WHITEPAPER

Making Safety Beautiful:

Functional Safety and Qt

Contents

Introduction

Functional safety requirements are increasingly relevant in a number of disparate markets – after all, who doesn't want safer highways, hospitals, and factories? Consistent processes and standards can help us design and build products that make the systems upon which these institutions operate safer. But the conservative approach of functional safety is often at odds with many aspects of the standard software development process – modern tools that make development easier, dynamic frameworks that allow software to adapt, and product requirements that call for attractive, modern, and updatable UIs rely on practices often forbidden in a functional safety context.

At the Qt Company, we increasingly see the intersection of these two disciplines. End-customers demand hardened products be as capable, flexible, and user-friendly as their smartphones. Meanwhile, industry regulators and corporate safety officers demand products be as safe as possible and meet standards that are difficult for most UI frameworks to meet. Is it possible to build a product that is at once functionally safe, modern, and attractive?

The answer is yes, although unsurprisingly it does take more planning and work to achieve as well as some experience to avoid a number of pitfalls. In this white paper, we'll discuss how to go about incorporating a modern, dynamic, and capable UX into a functionally safe product. Although we'll specifically be using Qt as our UX framework in this whitepaper, we'll also cover some generic advice whenever possible.

What Is Functional Safety?

Let's start with a brief introduction to functional safety. Its basic goal is to avoid unacceptable harm to people by lowering the impact of failures or by eliminating them whenever possible. This can be broken down – with apologies to safety experts everywhere for its simplicity – into four essentials.

1) Avoiding faults.
The goal isn't to remove all faults because that's impossible to achieve except for the most trivial software. Rather, it's about trying to avoid systematic faults when possible and properly control them when not. For example: because dynamic memory allocation can fail, a functionally safe approach would statically allocate all memory. In a functionally safe system you would also try to avoid or control random faults or failures, which could mean introducing redundancy in both software and hardware, or introducing keep-alive or heartbeat mechanisms to ensure software is running properly.

2) Managing risk.
Risk management starts with an understanding of the risk level through a risk assessment – that is, determining the worst that could happen if a component fails. This is determined by quantifying the risk in three ways: the severity of an injury (minor to death), the frequency of occurrence (seldom to continuous), and the ability to avoid (possible or unavoidable). If the failure of a train-control mechanism could cause a derailment at every junction, this would be considered a severe, frequent, and unavoidable risk – and would absolutely require extreme risk reduction to mitigate failure. By comparison, faults that could cause a person to get a small cut if they don't move their hand fast enough when changing a toner cartridge are minor, infrequent, and avoidable. Understanding risk level is critical to knowing how to manage risk; this is typically detailed for different industries via terminologies like Safety Integrity Level (SIL) or Class. More on this later.

3) Being consistent.
Early safety critical systems were created before functional safety standards so it is clearly possible to create safe systems without following a set of rules. However, we as software engineers have learned from examining the mistakes of others how we should and shouldn't build safe software. This has allowed experts to capture best practices in various standards. Each industry has their own way of defining processes to build safe software but they're all similar in that they enforce repeatable, consistent ways of doing things. You don't create safe software if you're a cowboy coder.

4) Incorporating safety from inception
Software needs to be designed with safety in mind from the beginning – it's extremely difficult to build a truly safe system by addressing safety as an afterthought. Acknowledging safety requirements at the architectural and design stage is important because some system characteristics are vitally different when some components need to be (nearly) fail-proof. If you're building a device that isn't allowed to fail, your approach to a graphics subsystem will be very different than if infrequent, isolated failures can be tolerated (and fixed by the occasional reboot).

Functional Safety Standards and Vertical Markets

There is no shortage of acronyms and numbered standards in the safety world so wouldn't it be nice if there was one single standard that applied to every industry? Although there are different standards in use – for example, medical systems use IEC 62304 and automotive systems use ISO 26262 – many are related to the granddaddy of safety standards, IEC 61508, used by industrial automation.

IEC 62304 (for medical) specifies the software lifecycle process when building medical devices. Although certification to IEC 61508 is not required by the medical device industry, since it contains a good deal of practical considerations for building safety systems it can be very beneficial to refer to IEC 61508 in conjunction with IEC 62304.

