

Lecture 9  
**Reliability and fault tolerance**

Monday Feb 14, 2011

Philipp Rümmer  
 Uppsala University  
 philipp.ruemmer@it.uu.se

- Notion + definition of reliability
- Notion of fault tolerance
- Typical classes of faults in embedded systems
- Techniques to improve fault tolerance

1/37

2/37

Correctness vs. reliability

- Last lecture: **Correctness** of software  
 → Absolute notion, assessed using V&V
- In practice: Every system will fail (sooner or later)
  - Totally correct of software is usually too expensive
  - Hardware faults (unavoidable)
- More relative notion: **reliability**

3/37

Reliability

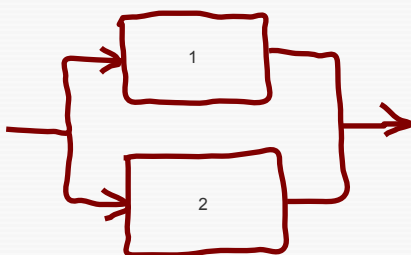
- **Probability**  $R(t)$  that a device/system fulfils its **intended function** for a period of time  $t$ , given **precisely stated conditions**
- If  $T$  is time (random variable) when first failure occurs, then:

$$R(t) = P(T > t)$$

Assumes knowledge about probability distribution

4/37

Reliability block model: parallelism ("or")



$$R(t) = R_1(t) + R_2(t) - R_1(t) \cdot R_2(t)$$

5/37

Reliability block model: series ("and")



$$R(t) = R_1(t) \cdot R_2(t)$$

- Compare with: **fault tree analysis**

6/37

Mean time between failures (MTBF)

- E.g., "power supply A has MTBF of 40000 hours"
- Determined by testing  $N$  devices for  $T$  hours each, counting the number  $R$  of failures

$$\theta = \frac{N \cdot T}{R}$$

- Inverse: **failure rate**  $\lambda = \frac{1}{\theta}$

7/37

Lusser's equation

- Relationship between reliability and MTBF:

$$R(t) = e^{-\frac{t}{\theta}} = e^{-t\lambda}$$

(special case of Weibull distribution)

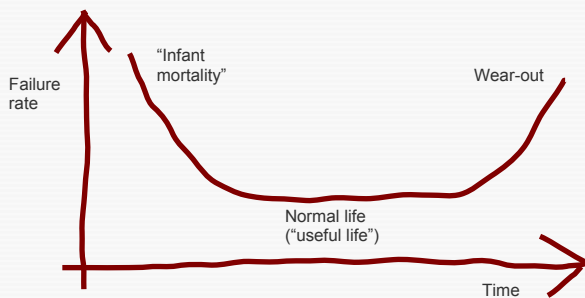
- Interesting consequence: Devices will survive their MTBF only with likelihood

$$R(\theta) = e^{-\frac{\theta}{\theta}} \approx 0.37$$

8/37

## Bathtubs

- Failure rate of hardware is considered to follow a “bathtub” distribution



9/37

## Faults categories

- Software faults
- Environment faults
  - E.g., wrong sensor data
- Internal hardware faults
  - E.g., bit flips, defective components

11/37

## Typical faults in embedded systems

13/37

## Testing memory faults

- Simple approach:  
Let system run for a long time, observe failures
- Problem: faults might only occur under special circumstances
- Acceleration through **fault injection**

15/37

## Improving reliability of a system

- Define **fault model** describing what could go wrong
  - E.g., bit-flips in memory, spurious interrupts, noisy sensor data
- Assess **fault tolerance** of system
  - Can a fault lead to a failure? How critical?
  - Testing, fault injection
- Measures to improve fault tolerance
  - E.g., error correction codes, redundancy

10/37

## Fault persistence

- **Transient** faults
  - Isolated event during system execution
  - E.g., overheating, glitch in a power line
- **Intermittent** faults
  - Malfunction that occurs repeatedly (periodic or aperiodic)
  - E.g., damaged sensor
- **Permanent** faults
  - Permanently defective component
  - E.g., stuck signal or memory cell

Could each be caused by software bug!

