DEALING WITH SOFTWARE HAZARDS

A short tutorial on designing software to meet safety requirements

M. Squair

www.criticaluncertainties.com

This tutorial is drawn from a number of sources, whose assistance I gratefully acknowledge

Except for images whose sources are specifically identified, this copyright work is licensed under a Creative Commons Attribution-Noncommercial, No-derivatives 4.0 International licence (CC BY-NC-ND 4.0). To view a copy of this licence, visit http://creativecommons.org/licenses/by-nc-nd/4.0/
Module Learning Objectives

A short tutorial on the basic techniques of safety design to software. We will be focusing on problematic areas of safety critical software design (software design hazards), and particularly dealing with:

- Real-time,
- Concurrency, and
- Asynchronous behaviour
Overview

“Software temptations are virtually irresistible. The apparent ease of creating arbitrary behavior makes us arrogant. We become sorceror’s apprentices, foolishly believing we can control any amount of complexity. … We would be better off if we learned how and when to say no.”

G.F. McCormick

When Reach Exceeds Grasp

“Better is the enemy of good enough”

Anon.
Software Hazards

- **Software hazards, system hazards and accidents**
  - A software ‘hazard’ is a software error state that can potentially be linked to a system level failure (accident) accident
    - Equates to an error in output timing or data
    - System hazard = System level service error
    - System accident = Resultant failure state
    - System accident severity = Fault state
  - Some authors consider a hazard as any error state that can be causally linked to an accident
  - Subsystem hazards = those error states of a subsystem that can propagate to a system hazard

- High yaw rate command
  - Aircraft in flat spin
    - Aircraft crashes
      - Loss of crew
Design Strategies for Software Safety

■ Context
  – Requirements or design drivers
    • The systems safety case requires that the software achieve a specified level of dependability or a verifiable absence of certain failure modes
  – Our objective:
    • To prevent the occurrence of an unplanned hazardous states
  – Our general strategy:
    • Develop robust, simple, deterministic, verifiable and understandable software
    • Implement specific countermeasures for known software design faults
    • Note. These general attributes and countermeasures are not independent
  – Countermeasures can be applied at various levels of design
    • Architectural
    • Detailed design (mechanistic)
    • Implementation

■ We will not consider
  – Human operator error
General Attributes of Safe Software

■ Safe software should behave predictably/deterministically
  – Predictable future behavior (value & timing (ordinal/cardinal)) given:
    • A known current state, and knowledge of future changes to its environment
    • Unpredictable behaviour is not ‘safe’ as we can’t confirm its liveness over time
  – It also masks software faults as ‘transient’ hardware faults

■ Safe software should be robust
  – It should withstand a specified set of faults (fault hypothesis)
  – But if the fault-hypothesis is violated the system will fail
  – If the violation is transient a restart of the whole system will succeed
  – ‘Never give up’ algorithms should detect this violation and initiate a restart
    • Ensure the environment is safe (e.g., freezing of actuators) while restarting
    • Provide an upper bound on the restart duration as a architecture parameter

■ High levels of safety strongly implies
  – A defence in depth approach

See Annex D
For a full definition of the Fault Hypothesis
General Attributes of Safe Software

- That layers of defence should be independent
- That the architecture should be simple to make hazard analysis and safety verification tractable

■ Safe software should be verifiable
  - Should not rely on probabilistic analysis
  - It should be testable:
    • Test results should be observable and test parameters should be controllable
    • Implies that there is a limited number of states to test
  - Verification includes self test and diagnosis
    • Capable of identifying objects violating the specification
    • Ability to access faults masked by fault tolerance layer
    • Ability to identify faults based on specified black box behaviour (e.g. time/value) visible at the interface
    • Ability to trap anomalies as the occur for later diagnosis
Case Study: CANBUS Error Diagnosis [15]

CANBUS does not allow identification of the source of a message sent with the wrong identifier.

Error CAN message AND wrong identifier

CC: Communication Controller
General Attributes of Safe Software

- Safe software should be simple [2]
  - Complex interactions increase the likelihood of design faults
    - Unanticipated side effects and interactions (design faults)
    - Increase probability of fault propagation
    - Harder to verify safe behaviour
  - Interface complexity is a cardinal indicator of overall complexity:
    - Often where design faults are found (e.g. false assumptions about behaviour)
    - Reflects a high degree of coupling between components (the root cause)
  - Should exhibit
    - A limited number of components/objects and interfaces
      - Measure code complexity with McCabe's cyclomatic measure*
    - Use of proven state of technology design not state of art
    - Avoidance of distributed systems especially for control systems
    - Limited states which strongly implies a design which is deterministic, single-tasking rather than multi and period or time driven

* McCabe's cyclomatic No. = e-n+2p where e = no. of arcs, n = nodes and p = no. of independent graphs being examined (usually one). Arcs represent non-sequential flow of control, nodes the sequences of instructions.
General Attributes of Safe Software

- Interactions between components should be limited and straightforward
- Worst-case timing are determinable from the design
- The code (implementation) contains only the minimum features required
- A top down hierarchical design employing encapsulation and data hiding
  - Be disciplined, a key driver of complexity is implementing ‘wants’ rather than ‘needs’

The design of safe software should be understandable

- Some general techniques
  - Separation of concerns principles
  - Abstraction of the design
  - Information hiding
  - Coupling and cohesion principles
  - A simple design is more easily communicated and described to others
  - The relation between safety case arguments and the architecture should be established at the highest practical level of abstraction
  - The fault hypothesis* should be clearly stated

---

*Inherent concurrency. It should be possible to understand the "inherent concurrency" of a program by examining it…

*A special case of the more general design hypothesis which is the set of assumptions that are made in the design of the system.
Case Study: Honeywell JA37B Autopilot [2]

- First full authority fly-by-wire system
  - 15 years of use without one in-flight anomaly
  - Purposeful pursuit of simplicity in design
  - So called rate based architecture

- Design rules
  - NO interrupts
  - NO back branches in the code
  - Control flow ‘unwound’ into a single loop executing at a constant rate

- Resultant design
  - No use of subroutines
  - Inter-module communication by post-box
    - Multiple one bit flags
    - Could be monitored as a flag vector during testing

- Complexity of data structure gave simplicity of control structure
Software Safety Design Principles

“Software is the resting place of after-thoughts”

J. Shore

The Sachertorte algorithm and other Antidotes to Computer Anxiety

See Annex C for a full listing of hazard countermeasures
Trade-Offs between Safety and Capability

- For example
  - Eliminate a hazard by substituting a robust and simple hardware interlock for a software one

- But there may be trade-offs, such as
  - Requirement for safe separation of hardware
  - Less capacity for growth
  - Less flexibility in the implementation and during subsequent modifications

- Documenting such tradeoffs is essential
  - Use trade study reports for major decisions
  - At a minimum document in design files or using engineering memo’s

THERE IS NO TECHNOLOGICAL IMPERATIVE THAT SAYS WE MUST USE SOFTWARE
Integrated versus Decoupled Design and Robustness

■ Current safety critical systems are more integrated
  – Original software systems were federated (one processor per function)
    • Low coupling minimised fault propagation and enhanced robustness
  – Challenge is to restore/maintain decoupling in modern integrated systems
    • Even low levels of coupling can be hazardous
      – ‘Small world’ syndrome where single links markedly increase overall coupling
        • A principal concern for both distributed and redundant system design

