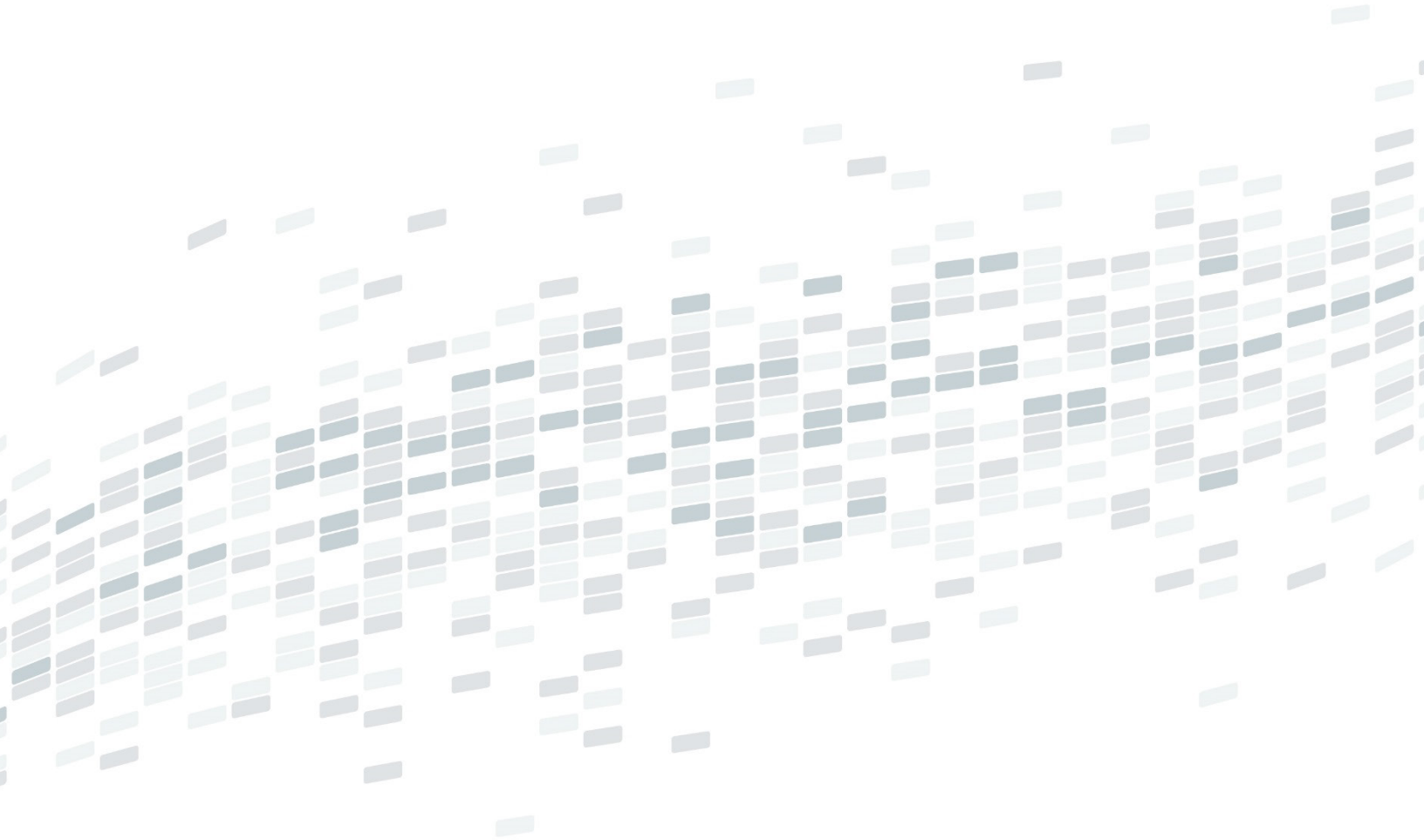
WHITE PAPER
www.visteon.com

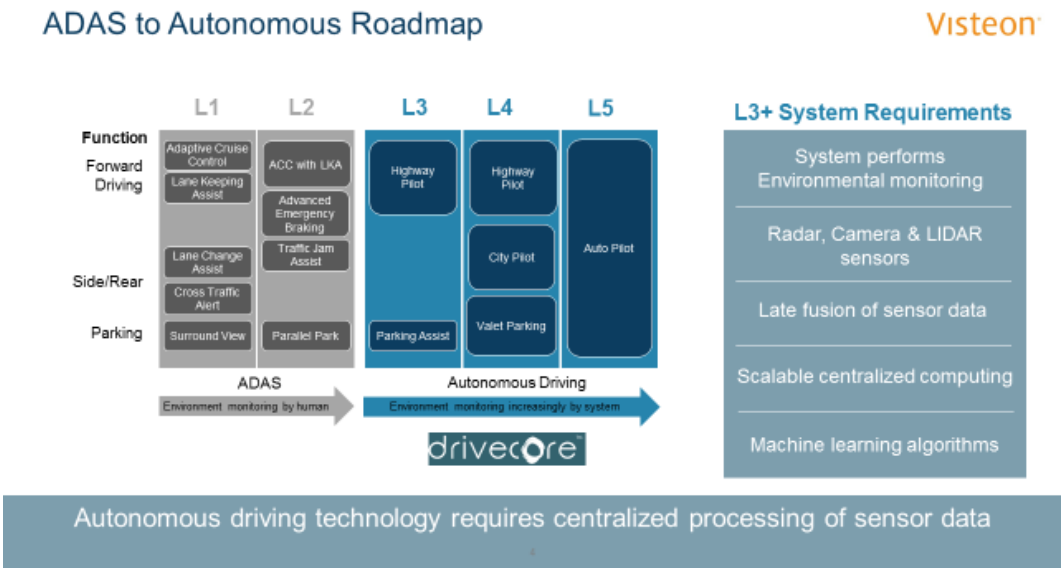
Visteon®