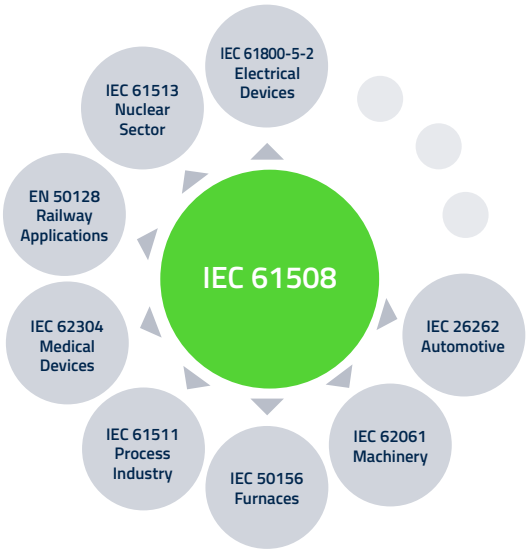


Figure 1: IEC 61508 and related standards

ISO 26262 (for automotive) is a functional safety specification customized for passenger vehicles and is a derivative of IEC 61508. Adaptations have been made to comprehend the particulars of designing and building cars, as well as the unique nature of a car's lifecycle. Although ISO 26262 covers cars, IEC 61508 is still often used for commercial, off-road vehicles

Safety Integrity Levels

One key aspect of nearly all functional safety standards is that they divide the failure risk into different discrete safety levels, where each level demands different treatment of the software in question. For example, IEC 61508 establishes levels based on whether or not the device is in high demand (used more or less continuously) or low demand (used at most once a year). For high-demand operation, the safety levels are defined as follows:

- Safety Integrity Level (SIL) 4 – between 10^{-9} and 10^{-8} failures per hour or one failure in 11,400 years of operation
- SIL 3 – between 10^{-8} and 10^{-7} failures per hour or one failure in 1140 years
- SIL 2 – between 10^{-7} and 10^{-6} failures per hour or one failure in 114 years
- SIL 1 – between 10^{-6} and 10^{-5} failures per hour or one failure in 11 years

Similarly, while ISO 26262 originally comes from IEC 61508, the two specifications use different definitions for various safety levels. The safety levels for ISO 26262 are not as prescriptive as those in IEC 61508 but are goal-oriented instead. They are based on three separate factors – severity, exposure, and controllability – that combine to form an Automotive Safety Integrity Level (ASIL). Because they are defined differently, the ASIL levels of ISO 26262 and SIL levels of IEC 61508 do not have a one-to-one mapping although they can be roughly correlated (see table 1). In practice, this means that software libraries for functional safety purposes cannot be directly leveraged between the two standards. Attempting to address both an industrial and automotive market would require both certifications.

	IEC 61508 Industrial Automation	ISO 26262 Automotive
Highest safety level	SIL 4	–
	SIL 3	ASIL D
	SIL 2	ASIL B/C
Lowest safety level	SIL 1	ASIL A
No safety requirement	–	QM (Quality Management)

Table 1. Rough comparison of Safety Integrity Levels [1]

1) From Safety Integrity Level to Assured Reliability and Resilience Level for Compositional Safety Critical Systems, Eric Verhulst, Altreonic NV

Functional Safety and Qt

With the basics out of the way, it's time to address how functional safety works with Qt because Qt provides the user interface for a great many embedded devices. To determine if it's possible to use Qt in a functionally safe system, we at the Qt Company conducted two separate studies with the certification authority VTT Expert Services. concluded that it is feasible to certify the Qt framework for functional safety by separating out the functionally safe component of a product from the remainder of the system. The functionally safe piece runs within an isolated environment and is used for the code portions that must be certified – including critical display graphics. The non-safe part is used for everything else, including the majority of the Qt-based user interface. Of course, the non-safe portion may still be robust and resilient code but because it isn't necessary to be certified, the non-safe code doesn't need to conform to the same rigid rules as the safe portion. However, the two pieces still need to share enough system state to allow for this separation.

The studies indicated several reasons that extracting and isolating the functionally safe portion of the system is the most appropriate course of action:

- Parts of the Qt system would be extremely difficult to retrofit for a functionally safe environment. (Even Qt core would need to be heavily modified to be certified.) Removal of these components would leave a significantly lighter and less valuable framework.
- As Qt makes substantive use of modern C++ features, with much of it of questionable suitability for a functionally safe system, a large subset of code would be adversely impacted.
- Many changes would also be needed to reduce or remove dynamic objects, pointers, and automatic type conversions from Qt. These changes would fundamentally alter the API, making a certified Qt framework look significantly different than Qt today.
- The API changes required would break any existing code and libraries. New APIs would also need new documentation, training courses, and materials so that engineers could properly use the new APIs.
- A separate source fork would make it nearly impossible to keep a "certified Qt" in sync with the standard Qt baseline. Coupled with the frequent feature additions and rapid release cadence, this would mean a safety-based Qt framework would increasingly drift away from the normal Qt, doubling the engineering efforts to maintain it.