■ Robustness and decoupling
  – Define the required service to be maintained in presence of faults
    • Graceful degradation
    • Never give up’ strategy
  – Define fault hypothesis or what the system is designed to withstand
    • Modes, fault numbers, arrival rate and fault containment zones (FCU))
  – Justify independence of FCU from cross coupling & common mode errors
  – Justify assumptions made in fault hypothesis as to fault types
Software Architectural Design

“Make it right before you make it work”

B.P Douglass
Safety Critical Systems Design
Safety Architecture Patterns

<table>
<thead>
<tr>
<th>Architectural pattern</th>
<th>Hazard countermeasure class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single channel* protected</td>
<td>Control by forward recovery/failsafe design</td>
</tr>
<tr>
<td>Multi-channel voting</td>
<td>Control by backward recovery</td>
</tr>
<tr>
<td>- Homogenous redundancy</td>
<td>Ditto (e.g. fault masking)</td>
</tr>
<tr>
<td>- Diverse redundancy</td>
<td>Ditto</td>
</tr>
<tr>
<td>-- Different but equal</td>
<td>Ditto</td>
</tr>
<tr>
<td>-- Lightweight redundancy</td>
<td>Control by forward recovery</td>
</tr>
<tr>
<td>--- Monitor/actuator</td>
<td>Control by forward recovery</td>
</tr>
<tr>
<td>Watchdog</td>
<td>Control by failsafe design</td>
</tr>
<tr>
<td>Safety executive</td>
<td>Eliminate by decoupling</td>
</tr>
<tr>
<td>Robust network</td>
<td>Reduce by use of physical redundancy</td>
</tr>
</tbody>
</table>

Safety is (in part) based on assumptions as to valid fault modes

- Need to verify that such modes cannot occur
- Need to consider ‘never give up’ response if fault hypothesis exceeded

* A channel is a set of devices (including processors and software) that handles a related, cohesive set of data or control flows from incoming environment events to an ultimate system output.
Single Channel Protected Design Pattern

The ‘de-minimis’ architecture

- Simple design
- Vulnerable to SPOF
  - Safety can be assured by adding hazard (error state) controls
    - Watchdog timer
    - Time outs on down stream processes
    - CRC and periodic ROM testing
    - Go to fail-safe state if channel stops operating
    - PRN signals between components
- Problems (potential hazards)
  - SPOF vulnerability
  - Fault detection may take up appreciable processor time
  - Redundant processing is common mode failure vulnerable
  - May require a ‘never give up’ mode of operation
Multi-Channel Voting Pattern

- Odd number of redundant* channels (channel ~ FCU)
  - Vote taken on validity of input and/or output from channel
  - Majority rule taken on validity of data
  - May be homogenous or diverse
  - Vulnerable to asynchronicity
  - Safety (determinism) can be achieved via:
    - Wide ‘voting gates’
    - Reduced control gain increasing reaction time
    - Synch. at control law decision points
    - Amp in/Amp out channels at decision points
  - Problems (potential hazards)
    - Asynchronous sampling
    - Increased failed sensor ‘average drag’
    - Sensor noise and sampling skew
    - Abrupt value changes on vote IN and OUT of channels (Thumps)
    - Bifurcation. Processes taking separate paths at critical points in control laws

*Redundancy can also be extended to data as well.
Case Study: ATFI-16 Program Multi-Channel Voting Pattern

■ Flight test event
  – 3-second departure Flight test 36:
    • Sideslip > 20 degrees
    • $\alpha_n >$ first -4g, to +7g
    • AoA to -10 degrees, then +20 degrees
    • Aircraft rolled 360 degrees
    • Vertical tail exceeded design load
    • All control surfaces operating at rate limits
    • Hydraulic & canard actuator failure indication

■ Side air-data probe was blanked by the canard at high alpha and sideslip
  – The wide input threshold passed the incorrect value through
  – Different channels took different paths through the control laws
  – Analog reversion for areas of flight envelope

■ Discussion:
  – What is the failure?
  – What is the fault?
  – Is it HW, SW other?
  – What was done right?
Multi-Channel Voting Pattern – Design Issues

- Approximate versus exact channel agreement
  - Voting requires a consistent notion of state in distributed FCUs
  - Preceding example based on approximate agreement
    - Architecture based on belief that coordination of sampling equates to coupling
    - Fault containment maintained by use of approximate comparison
    - Introduces coupling at ‘state synchronisation points’
    - Still have problems of non-deterministic behaviour
  - Alternative is exact agreement
    - Coordinate system data so each process gets same data at same time
      - A consistent notion of system state is required to be distributed (value + time)
    - Use the concept of system state to ‘decouple’ the system from the past
      - Decisions are made solely on current state + future inputs
    - Need to provide interactive (Byzantine) consistency e.g. resistance to failures in distributing the same data to different process nodes
    - ‘Slightly Out of Spec’ failures are an example of real world Byzantine faults
Case Study: Slightly Out Of Specification (SOS) Fault [15]

Byzantine (asymmetric) fault

Drive by Wire
Homogenous Redundancy Pattern

- Identical channels used to increase redundancy
  - Voting as for multi-channel voting channel
  - Improved robustness in channel dependability
  - Low development overhead
  - Follow self-confidence principle
    - A channel will assume its correct unless two other channels tell it its wrong
      - Ensures a correct channel will make correct decision
      - Only a faulty channel will make incorrect decision
  - Safety (determinism) can be assured by
    - Self confidence voting principle
    - Same principals as multi-channel
  - Problems
    - Full redundancy must be ensured to prevent SPOF
    - Can only protect against random hardware failures
    - Latent hardware OR software design faults will trigger multi-channel failure
    - Increase in recurring cost due to hardware duplication
Case Study: Homogenous Redundancy Pattern – Ariane 5

- **Accident**
  - On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. At about 40 s after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded [5].

- **Ariane 5 architecture**
  - Dual redundant channel homogenous architecture for INS system
  - Identical code contained same software fault
  - Both channels failed simultaneously

- **Design assumptions (Fault hypothesis)**
  - Common mode (design) faults not considered
    - Either software or hardware
  - Channels were designed to protect against random hardware failure
    - Homogenous architecture valid for this fault type
**Diverse Redundancy Pattern**

- Different channel designs used to reduce common cause vulnerability
  - Different but equal channels, or
  - Lightweight redundancy – reasonableness check
    - Particularly useful for complex algorithmic design
    - Trap instances where input exceeded a ‘reasonable’ value
  - Separation of monitor/actuator
    - Can continue to operate in face of design faults
  - Problems
    - Different may not be so different [11]
      - Designers tend to think alike
      - Small correlations in design, markedly decrease independence
    - Lightweight checks rely on additional recovery method
    - Small differences in rounding errors can lead to markedly different outputs
Monitor Actuator Pattern

■ Separate monitoring and actuating channels
  – Channel includes sensors
  – Monitor should ideally look at environment (system) not just actuator
  – Can exchange heartbeats between channels to confirm liveness
  – Think of it as an intelligent version of the interlock

■ Monitor channel identifies actuator channel failures
  – If monitor fails the actuator channel still operates
  – Satisfies SPOF resistance criteria
  – Monitor can be hardware/software or person (traditional)

■ Problems
  – Still have to eliminate common mode failures
  – Some increase in recurring cost
  – Still need some other (unspecified) error recovery function
Monitor Actuator Pattern – ‘Fire’ Control

- Controller
  - Initiates a ‘fire’ action for safety critical function
  - Also provides initialisation data to function

- Monitor
  - Provides safety interlock against inadvertent firing
  - Traditionally provided by operators making a release consent switch

- Safest action = no ‘fire’

