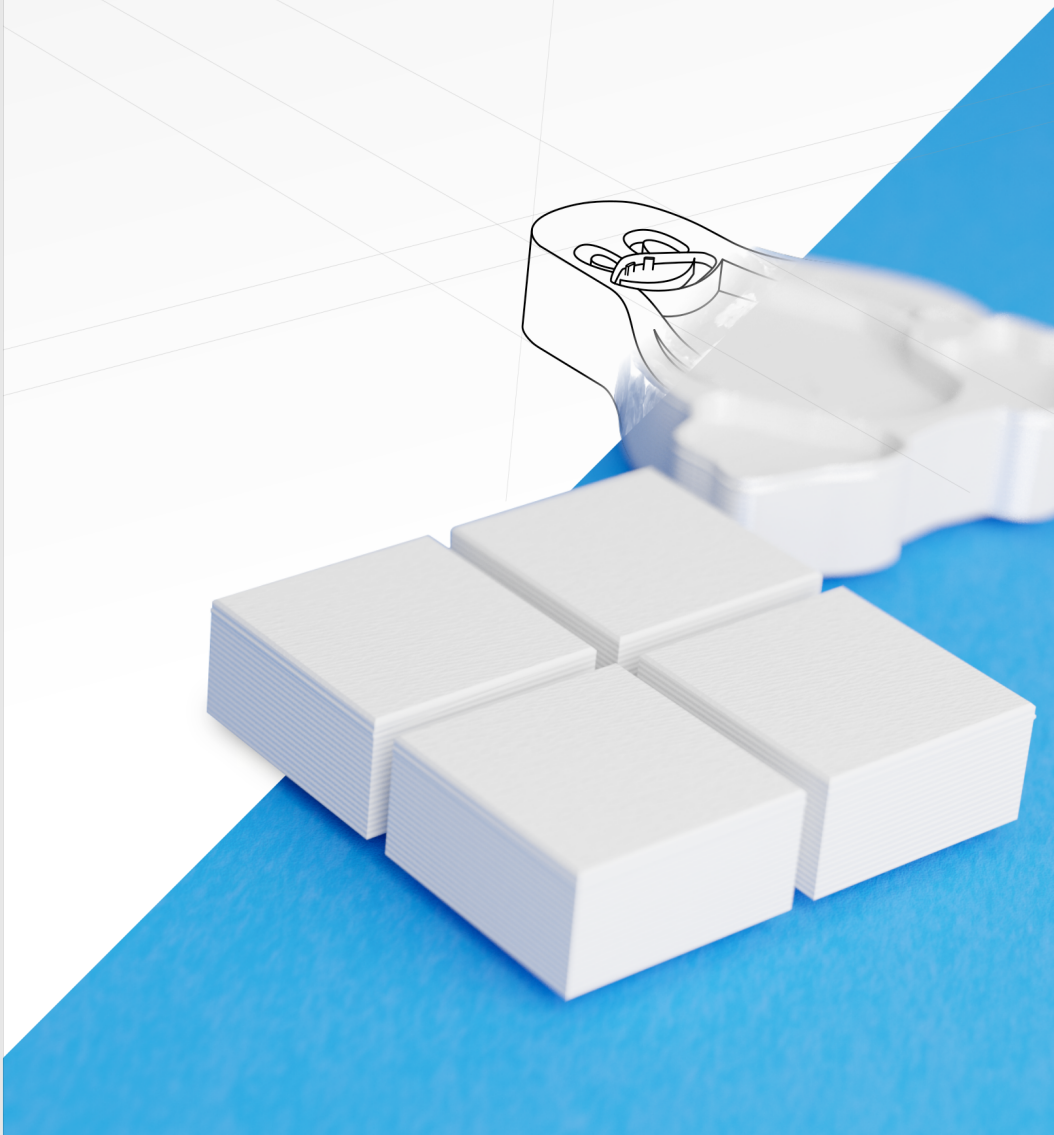
Windows CE to Embedded Linux Migration

A Practical Guide

Miłosz Kosobucki
Senior Software Engineer



Contents

Why migrate from Windows CE

Windows CE was once a popular choice for embedded development and still powers millions of devices, but it is clearly showing its age. It no longer receives updates, support ended in 2023, and Microsoft will stop selling new Windows CE licenses after May 31, 2028 (https://web.archive.org/web/20150410235645/http://www.microsoft.com/windowsembedded/en-us/product-lifecycles.aspx).