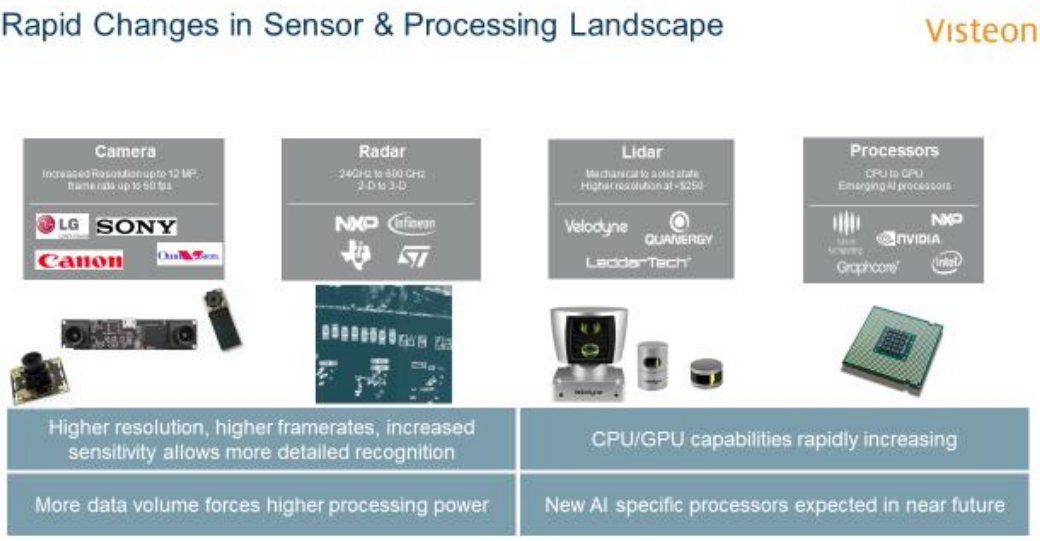
Communication Patterns in Safety Critical Systems for ADAS and Autonomous Vehicles