![Diagram of Monitor Actuator Pattern – ‘Fire’ Control]
Watchdog Timer (WDT) Pattern

■ Probably the oldest safety behaviour
  – Receives periodic ‘heartbeats’ from subsystems
  – Restarts system if error (No heartbeat) detected
  – Can be periodic or sequential
  – Can be very simple (hardware or firmware)
  – Cheap to implement
  – Applied to HW/SW state machines

■ Is a safety device outside of the processor
  – No knowledge of what the processor is doing
  – Relies on being periodically reset
  – Can not catch all errors
  – Can catch certain lockup states or sets of states
    • Use keyed (event/calculated key) to protect against ‘endless loops’ errors [14]

■ Supports forward recovery strategies
Watchdog Timer (WDT) Pattern – Design Issues

■ WDT Size
  – Maximum size (time) bounded by time to prevent system damage
  – Minimum size bounded by false triggering margin + Software growth + test & diagnostics modes

■ Single event latch-up power supply events can fail the WDT

■ Record types of error and numbers
  – Use a saturating counter stored in non-volatile always powered memory
  – Record both the type and number of resets

■ Disabling the WDT
  – Disable function to avoid SPOF and allow diagnostics/test/software load
  – Automatic resumption upon completion of test or for unforeseen events
  – Control with a non-volatile hardware discrete i.e. a latching relay

■ Default WDT state
  – Default On – Simple, safer but may make recovery difficult
  – Default Off – makes software maintenance easier, but introduces SPOF
Safety Executive Pattern

- A centralised coordinator for safety
  - Detect faults & coordinate error recovery
  - Typically monitors for:
    • Watchdog timeouts
    • Software error assertions
    • Monitor/Actuator identified faults
    • Health state of the processor (memory, timer etc)
  - Advantages
    • Separate fault tolerance from application code
    • Layering of design (policy vs mechanisms)
    • Encapsulation
    • Use to keep operator informed of system state
    • Can make RTOS system calls to terminate faulted thread or entire process and restart it
  - Problems
    • There is some overhead and code size associated
    • Need to decide whether policy is application or kernel enforced
Safety Executive Pattern – Deployment Diagram

1. Identify safety policies in requirements specification
2. Specify as invariants (timing, values etc), use code fragments to enforce

Failure above kernel
Handled using safety policy
1. On request → ‘filter’ style check of validity ∧ legality for the system state OR
2. Upon detection of fault state → specify response for system states

Failure below kernel
OS service failure viewed by Kernel as other subsystem failure
Minimise failure impacts by minimising dependence of executive and application
Handled using WDTs, liveness checking, data coding etc
Safety Critical Network Pattern

- **Time driven star topology**
  - Uses a central hub interconnect
  - Time driven (either distributed/central)
    - Bus guardian (independent of FCU)
    - Global time synchronisation
  - Advantages
    - Far more resilient to random damage
      - Probability of 1 in N that node will be hit
    - Node damage will not disrupt bus comms
    - Hub can still isolate damaged nodes
    - Hub physical (separated) redundancy can eliminate it as a SPOF
    - Bus performance defined at design time
  - Problems
    - More wiring and additional hub
    - Time synchronisation (if distributed)
Software Mechanistic and Detailed Design

“Time is what prevents everything from happening all at once”

John Archibald Wheeler
Concurrent and Asynchronism

■ Why concurrency? [8]
  – The degree to which a system must be distributed
  – The degree to which a system is event-driven (and how it handles them)
  – The computational intensity of key algorithms
  – The degree of parallel execution supported by the environment

■ Concurrency introduces conflict and synchronisation issues
  – Asynchronous conflicts over resources will arise
  – It would be nice if everything could be synchronised, this won’t happen
  – Multiple dimensions to problem (pipeline processing, redundancy etc)
  – Synchronism and asynchronism are not dichotomous [8]
Software Concurrency – Shared Resources

TRUE CONCURRENCY

PSEUDO CONCURRENCY
Thread Interleaving required

Heavyweight threads (process)
1. Have own data space
2. More robust encapsulation
3. Inter task messaging
4. Common RTOS access

Lightweight threads (process)
1. Share process context (data)
2. Quicker communication
3. Less robust communication

Thread context
Process context
Shared -
Case Study: Shared Resources in C [16]

- Question: Which variables in a threaded C program are shared?
  - Hint: It’s not as simple as “global variables are shared” and “stack variables are private”
  - Requires answers to the following questions:
    - What is the memory model for threads?
    - How are variables mapped to memory instances?
    - How many threads reference each of these instances?

- Operationally this context model **may not be** strictly enforced:
  - Register values may be truly separate and protected
  - But if any thread can read and write the stack of any other thread...a hazardous conflict

- This mismatch between model and reality is one source of Heisenbugs

```
char **ptr;  /* global */
int main()
{
  int i;
  pthread_t tid;
  char *msgs[N] = {
    "Hello from foo",
    "Hello from bar"
  };
  ptr = msgs;
  for (i = 0; i < 2; i++)
    Pthread_create(&tid,
                   NULL,
                   thread,
                   (void *)i);
  Pthread_exit(NULL);
}

/* thread routine */
void *thread(void *vargp)
{
  int myid = (int)vargp;
  static int svar = 0;

  printf("[%d]: %s (svar=%d)\n",
         myid, ptr[myid], ++svar);
}
```

*A ‘Heisenbug’, as the name implies, exists until you try and test for it...
<table>
<thead>
<tr>
<th>Hazard Class</th>
<th>Example</th>
<th>Example countermeasures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exclusion Violation</td>
<td>Use of interrupts – Such that they break assumptions of atomic action</td>
<td>Decouple either:</td>
</tr>
<tr>
<td></td>
<td>Use of pointers – Such that they allow inadvertent concurrent access to the same memory</td>
<td>Physical – Separate processor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Logical – Separate address space</td>
</tr>
<tr>
<td>Race Condition</td>
<td>Preceding architecture example (macro) + critical sections of code</td>
<td>Disjoint variables</td>
</tr>
<tr>
<td></td>
<td>Two threads accessing same critical section</td>
<td>Global invariants</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Synchronisation</td>
</tr>
<tr>
<td>Calling thread unsafe function</td>
<td>Thread calls a thread unsafe function</td>
<td>Reliance on persistent state data across invocation</td>
</tr>
<tr>
<td>Deadlocked Resources</td>
<td>Unbounded priority inversion between task priority and priority of a mutual exclusion semaphore*</td>
<td>Redundant links (to/from)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Safe link updating</td>
</tr>
<tr>
<td>Time Starvation</td>
<td>High priority critical task starved of time by other lower priority non-critical tasks</td>
<td>Use of pre-emptive cyclic schedule</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Allocate fixed time budgets</td>
</tr>
<tr>
<td>Memory Starvation</td>
<td>Faulty thread spawns more and more kernel objects until memory exhausted</td>
<td>Allocation of fixed memory budgets to threads</td>
</tr>
<tr>
<td>Bad RTOS call**</td>
<td>Thread makes a ‘bad’ call to the kernel.</td>
<td>Opaque handlers for kernel objects</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Validate system call parameters</td>
</tr>
</tbody>
</table>

* But semaphores are also a control for exclusion hazards.
** Not covered in detail
Case Study: Hardware Protected Address Spaces

- **Process/heavyweight thread based RTOS**
  - Define a RAM structure for each process
  - 1 thread = 1 process at extreme (i.e. a safety kernel)
  - HW Memory Management Units (MMU) enforce protection
  - New logical address set switched in on context switch
    - MMU maps logical $\rightarrow$ physical address via the current map
    - MMU flags code attempts to access illegal logical addresses