If your business sells WinCE-based devices, you need to move to another platform before this date. In the meantime, your devices may already contain serious security vulnerabilities, which is especially problematic in Europe in light of the Cyber Resilience Act applying fully from December 2027, as manufacturers will then bear complete responsibility and liability for the software they place on the EU market.

The time to act is **now**.

This whitepaper discusses the specific challenges of migrating away from Windows CE and practical strategies for approaching such a project, especially in areas that most embedded systems have to deal with and which often cause problem when migrating from WinCE.

For more general discussion of migration projects, not specifically related to WinCE, see KDAB's whitepaper on the subject: https://www.kdab.com/modernizing-legacy-systems-brochure/.

Why Linux?

There are several platforms you can migrate to from WinCE, including:

- **Windows IoT** – Microsoft's new offering for embedded devices – some shortcuts are possible due to similarities with the underlying OS. However, the application distribution model (UWP) and licensing is challenging. Not many vendors offer Windows IoT based boards.
- **QNX** – an established, robust OS with realtime capabilities. Its licensing and runtime costs can be appropriate for applications that explicitly require hard realtime features.
- **Embedded Linux**

This guide focuses on the last one. Linux is an excellent choice for embedded devices for the following reasons:

- It is scalable from tiny IoT appliances to giant supercomputers and has proven its versatility for almost 35 years.
- The majority of the software that constitutes the operating system is licensed with free/open software and doesn't require licensing fees.
- Source code is available for the whole OS except for certain proprietary device drivers.
- Due to Linux's popularity there is a large amount of learning material and a vibrant community of users and developers.

The Embedded Linux ecosystem is broad and can feel overwhelming, but with experienced guidance from a partner like KDAB it becomes very manageable. Like your migration effort, this whitepaper is divided into two parts:

- porting your application to technologies supported on Linux.
- adapting the OS specific parts so that they can talk to Linux.

Strategies for migrating your application

Migration vs rewrite

As discussed in the "10 Step Guide to Software Migration" the switch of technology may appear to be a good reason to start with a clean slate and rewrite the application completely.

The problem with a full rewrite is, that it discards not just the legacy code but also decades of accumulated bug fixes, edge-case handling, and institutional knowledge expressed in that code. In certain situations a rewrite is justified, but more often, if you can preserve the business logic of your product, a migration is preferable. That is the perspective this guide takes.

Preparation for migration

Preparing the code

If you have followed best practices of software engineering, the migration will become significantly easier.

A codebase that:

- is neatly layered, for example into UI, business, data access and hardware access

- is thoroughly tested with automated unit and integration tests
- abstracts OS-specific APIs behind interfaces, ideally with dependency injection

will be much easier to migrate than code that uses OS-specific types throughout the stack or does not have a test suite that lets you confidently make changes.

Refactoring towards this structure before starting the migration is usually a good idea. It lets you change code more fearlessly and often makes it possible to bring the business layer over to Linux almost unchanged. But beware of the temptation to refactor and clean during the migration. You will end up with two sources of problems.

"Give your teams dedicated time and space to learn Linux: formal trainings, small prototype projects, or internal hackathons all work well"

Preparing the people

Because of its Unix heritage, Linux is very different from Windows. If your company has been Windows-centric for a long time, your employees are likely very accustomed to the Microsoft ecosystem.

