voter failure probability is negatable small
- How can such a voter be implemented?



Voter Types

- All voter types try to determine a correct sensor reading
- The top 3 voters may be used to identify faulty units
- Formalised majority voter (FMV)
  - All inputs are equal, selects **absolute** (> 50%) majority
- Generalised median voter (GMV)
  - All inputs are equal, selects the median of the values
- Formalised plurality voter (FPV)
  - All inputs are equal, partitions the set of inputs based on metric equality and selects the output from the largest group, i.e. picks most common value
- Weighted averaging (WA)
  - Combines the outputs in a weighted average (mean)

1, 3, 3, **6**, 7, 8, 9

Median = 6

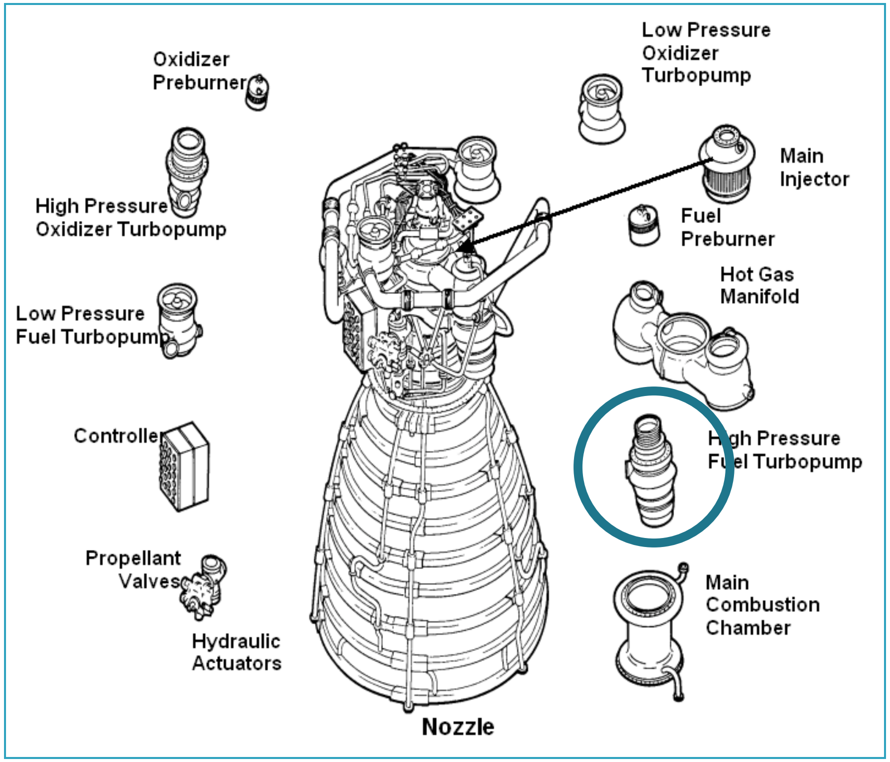
1, 2, 3, **4**, **5**, 6, 8, 9

Median = $(4 + 5) \div 2$

= 4.5

Case Study

- Consider a rocket engine as shown in the diagram
- Its high-pressure fuel turbopump has to operate within a certain fuel pressure range that is constantly monitored by a range of pressure sensors (not shown)
- Because of the extreme environment the sensor readings have a significant error, so the readings or multiple sensors are processed using a voter
- The diagram on the next slide shows a set of readings (in Megapascal) provided by 5 pressure sensors
- The sensors have a different weight depending on their perceived reliability



Calculate FMV, GMV, FPV and WA Output for the Data below

Tag	Input 1	Input 2	Input 3	Input 4	Input 5
Weight	0.2	0.3	0.1	0.3	0.1
Val	10	11	12	11	8
FMV					
GMV					
FPV					
WA					

- Calculate **FMV, GMV, FPV and WA** output for the data above
- Do not round results, i.e. **return decimal values where appropriate**
- Note that the weight row adds up to 1.0

Calculate FMV, GMV, FPV and WA output for the Data below

Tag	Input 1	Input 2	Input 3	Input 4	Input 5
Weight	0.2	0.3	0.1	0.3	0.1
Val	10	11	12	11	8
FMV	n/a				
GMV	11				
FPV	11				
WA	10.6 (rounded 11)				