- **Advantages**
  - Robust hardware enforced safety policy
  - Map independent pages to a logical address space for code, data & stack
  - Page of logical addresses after each threads stack unmapped to trigger MMU on thread overflow

- **Cons**
  - Overhead of memory access via lookup table
  - Inter-process communications

**Note.** Memory protection is also useful during development to pick up exclusion design faults by throwing a hardware ‘hard’ exception whenever a violation occurs.
Race Condition Hazard And Countermeasures

■ Shared resource violation hazard [16]
  – Thread trajectory is safe IFF it doesn’t touch any part of an unsafe region
  – Race condition hazard exists in unsafe region
  – Countermeasures
    • Disjoint variables
    • Weakened assertions
    • Global invariants
      • Software (Dekker’s algorithm)
      • Hardware (test and set atomic operation)
      • OS solution (semaphores)
      • Program Library solution (monitors)
    • Distributed/partitioned OS solution (IPC)

■ Also, need to consider the hazards of calling ‘thread unsafe’ functions*:
  – Failing to protect shared variables (see above)
  – Relying on persistent state across invocations
  – Returning a pointer to a static variable
  – Calling thread-unsafe functions

*Note that these hazards are not disjoint.
Race Condition Hazard And Countermeasures

■ Semaphores can interact with scheduling
  – ADA 83 only allowed FIFO synchronisation*
  – Can lead to priority inversion, deadlock or starvation

■ May also need to consider
  – Atomicity (Dekker’s algorithm)
  – multiple resources (tokens)
  – parallel processors (spin-locks)

■ Non-atomic action hazard
  – What if there is a task switch at the exact moment between falling through a ‘wait()’ loop and setting the semaphore?
  – Countermeasures:
    • Atomic wait() loop by disabling task switching before the test is done and re-enabling it later
    • Use condition variables to sleep tasks
    • Use a barrier condition variable

*Fixed for ADA 95, but a good example of language semantics which can increase the occurrence of design faults.
Race Conditions Hazard And Countermeasures

Example: Linked list ready queue with no critical regions [8]

HAZARD CONTROL
ENFORCE A SAFE SEQUENCE OF ACTIONS

Question: What type of countermeasure is this?
## Deadlock Hazard And Countermeasures

- Deadlock IFF 1 to 4 conditions true
  1. Mutual exclusion
  2. Hold and wait
  3. No pre-emption
  4. Circular wait

- Countermeasures
  - Eliminate* or
  - Control/reduce by detecting and recovering from deadlock

- Note
  - The further down the precedence is countermeasure the more complex the solution
  - Self deadlock is also a possibility
    - Make semaphoring recursive

<table>
<thead>
<tr>
<th>No.</th>
<th>Type</th>
<th>Countermeasure</th>
</tr>
</thead>
<tbody>
<tr>
<td>②</td>
<td>Eliminate</td>
<td>Eliminate hazard by enforcing system policy requiring release of one resource if calling another resource</td>
</tr>
<tr>
<td>③</td>
<td>Eliminate</td>
<td>Release held resources and request total of held + new resources</td>
</tr>
<tr>
<td>④</td>
<td>Eliminate</td>
<td>Eliminate hazard by blocking all other tasks from continuing until task releases critical resource</td>
</tr>
<tr>
<td>④</td>
<td>Reduce</td>
<td>Dynamically examine the resource-allocation state to exclude a circular-wait condition**</td>
</tr>
<tr>
<td>④</td>
<td>Control</td>
<td>Control hazard by use of a timed rendezvous to break a deadlock</td>
</tr>
</tbody>
</table>

* Conditions 2 & 4 are usually easiest to negate.

**A complex countermeasure that may result in process starvation due to recurring rollback.
Priority Inversion Hazard And Countermeasures

- Undermines efforts to determine schedulability of code
  - RMA assumes highest priority threads execute first
  - Increases complexity and non-deterministic behaviour

- Special case of deadlock
  - A higher priority thread is blocked by a mutex (or binary semaphore)
  - Mutex is owned by lower priority thread
  - Lower priority cannot run because of medium priority task

- Countermeasures
  - Priority inheritance mutex
    - Low priority thread gets highest priority (of using threads) when it uses a mutex
    - Allows it to run
    - Relinquishes when it releases the mutex
    - Cons: Doesn’t prevent chained blocking or a task suspending itself while holding a resource locked (not good in a safety critical application)
  - Priority ceiling protocol
Time Starvation Hazards

■ Many strategies for handling concurrent thread scheduling
  – FIFO run to complete
  – Non-preemptive task switching
  – Time slicing
  – Cyclic executive. Popular in safety critical applications
    • Statically ordered threads
    • Run to completion semantic
    • Advantage: very simple to write
    • Disadvantage: tightly coupled to the timing & feature sets
  – Priority based preemption. Popular in RTOS applications
    • Time allocation problem for priority/pre-emptive scheduling
    • Threads of same priority are allocated same time ‘slice’

■ Hazards
  – No provision for guaranteeing processor time for safety critical threads
  – No way of informing the scheduler of the criticality of a specific thread
Time Starvation Countermeasures

■ **Time starvation hazard countermeasures**
  – Weight threads so that they always receive a fixed ratio
  – If a thread spawns another of same priority it must share this time
  – Ensure critical threads get a fixed ‘safe’ amount of time to execute

■ **Scheduler is ignorant of context of threads**
  – Solution: Provide heavyweight (process) scheduling
    • Only its threads can run
    • System designer provides pre-emptive time partitioning
    • Provide a ‘background’ partition for when a process does not contain threads in a waiting state and use it for BIST etc
  – Cons: If partition scheduling is ‘on top’ of a RTOS then switching will require halting all threads in a partition resulting in a linear increase in context switch overhead with the No. of threads
Memory Starvation Hazard And Countermeasures

- Most RTOS allocate memory for process control blocks from a central store
  - Kernel allocates memory for thread, semaphore or other kernel object
  - A design fault in a thread creating kernel objects can lead to memory exhaustion

- Memory quota countermeasure
  - Preemptive allocation of memory to processes
  - A process must use its own allocated memory to create new kernel objects (if quota is 30% then 15% (half of quota) might be allocated)
  - Prevents exhaustion (e.g. the Unix ‘fork bomb’)

- Local quota dynamic allocation countermeasure
  - If each process allocates memory from its own quota NOT central store then dynamic allocation for non-critical tasks can be tolerated
### Hazards Introduced By Asynchronous Behaviour

<table>
<thead>
<tr>
<th>Hazard Class</th>
<th>Example</th>
<th>Example countermeasures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re-initialisation</td>
<td><strong>Example</strong> Task is suspended by interrupt and re-started in with</td>
<td>Re-initialise sensitive variables</td>
</tr>
<tr>
<td>hazard</td>
<td>variables set to the values at suspend.*</td>
<td></td>
</tr>
<tr>
<td>Broken atomic</td>
<td><strong>Example</strong> Semantics of an atomic action (load, store et al) is not</td>
<td>Mask interrupts</td>
</tr>
<tr>
<td>action hazard</td>
<td>actually enforced in practice.</td>
<td></td>
</tr>
<tr>
<td>Masked interrupt</td>
<td><strong>Example</strong> A masked kernel task duration is longer than safety critical</td>
<td>Log and postpone</td>
</tr>
<tr>
<td>hazard</td>
<td>data latency required by the masked interrupt</td>
<td>interrupts</td>
</tr>
</tbody>
</table>