Give your teams dedicated time and space to learn Linux: formal trainings, small prototype projects, or internal hackathons all work well.

Consider hiring developers with Linux expertise and encourage them to share their knowledge through internal workshops.

Bottom line: Do not expect that Windows developers will "pick up" Linux on the side while driving a migration. The differences are substantial, and without proper onboarding you risk long, frustrating debugging sessions that could have been avoided with a basic understanding of the Linux environment.

The actual migration

Every embedded project is different, but from experience the migration effort typically clusters around three major areas:

- Business logic
- User interface (UI)
- OS and hardware interactions

The cost of migrating each area depends on your existing technologies and on how cleanly your code is structured. If the

UI communicates with the business logic through well-defined, portable interfaces, migration will be much faster.

Business logic

In a well-architectured system, the business layer is often written in a portable way and can be migrated with minimal changes. It is usually the layer that encodes most of your company's secret sauce. If it is not too coupled with the UI and OS communication layers aim to keep it as unchanged as possible. If it is entangled, first separate it from UI and OS concerns, then start the migration.

User interface

The UI is often the most challenging part. It likely uses a graphical framework that does not have a Linux equivalent. In such case, a complete rewrite of this layer is required.

OS and hardware touchpoints

If you followed good practices, your business layer talks to the OS through abstracted interfaces and does not make many assumptions about specific implementation. This area is covered in more detail in a later section.

Side note
If you think you need to abandon C# because you are moving to Linux there is good news. Read more in the "Migrating from C# with WinForms" section.

High level migration approaches

Staying in the source language

If your application's language is well supported on Linux, keep it. You will need to adjust to a different toolchain and some implementation details, but those issues outweigh the advantages of keeping a known and trusted codebase.

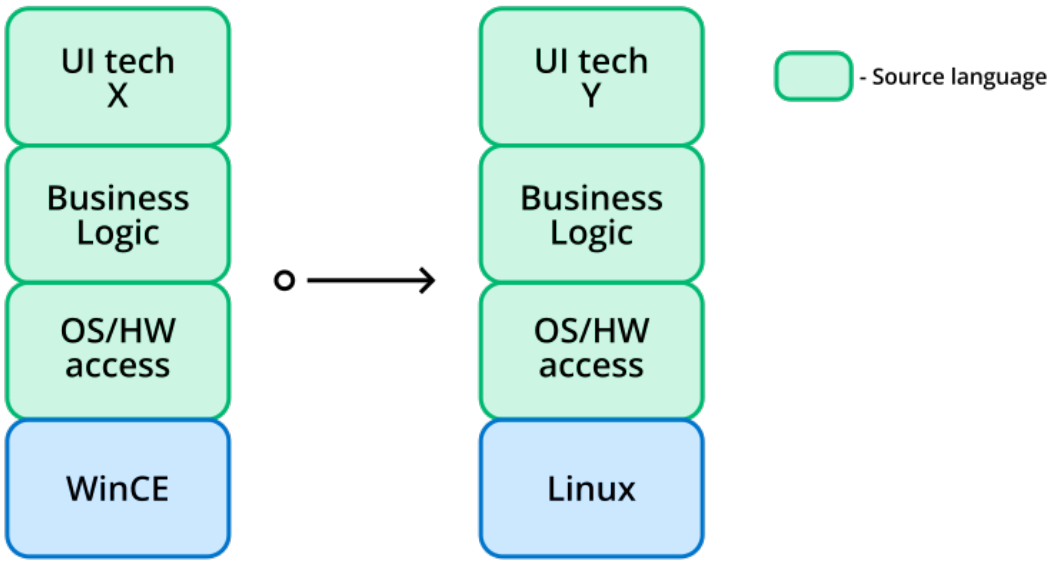


Figure 1: Staying in the source language

During migration, stay as close as possible to the original structure - even if that means sacrificing the elegance and idioms from the target language.