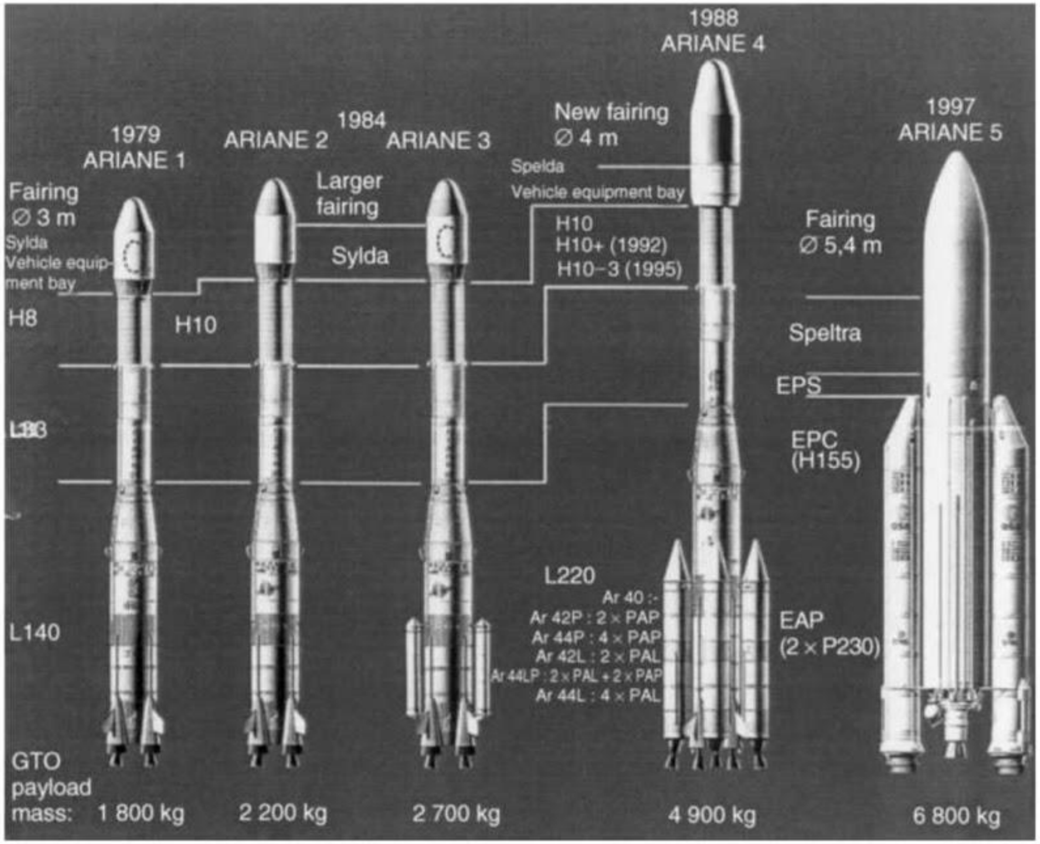
- Calculate **FMV, GMV, FPV and WA** output for the data above
- Do not round results, i.e. **return decimal values where appropriate**
- Note that the weight row adds up to 1.0

Increasing Software Reliability

- System faults may not only be the result of hardware issues, but can be a consequence of software problems
- Again, redundancy approaches can help, i.e.
  - Static software redundancy
  - dynamic software redundancy
- Let's start with a case study first (Ariane 5)

The Ariane Rocket Family

- Ariane is a series of a European civilian (ESA) expendable launch vehicles for space launch use
- GTO = [geosynchronous transfer orbit]