Communication Patterns in Safety Critical Systems for ADAS & Autonomous Vehicles

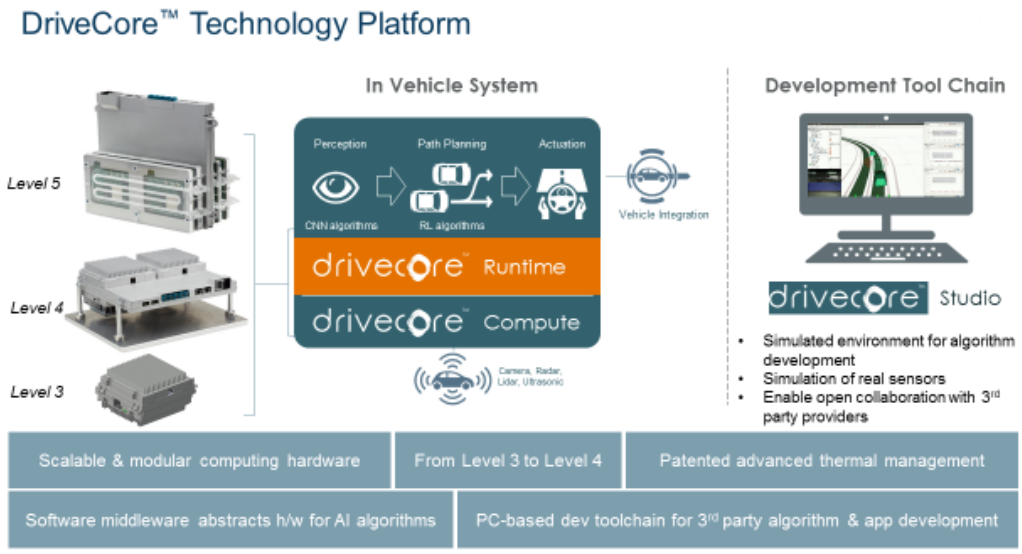
Visteon is on the path to innovation within the area of autonomous driving, with the aim to deliver Level 3-5 autonomous driving solutions - the first Level 3 functions to be delivered will be the highway autopilot and parking assist. As capabilities increase, this will be extended to Level 4 functionality which allows the driver even greater freedom to complete other tasks when not in control of the vehicle. These are the stepping stones that will eventually lead to fully autonomous vehicles which don't need a driver at all. This will require increased monitoring of the environment with radar, camera and LiDAR sensors. The system performance has to be increased dramatically as at the higher levels there is no driver supervision and so the system has the responsibility of detecting and executing which reaction is safe and appropriate. Fail operational systems demand higher levels of redundancy. To satisfy the requirements of different computing demands, Visteon is proposing a scalable system solution, which supports machine learning algorithms at different levels of the stack.





Looking at the rapidly changing sensor and processing landscape, a functioning system has to also be able to incorporate different suppliers' sensors with different needs of connectivity - from Gigabit Ethernet to various types of high frequency serial data links. The support of different CPU vendors enables Visteon to perform functional safety decomposition with increased freedom of interference.

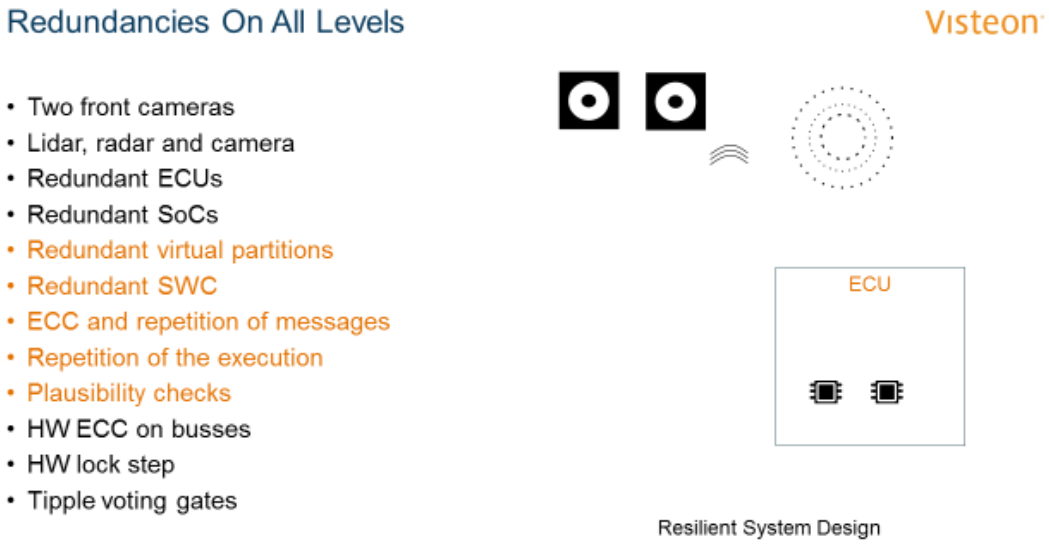
Visteon's answer to these challenges is DriveCore™:



DriveCore™ consists of *Compute*, *Runtime*, *Studio* and *Algorithms*. Compute is the modular scalable hardware to enable Visteon's customers to perform rapid prototyping and deliver fast to market solutions. The in-vehicle system consists of DriveCore™ *Runtime* and *Algorithms*, which implement the autonomous driving functions. The hardware details are abstracted by *Runtime* so that the underlying hardware and sensors can easily be changed. DriveCore™ *Studio* enables the developer to perform

simulation of the environments and real sensors – facilitating third-party providers and enabling collaboration in software development.