Switching a language

If your current language has poor Linux support, you may need to rewrite your software in a different one. During migration, stay as close as possible to the original structure - even if that means sacrificing the elegance and idioms from the target language. Mixing migration and refactoring is a recipe for disaster. Only start adapting your code to the new language after you have a working product.

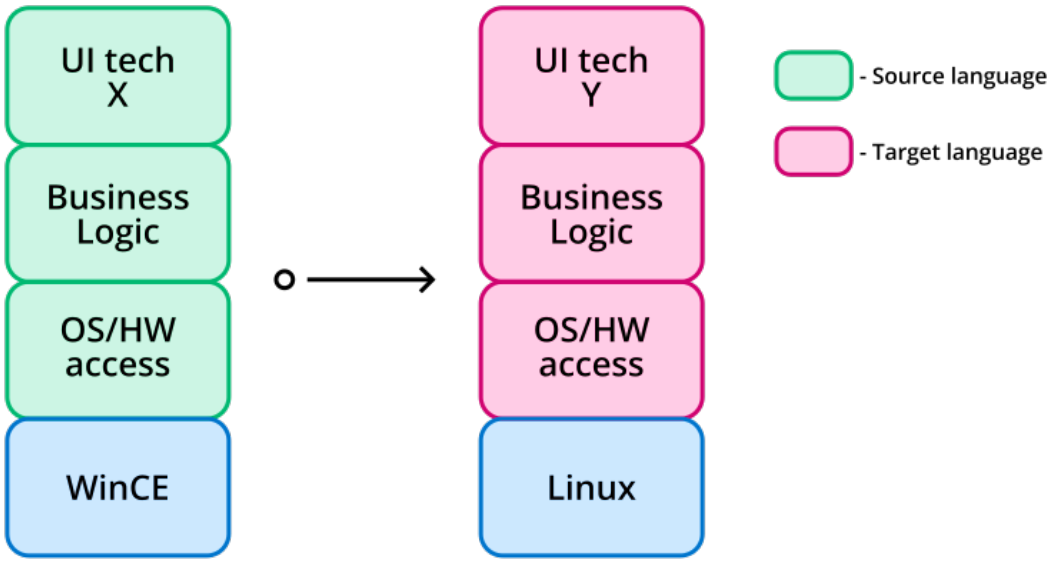


Figure 2: Switching a language

Separating the GUI into another process

If there is no suitable UI technology for the language used in your non-UI code and you want to avoid cost and risk of migrating that code to another language, you can move the UI into a separate process. The UI then communicates with the rest of the application over an inter-process communication solution (IPC). This gives you more UI technology options for the cost of more complicated architecture. Async cross-process RPC will always be more complicated than simple function calls and callbacks.