The Ariane 5 Accident

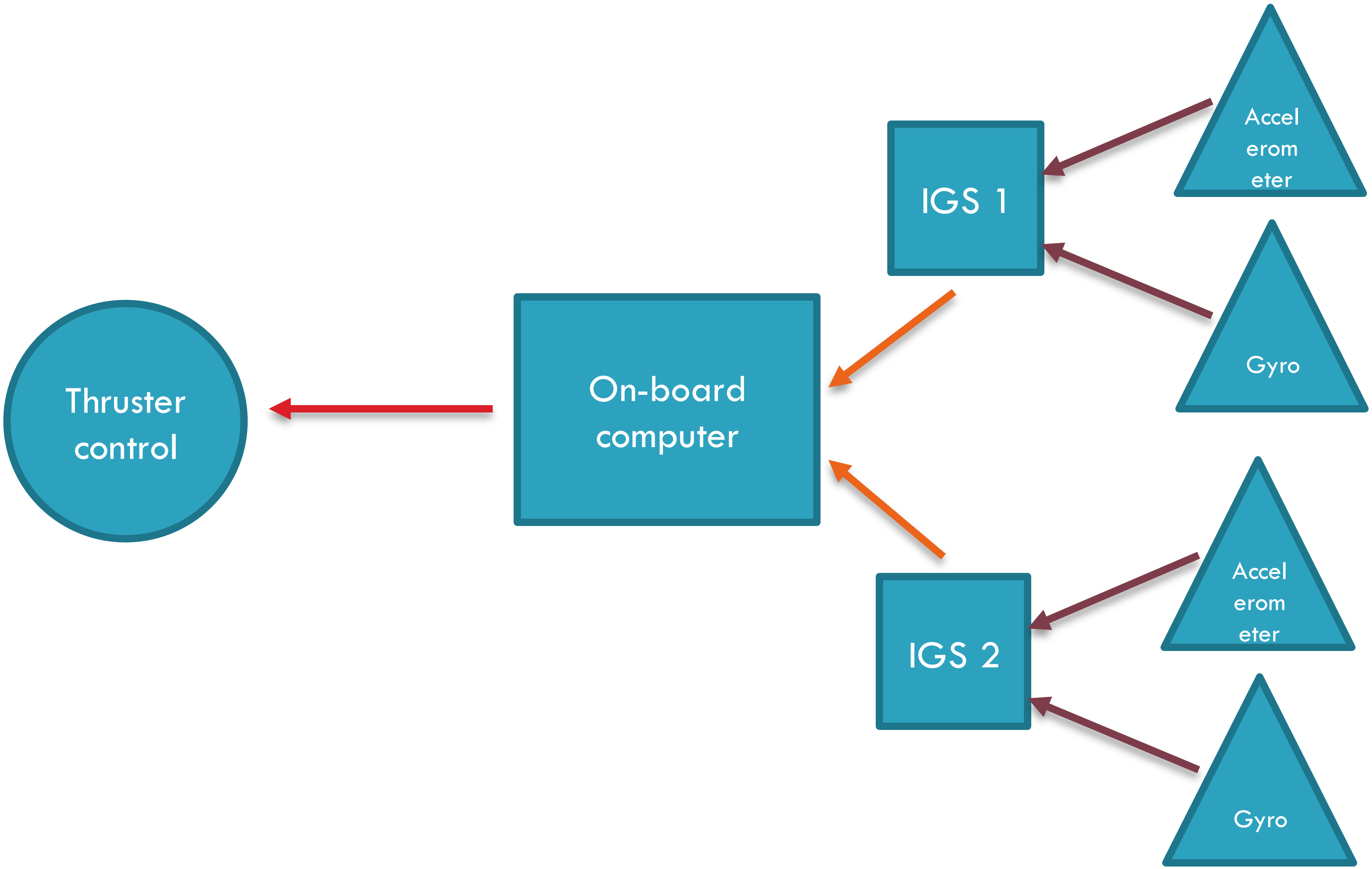
V

- Ariane 5 is a now retired European heavy-lift space launch vehicle
- The launch vehicle had 82 consecutive successful launches between 2003 and 2017
- However, its maiden flight on 4 June 1996 resulted in self-destruction after 37 seconds because of a malfunction in the control software
- See https://www.youtube.com/watch?v=gp_D8r-2hwk

One Bug – One Crash

- Steering was controlled by the **on-board computer**, which mistakenly thought the rocket needed a course change because of numbers coming from the **inertial guidance system (IGS)**. The IGS uses gyroscopes and accelerometers to track motion
  The numbers looked like bizarre flight data, but were actually a diagnostic error message. The guidance system had in fact shut down!
- This shutdown occurred 36.7 seconds after launch, when the **guidance system's own computer tried to convert one piece of data - the sideways velocity of the rocket - from a 64-bit format to a 16-bit format. The number was too big, and an overflow error resulted.** <u>When the guidance system shut down, it passed control to an identical, redundant unit</u>, which was there to provide backup in case of just such a failure. <u>But the second unit had failed in the identical manner a few milliseconds before, as it was running the same software</u>

Ariane 5 Flight Controller



The Ariane 5 Accident: Root Cause Analysis

- The software for the IGS was originally written for the Ariane 4 rocket and re-used for Ariane 5
- However, Ariane 5 is a more powerful rocket than Ariane 4, resulting in a sideways velocity that was not anticipated when the IGS was originally build, causing a numeric overflow
- As a result, the booster nozzles got incorrectly aligned by the on-board computer, which led to a rapid change of attitude, which caused the launcher to disintegrate due to aerodynamic forces