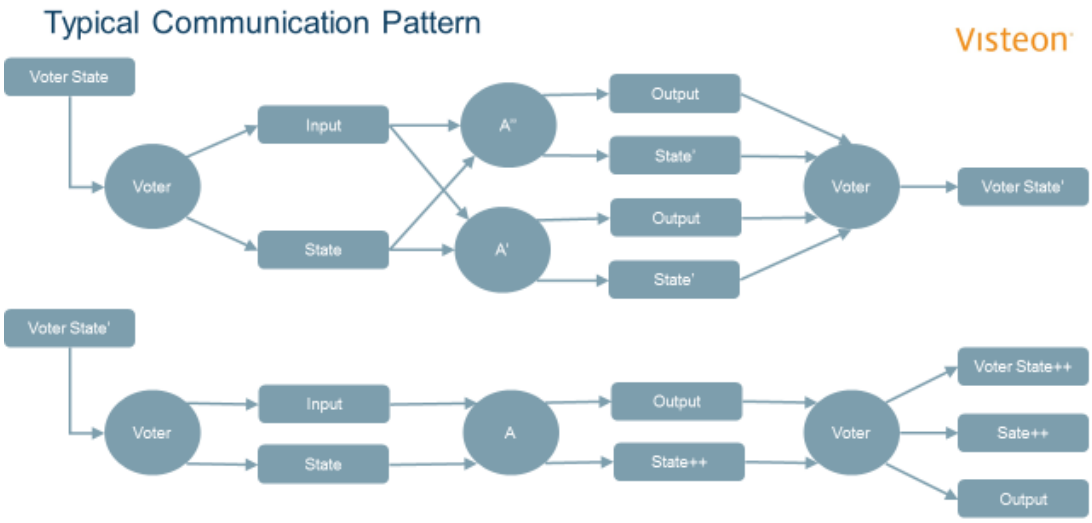
This paper focuses on the lower end of DriveCore™ *Runtime*. What are typical communication patterns? How does Linux fit into this setup? And how can the hardware be used even better?