Finally, functional safety was not one of the original design criteria of Qt, which implies a significant amount of work required to reassess, document, and rework the entire framework to comply with functional safety criteria.

Enter the Qt Safe Renderer

Thankfully, there is very little need to run software as complex as Qt for most functional safety systems – there's a much better approach. Ideally, a safety-separated Qt system would be responsible only for the functional safety portions of the UI, safely interact with the main software that does not have safety critical requirements, and be designed from the ground-up with safety considerations in mind. To reduce the possibilities for failure, it should preferably be as small and simple as required – but no less. And it should leverage as much as possible from the tools and frameworks that already exist in the normal Qt world.

These are the basic design requirements behind the Qt Safe Renderer, which was specifically created to address creating Qt applications in functional safety systems. Because it separates the safety critical portions from the primary Qt system, it also minimizes the impact of a functional safety certification on the mainline software development.

Essentially, the Qt Safe Renderer is a component that allows simple graphics to be displayed, such as indicators, warnings, alerts or pointers. It is isolated from the main Qt application in a separate container (for example, within a hypervisor environment) and overlays its imagery on top of the primary application via hardware layers or a safety-certified software compositor. It has the necessary certification evidence to feed into a functional safety certification. It monitors the execution of the main Qt application to ensure it's functioning properly and attempts to restart it if not.

Because the Qt Safe Renderer is designed to have a fail-proof and 100% verified code path, it necessarily restricts the options in how it presents items to the user interface. The Qt Safe Renderer provides the functionality that is most commonly needed in safety systems: displaying and repositioning static bitmaps. This handles situations with trouble or diagnostic indicators very well – which is typically enough to provide the functional safety portions of a medical, automotive, or industrial automation system.

Functionally Safe Qt Architecture

Software architecture for a system that combines Qt and Qt Safe Renderer breaks one CPU virtually in two. Depending on the operating system used and the level of functional safety certification that's required, this may be possible to do within the operating system itself. If it's not possible, the addition of a hypervisor can keep software in each portion isolated from one another. In either case, the goal is to certify the smaller portion for functional safety while the remaining software can be built using standard practices and existing methodologies. Here are the two approaches:

Figure 2. The Qt RTOS-only Environment for Functional Safety

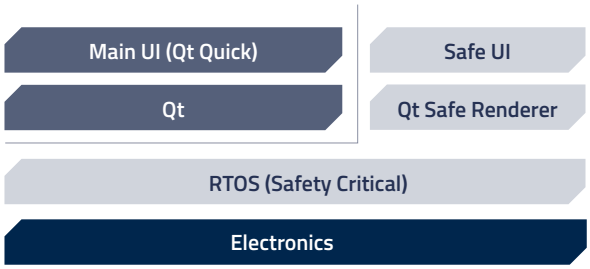
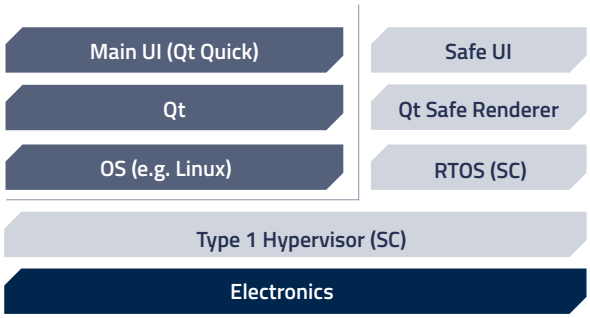