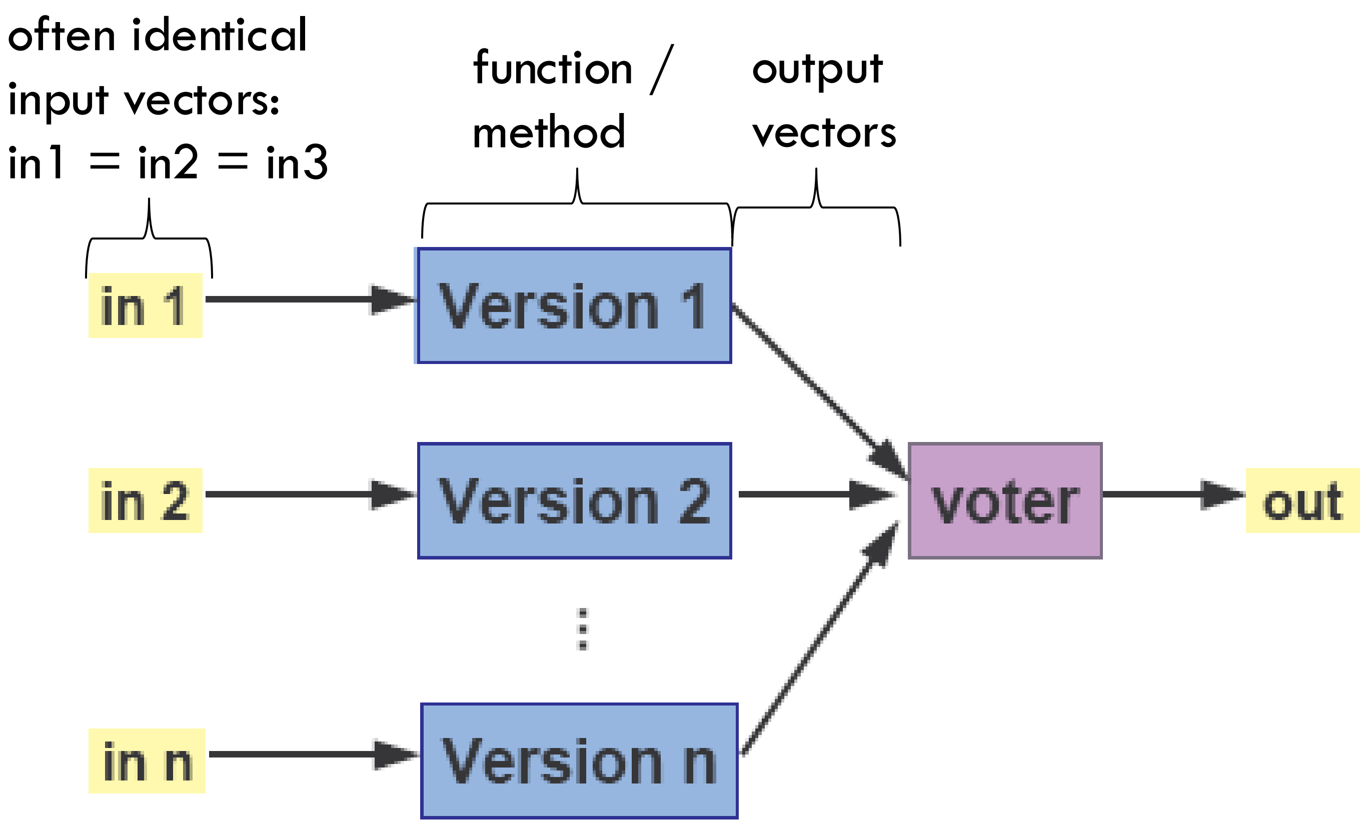
Software Redundancy

- The most certain and effectual check upon errors which arise in the process of computation is to cause the same computations to be made *by separate and independent computers*, *<u>and this check is rendered still more decisive if their computations are carried out by different methods</u>*

Static Software Redundancy

- N version programming (N>=2)
  - Independent generation of N functionally equivalent programs from same spec
  - Assumption: Developed independently→ will fail independently
- N versions run concurrently
  - Voter makes decision
  - Impacts on performance and synchronisation issues may have to be considered
  - Good match for NMR
- Also
  - Use of a common language may lead to common errors
  - Different compilers/hardware minimise risk of common failure

