

Software Reliability

The Silent Danger - How to Assess It



Suzanne Flynn
Cygnet Solutions Ltd

© 2011 Cygnet Solutions Ltd.



Cygnet Solutions

- **We are a small independent software development and consultancy company based in Scotland**
- **We write safety/business critical software for the utilities and defence industries**
- **We assess software-based systems against various standards including IEC 61508**
- **We carry out independent design and code reviews of systems**

Why am I Giving This Talk?

- Because software reliability is different from hardware
-
- Measuring software reliability is virtually impossible

Why is Software Reliability a Silent Danger?

- Because the software maybe well designed and tested

AND

- Its “reliability” may be independently expertly assessed

BUT

- A killer bug can remain silent for days, months, years



Some Definitions

Software Instructions to a computer

Hardware Wiring, power supplies, electronics, circuit boards

Firmware Is software, but stored semi- permanently

Compare and Contrast Software & Hardware

- **Design**
- **Reliability**
- **Proving**
- **Prototype**
- **Manufacturing faults**
- **Aging**

Necessity

- **Software needed and unavoidable so must ensure as reliable as possible**
- **Software enhances safety e.g. ABS on cars**



How are Faults Introduced - 1

- **Design**

- Metric/imperial
- Misinterpretation of requirements or design
- Design or algorithm is wrong or unsuitable

- **Coding**

- Wrong sign, operator
- Typo, cut & paste error
- Buffer overflow
- Rounding
- Loops off by one, counters
- Divide by 0
- Whole numbers v decimal numbers
- Initialising
- Data input not validated, data out of bounds

How Faults Are Introduced – 2

- **Compiler (translates the program into language of the computer)**
- **Operating System**
- **Wrong version of software used**
- **Processor - CPU Maths**
- **User incompetence/lack of knowledge**

Types of Failure and Safety

- **Safe:**

- Crash
- Hang
- Slow
- Will not accept input

Above can be handled and announced by the system, but may be inconvenient

- **Dangerous:**

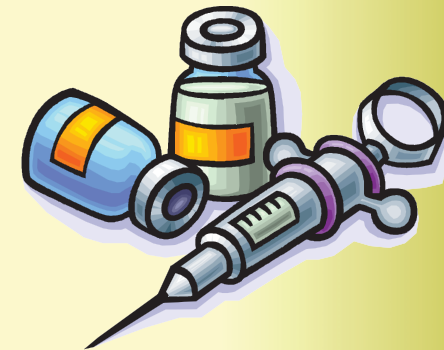
- Incorrect calculation
- Wrong data accepted
- Memory corruption/overflow
- No self-tests

Prevention v Cure v Assessment

- Prevention of errors by good design/review/testing

Better than

- 'Cure' by testing on finished article



- Assessment is to give user confidence
- Assessment examines how well prevention techniques have been applied

Prevention and Detection

- **Design Review**
- **Code Review**
- **Well trained competent programmers and testers**
- **Re-use of existing programs**
- **Psychological**
- **Test – module, functional**

