These tutorials are a simplified introduction, and are not sufficient on their own to achieve system safety. You are responsible for the safety of your system.
Are You Using A Good Safety Pattern?

- Anti-Patterns for Safety:
  - Mixed-SIL software without isolation
  - No redundancy for high criticality functions
  - Fault detection vs. availability confusion

- Appropriate pattern depends on the system
  - Cross-checked redundancy for fault detection
  - Standby redundancy for availability
  - Separation of Low SIL and High SIL functions
    - Each SIL must have its own isolated CPU
    - For discussion:
      » SIL 1 & SIL 2 are low criticality (e.g., non-fatal injuries)
      » SIL 3 & SIL 4 are life critical – requires same-SIL redundancy
Low SIL

Pattern: One Channel (1-of-1)
- Hardware: single CPU
- Software: no isolation

Pro:
- Simplest pattern
- Least expensive hardware
- Suitable for SIL << hardware failure rate

Con:
- All software promoted to higher SIL
- Only for low criticality (e.g., SIL 1, 2)
  - Fails "active" (i.e., many failures are unsafe)
  - HW failure rate has to be infrequent compared to SIL requirements

NOTE: Solid Box is a Microcontroller Chip

LOW SIL PRIMARY

Single CPU at SIL 1 or SIL 2 (Inputs/Outputs Not Shown)
Self-Diagnosis

Pattern: One Channel (1-of-1) + Built-In-Self-Test

- Hardware: single CPU
- Software: additional self-test libraries

Pro:
- Least expensive hardware
- Suitable for SIL < hardware failure rate
  - Permitted by IEC 60730 with self-test library

Con:
- All software promoted to higher SIL
- Only for low criticality (e.g., SIL 1, 2)
- Self-test does not provide high-criticality safety (e.g., SIL 3, 4)
  - Fails “active” (i.e., many failures are unsafe)

Single CPU at SIL 1 or SIL 2

Single CPU at SIL 3 or SIL 4
Partitioned Low SIL

- **Pattern:** One Channel with Software Isolation
  - Hardware: single CPU
  - Software: partitioned Low SIL / Higher SIL

- **Pro:**
  - Simplest mixed-SIL pattern
    - More or less this is an RTOS for task isolation
  - Relatively inexpensive hardware

- **Con:**
  - Requires SIL “isolation argument”
    - e.g., RTOS memory protection, task timing, I/O isolation, ...
  - **Only for low criticality (e.g., SIL 1, 2)**
    - Fails “active” (i.e., some failures are unsafe)
Low SIL, Fail Operational

Pattern: Two Channel Failover (1-of-2)
- Hardware: primary CPU and backup CPU
- Software: no isolation

Pro:
- Simplest high-availability pattern
- Failover for simple failure modes (low SIL)
  - e.g., loss of heartbeat from Primary

Con:
- All software promoted to higher SIL
- Requires standby diagnosis
  - E.g., via periodic role reversal and self-test
- Standby component does not improve SIL
  - Redundancy for availability, not fault detection
\section*{Voting Architecture}

\textbf{Pattern: Triplex Modular Redundancy (2-of-3)}
- Hardware: Three Primary CPUs plus HW majority voter
- Software: High SIL Primary

\textbf{Pro:}
- Improves availability without internal testing
  - Any fault gets voted out of the majority voter
  - Mismatching unit is most likely the faulty unit
- This pattern is about improving availability
  - Avoids diagnostic loopholes in failover pattern

\textbf{Con:}
- The voter is a single point of failure
  - High SIL fail-operational voter is challenging!
High SIL, Fail Silent

- **Pattern: Two Channel (2-of-2)**
  - Hardware: two cross-checked CPUs
    - Includes redundant, cross-checked I/O
  - Software: no isolation

- **Pro:**
  - Simplest High-SIL pattern
    - Suitable for life-critical SIL (e.g., SIL 3, 4)

- **Con:**
  - All software promoted to higher SIL
    - E.g., if one function is SIL 4, all software must be SIL 4
    - Potentially expensive software development
  - Fails “silent” (stops operation)
High SIL, Fail Operational

- **Pattern:** Dual Two Channel (Dual 2-of-2)
  - Hardware: two pairs of cross-checked CPUs
  - Software: no isolation

- **Pro:**
  - Simplest high-SIL availability pattern
    - Suitable for life-critical SIL (e.g., SIL 3, 4)
  - Fails operational via hot standby

- **Con:**
  - All software promoted to higher SIL
    - Potentially expensive software development
  - Requires ensuring standby is ready to go
    - E.g., via periodic role reversal
    - Periodic off-line self test improves reliability (proof testing)
Ariane 5 Flight 501 Failure

- June, 1996 loss of inaugural flight
  - Also lost $400 million scientific payload

- Primary/Backup Inertial Reference System
  - Reused from Ariane 4
    - But, Ariane 5 had higher horizontal velocity
    - 64-bit float to 16-bit integer overflow in backup
      ... followed by ...
      The exact same numeric overflow in primary
  - Both processors failed \(\Rightarrow\) loss of control