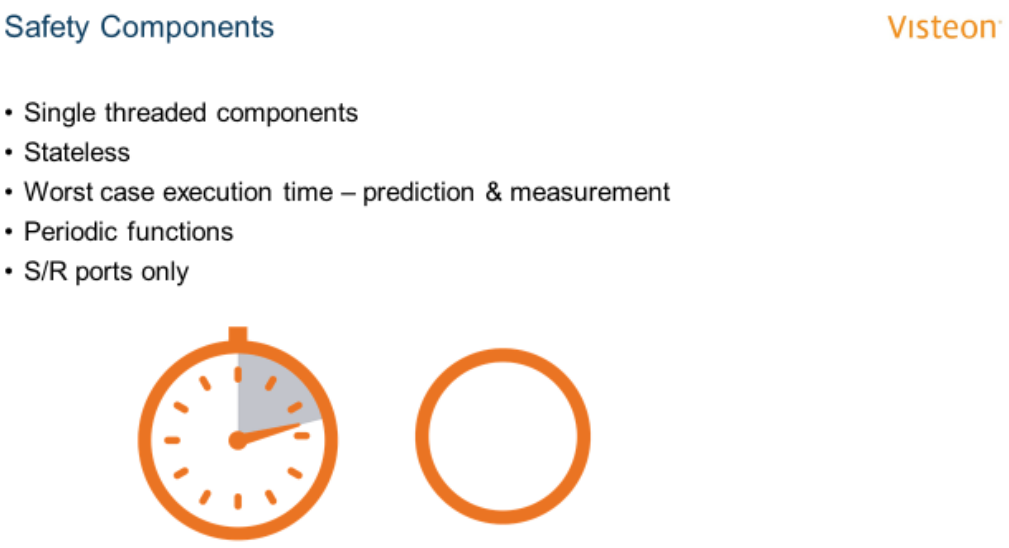
Resilient System Design

In a typical setup of a resilient system design, there are redundancies on all levels – this is required to detect errors and operate correctly to perform the correct actions even in case of a transient error. On the highest level there are redundancies on the hardware level. Sensors are duplicated – cameras, for example – and the coverage of sensors is redundant with different types of sensors, so that if one sensor fails there are two others to still detect objects or verify that there is no object. On a vehicle level, ADAS ECUs may be redundant and even inside ECUs the SoCs can be diverse redundant to allow functional safety decomposition.
On a software level there is redundancy, too. Virtual machines are used to have redundant operating systems - software components may be duplicated to implement software lockstep. Redundancy is added to the data in the form of ECC or CRCs and the repetition of safety critical messages.
The hardware on which the software runs contributes further redundancies in the form of ECC protection of the busses, hardware lockstep CPUs and even on Gate level, triple voting gates are employed to reach the reliability required by the ASIL level.

In software, the redundancy is implemented typically by the concept of software lockstep. The voter provides the state and the input to two instances of algorithm A, which independently produces the output and a new state – these can even be implemented by different teams to improve the independence of the implementation. The voter will then evaluate the output and depending on the nature of the data will decide if an error has happened or not. In case the result is not convincing, the voter will execute the algorithm and take the new results into account, too.

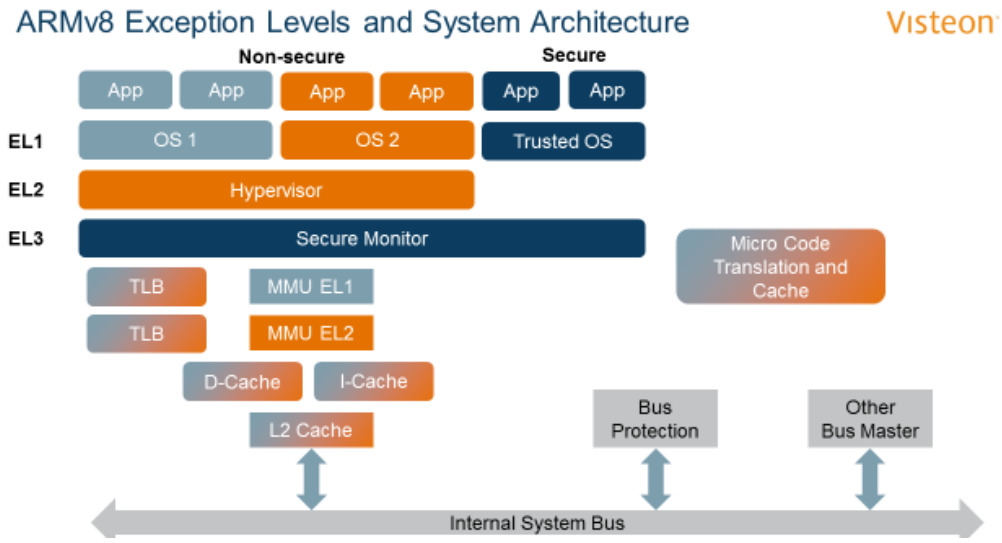


It is a very important observation that algorithms need to be stateless in order to be able to re-execute them. Safety components are single thredded components, which are stateless. To be able to have a predictable execution load, the worst case execution time of each periodic function has to be known by prediction or measurement on the real hardware under worst case conditions. As events would disturb this predictable execution of events sender/receiver ports are used only. Events would be handled only during the scheduled next execution of the periodic function.



To schedule this rate, monotonic scheduling is applied. However, rate monotonic scheduling will waste a lot of CPU time as the typical execution time will be far less than the worst case execution time which is the base of the scheduling plan. This problem has to be kept in mind and a solution will be proposed at the end of this white paper.

The ARMv8 architecture is shown here:



This architecture has three privileged exception levels. Exception level 1 is owned by the operating system and allows to configure the first level Memory Management Unit (MMU).
The hypervisor assigns the hardware and CPU cores to an operating system (OS) instance and is in charge to configure the second level of MMU. Each MMU has a translation lookaside buffer (TLB) which enables fast translation of virtual addresses to physical addresses. The cache and TLB contents depends on the code executed earlier and is determined by the OS executed.
The third exception level is owned by the secure monitor which is in charge of hosting the trusted OS – typically owned by the arm trusted firmware. This software also has to configure the Bus protection which is preventing the different Bus Masters (other CPUs, DMA, PCIe, etc.) to access memories which are not allowed to be written by each domain. With this concept the system is portioned into QM and ASIL functions.