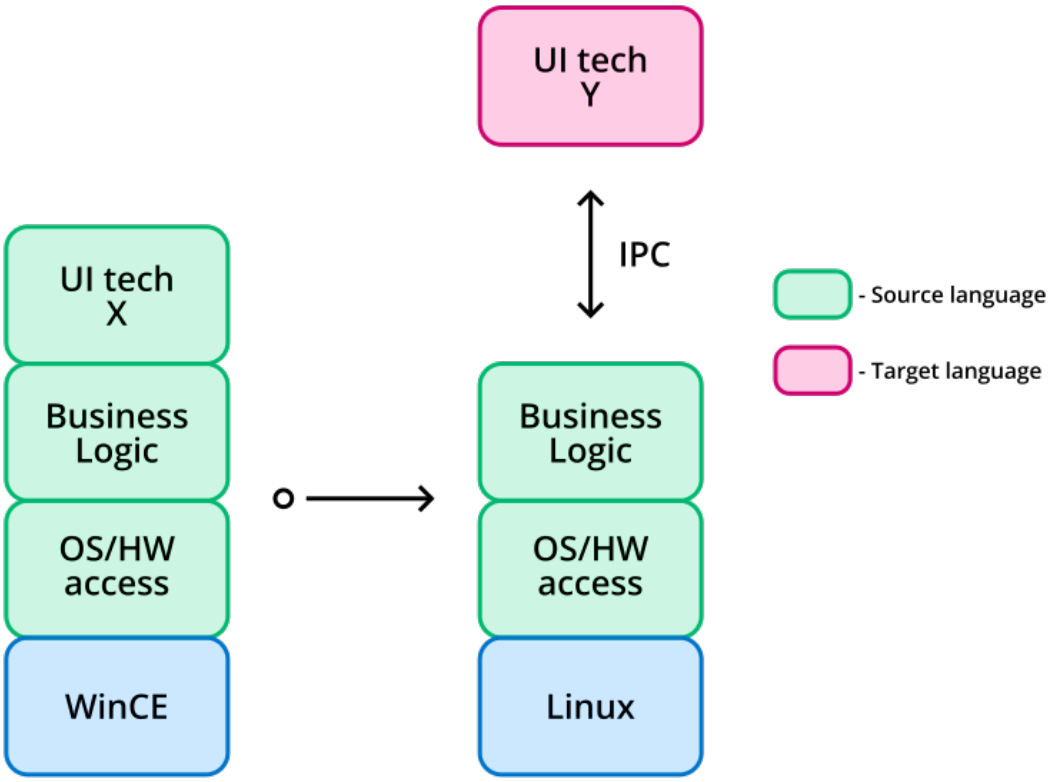


Figure 3: Separating the GUI into another process

Although potentially costly, this approach makes it possible to put the UI on a separate device.

Language-specific scenarios

C++

Migrating from C++ and Qt

This scenario likely requires the least effort. Both C++ and Qt are well supported on Linux, making the transition relatively smooth. You will probably need to update to a more recent Qt version, as the last one supported on WinCE is quite outdated. Fortunately, Qt upgrades and modernization are familiar territory for KDAB, where you can rely on our proven expertise.

Migrating from C++ and MFC

If your codebase primarily uses standard C++ types and has kept MFC dependencies out of the business logic, there is a good chance the business layer can be migrated with minimal changes.

The UI layer, however, will require a complete migration. Suitable options that work well with C++ include:

- Qt – a mature, crossplatform UI framework. KDAB offers dedicated MFCtoQt migration services with a gradual approach and a compatibility layer that lets MFC and Qt

coexist during the transition for better testability. See KDAB's whitepaper on the subject: https://www.kdab.com/mfc-to-qt-migration/.

- **Slint** – a declarative UI toolkit gaining traction in embedded scenarios, with strong C++ support and scalability from powerful SoCs down to microcontrollers. KDAB has been one of the earliest adopters and is an official service partner for Slint.
- **LVGL** – low level C library targeting primarily resource-constrained hardware

Migrating the build system

WinCE C++ projects are typically built as Visual Studio solutions. Those project files cannot serve as a native build system on Linux. You will need to migrate to a Linuxcapable build system for C++, with CMake being the defacto standard today. See: https://isocpp.org/files/papers/CppDevSurvey-2025-summary.pdf. KDAB provides CMake expertise and training to get your team up to speed.

Migrating from C# with WinForms

The idea that .NET and Linux do not fit together is outdated. Linux is a first-class platform for .NET these days. Since the advent of modern .NET, starting with .NET Core 1.0 released in 2014, Microsoft officially considers Linux a supported target. Unless your business logic uses Windows-specific .NET APIs, you should be able to migrate that code to Linux with relatively few changes.

Running the official Microsoft API compatibility analyzer tool is recommended (https://learn.microsoft.com/en-us/dotnet/core/porting/github-copilot-app-modernization/overview) to give you an overview of necessary changes.

Replacing Windows Forms

WinForms does not work on Linux. It is tightly coupled to Windows' windowing system and drawing APIs. Microsoft has not ported it to Linux.

