and stop the car in case of an emergency. A reset or complete deactivation of a system are considered as safe states. Such systems are fail-safe.

The software of fail-safe systems must detect and handle both, hardware and software fault within a specified tolerance time interval. Often, deadline monitoring for certain software functions or additional checks for integrity and liveness are used for this purpose. An example of the latter is end-to-end protection (E2E).

In automated driving, the driver no longer serves as fallback solution in the safety concept. There must be an electronic solution. This is typically implemented by a second channel – i.e. by redundancy (Figure 1). In case of a fault, the active channel is deactivated, and a second channel takes over control of the vehicle. Such a system is fail-operational.

Fail-Safe and Fail-Operational

Today, almost all systems in a car are designed to work with a single channel. Single channel means that if an element in the chain from sensor via logic to the actuator fails, then the entire system fails. Redundancy by, for example, a second sensor or microcontroller is introduced exclusively to reliably detect a fault in this chain. If an error occurs, a safe state must be established. This is usually done by resetting the control unit. The driver must adapt to the new situation and stop the car in case of an emergency. A reset or complete deactivation of a system are considered as safe states. Such systems are fail-safe.

The software of fail-safe systems must detect and handle both, hardware and software fault within a specified tolerance time interval. Often, deadline monitoring for certain software functions or additional checks for integrity and liveness are used for this purpose. An example of the latter is end-to-end protection (E2E).

In automated driving, the driver no longer serves as fallback solution in the safety concept. There must be an electronic solution. This is typically implemented by a second channel – i.e. by redundancy (Figure 1). In case of a fault, the active channel is deactivated, and a second channel takes over control of the vehicle. Such a system is fail-operational.

Failures in Hardware and Software

In hardware development the assumption is well established that a system will fail at any point in time with a certain probability. There are different mathematical models for this probability. With these models the developers evaluate and argue the safety of their hardware. The crucial point is that such models exist at all.
In contrast, all faults in software are of systemic nature. All attempts to establish a generally accepted model for probabilities of systemic faults have not succeeded to date. Thus, systemic faults must be avoided in order to keep the probability-based argument of the system analysis valid. For automated driving, the detection of faults is no longer sufficient. Systematic faults can only be prevented by an appropriate development process. The safety standard ISO 26262 defines a lower bound for the activities and methods. In contrast to fail-safe systems, many more functions become safety-related in fail-operational systems, not only the functions that detect and mitigate faults. One example of a safety requirement for the AUTOSAR basic software is reliable communication via the bus systems. The ISO 26262 standard defines these types of requirements as safety-related availability requirements. They lead to a typical design of today’s control units for highly automated driving: a microprocessor-based part that performs the nominal function and a microcontroller-based part that serves as a monitor to the microprocessors and as a fallback level. This ensures that driving continues in the event of a microprocessor fault (Figure 2). There are multiple reasons for these kinds of systems architectures. The development process of high-performance microprocessors does not yet reach the current automotive engineering standards. Due to the significantly higher complexity of such microprocessors, systemic faults are not yet sufficiently avoided. The number of built-in safety mechanisms is also smaller. Crucial software, such as reliable, safety-qualified communication stacks, are currently only available for microcontrollers. This lack of mechanisms and confidence leads to the supplementary use of microcontrollers with well-known software which has proven itself for years in automotive engineering.

Challenges for Operating Systems in High-Performance Control Units

Many microcontroller-based ECUs typically run OSEK/AUTOSAR operating systems. High-performance ECUs, on the other hand, use operating systems known from the IT world – such as Linux, QNX or PikeOS. In contrast to AUTOSAR, these operating systems provide mechanisms to easily install additional software on the ECU without reconfiguring or recompiling the operating system. This makes it very tempting to provide the end customer with an “empty” ECU and install the software afterwards. But is this approach also applicable for safety-related software such as Lane Keep Assists or Highway Pilots? With the number of downloadable, safety-related applications, the testing effort for these systems increases, since every combination of software on the ECU must be tested, as specified in ISO 26262-6:2018 Annex C. In order to reduce the effort, the operating system needs to guarantee that the installed applications do not interfere with each other. Newly added applications must therefore have no influence on the behavior of already installed applications. Interference here has multiple aspects, i.e. memory, timing and communication. Especially for Linux such a variant is not available, which makes a qualification according to ISO

![Figure 1](image)

**Figure 1**: Architecture of a fail-operational system. Redundancy is used to switch to the other channel in the event of a fault.
Non-deterministic time for allocation and deallocation of memory
Fragmentation of the (heap) memory
Memory leaks, i.e. memory that was forgotten by the application and not returned to the operating system
Exhaustion of memory

A big problem in software development has always been the access to invalid memory, which does not contain the intended information. This problem is even more complex to solve with dynamic memory allocation. For fail-safe systems, only access to invalid memory is critical – for example via “use-after-free” access. All other problems are simply detected, and the system can be shut down safely. Memory leaks are not dangerous on their own but lead to a faster memory exhaustion than expected. This is also easily detected. Fragmentation of memory leads to a non-deterministic time behavior, when allocating and deallocating. However, this can also be easily detected by proven means.

26262-8:2018 Clause 12 necessary. There are some projects that try to give guidance, how this qualification could be done [1], but it will take a while until their results are usable. Commercially available operating systems, which were developed specifically for use in safety-related systems, offer benefits here.