There is the risk of interference between QM and ASIL codes in three main areas: temporal, spatial and logic interference.

Freedom from Interference

Visteon

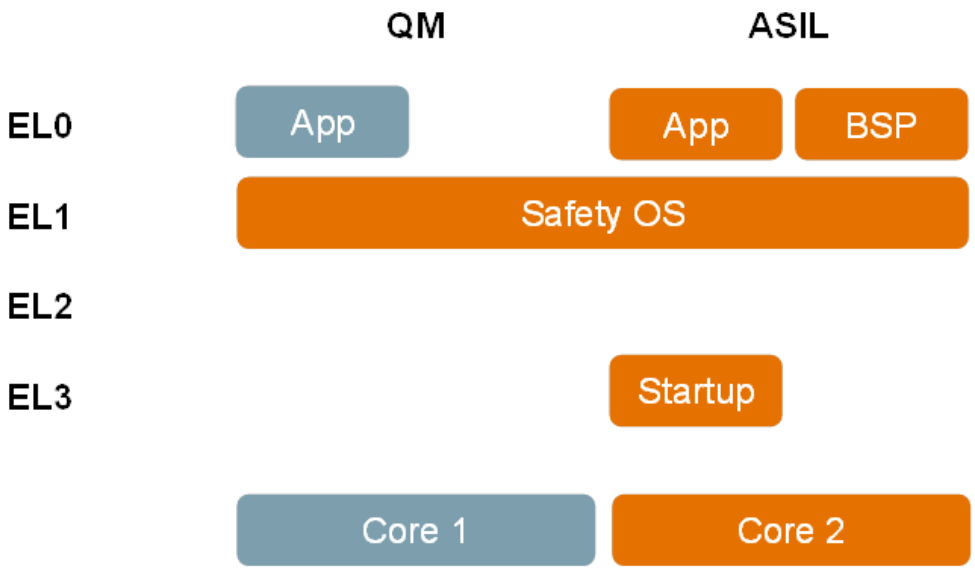
Interference	Countermeasure
Temporal Interference	**Temporal Countermeasure**
Interrupt disabled	NMI for safety
Wrong priorities	Highest priority for ASIL – less priorities
Bus availability	Bus periodization
Cache contents	Clear the cache after scheduling
Translation lookaside buffer	Clear TLB after scheduling
Spatial Interference	**Spatial Countermeasure**
Memory not protected	Use secure memory for safety, bus protection
DMA into safety RAM	Bus protection
mmap the same memory or device	BSP safety certified
Logic Interference	**Logic Countermeasure**
Wrong instructions	Verify before execute before failure
Wrong microcode	Avoid CPUs with microcode

Temporal interference means that the code is not executed with the expected speed. This can happen if a higher interrupt is executed at a high exception level – for example, a timer interrupt on exception level 3 cannot be disabled by the OS or hypervisor and will gain the CPU access in any case.

Wrong priorities are another issue. ASIL code should run on a higher priority than QM code. But if ASIL code is accessing QM Code the Priority Ciling Protocol will lead to increased priority of the QM code. The same priority issue exists for Bus access – ASIL code should have the highest Bus priority on the Bus. Caches and TLB contents may be kind of flushed by QM code if it has a huge active working set which can impact the performance of the ASIL code. As a counter measure a cache flush and TLB flush may be performed to measure the worst case execution time. To prevent information leakage on speculative executed cache loads, the cache should be flushed while the switch between QM and ASIL functions happen.

Spatial interference means that the data is overwritten by other components. This can happen if the MMU or Bus protection was not properly configured. The ARMv8's secure memory feature can be utilized to separate QM (non-secure) from ASIL (secure) memories.