**Masking and interrupt latency issues**
- Not all are created equal (50 Hz timer tick versus flight control update)
- If the masking for the longest kernel task > than data latency then a hazard exists
- If you partition functionality then you will need to partition interrupts
- Alternative (more difficult) approach is to never disable an interrupt
  - Log then postpone handling of low priority interrupts until critical task finished
  - Use for RTOS kernel calls in particular
  - **Assumption** is that all kernel calls are short or can be restarted

* Can also be a problem for software which is ‘multi-pass’ in nature with successive calls of the software invoked as a functional sequence.
Asynchronous Behaviour Hazards Countermeasures

■ Example [8]
  – ‘X’-by-wire system used floating point operations in its interrupt handling routines
  – Programmer made sure that critical regions were observed and all used floating point registers were saved and restored
  – But each floating point register was 2 words and the load and store operations were not atomic!
  – If the interrupt occurred between the first and second word of a load or store and the interrupt handler used that register, the second word would be lost

■ Countermeasure:
  – Mask interrupts during critical tasks such as manipulating internal data structure esp. for kernels
  – If interrupt is a low priority log and respond when task complete

■ This is a good example of how non-deterministic behaviour introduces both hazards and further complexity
Case Study: 1981 STS-2 Preflight Shuttle Software Freeze

- Transatlantic abort simulation being run [7]
  - Prior to fuel dump task in abort sequence all 4 mission computers froze (same software load)
  - OFP code branched to non-program space
  - Resultant OS short loop trying to service the interrupts
  - During the abort fuel is dumped
    - Control module is multi-purpose code used to dump fuel
    - Could be called at multiple points in flight
    - Had been invoked before & interrupted (by crew) BUT some counters weren’t re-initialised
    - Module code contained a GO-TO statement expecting counter value 1..X
    - When module called during abort GO-TO got value >X and jumped to non-program space

- The fault resembled other more benign faults
  - Always involved a module with N>1 processing passes to finish
  - All had been interrupted and did not work correctly when re-started

- A review of all the code was conducted
  - 17 discrepancies found one of which was a serious launch related hazard
  - Loss of safety checking on the solid rocket boosters during ascent, patched for STS-2 flight
Hazards Introduced By Loss of Data Integrity

<table>
<thead>
<tr>
<th>Hazard Class</th>
<th>Example</th>
<th>Example countermeasures</th>
</tr>
</thead>
<tbody>
<tr>
<td>External data corruption</td>
<td>EMI can affect electronics and can cause single bit hardware upsets</td>
<td>Utilise multiple bits for critical data</td>
</tr>
<tr>
<td>Unauthorised access</td>
<td>Unauthorised software module (or person) accesses critical data and changes it to an incorrect value</td>
<td>To-From programming technique Security kernel and access control list</td>
</tr>
<tr>
<td></td>
<td>Also need to consider ‘jumping into the middle’ of critical code scenarios</td>
<td></td>
</tr>
</tbody>
</table>

- Not just a real time system problem
  - Example: Red cross blood bank software error (a race condition) caused status of blood (critical data) to be corrupted

- Often there is a tacit assumption that data = software
  - But unlike static code, data is dynamic (both mission and configuration)
  - Data is potentially spread through the software
  - Makes safety verification of data potentially difficult

- Data safety can be addresses at both architecture and detailed levels
Data Access Hazard Countermeasures

- Data hiding (decoupling) countermeasures
  - Information should only be exposed to those program parts which need it
  - Create objects which maintain data state and operations on that state
    - Strongest form is an ‘encapsulation kernel’ to strictly controls calls
    - Reduces the probability of
      - Accidental corruption of data
      - Propagation of error by presence of data ‘firewalls’
    - Reduces programming and reviewer errors
  - Access control constraints
    - Access control list – Called object oriented
    - Capabilities list – Calling object oriented
    - Should be; always invoked, non-bypassable, tamper proof and evaluable
  - Reference monitor
  - Software batons
Data Corruption Hazard Countermeasures

- Data corruption countermeasures
  - Based on redundancy of critical data
  - For example:
    - Use multiple bits $N > 4$ to represent armed and safe states
    - Do not use logic 0 or 1
    - Use a pattern for the safe state that cannot, due to a one or two bit error, represent the armed state
    - Monitor for a pattern other than the two allowed
  - Use linked lists with backwards and forward pointers
  - Store data in multiple locations and compare
    - Store one copy as one’s complement form and compare to other
  - Use check bits and cyclic redundancy checks
  - Use sub-range constraints
This pseudo-code represents how the exposure to a hazardous state (object in an armed state) can be minimised

- Entry is controlled to the safety critical object via a top entry baton that checks correctness of call
- Object is deliberately kept small (simple)
- Safety critical data is hidden within the object
- NB. Only single point access to public safety critical data should be allowed
- The baton is checked twice for software in order to confirm that the software has legally branched to the object
- The time that the object is active is minimised (pseudo atomic)

```
MODULE SC123()
ENTER
CHECK TOP ENTRY BATON FOR LEGITIMATE ENTRY
---
PERFORM MINIMUM PRECURSOR TASKS
---
READY ARM HARDWARE
---
CHECK TOP ENTRY BATON FOR LEGITIMATE ENTRY
TURN ON HARDWARE
CONTINUE
```
## Hazards Introduced By Transmission of Data Errors

<table>
<thead>
<tr>
<th>Hazard Class</th>
<th>Example</th>
<th>Example countermeasures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data transmission corruption hazard</td>
<td>EMI on bus causes corruption of bits within transmitted message</td>
<td>Use a transmission protocol designed for critical applications*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Checksums and cyclic redundancy checks for messages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Split signals and run via separate paths (redundant bus)</td>
</tr>
<tr>
<td>Wrong addressee hazard</td>
<td>Error in dynamic allocation of bus addresses leads to messages for front ABS brakes to be sent to rear</td>
<td>Verification response from recipient</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Restricted addressee list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Separate analog consent signal</td>
</tr>
<tr>
<td>Babbling idiot hazard</td>
<td>Failure of FCU causes it to ‘talk out of turn’ on TDMA bus</td>
<td>Bus Guardian or Master/Slave bus protocols</td>
</tr>
</tbody>
</table>

- Error propagation can defeat assumptions of independence made as part of the safety case or fault hypothesis

* If using a general communication protocol standard consider the use of a subset optimised for safety
Case Study: Safety Critical MIL-STD-1760C Signals

- MIL-STD-1760 defines implementation requirements for the Aircraft /Store Electrical Interconnection System (AEIS)
  - Implements a subset of MIL-STD-1553 messaging
  - Specified safety critical messages
    - Store control message
    - Store monitor
    - Store description
  - Fixed message address space
  - Protocol checks on safety critical messages
    - Verification of checksum (Rotated Modulo 2)
    - Verification of message header
    - Verification of critical authority and control batons
  - Separate power supply for safety critical functions
  - Independent of software safety consent signal
    - Used in conjunction with 1553 commands
    - Release consent originates separate to the software but may be software steered
    - Release consent NOT used for non-safety functions
Hazards Introduced by Error Recovery Functions

- Error recovery should detect error states and minimise or reverse their effects
  - An error can be an internal message or a control output to the environment
  - Error recovery software is usually asynchronous (responsive)
  - Non-critical tasks can be halted BUT critical tasks may have to continue
  - Most likely to run during highly critical events and/or hardware failures
  - **Can potentially violate safety constraints of mission application**
  - Nor can we assume atomic actions if external hardware is involved