Compared to ensuring freedom from interference with respect to memory, showing the freedom from interference with respect to timing is even harder. Operating systems designed for the IT domain typically do not offer hard real-time scheduling. Linux can e.g. be extended with the preempt_rt patch [2], providing the “earliest deadline first” (EDF) scheduling algorithm and optimized kernel locking. But even then, newly added processes may impact other applications, for example if too many threads are created. PikeOS as a counter example [3] provides separate time domains for different applications, bounding the execution time and in turn ensure freedom from interference. Thus, applications developed according to QM level are executed in their own time domain with dynamic use of the CPU.

If open source software (OSS) is to be used in safety-related applications, additional qualification measures according ISO 26262 must be performed. In addition, the maintenance strategy must be carefully considered. In case of a defect in the OSS, short-term bug fixing is sometimes difficult because employees or external partners are likely not familiar with that part of the code. This is especially true for large OSS projects, such as the Linux kernel itself. The decision to use for OSS in safety-related projects should therefore be carefully evaluated.

Safe Application Development with C++
A typical question when using C++ is the handling of dynamic memory allocations, i.e. memory that is requested and released by the operating system during runtime. This creates a whole new class of potential faults, which previously played no role in automotive software development:

- Non-deterministic time for allocation and deallocation of memory
- Fragmentation of the (heap) memory
- Memory leaks, i.e. memory that was forgotten by the application and not returned to the operating system
- Exhaustion of memory

A big problem in software development has always been the access to invalid memory, which does not contain the intended information. This problem is even more complex to solve with dynamic memory allocation. For fail-safe systems, only access to invalid memory is critical – for example via “use-after-free” access. All other problems are simply detected, and the system can be shut down safely. Memory leaks are not dangerous on their own but lead to a faster memory exhaustion than expected. This is also easily detected. Fragmentation of memory leads to a non-deterministic time behavior, when allocating and deallocating. However, this can also be easily detected by proven means.
Outlook

With the methods described, fail-safe systems can be developed already today. The limiting factors for fail-operational systems with high performance ECUs are the availability of reliable hardware and important operating system features. For example, the communications stack must have high integrity and availability. For some operating systems, development has already been started in this direction, but they will not be ready for production before 2021/2022. Therefore, the first generation of high-performance ECUs will be based on fail-safe systems.

Another question concerning C++ is the error handling using exceptions. Their use requires dynamic memory allocations and complex unwinding of the stack. Moreover, it is not easy to implement “exception safety”. “Exception safety” means that the current object is still in a valid state even if an exception is thrown. The most urgent problem, however, is that static code analysis with today’s tools is almost impossible to do. For example, until a change in the C++ standard has become established, it is advisable to use result data types – a concept that is used among others in the Rust programming language.

Safety Mechanisms in the AUTOSAR Adaptive Platform

The AUTOSAR Adaptive Platform offers very similar safety mechanisms as the Classic Platform. For example, the integrity of communication between ECUs is ensured with end-to-end protection (E2E). In this communication pattern, the sender sends information periodically on the network to the receiver. This information is enriched with a sequence counter, a data ID and a CRC checksum. This enables the receiver to detect the relevant faults in the communication (Table 1).

For intra-ECU communication, the E2E mechanism is typically not used because of performance reasons. Thus, in AUTOSAR Adaptive systems, the operating system must provide a safe channel. For ECUs using a POSIX operating system, this requirement is difficult to achieve. Although, other applications cannot directly compromise the channel by using the Memory Management Unit, the kernel itself is a threat. Either the entire kernel is developed according to a safety-oriented process, or another argumentation exists, why the kernel cannot compromise the channel. Besides process separation, the safe communication channel is the second most important safety requirement for such an operating system.

Table 2: Overview of timing faults and their countermeasures

<table>
<thead>
<tr>
<th>Which faults are possible?</th>
<th>Countermeasures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution of code without request</td>
<td>Deadline Monitoring</td>
</tr>
<tr>
<td>Code not executed although requested</td>
<td>- Applicable for aperiodic functions</td>
</tr>
<tr>
<td>Execution of code started too early or too late</td>
<td>- Time between two checkpoints is compared to min/max values</td>
</tr>
<tr>
<td>The execution time of a code is longer or shorter than expected</td>
<td>Alive Monitoring</td>
</tr>
<tr>
<td>The program flow of a code differs from the expected behavior</td>
<td>- Applicable for periodic functions</td>
</tr>
<tr>
<td></td>
<td>- Number of checkpoints in interval is monitored</td>
</tr>
<tr>
<td></td>
<td>Logic Monitoring</td>
</tr>
<tr>
<td></td>
<td>- Detect wrong execution order</td>
</tr>
<tr>
<td></td>
<td>- Validate checkpoint activation sequence against preconfigured execution graphs</td>
</tr>
</tbody>
</table>

Table 2: Overview of timing faults and their countermeasures

AUTOSAR Adaptive also provides mechanisms for timing monitoring. Even if the operating system’s scheduler has been developed according to the highest safety integrity level, a timing monitoring using an external watchdog is advised. In AUTOSAR Adaptive, this functionality is performed by the “Platform Health Management” (PHM). The application regularly reports to the PHM, which then decides based on its configuration whether the application is still running within its time limits. There are several potential faults and reactions (Table 2).

Published in the German magazine ATZelektronik 7-8/2020

Image rights: Vector Informatik GmbH

Literature References:
[1] https://elisa.tech/