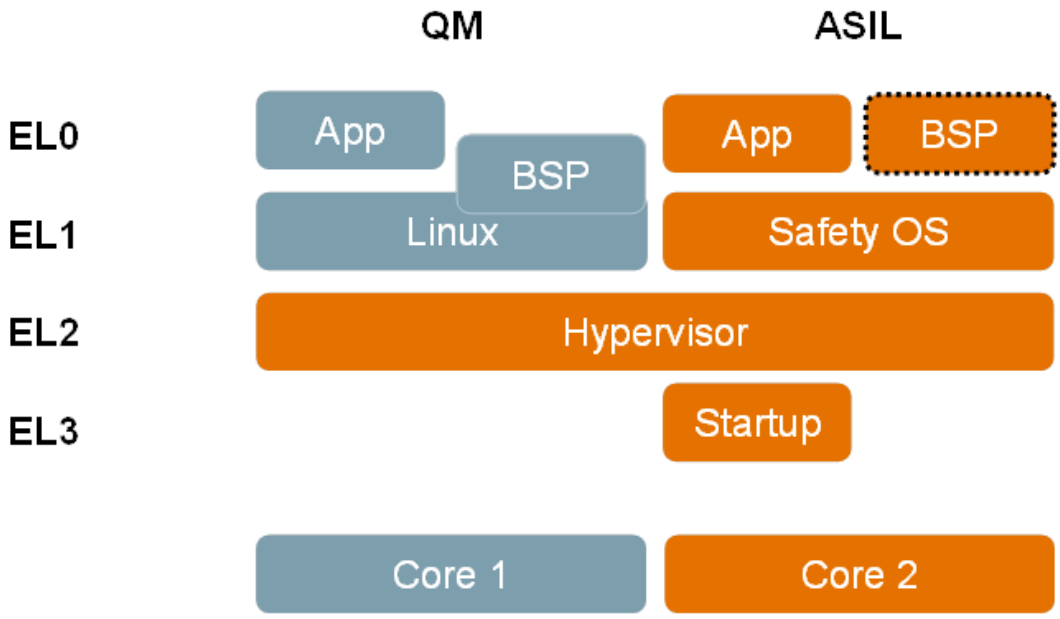
Finally there can be logic interference where wrong instructions or wrong microcode is executed. For functional safety, one may want to avoid CPU cores which perform internal code translation and signature check the code before execution.



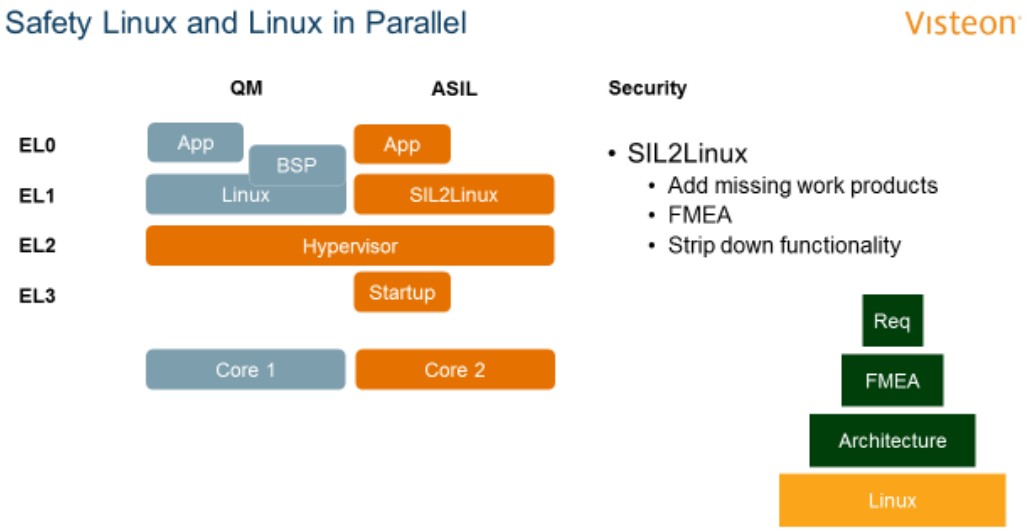
The simplest setup at first glance is to have a safety operating system, which is safety certified and executes QM and ASIL functions. Ideally they are dedicated to different cores so that the Bus Protection can discriminate between them. Performing a fine grained prevention of wrong map access on OS level is possible, but tedious and the QM applications would have to be modified. Most likely the BSP has to be safety certified as well.

The exception level 3 code is startup code, which is required to write the Bus protection. On some SoCs a later read back of this configuration is not possible as the access is only allowed from a secure domain – which is not present in this setup.

A different approach is to use a hypervisor to separate QM and ASIL code. One OS hosts only QM functions and the other safety OS hosts only ASIL functions. It is very likely that the BSP on the safety OS could be reduced to a minimum. The QM OS hosts the majority of the BSP. Linux may be used as only QM functions are allocated.