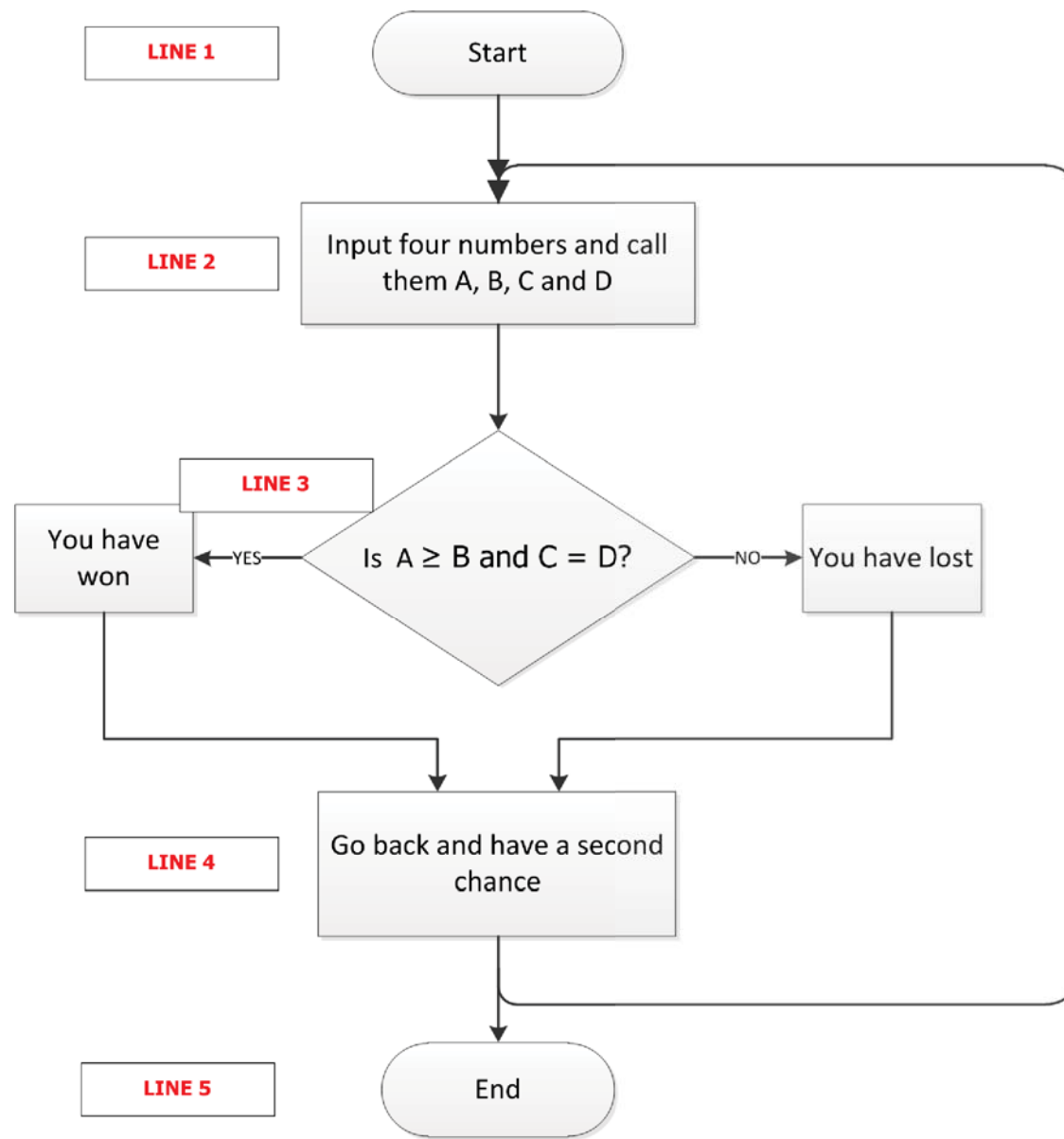
Testing Problems

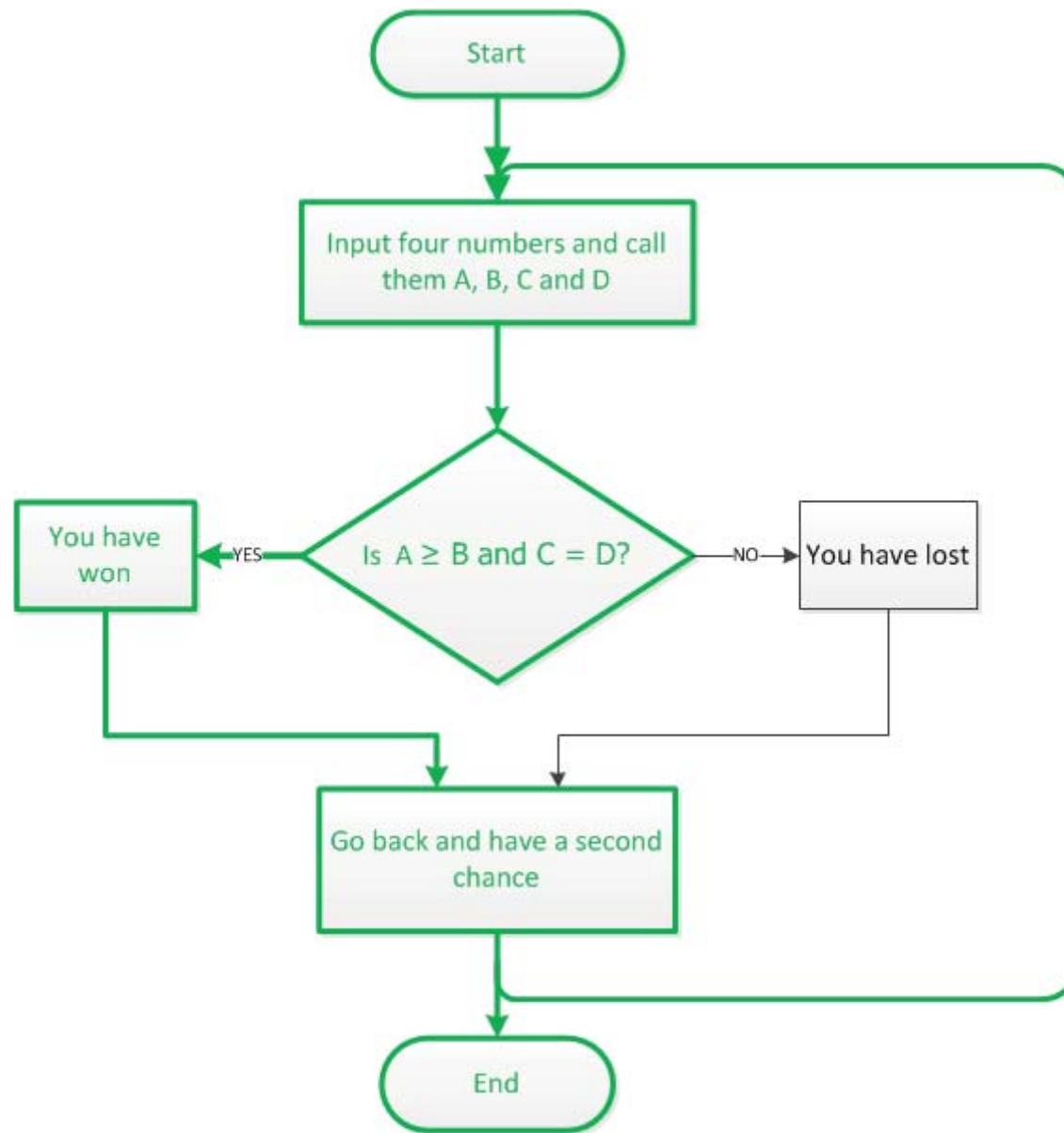
- **Tester tests how tester thinks it should work**
- **Independence**
- **How much testing:**
 - Lines of code
 - Number of decision
 - Number of conditions
 - Number of jumps