Figure 3. The Qt Hypervisor Environment for Functional Safety



1. Certified Operating System – operating system (typically a real-time OS) that can be safety certified such the QNX Neutrino RTOS Safe Kernel or the Green Hills INTEGRITY RTOS. The role of the RTOS is to provide the services needed to load and execute the Qt Safe Renderer – primarily memory allocation for initial execution, task scheduling, and graphical-rendering services.
2. Qt Safe Renderer and Safe UI – component that handles graphical rendering for the parts of the system that fall under functional safety requirements. As the names would indicate, the Safe UI is the application responsible for the functionally safe UI, and the Qt Safe Renderer is the engine that does the actual rendering on behalf of the Safe UI. The Qt Safe Renderer also monitors the health of the primary application and restarts it if it's not operating properly.
3. Operating System – OS that runs the main Qt application. This OS doesn't have to pass a functional safety certification so it can be something with less rock-solid safety credentials like Linux. However, there's no reason it couldn't run the same OS as the safe operating system, contributing fewer components and making the overall system simpler. The primary OS needs to provide all the standard services required by the primary app – memory management, file systems, threading, synchronization primitives, drivers, etc.
4. Qt – Qt framework libraries used by the main UI. These are all the Qt components that the app requires at runtime.
5. Main UI – main application containing all the primary non-safety functionality. The application provides a periodic heartbeat to the Qt Safe Render so it can monitor the main UI application for crashes or misbehavior, and respond appropriately. The application does not need to pass certification, however, it's still important that it is highly reliable and as fail-proof as possible. You may want to consider developing the application using the same or similar processes as you use for the functionally safe portion.

The architecture in figure 2 may work fine if you are using a safety certified OS as a starting point. However, if your system design uses Linux or another OS that can't be certified, you may need to add a type 1 hypervisor to the system to ensure the split between the application OS and the safety OS is isolated and more easily certified – see figure 3. This may also be beneficial if you're seeking a higher level of functional safety certification (for example, ASIL D). In this case, most of the pieces stay the same as in figure 2 with one obvious addition:

6. Certified Type 1 Hypervisor – piece of software that allows the safety and non-safety environments to run on the same processor in isolated containers. The hypervisor also uses hardware support to let those two disparate environments share the same resources: CPU, GPU, memory, flash, and peripherals. It provides the best opportunity for developing a safety system that is definitively isolated and protected, and hence easier to pass certification.

Dual Processor Approach to Safety

Another possible alternative to the hypervisor model for a functionally safe architecture is to use independent processors, as shown in Figure 4.

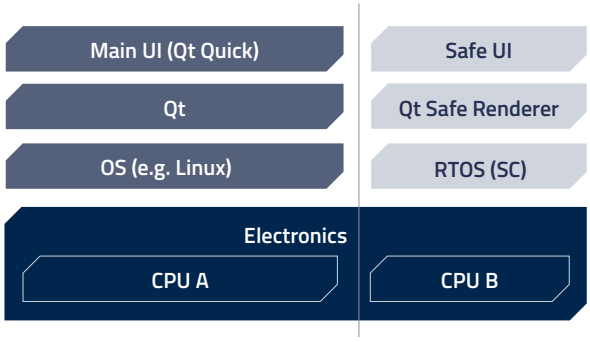


Figure 4. Functional Safety using two CPUs

This approach has a number of benefits – one of the biggest being near complete functional safety isolation through hardware. As the safety critical portion of the code is typically limited in scope, a lightweight micro-controller running a simple no-frills RTOS may be able to handle the processing needs while the full-featured CPU and OS combination handles the remainder of the system. The two CPUs may also be able to share resources – RAM, non-volatile storage, or select peripherals – as long as their interaction can guarantee integrity of the safety critical software.

Building a Safety-separated Qt System

With the introduction of a new safe partition into a Qt run-time system, your main question may be, "How does any of this work with my current Qt workflow?" Let's look at several pieces in a typical Qt tool chain and review how they interface with the functionally safe part of the system and Qt Safe Renderer.

Figure 5. ISO icons added in Qt Quick Designer for use by Qt Safe Renderer

- **Qt Quick and Qt Quick Designer**

Qt Quick (and QML) provides the overall system with the ability to create declarative interfaces. Any indicator assets created in Qt Quick Designer and used by the QML interface can be extracted and built into the Qt Safe Renderer binaries. This means the same tool chain and workflows that build the primary user interface can also be leveraged for the Qt Safe Renderer assets.

- **Qt for Device Creation**

The Qt Safe Renderer build-time tools integrate with either standard Qt or Qt for Device Creation but because embedded systems are the primary use case for functional safety, Qt for Device Creation will almost always be preferable. (Because the Qt Automotive Suite is bundled with Qt for Device Creation, automotive designs will also fall under this umbrella.)