Instead, there are some options for Linux-compatible C# GUIs:

- **AvaloniaUI** – a mature, open-source UI framework that supports Linux and Windows (in case you want to keep running and testing your code on Windows). It uses AXAML, a XAMLderived language, for UI definitions. The

Regardless of which option you choose, KDAB and its partners can support your migration to Linux.

Avalonia team recently announced a .NET MAUI Linux backend with Avalonia as the underlying implementation (https://avaloniaui.net/blog/net-maui-is-coming-to-linux-and-the-browser-powered-by-avalonia). This may be useful, if you have other divisions in your organization that work with MAUI.

- **Uno platform** - another XAMLbased C# UI framework that supports Windows, Linux, and other platforms, with commercial support options.

Regardless of the option you choose, KDAB and its partners can support your migration to Linux.

"Choosing Mono would effectively mean moving from one obsolete technology to another only slightly less obsolete"

What about Mono?

Mono is an older implementation of .NET that ran on Linux long before .NET Core was even planned.

It has a WinForms implementation for Linux, which can make it tempting to just patch the Mono incompatibilities in your application and avoid actual migration of the UI.

This approach is not recommended. Mono no longer has commercial support. Microsoft transferred it to the Wine project (https://github.com/mono/mono/issues/21796) mostly to provide a .NET compatibility layer for Windows application emulation under Linux. For the purpose of running embedded applications, it is effectively unmaintained.

Other problems you will face with Mono include:

- Recent versions are not packaged for popular distributions – you will need to compile it yourself to stay up to date
- No native Wayland support
- Performance price of the Windows compatibility layer

Choosing Mono would effectively mean moving from one obsolete technology to another only slightly less obsolete.

Migrating the OS touchpoints

Any nontrivial embedded application interacts with hardware like sensors and actuators, talks over the network, or read and write from storage devices. All these interactions cross the application/operating system boundary and this code will have to change, because Linux behaves very differently from WinCE in these areas.

In this section we will discuss the most common OS touchpoints and how to assemble the operating system itself.

Assembling the Operating System

With WinCE you likely got a system image from your hardware vendor, perhaps with custom drivers included.

On Linux there are more options. Your hardware vendor may still provide a reference system image, but it will often need modifications or extensions to match your product's requirements.

The full landscape of options to create such a system image is too broad to be presented within the scope of this guide. Technologies you may encounter include:

- **Yocto** – a meta-distribution that lets you build your own system from the ground up. A popular choice for the base Linux system.
- **Containerized Linux** – your application runs in a sandboxed container with a more generic Linux underneath. Torizon from Toradex is one example of this approach. KDAB's whitepaper on Containerization on Embedded Linux provides more detail.

System control with systemd (startup, reboot, date and time)

Your application will not be the only one running with Linux. The operating system runs numerous processes that control aspects like time synchronization, network management, DNS resolution and more.

These are orchestrated by a service manager: the first user-space program that the Linux kernel starts after booting. It will also start your application.

In recent years, the most popular choice for this role has been systemd (https://systemd.io/). Apart from process management it also provides a set of tools and daemons that run in the background and implement important OS functionality in a distribution-agnostic way.

You can talk to systemd over D-Bus (https://www.freedesktop.org/wiki/Software/dbus/), a de-facto standard inter-process communication technology. Most mainstream programming languages provide DBus bindings. For C++ we recommend sdbus-cpp and for C# the Tmds.Dbus package.

The D-Bus interfaces implemented by systemd and adjacent tools that are particularly relevant include:

- **login1** – for controlling user sessions, logins, rebooting the system
- **timedate1** – for working with time and date settings, time zones, and NTP configuration.
- **hostname1** – basic information about the machine: kernel version, hostname info, firmware version for example

Hardware communication: CAN, gpio, I²C etc.

Common hardware interfaces such as CAN, GPIO, and I²C are all well supported on Linux, but the way your application talks to them will need to change. If your application is well-structured, it should be a matter of providing a new implementation of an interface and wiring up dependency injection.