12/37

## Fault model: memory faults

- Can affect internal RAM, ROM, registers, external RAM, etc.
- **Soft error:** memory contents are modified (→ transient)
- **Hard error:** memory cell is damaged, e.g., stuck at zero (→ permanent or intermittent)
- Many possible causes: electrostatic discharge, power surging, vibration, radiation (single-event upsets)

14/37

## Memory fault injection

- **Hwifi:** hardware-implemented fault injection
  - Can be manual or automatic
  - E.g., through heavy-ion radiation
- **Swifi:** software-implemented f.i.
  - Simulate faults by code instrumentation
  - More systematic, more possibilities for optimisation
  - Tools available to automate Swifi
  - Related to **mutation testing**

16/37

## Swifi example

```
for (;;) {
  if (GPIO_ReadInputDataBit(GPIOC, SwitchPin)) {
    ++count;
  } else if (count != -1) {
    GPIO_WriteBit(GPIOC, ON1Pin, Bit_RESET);
    GPIO_WriteBit(GPIOC, ON2Pin, Bit_RESET);
    count = -1;
  }

  if (CONDITION)
    count ^= 1 << BIT;

  if (count == 10) // 0.2 seconds
    GPIO_WriteBit(GPIOC, ON1Pin, Bit_SET);
  else if (count == 100) { // 0.2 + 1.8 seconds
    GPIO_WriteBit(GPIOC, ON1Pin, Bit_RESET);
    GPIO_WriteBit(GPIOC, ON2Pin, Bit_SET);
  }
}
```

Instrumentation  
simulating a bit flip  
(e.g., triggered randomly)

17/37

## Swifi example (2)

- Instrumented code is tested
  - If no failures occur, software is *tolerant* w.r.t. the injected fault
- Alternative method:  
Compare outputs of the original and the instrumented software during tests
  - If no difference in outputs can be observed, fault is tolerated

18/37

## Protection from memory faults

- Error-correcting memory
  - RAM: error-correcting codes (ECC), traditionally Hamming codes
  - ROM: checksums (e.g., CRC)
  - Can be implemented in software or hardware (e.g., some CORTEX M3 have error-corrected flash memory)
  - Memory “scrubbing:” fix errors in memory in the background
- Check-point schemes, runtime monitors, self-stabilising algorithms, ...

19/37

20/37

## CPU/MCU defects

- Components of controller or system can be defective
  - Usually permanent fault
- Might be a **byzantine fault**
  - Component continues operating in a faulty manner
  - “Babbling idiot”
- Difficult to predict or inject

## Detection of defects

- **Built-in self-test (BIST);**  
aka **built-in test software (BITS)**
- Permanently run software tests in the background (when system is idle)
- Built-in watchdogs
- When fault is detected, component can be switched off or replaced, fault can be reported

21/37

## Spurious interrupts

- Defective components might erroneously trigger interrupts
  - Can destroy real-time properties if load becomes too high
- Effects can be mitigated by redundancy
  - Additional checks whether interrupts are genuine (e.g., confirmation flag set by component via DMA)
  - Interrupts can be disabled if spurious interrupts are detected

22/37

## Precision of sensors

- Most sensors exhibit some amount of noise
  - E.g., GPS usually off a few meters
  - Position sensors (like in the elevator lab) might count wrongly
- Improve precision by combining different sources of information
  - Multiple sensors
  - Expected sensor values, domain knowl.
  - Combined using **Kalman filters**

23/37

## Networking faults

- Various possible problems
  - Transmission errors, dropped messages
  - Defective (byzantine) component sends spurious messages
- Crucial: error-correcting protocols
- Can be tested for:
  - Simulation of faulty channels
  - **Fuzzing** techniques