N-Version Programming

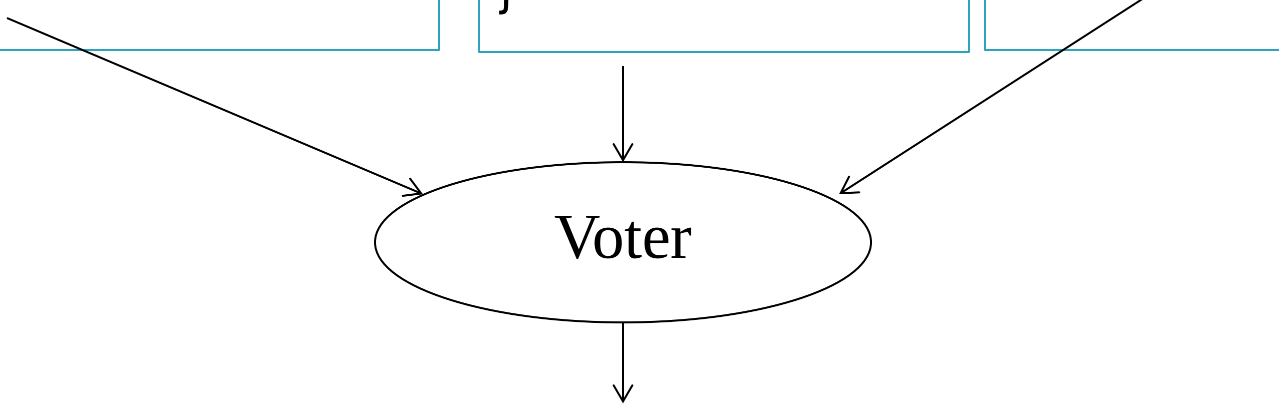


N-Version Programming: Example Factorial

```
int fac_v1(int a) {
int res = 1, i;
for (i = 2; i <= a; i++)
   res *= i;
return res;
}
```

```
int fac_v2(int a) {
int res = 1, i;
for (i = a; i > 1; i--)
   res *= i;
return res;
}
```

```
int fac_v3(int a) {
if (a < 2) return 1;
else return (a * fac_v3(a - 1));
}
```