- **Qt 3D and Qt 3D Studio**

If an application needs 3D rendering, you'd use Qt 3D Studio to develop assets and Qt 3D to use them at run-time. Just like QML, Qt Safe Renderer can interface with the Qt 3D Studio tools at build time to import 2D assets, which allows use of a single tool to service both the main UI and the functionally safe compartment without requiring duplicate resource creation.

- **Entire Qt framework**

Although Qt itself cannot be run in the functionally safe environment, it does contribute to the overall system by providing a reliable, stable, and proven UI-development platform for the main application. The modules that interact with the Qt Safe Renderer to control position and visibility of indicator icons will be part of this application.

Functionally Safe Qt in Different Domains

How would the proposed system decomposition fit in different problem domains? Here are a few use cases for three selected vertical markets that require functional safety.

Automotive

The primary use case for functionally safe Qt in the car is in digital instrument clusters. Knowing your engine's RPMs may not save your life but knowing that your brake system failed just might. Note that the rest of the vehicle contains many modules that may have safety requirements – anti-lock braking systems, steer-by-wire or drive-by-wire, engine control units, etc. – but as these components do not have GUIs, they would not need Qt. The other place in the car for a screen is the infotainment (or navigation) system. This does not typically have a functional safety requirement but, when it does, developers can use the Qt Safe Renderer for safety critical portions of the HMI.

Previous generations of car cockpits have used separate LED indicators for things like engine, brake, or airbag failure. However, in many of today's car designs, the entire driver interface is being hoisted onto a single LCD or OLED display. That means that to remain functionally safe, these digital instrument clusters must ensure indicators are controlled and contained independent from the main instrument cluster application. Qt Safe Renderer is perfectly designed for an automotive instrument cluster – indicators are displayed by code in a completely isolated container but can be repositioned by the main application to allow for different instrumentation layouts.

Duplicating the driver's traditional analog instrument cluster with a screen has also given designers new freedom with fully configurable gauges, sophisticated 3D rendered digital gauges, and 3D rotating vehicle models to more attractively and more accurately communicate with the driver. Even if designs are created within the Qt 3D Studio, 2D indicator assets can be used in conjunction with the Qt Safe Renderer as required.

The main application would be responsible for not only drawing the gauges (through Qt3D or QtQuick/QML) but also interfacing with the vehicle bus interface (CAN/MOST), steering wheel controls, and infotainment system. It would normally be built with either Qt for Device Creation or Qt for Automotive Suite (which is actually a superset of Qt for Device Creation).

Medical

Medical devices with the lowest patient risk are called Class I devices, although the specifics of the classification vary somewhat based on country (see table below). Examples would be gait analyzers, thermographic cameras, or blood pressure monitors – devices where the display can fail with no (or trivial) risk of harming a patient.

	Europe	United States	Canada
Highest safety level	Class III	Class III	Class IV
	Class IIb	Class II / III	Class III
	Class IIa	Class I / II	Class II
Lowest safety level	Class IIa	Class I	Class I

Table 3. Rough comparison of regional medical safety classifications [2]

Devices that are Class II (including the IIa/b European variants) pose more risk to a patient – generally, if these devices fail a patient may be either harmed or the failure may allow for a life-threatening condition to go undiagnosed. Examples of these types of devices are electrocardiographs (ECGs), electroencephalographs (EEGs), ultrasound machines, and stress exercise monitors. Like the automotive use case, these devices have some critical monitoring function that can be run in a safe container, and use canned icons to alert the user to a danger state. For example, an ultrasound machine may display the black and white ultrasound image in the main application while the safety portion flashes a large red exclamation point when it encounters any abnormalities.

Class III devices pose the highest risk if they fail. They are regulated strictly and require the highest level of certification. Class III devices are often responsible for providing life-support functions and if misused can present a risk of serious injury or death. Examples of these are infusion pumps, implantable nerve stimulators, implantable pacemakers, or automated defibrillators. Even though implantable devices clearly don't need a screen, they may be programmed with an external control pad running Qt, making the entire system a class III device.

[2] Medical device regulations, classification and submissions, MaRS Discovery District.

The need for more modern, more responsive, and more reliable user displays is most urgent for the nurses, doctors, and technicians running the equipment. These people need to make very quick decisions in life-or-death situations – delivering a drug, shocking a heart, sustaining breathing, or providing other life-critical functions – and they need medical devices with straightforward, intuitive, responsive, and reliable user interfaces. The importance of error-free human-machine interfaces in medical devices means that a great user interface isn't just cosmetic – it's critical.