- For safety critical sequences need to verify that error recovery
  - Continues to satisfy domain level precedence constraints (sequences)
  - Maintains data race condition (read/write) constraints
  - Satisfies timing constraints
    - For example, ADA & C++ native error (exception) handling via unwinding the stack is inherently unpredictable
Error Recovery Behaviour Pattern

■ First lane represents standard flow (unmodified)
■ Additional lanes added to address exceptions
  – Exceptions handled locally with recovery to local task OR
  – Execute an exception exit of the composition
  – Can be applied to external faults (noise, spoofing etc)
  – Hierarchy of error handling from local level to system

■ Recovery strategies
  – Backwards
    • Recovery blocks, transactions & checkpoints
    • N version programming
  – Forward
    • Robust data structures (detect and correct)
    • Redundant pointers
    • Error coding
    • Dynamic alteration of control flow (delay non-critical tasks)
    • Coast across single cycle errors
  – Compensation
    • Safe hold states
    • Voting and averaging of values
Error (Exception) Monitor Design Pattern [14]

- Exception Safe Class (abstract)
  - Collaborating objects subclass to this class

- Exception
  - Reifies abstract exception class
  - Classifies exception
  - Associates to the handler

- Exception handler
  - Handles exception, known to exception at creation
  - Can handle multiple exceptions
  - Optional association to client to allow it to invoke operations (HW safe holds etc)

- Exception monitor (mediates)
  - Logs exceptions*
  - Passes exception to global handler if not handled
  - Destroys exception if successful

- Log
  - Records exceptions

- Global exception handler
  - Handles exceptions that cannot be handled locally within the collaboration

This pattern supports hierarchical error recovery and eliminates the issue of unpredictable exception handling times in native ADA and C++ handling mechanisms.

*Important to allow detection of faults masked by error recovery
Case Study: Fault Detection and Error Recovery - Apollo 11

Apollo 11 1202 and 1201 alarms during landing

Alarm cause

- The Guidance & Control System (GCS) was an interrupt & priority scheduled real time multi tasking OS
- Radar was turned on for landing (at the last minute) which was not simulated (external fault)
- Repeated jobs to process rendezvous radar data were scheduled by the GCS
- There was insufficient memory to run task, causing an alarm/abort (fault detection)
- The executive then rebooted and re-initialised the computer (error recovery) ...

- Software was fault tolerant by design
  - Graceful degradation in face of system overloads
  - Fault tolerance & error recovery was extensively tested prior to mission

- Pilots could manage fault indirectly by minimising other processor loads (e.g. not requesting pitch over TTG from computer)
## Hazards Introduced by Algorithm Design

<table>
<thead>
<tr>
<th>Hazard Class</th>
<th>Example</th>
<th>Example countermeasures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm fault hazard</td>
<td>Numerical method sensitivity to initial start points</td>
<td>Inversion of calculation</td>
</tr>
<tr>
<td></td>
<td>Method may be inherently inaccurate</td>
<td>Use a lightweight ‘second guess’ algorithm for more complex cases</td>
</tr>
<tr>
<td></td>
<td>Converting from analog to discrete equations can introduce inaccuracies</td>
<td>Use domain ‘reasonable’ values</td>
</tr>
<tr>
<td></td>
<td>Rounding errors in floating point calculations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>You may have violated pre-conditional invariants</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vulnerability to equation asymptotes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Errors in programming for complex equations</td>
<td></td>
</tr>
</tbody>
</table>

- Deterministic performance of algorithms may be more important than efficiency
- Simple less accurate algorithms may be better than more complex algorithms which are harder to prove correct
- Use reasonableness checks, e.g. “if your software says your breaking light-speed, you’re probably not…”
Case Study: Flight Controls Fail At Supersonic Transition

- An operational fighter underwent tests after a mod.
  - One particular weapon test required significant telemetry
  - The telemetry showed all flight computers failing at Mach 1
    - Ceased to function and then restarted at a certain airspeed
    - The aircraft had sufficient momentum and mechanical control of other systems so that it "coasted" through this anomaly unnoticed
  - One flight control equation asymptotically approached infinity at Mach 1

- System & software engineering did not understand this
  - The software specification did not include the limitations of the equation describing a physical science event
  - The computer hardware accuracy was not considered in the limitations of the equation
  - The various levels of testing did not validate the computational results for the Mach 1 portion of the flight envelope
Software Implementation

“C makes it easy to shoot yourself in the foot. C++ makes that harder but when you do, it blows away your whole leg.”

Bjarne Stroustrup, inventor of C++

“C treats you like a consenting adult, Pascal like a naughty child and Ada like a criminal”

Anon.
Software Implementation

- Implementation needs to address safety
  - Not enough to develop a safe design
  - Needs to translate into the source code and compiled software object
  - Similarities with production quality control in other critical domains

- Implementation issues
  - Language choice
    - Exceptions versus error codes
  - ‘Safe’ language subsets
  - Unoccupied memory, dead and co-resident code
Language Choices

■ Software coding faults
  – Language selected can affect human errors (C++ versus ADA)
  – People commit programming errors
  – Language = a performance shaping factor

■ Some characteristics that improve safety
  – Strong compile time checking
  – Strong run time checking
  – Support for encapsulation and abstraction
  – Exception handling (Ada vs C++ vs Pascal)
    • Idealy hierarchical and class based
  – Safe subsets

■ Compiler issues
  – Validated
  – Pragma free

<table>
<thead>
<tr>
<th>Application</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boeing 747-400</td>
<td>75 languages</td>
</tr>
<tr>
<td>Boeing 777</td>
<td>Ada subset</td>
</tr>
<tr>
<td>ESA – mission critical</td>
<td>Ada</td>
</tr>
<tr>
<td>Car ABS</td>
<td>Assembler</td>
</tr>
<tr>
<td>Train controls</td>
<td>Ada</td>
</tr>
<tr>
<td>Medical systems</td>
<td>Ada + Ass</td>
</tr>
<tr>
<td>F-18 Mission Mngt</td>
<td>Ass + CMS</td>
</tr>
<tr>
<td>F-18 Stores overlays</td>
<td>Ada x-compiled to Ass</td>
</tr>
<tr>
<td>Nuclear reactor protection system</td>
<td>Ass moving to Ada</td>
</tr>
</tbody>
</table>

See Annex B
For a full listing of error prone semantics
Hazards Introduced By Dead And Co-Resident Code

■ Dead code
  – The ‘...and do nothing more’ hazard
    • Ariane 5 and Titan II accidents
  – A COTS issue where ‘unwanted/unknown’ functions are also provided

■ Counter measures
  – Generally reduce hazardous conditions:
    • Remove redundant elements of the design and code
    • Initialise unwanted processor memory with a pattern that will cause a safe state to occur if the software ‘jumps into the unknown’
    • Overlays (if used) should occupy same amount of memory and unused memory should be filled with a safe jump pattern

■ Co-resident code
  – What happens when you have multiple software variants?
  – How do you manage the different configurations?
  – What tells the software that it’s in a particular configuration/site/tail no.?
Hardware as Software
Hardware as Software [12]

- VHDL + FPGAs allow designers to treat design as software
  - VHDL design represented by logical equations from which a design is synthesised
    - Unlike schematics not WYSIWYG
    - Connectivity hard to follow in VHDL files
      - Especially true for translations from schematics
    - Signal flows through sequential circuits can be hard to follow
    - Interactions between logical blocks can be difficult to understand
    - Spelling errors → undetected circuit errors
  - Over reliance on ‘simulations’ versus knowing what the circuit is doing
    - Especially for those FPGAs for which there is no reprogramming penalty
  - Rather than designing by analysis, designers simply “try” design concepts