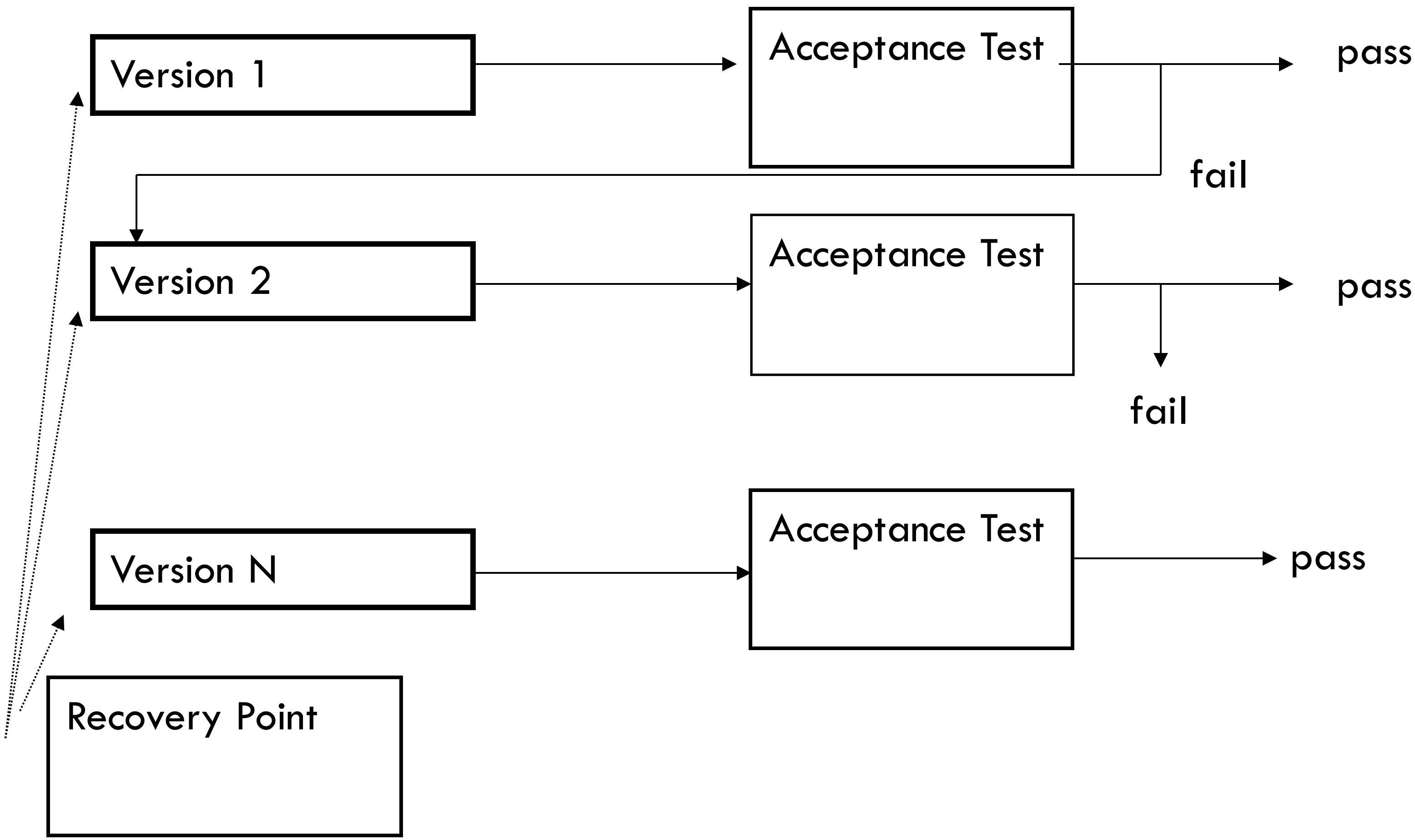
N-Version Programming Issues

- Redundant code runs regardless of whether faults are present
  - Additional CPU-resources required
  - Redundant code may have different execution times, may cause synchronisation problems
- Initial spec may provide common failure mode regardless of subsequent strategies
- This requires the creation of diverse and equivalent specifications so that programmers can design software which do not share common faults
- Teams of programmers may have similar bad habits and/or biased programming techniques
  - Especially if same language is used (think of pointers in C)
- Overall, very costly process → limited application (e.g. avionics, military)

Dynamic Software Redundancy

- In this approach, redundant components run only under fault conditions
  - This assumes that a fault can be detected, e.g. via overflow, out-of-bounds, or acceptability checks
- After a fault has been detected, **backward recovery** takes place:
  - Go back and restore system to safe state prior to error
  - Avoid problem on repeat by using different implementation

Recovery Block Approach



Recovery Block Approach via Java Exception Handling

```
Element getElement(x) {
    try {
        return repository.getElement(x);
    } catch (NotFoundException e) {
        return fallbackToSimilar(x);
    }
}

Element fallbackToSimilar(x) {
    try {
        return repository.getSimilarElement(x);
     } catch (NotFoundException e1) {
        return fallbackToParent(x);
     }
}

Element fallbackToParent(x) {
    try {
        return repository.getParentElement(x);
    } catch (NotFoundException e2) {
        throw new IllegalArgumentException(e);
    }
}
```

- Consider three versions, i.e.
  - getElement()
  - fallbackToSimilar()
  - fallbackToParent();
- If *getElement()* throughs a *NotFoundException* object, *fallbackToSimilar()* will be called
- If *fallbackToSimilar()* throughs a *NotFoundException* object, *fallbackToParent()* will be called

Summary

- Hardware-, software-, and information redundancy are the building blocks for RTSCS
- They increase system reliability and subsequently system safety
- They can be found in many industries including
  - Automation
  - Avionics (next slides)
  - Military
  - Medical device
  - Robotics

Final Example: Airbus 340

- Hardware & software redundancy
  - NMR redundancy with
    - 3 main flight controllers
    - 2 backup flight controllers (that replace faulty unit on-the-fly)
  - Software developed by different teams and on different platforms

Final Example: Boeing 777

- Hardware Redundancy
  - Motorola
  - AMD
  - Intel
    - Example Pentium FDIV bug
- **Ada** programming language used, but different compilers

The correct value is:

$$\frac{4{,}195{,}835}{3{,}145{,}727} = 1.333820449136241002$$

When converted to the hexadecimal value use
at or beyond four digits:[9][10]

$$\frac{4{,}195{,}835}{3{,}145{,}727} = 1.333739068902037589$$