Industrial Automation

Perhaps the most dramatic differences between products are within the industrial automation sector, which is often a catch-all category for products that don't clearly fit into other domains. Industrial automation devices with functional safety needs that also need LCD/OLED displays range widely: laboratory automation, robotic manufacturing, building automation, material inspection machines, CNC machines, warehouse management systems, and conveyor systems are a small sample.

Common to many of these systems is that a calibration or other fault could lead to a critical error and dangerous behavior if ignored. In this way, even these disparate systems are similar to the automotive example – a well-placed, prominent error indicator can ensure that an operator takes appropriate action to shut down the assembly line, turn off feed stocks, or whatever other means are needed to prevent cascading failure and injury should the device malfunction. Systems with the need for critical error indication states can be handled with the same type of system partitioning as discussed in the other examples – running Qt Safe Renderer in an isolated safety partition or virtual machine, while the remainder of the UI that doesn't require a safety certification remains outside the safe partition.

Tackling Functional Safety Challenges

A functional safety-auditing firm will be your best source of knowledge and guidance while you are undergoing the certification process. However, you may have many questions before entering into a functional safety project – especially if this is your first. While we would always recommend talking to the experts, here are a few assorted items to consider as you determine your path.

- Functionally safe products can end up saving money. While there is a considerable effort in tools, education, engineering, documentation, and process required to develop a functionally safe product, at the end of it all your company will have reassurance that it is building more reliable products. Functional safety provides a measure of insurance against liability lawsuits and publicly brand-damaging failures as well as simplifying regulatory compliance, especially for multiple markets. And by removing defects from your software early on, it can make a very real contribution to cost savings. A comprehensive analysis of costs-per-defect ("A Short History of the Cost Per Defect Metric", Jones, 2012), shows that creating excellent quality software can cost 33% less over the length of the program than average quality software.
- Software documentation is critical. Make sure you have your documentation in order, because lack of sufficient documentation is responsible for a surprising number of certification failures. One study ("Analysis of Premarket Review Times Under The 510(K) Program", US FDA, 2011) found that 20% of certification submissions were rejected due to lack of proper software documentation. The importance of your documentation is always better to understand before coding begins.
- Functional safety can't tell you how to design a good user interface but a bad user interface will have a negative impact on your certification. Bad user interfaces can cause user hesitation or control misuse, lead to misinterpretation of critical data, or make it possible to input erroneous information. When designing your product workflows, make sure you consider human factors issues with a safety component in mind. Some industries have standard guidelines for user interface design – for example, medical products can refer to ISO/IEC 62366:2007 and AAMI/ANSI HE75:2009 for usability concerns.
- Standards such as IEC 61508 don't require certification for individual components, which means that functionally safe products could use commercial-off-the-shelf (COTS) software – even if it wasn't certified. However, functional safety standards do require proof of dependability and suitability for COTS components, which is significantly easier to accomplish when those products have already been pre-certified. Certification for functional safety software components requires creation of specific specifications, documentation, reports, and plans that feed into the certification process. Clearly this cannot be accomplished without full support of the vendor.

Summary

Making embedded systems both functionally safe and user friendly is a modern global trend thanks to the growing demands of government agencies for safer products and of end-users for the attractive and intuitive interfaces to which they've increasingly become accustomed.

Qt provides the user-friendly interface for a great many embedded devices that must increasingly meet these functional safety standards. Thankfully it is feasible to certify the Qt framework by separating a system into functionally safe and non-functionally safe components – and to facilitate this process, we created the Qt Safe Renderer to make it easier to create safety-critical user interfaces. This functionally safe component not only allows for separation of the safety critical portions from the primary Qt system, it minimizes the impact of functional safety certification on the mainline software development while working within your current Qt workflow.

And if you need a Qt-savvy partner to help you through your functional safety journey, we're more than happy to help.

The Qt Company develops and delivers the Qt development framework under commercial and open source licenses. We enable the reuse of software code across all operating systems, platforms and screen types, from desktops and embedded systems to wearables and mobile devices. Qt is used by approximately one million developers worldwide and is the platform of choice for in-vehicle digital cockpits, automation systems, medical devices, Digital TV/STB and other business critical applications in 70+ industries. With more than 250 employees worldwide, the company is headquartered in Espoo, Finland and is listed on Nasdaq Helsinki Stock Exchange. To learn more visit **http://qt.io**