- Sound familiar?
Key Points and Issues
Key Points and Issues

■ Key points
  – Remember the precedence of countermeasures
  – Always base the design on the requirements of the safety case
  – A layered set of independent measures supporting a defence in depth approach will be usually be the most successful (and verifiable)
  – It is important to document your assumptions as part of the design
  – We can introduce new hazards with our countermeasures
  – No architecture/design/implementation is perfect
  – Design faults are to be expected as are random physical failures
  – Often we introduce new hazards as we control old ones
  – Think in terms of hazard and countermeasure
  – We’ve focused on the ‘difficult hazards’ of design

■ Question: Do you know what your software is doing right now?
References

1. Jaffe, M., Architectural approaches to limiting the criticality of COTS (or other re-used software) components in avionics systems, Computer Science /Engineering Department, Embry-Riddle University, Prescott, Arizona.
5. Buchanan, M., Ubiquity, the science of history … or why the world is simpler than we think, Phoenix press, 2000.
References


16. CS 475, Lecture 5: Programming with threads, Geoge Mason University, Computer Science Dept.
Annex A

Software Hazards
Software Hazards

- This annex collates software hazards identified in this module
- Hazards are grouped by class
  - Synchronisation hazards
  - Real Time Operating System (RTOS) hazards
  - Data hazards
  - Algorithm hazards
  - Dead code hazards
  - Transmission hazards
  - Co-resident hazards
- Within these classes hazards are not disjoint and may interact or lead to other hazardous states
- As with all hazard checklists this list is not complete
  - Use it as a start point for further analysis
  - Tailor it to your application and domain
## Software Hazards

<table>
<thead>
<tr>
<th>HAZARD</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>■ Synch hazards</td>
<td></td>
</tr>
<tr>
<td>Re-initialisation hazard</td>
<td>A critical software thread is capable of being interrupted by another without re-initialising the persistent data used for internal task control leading to an unsafe control output</td>
</tr>
<tr>
<td>Broken atomic task hazard</td>
<td>A software task is assumed to be atomic (un-interruptible) but can be broken by an interrupt leading to data corruption and a resultant unsafe control output</td>
</tr>
<tr>
<td>Masked Interrupt hazard</td>
<td>A safety critical interrupt is masked during a critical task leading to a failure of the software to respond and an unsafe control output</td>
</tr>
<tr>
<td>Shared data hazard</td>
<td>Two (or more) concurrent threads have mutual AND non-exclusive access to shared data used by a safety critical process resulting in an unsafe control output</td>
</tr>
<tr>
<td>Persistent state reliance</td>
<td>A safety critical thread calls a function which implicitly relies on persistent state data across the invocations</td>
</tr>
<tr>
<td>hazard</td>
<td></td>
</tr>
<tr>
<td>Pointer return hazard</td>
<td>Returning a pointer to a static variable.</td>
</tr>
<tr>
<td>Calling thread-unsafe</td>
<td>A safety critical thread function calls one thread-unsafe function making the entire function thread-unsafe</td>
</tr>
<tr>
<td>function hazard</td>
<td></td>
</tr>
<tr>
<td>Deadlock hazard</td>
<td>A safety critical thread is placed in a locked condition indefinitely resulting in a service failure of a safety critical output</td>
</tr>
<tr>
<td>Race condition hazard</td>
<td>The relative speed of two threads affect a safety critical outcome resulting in an unsafe control output</td>
</tr>
</tbody>
</table>
# Software Hazards

<table>
<thead>
<tr>
<th>HAZARD</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time starvation hazard</td>
<td>A safety critical thread is starved of process time resulting in a service failure of a safety critical output</td>
</tr>
<tr>
<td>Memory starvation hazard</td>
<td>A safety critical thread is starved of memory resulting in a service failure of a safety critical output</td>
</tr>
<tr>
<td>Error recovery hazard</td>
<td>An error recovery process can violate the safety constraints placed upon a safety critical thread (timing, precedence, race conditions etc) leading to an unsafe control output</td>
</tr>
<tr>
<td>■ RTOS hazards</td>
<td></td>
</tr>
<tr>
<td>System call error</td>
<td>A thread incorrectly accesses the RTOS kernel shared by a safety critical thread leading to a kernel error state and an unsafe control output</td>
</tr>
<tr>
<td>■ Data hazards</td>
<td></td>
</tr>
<tr>
<td>External data corruption hazard</td>
<td>A hardware fault or environmental input (radiation, EMI) leads to corruption of safety critical data and an unsafe output</td>
</tr>
<tr>
<td>Unauthorised access hazard</td>
<td>Safety critical data is accessed by an unauthorised thread leading to data corruption and an unsafe control output</td>
</tr>
<tr>
<td>■ Algorithm hazards</td>
<td></td>
</tr>
<tr>
<td>Algorithm failure hazard</td>
<td>An algorithm design fault (rounding error, asymptote, invariant violation) in a safety critical calculation leads to calculation of an unsafe control output value</td>
</tr>
</tbody>
</table>
# Software Hazards

<table>
<thead>
<tr>
<th>HAZARD</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Over/underflow hazard</td>
<td>An algorithm design fault (rounding error, asymptote, invariant violation) leads to an overflow/underflow condition and violates the safety constraints placed upon a safety critical thread (timing, precedence, race conditions etc) leading to an unsafe control output</td>
</tr>
<tr>
<td>Dead code hazards</td>
<td></td>
</tr>
<tr>
<td>Dead code hazard</td>
<td>Dead code inadvertently interacts with safety critical live code during program execution leading to an unsafe control output.</td>
</tr>
<tr>
<td>Transmission hazards</td>
<td></td>
</tr>
<tr>
<td>Critical data corruption</td>
<td>Critical data is corrupted during transmission leading to an unsafe control output.</td>
</tr>
<tr>
<td>Wrong addressee hazard</td>
<td>Critical data is sent to the wrong addressee leading to an unsafe control output.</td>
</tr>
<tr>
<td>Babbling idiot hazard</td>
<td>A communication node fails active leading to failure of a communication bus carrying safety critical data or commands.</td>
</tr>
<tr>
<td>Co-resident hazards</td>
<td></td>
</tr>
<tr>
<td>Inappropriate version hazard</td>
<td>An inappropriate version of the software is activated for the particular platform leading to unsafe control outputs.</td>
</tr>
</tbody>
</table>
Annex B

Error Prone Coding Semantics
Error Prone Coding Semantics

- Certain constructs are consistently present in design faults
  - The way in which a language implements these semantic constructs are performance shaping factors
  - Many errors appear to relate to underlying human errors types

- Empirical evidence
  - Logic, timing, variable, data access and exceptions semantics/syntax errors (slips) 46%
  - Incorrectness (mistakes) / incompleteness (lapse) 30%
  - Violations of specification or design standard (violation) 24%

- Countermeasures must vary as well

<table>
<thead>
<tr>
<th>Design Error Type</th>
<th>% of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect/incompleteness</td>
<td>30</td>
</tr>
<tr>
<td>Deviated from specification deliberately</td>
<td>13</td>
</tr>
<tr>
<td>Errors in logic</td>
<td>13</td>
</tr>
<tr>
<td>Violation of standards</td>
<td>11</td>
</tr>
<tr>
<td>Errors in data access (SW)</td>
<td>11</td>
</tr>
<tr>
<td>Invalid Timing</td>
<td>10</td>
</tr>
<tr>
<td>Improper Interrupt (Exception) Handling</td>
<td>4</td>
</tr>
<tr>
<td>Wrong Values of Variables</td>
<td>4</td>
</tr>
</tbody>
</table>