In case the safety OS only uses generic functions of the ARMv8 architecture it may be even feasible to qualify as stripped down Linux to perform the functionality of the safety OS.



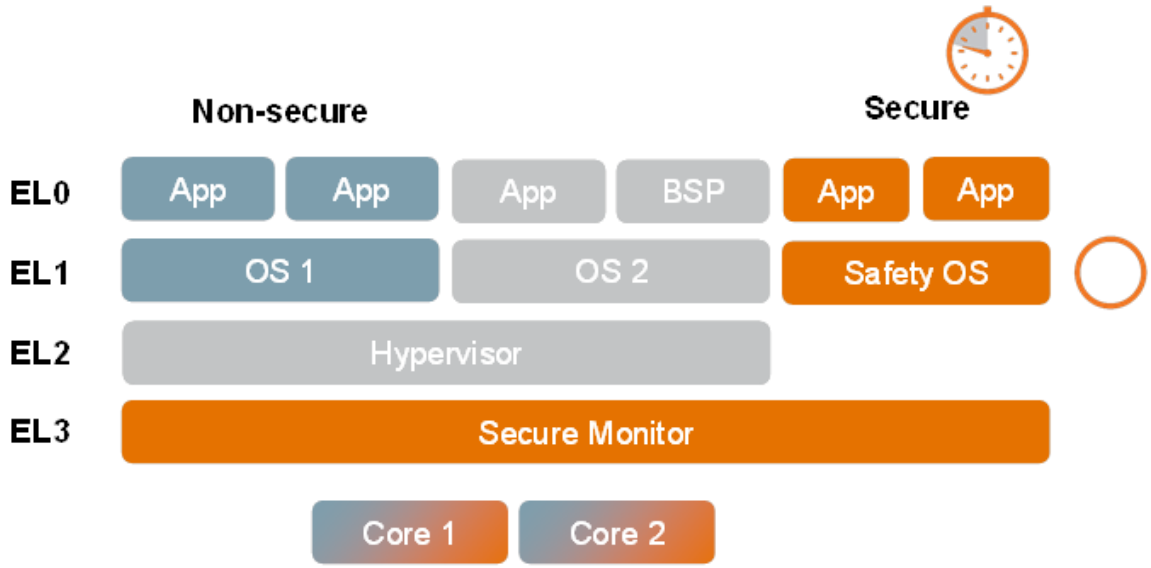
This and even more is done by the SIL2Linux project. The safety relevant work products have to be provided to be able to qualify Linux. The change rate and code size are problems for this approach. So it

is very unlikely that Linux will be updated often – however all changes have to be tracked and evaluated if they fix a bug, which is to be also fixed in the safety certified version – this can be a lot of noise.
The other problem is the code size; this can be partially fixed through the strip down process.

Looking at these setups what are the main requirements for a safety OS?

- Rate monotonic periodic scheduling of execution sets
- Strict memory protection
- Re-execution of components possible
- Highest interrupt priority
- Uninterrupted execution of component
- Clear cache and TLB before execution of component (WCET)
- Support functions (watchdog, signature checks, start/stop of components)
- Boot and device support as needed
- Protect against side channel attacks

One issue with the setups so far is that they do not exploit exception level 3 and the secure domain of ARMv8.



This could be solved if the secure monitor is used and the safety OS is executed in the secure domain of the ARMv8 architecture. There, simple components can be executed in a rate monotonic fashion. The switch back to the QM world is as simple as a return from exception level. The remainder of CPU time not used by the safety domain can be utilized by the QM domain running in the non-secure world. The disadvantage is that the CPU is blocked by the safety domain - increasing the latency. This problem can be mitigated if one core is reserved for QM functions with low latency requirement.
The safety OS can be implemented in a generic way by using only generic ARMv8 functionality, with reduced hardware dependencies. This safety OS can then be also the vehicle to implement secure functionality found in the existing TrustZone code. Existing TrustZone code has a similar problem to Linux – it was not developed with safety and ASPICE in mind in the first place.

Conclusion

Redundancies exist on all levels in ADAS systems. Linux could be used isolated or in the SIL2Linux way. A dedicated safety OS exploiting the security features of the hardware is proposed. This enables to mix rate monotonic scheduling with priority-based scheduling to maximize hardware utilization.