- Software is a single point of failure
  - Redundant SW fails the same way
Pattern: Same-CPU Doer/Checker Pair (mostly fail silent)

- Hardware: single CPU
- Software: Doer=Low SIL; Checker=Low SIL

Pro:
- RTOS can provide some Doer/Checker Isolation
  - Perhaps Checker at SIL 2, Doer at SIL 1
  - Permitted by IEC 60730
- Might be able to take credit for higher SIL checker

Con:
- Requires Doer/Checker isolation argument
  - Or, Doer and Checker both need to be at the same, higher SIL
- Only for low criticality (e.g., SIL 1, 2)
  - Fails “active” (i.e., some failures are unsafe)
Pattern: Low SIL Doer/Checker Pair
- Hardware: Primary plus Checker CPU pair
  - Sometimes called an “E-quizzer” pattern; needs I/O checking!
- Software: Doer=Low SIL; Checker=Low SIL

Pro:
- Hardware isolation between Doer/Checker
  - E.g., SIL 1 Doer, SIL 2 Checker with some SW diversity
- Can lock down checker image despite Doer updates
- Non-Desktop OS in Checker could help with security

Con:
- Requires self-test for Checker to ensure it’s alive
- Only for low criticality (e.g., SIL 1, 2)
  - Checker self-test can’t be perfect; Fails “active”
Pattern: Attempted High SIL Doer/Checker Pair
- Hardware: Primary plus Checker CPU pair
  - Sometimes called a High SIL “E-quizzer” pattern
- Software: Doer=High SIL; Checker=High SIL

Con: Checker can’t be trusted
- Checker self-test will not find all faults
  - Single fault containment region cannot self-diagnose 100% at SIL 3 or SIL 4
- Doer cannot detect all possible Checker faults
  - “Sanity checks” and “quizzing” will only find some faults
  - Doer & Checker have different SW – NOT a 2-of-2 pattern!
- Therefore, Checker will have undetected faults
  - Use for High SIL applications is likely to be unsafe
    » Except for one special case .... see next slide
Pattern: High SIL Doer/Checker with Isolated Checker

- Hardware: Primary Doer/Checker CPU plus Checker CPU
- Software: Doer=High SIL; Checker=High SIL
  - Checker #1 exactly models Checker #2 behavior

Pro:
- Fail-silent behavior with simpler checker CPU
  - Potentially suitable for life-critical SIL (e.g., SIL 3)

Con:
- Requires all High-SIL software; fail-silent
  - Must do proof tests as with dual 2-of-2 architecture
  - Must be careful with potentially coupled Doer/Checker #1 faults
- Requires Doer/Checker software architecture
  - All software must be at the same SIL; mixed SIL is unsafe
**Pattern:** Mixed SIL Doer/Checker

- Hardware: Primary CPU plus 2-of-2 Checker CPU pair
- Software: Doer=Low SIL; Checker=High SIL

**Pro:**
- Isolates High SIL software from Low SIL
  - Suitable for life-critical SIL system (e.g., SIL 3, 4)
  - Checker SIL responsible for system safety
- Only critical software developed at high SIL
  - Enables Low SIL software updates to Doer
  - Checker CPUs can often be small and cheap

**Con:**
- Fail-Silent behavior
- 3 CPUs instead of 2 for fail-silent system
**Pattern:** Mixed SIL Dual Doer/Fail-Stop Checker
- **Hardware:** Dual Primary CPU plus 2-of-2 Checker CPU pair
- **Software:** Doer=Low SIL; Checker=High SIL

**Pro:**
- Likely to be less expensive than dual 2-of-2
  - Only critical software developed at high SIL
  - Checker CPUs can often be small and cheap
- Suitable for life-critical SIL (e.g., SIL 3, 4)
  - Less likely to have an outage due to Doer fault

**Con:**
- Need to structure software as Doer/Checker pair
- **Not fail operational!**
  - Low SIL Doer software fault can shut down system
Best Practices For Safety Architecture

- Consider both HW & SW redundancy
  - Doer/checker provides some diversity

- Use building blocks as appropriate
  - Failover for availability
  - 2-of-2 for same-SIL fault detection
  - Doer/checker for mixed-SIL fault detection

- Pitfalls:
  - Don’t double-spend redundancy
  - “Clever” shortcuts usually don’t work
  - Avoid single points of failure
    - Don’t forget I/O connection redundancy issues!
  - Acceptable patterns depend upon your safety argument
Next Steps: Are you using the right safety architecture pattern?

- Edge Case Research can suggest cost effective ways to build critical systems

Specialists in Making Robust Software:

- Embedded software quality assessment and improvement
- Embedded software skill evaluation and training
- Lightweight software processes for mission critical systems
- Lightweight automation for process & technical area improvement

info@edge-case-research.com
http://www.edge-case-research.com/