Source: 1970s TRW study
# Error Prone Coding Semantics

<table>
<thead>
<tr>
<th>SEMANTIC</th>
<th>TYPICAL ERRORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointers</td>
<td>Pointers referring to the wrong memory areas can corrupt data, especially when raw pointers are used in the presence of exception handling</td>
</tr>
<tr>
<td>Aliasing</td>
<td>More than one pointer aliasing for storage can make programs difficult to understand and change</td>
</tr>
<tr>
<td>Control transfers</td>
<td>Can introduce inadvertent interactions in the code</td>
</tr>
<tr>
<td>Default and implicit typing</td>
<td>Requires consistent use of the typed variable and can lead errors which will not be checked by the compiler or at run time.</td>
</tr>
<tr>
<td>Use of ‘global variables’</td>
<td>We may introduce inadvertent connections and expose critical data to inadvertent corruption</td>
</tr>
<tr>
<td>Overloading operators</td>
<td>Operators become context specific which can introduce subtle errors that are very difficult to find</td>
</tr>
<tr>
<td>Conditional statements</td>
<td>Statements may not be logical tautologies, humans have difficulty in thinking about and testing for ‘null’ or alternate cases</td>
</tr>
<tr>
<td>Dynamic memory allocation</td>
<td>Run-time allocation can cause memory overflow as well as introducing problems of memory management such as garbage collection and fragmentation</td>
</tr>
<tr>
<td>Inheritance</td>
<td>Code is not localised and can result in unexpected behaviour when changes are made and problems of understanding</td>
</tr>
<tr>
<td>Recursion</td>
<td>Errors in recursion can cause memory overflow</td>
</tr>
<tr>
<td>Floating point numbers</td>
<td>Inherently imprecise and may lead to invalid comparisons</td>
</tr>
</tbody>
</table>
Annex C

Software Countermeasures
# Software Countermeasures

<table>
<thead>
<tr>
<th>COUNTERMEASURE</th>
<th>STAGE</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Elimination</strong></td>
<td></td>
<td>Eliminate the hazardous state or the losses associated with it</td>
</tr>
<tr>
<td>Substitution</td>
<td>A</td>
<td>Substitute a hardware function for a software one</td>
</tr>
<tr>
<td>Simplification</td>
<td>A</td>
<td>A priori scheduling</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>Use of polling rather than interrupt driven architectures</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Exclusive mode definitions</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Use state transition tables, using only current state, to make control decisions</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Eliminate effects of common hardware failures upon the software</td>
</tr>
<tr>
<td><strong>Decoupling</strong></td>
<td>A</td>
<td>Modularise safety critical code</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>Minimise number of safety critical modules and resultant module interfaces</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>Safety kernel or executive to protect safety critical functions</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>Use a firewall technique (logical or physical) to protect critical software</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Use of disjoint variables to eliminate synchronisation hazards</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Use weaker assertions to reduce synchronisation requirements</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Use of global invariants</td>
</tr>
<tr>
<td></td>
<td>I</td>
<td>Don’t use self modifying code</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Separate input/output ports for critical and non-critical code</td>
</tr>
</tbody>
</table>

*A = Architecture, D = Design and I = Implementation*
# Software Countermeasures

<table>
<thead>
<tr>
<th>COUNTERMEASURE</th>
<th>STAGE</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human error</td>
<td>I</td>
<td>Choose a language whose semantics support error free coding</td>
</tr>
<tr>
<td><strong>REDUCTION</strong></td>
<td></td>
<td><strong>Reduce the likelihood of a hazard (or its causal factors) occurring</strong></td>
</tr>
<tr>
<td>Controllability</td>
<td>D</td>
<td>Design processes to be incremental with exits to a safe state at each step</td>
</tr>
<tr>
<td></td>
<td>A/D</td>
<td>Provide backup modes of operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Provide decision aids to the operator</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>Provide a channel monitor</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Monitor with a hierarchy of self checks hardware → code → Audit → supervisory</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>Use a safety kernel/executive to coordinate checking</td>
</tr>
<tr>
<td>Barriers</td>
<td>A</td>
<td>Apply security kernel to protect safety critical code and data*</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Utilise software batons to ensure valid access</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>Use a separation kernel to enforce encapsulation and control communication</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Use condition variables to sleep tasks</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Robust behaviour in the face of erroneous or out of spec external faults</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Use semaphores or rendezvous objects to lock shared parts of code</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Use opaque handlers for kernel objects to defeat bad RTOS calls</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Use ‘come from’ batons to confirm correct sequence and calls of a module</td>
</tr>
</tbody>
</table>

* Is also a useful mechanism with which to justify negative properties, i.e. things that should not happen.
## Software Countermeasures

<table>
<thead>
<tr>
<th>COUNTERMEASURE</th>
<th>STAGE</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimise failures</td>
<td>A</td>
<td>Homogenous or diverse control channel redundancy</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>Monitor channel redundancy</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Data redundancy, check bits, cyclic redundancy, message sequence numbers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>… duplicate pointers, send/receive addresses</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Control channel redundancy, reasonableness checks, N version voting</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Backward recovery via recovery blocks, transactions &amp; checkpoints</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>Backward recovery via n-version programming</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Forward recovery via robust (redundant) data structures</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Forward recovery by altering the flow of control to allow critical tasks to continue</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Compensation schemes</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Forward recovery via Coast across (ignore) single cycle errors</td>
</tr>
<tr>
<td>CONTROL</td>
<td></td>
<td><strong>Reduce the likelihood of a hazard (if it occurs) propagating to an accident</strong></td>
</tr>
<tr>
<td>Minimise exposure</td>
<td>I</td>
<td>Set critical flags as close as possible to protected code</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Maximise time spent in safest state, minimise time spent in hazardous states</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Remove redundant code</td>
</tr>
</tbody>
</table>
## Software Countermeasures

<table>
<thead>
<tr>
<th>COUNTERMEASURE</th>
<th>STAGE</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail Safe Design</td>
<td>A</td>
<td>Watchdog timer and heartbeat</td>
</tr>
<tr>
<td></td>
<td>I</td>
<td>Initialise unused memory with a ‘jump to safe’ state command</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Use Spin-locks for parallel processors and time-out semaphores</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Provide fallback degraded or ‘never give up’ operational modes</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Deadlock detection and recovery (process rollback) algorithms</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Provide a capability to recover from degraded modes</td>
</tr>
</tbody>
</table>
Annex D

Fault Hypothesis Definition
Fault Hypothesis

The following questions define the fault hypothesis

- Fault-Containment Unit (FCU): Which are independently affected by faults

  - Fault modes (dimension and effect)
    - **Dimension**: Value, timing or spatial (physical destruction)?
    - **Manifest**: Self evident and reliably detectable e.g. ‘fail silent’
    - **Symmetric**: Same for all observers e.g. an ‘off by one error’
    - **Arbitrary**: May be any value and asymmetric, i.e. perceived differently by different observers*

  - **Frequency of faults**: What is their arrival rate

- Number of Faults: What total number of failures can be tolerated

- Detection: How are faults detected? How long is the detection latency?

- State Recovery: How long does it take to repair corrupted state (in case of a transient fault)?

The fault hypothesis can affect the design markedly, e.g. if we assume strict ‘fail silent’ faults then error detection is not needed

*e.g. a ‘slightly out of spec’ error is perceived differently by different nodes due to a slow signal edge rise or closeness to a clock edge.