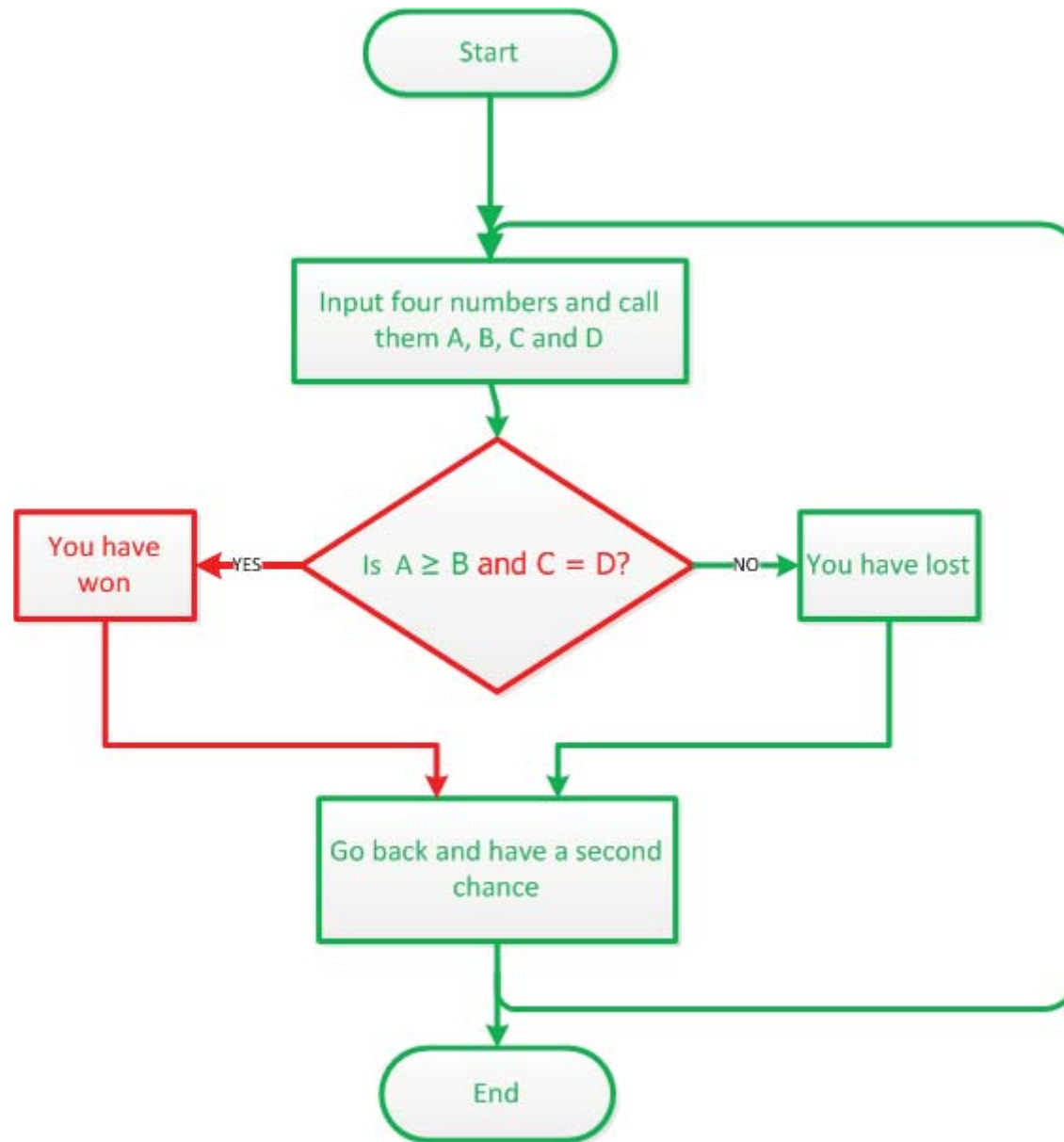
An Illustration

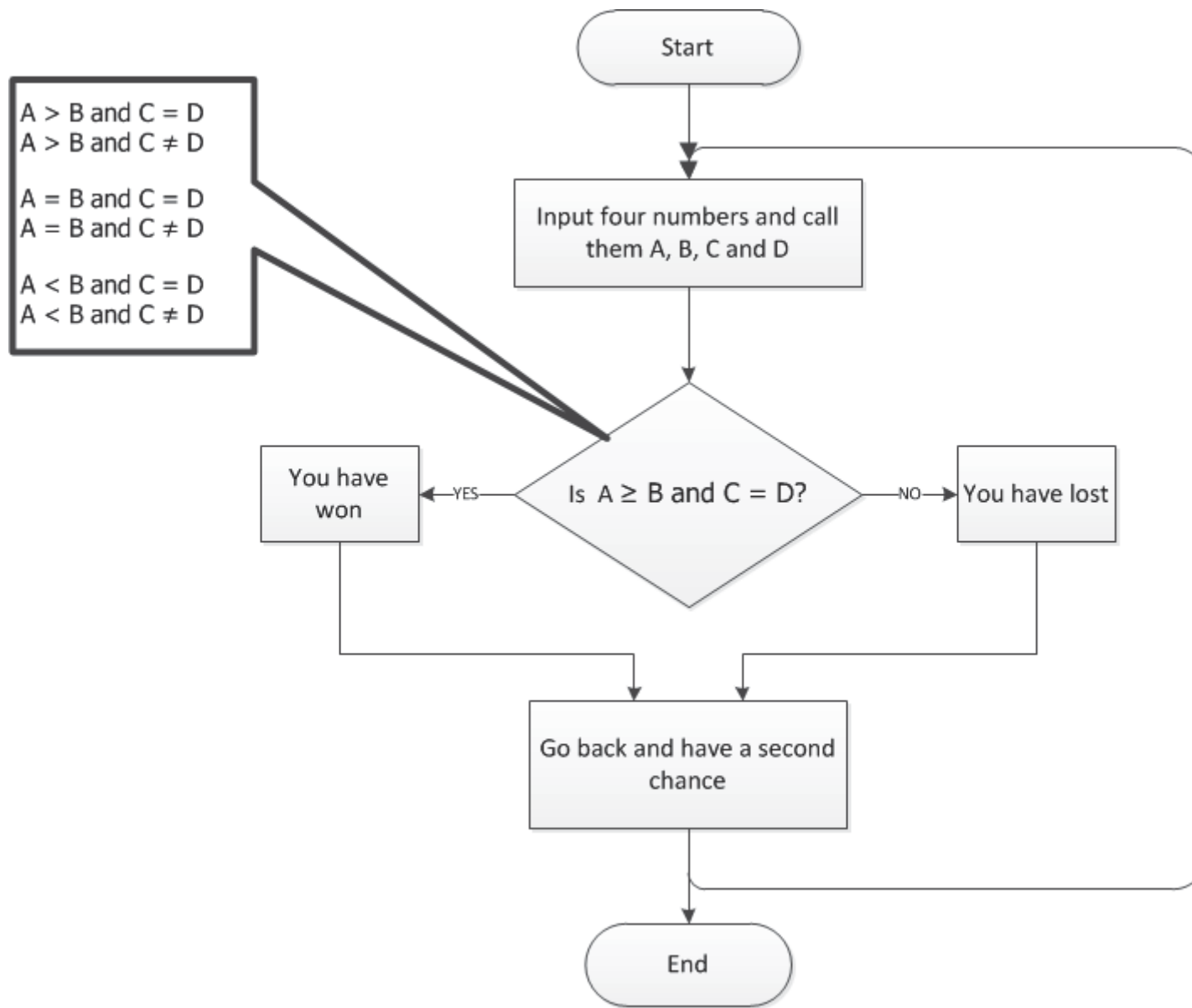
A program takes in 4 numbers and if they meet the criteria outputs one message and if not outputs another message.

Just five lines of code as shown in the following flowchart









- **SIX** tests for just **FIVE** lines of code -----

And there's more

- **Loops**
- **Whole numbers and decimal numbers**
- **Domain e.g.**
 - Aircraft Autopilot
 - ABS
- **Conclusion: Testing is as difficult as the design and requires as much time spent on it**

Assessing Software Reliability

Quantitative

- Automated testing
- Statistical Testing and Statistical Models
- Estimation and Prediction
- Field Experience
- Static Analysis

Quantitative – but it is measuring aspects of software **related** to reliability NOT the reliability itself

Static Analysis

- Tool based so quick and easy to use.
- Measures, identifies:
 - Complexity
 - Inputs and outputs
 - Flow paths between modules
 - Unreachable code
 - Bad programming practice



Assessing Software Reliability

Qualitative

- Competence of programmers, testers etc
- QA processes followed:
 - Defined Lifecycle and Documentation
 - Design
 - Review
 - Code Review
 - Module testing
 - Test coverage, statement, decision, condition
 - Functional Testing – all functions tested?
- Independence of reviewers and testers
- Static Analysis
- Prior use data



Assessing Software Reliability – Standards

- **IEC 61508 - covers the electronics and software**
- **NASA 871913 Software Safety**
- **US DOD 178B Software for Airborne Systems**
- **Carnegie Mellon SEI CMM**

Also

- **Programming language standards e.g. MISRA C, ESA BSSC 2005(2) Java**



Measures of Reliability

- **Reliability expressed in terms such as :**
 - Mean time to failure
 - Probability of failure in a set time or on demand
 - Often expressed in discrete safety integrity levels (SIL)
 - SIL 1 is the lowest level: SIL 4 the highest
 - E.g. SIL 1 is expected to fail between 1 in 10 and 1 in 100 times of operation
 - Proportion of dangerous detected failures and safe failures