24/37

## General approaches to fault tolerance

25/37

## N-version programming

- Implement  $N$  completely independent control systems
  - Independent hardware
  - Independently developed software
  - Independent programmers
  - But same specification
- Main idea: different systems will contain different kinds of bugs
  - Unlikely that all fail at the same time

26/37

## N-version programming (2)

- Main idea: ... **different kinds of bugs**
- Not so clear whether this is actually true:
  - Studies show that even independent teams tend to make the same mistakes
- Nevertheless: this is one of the standard techniques

27/37

## N-version programming (3)

- Different topologies possible
  - Voting scheme:
    - Majority wins
  - Master/slave
    - If master system fails, slave takes over

28/37

## Checkpoints

- Record system state at particular points during execution
- E.g., write to file, send to other system over network
- Can be used for diagnostic purposes, or after a system failure

29/37

## Recovery blocks

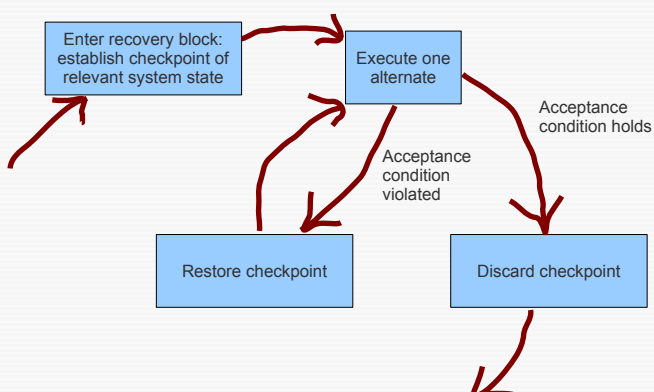
- *Extended version of checkpoints:* Split computation into *recovery blocks*, which can be rolled back if a fault is detected
- Standard version:

```

ensure    <acceptance test>
by        <primary alternate>
else by   <alternate 2>
...
else by   <alternate n>
else error
    
```

30/37

## Basic recovery schema



31/37

## Properties of recovery blocks

- Can mitigate various faults
  - Software faults in one of the alternates
  - Transient hardware faults
  - Transient erroneous sensor data
  - Timing problems:
    - first try expensive computation, abort if it takes too long and use a faster (but sub-optimal) variant

32/37

## Properties of recovery blocks (2)

- Of course, introduces new problems
  - Side-effects of alternates can be hard to undo
  - Some computations might be impossible to repeat
  - Checkpointing can be expensive
  - Maybe no time for repetition
- Concept is related to *transactional memory*

33/37

## Watchdogs

- One of the most important techniques
- Watchdog is a component that monitors the system
  - If system does not react any more, watchdog restarts it
  - Should be as independent as possible from system
  - But: most micro-controllers have built-in watchdogs (often with independent clock)

34/37

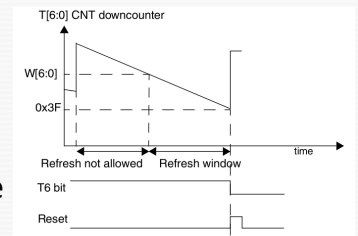
## Watchdogs (2)

- Typically: watchdog works like a timer
  - Continuously counts up to some limit
  - If limit is reached, system is restarted
  - System has to reset the timer regularly to prevent restart
- STM32 CORTEX M3 has two built-in watchdogs
  - “Independent watchdog” (own clock)
  - “Window watchdog”

35/37

## Window watchdog example

- 8-bit timer, counting downward
- Timer has to be refreshed within a certain “window”
- Too early or too late refresh → System is restarted
- Refreshing can be done using an interrupt



36/37

## Watchdogs in general

- Watchdogs have to discriminate:
  - Normal system execution, from
  - System that hangs, or that is running rogue
- Difficult in general:  
*Byzantine* system can do anything
- Interesting read:  
<http://www.ganssle.com/watchdogs.htm>

37/37