How SIL is Applied for Software

- **Qualitative assessment often based on IEC 61508**
- **IEC 61508 has general requirements for software and “techniques and measures”**
- **Leap of faith based on experience from the assessments to the SIL level**
- **Hardware – also uses the above, but in addition can predict mean time to failure from knowledge of the failure rates of its components**

Software SIL - Summary

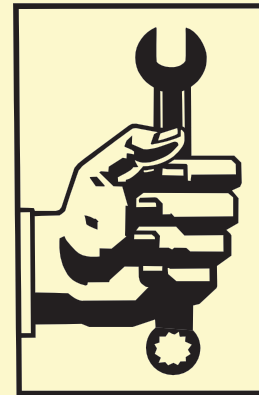
- **Techniques used are best available**
- **Nothing easy, many are not practical**
- **Assessment tends to be cautious**
- **SIL levels achieved in these assessments seem to be justified in practice**
- **Assessing the device and assigning a SIL at the very least categorises the device as very reliable, reliable or unreliable!**

How to Use SIL

- **Don't just rely on the SIL assessment of an individual device; make sure the overall system design is safe!**
- **Carry out extra testing**
- **Review the code**
- **Measure the complexity of the code**

Software Maintenance

- **Every change introduces new errors**
 - E.g. Windows, iPhone



Practical Experience

Legacy Process Controller

- **Originally developed by another company**
- **Records missing**
- **Gaps compensated for with**
 - Code review
 - Static Analysis
 - Witnessed re-run of module tests
 - Statistical testing based on its use on plant
- **Manufacturer's co-operation**
- **Working well on plant now with no problems**

Practical Experience

SIL 3 Device

- **Independently assessed to SIL 3**
- **Programmed in Assembly Language: not a language of choice for safety**
- **Inadequate documentation**
- **Unmaintainable**
- **Critical bug (memory overwrite) that should have been avoided and detected**
- **Similar bugs may still lie undetected in the software**



Practical Experience

Device developed to IEC 61508

- **Assessment was straightforward**
- **But still issues with the failure rate prediction methodology**
- **Techniques and measures only “Highly Recommended” ones used**

IEC 61508 - “Highly Recommended” v “Recommended”

- **If only Highly Recommended Techniques and Measures are applied, the following are not required for SIL 1 and 2**
 - Self-tests
 - Stress testing
 - No complexity metrics at any level
- **Typically manufacturers and independent assessment houses judge the SIL level only on highly recommended parts of IEC 61508**

Infamous Bugs



- **NASA Mars orbiter lost.** Use of imperial units of measurement when metric should have been used.
- **ESA Ariane 5 lost** Ariane 4 rocket software reused in the Ariane 5. Certain types of numbers were larger in the Ariane 5 than in the Ariane 4, triggering an overflow condition that resulted in the rocket self-destructing.
- **National Cancer Institute, Panama City.** Radiation wrong dosage. The software miscalculated the proper dosage of radiation for patients undergoing radiation therapy.
- **Intel Pentium floating point divide.** There were a few missing entries in the lookup table used by the digital divide operation algorithm. Only occurred in 1 in 9 billion division.



Millennium Bug- Leap Year

- A year is a Leap year if it is divisible by 4, but not if divisible by 100, unless divisible by 400.
- By Rule 3, Year 2000 was a leap year.
- Many systems got it right for the wrong reasons using Rule 1
- In a example in the nuclear industry, one programmer applied Rule 3 to part of the system and another programmer applied Rule 2
- When the two halves of the system tried to talk to each other, they “argued” over the date on 28th Feb 2000
- Consequence was the system could not do its intended job of measuring and failed catastrophically

Conclusion

- **Software reliability is not measurable or predictable**
- **Aspects of software related to quality and reliability can be assessed that and the assessment used to arrive at a probable reliability figure**
- **Software reliability should not be used in isolation**



Thank you

Any Questions