C#

For modern C#, support for several of these interfaces is already available in the language's standard library:

- `System.Device.I2c` namespace: https://learn.microsoft.com/en-us/dotnet/api/system.device.i2c
- `System.Device.Gpio` namespace: https://learn.microsoft.com/en-us/dotnet/api/system.device.gpio

For other interfaces, libraries developed by the community are available, like the C# wrapper for SocketCAN.

C++

For popular protocols, there are usually Linux-compatible libraries in C or C++, likely already packaged for your distro and

supported by Yocto. For example:

- **I²C – libi2c**: https://github.com/amaork/libi2c
- **gpio – libgpiod**: https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git/
- **CAN bus – SocketCAN**: https://docs.kernel.org/networking/can.html

"There are multiple projects that expose high-level networking functionality on top of the low-level kernel interfaces"

Networking

If your device is connected to a network — and these days it likely is — you will probably need to allow users to configure connection settings, select WiFi network, provide a password, or connect to a VPN.

On Linux, networking is configured very differently than on Windows CE. First of all, there is no single, OS-provided way. There are multiple projects that expose high-level networking functionality on top of the low-level kernel interfaces. Most of them can be controlled via DBus, using the same messaging system mentioned above.

The most common networking packages include:

- **NetworkManager** – probably the most popular choice. From C and C++ you can talk to it through the libnm library. In other languages, you talk to NetworkManager's D-Bus endpoint through your language's D-Bus bindings. NetworkManager is very complete in its support of networking technologies. For certain usecases it can be too heavy though.
- **systemd-networkd** – one of the sub-tools of systemd. Not as feature-complete as NetworkManager but lighter in implementation and also uses D-Bus as an API.
- **Connman** – an embedded-oriented networking manager. It also exposes a D-Bus interface for programmatic control.

Storage (SD cards, USB sticks)

If your device uses dynamically attached storage like SD cards and USB sticks you need a way to automatically mount them to the device's file system. A common solution for this is the udisks daemon, which listens to kernel hardware events and mounts detected devices.

You may also query information about storage devices handled by udisks using its D-Bus API.

Remote access

Some devices need to be operated or monitored remotely. Depending on your situation and requirements, options include:

- **AvaloniaUI VNC backend** – if you use Avalonia as your UI framework, the Headless VNC rendering backend is an option.
- **Weston VNC and RDP backends** – the Weston compositor, that you may use for window management, can expose VNC or RDP servers.

Registry

There is no direct replacement for the Windows Registry. Programs usually store their configuration in files on the file system according to the "XDG Base Directory Specification" (https://specifications.freedesktop.org/basedir/latest/). Common formats used for that are Ini, JSON, YAML and TOML. Sometimes, lightweight databases like SQLite are used, if transactionality is important.

Touchscreen calibration

If your device uses a restive touchscreen — still common in industrial environments where users wear gloves —, you need a way to calibrate it.

Windows CE provided a builtin calibration tool. On Linux, calibration depends on your chosen windowing and input setup. Common solutions:

- **Weston's weston-calibrator** – if your UI runs under Weston compositor, it contains a tool called weston-calibrator that launches a simple fullscreen calibrator application. Documentation is sparse, but it works reliably in practice.
- **Tslib** – a library that sits between the device's input system and your application. It also provides a calibration application.

Key Insights

Migrating away from Windows CE is not just a simple technical task but a major shift in how your project works. It will require changes not only to your code but also its operating environment and tooling.

In summary, here are the key insights:

- **Do not wait.** Start as soon as possible. Technical and regulatory pressure will only increase. It's better to migrate when there's still time than to scramble just before Windows CE sunset deadlines.
- **Prepare.** Both technically and organizationally. Gather knowledge and give your employees space to learn the new tools. Invest in architecture improvements and test coverage before the migration starts to reduce the risks.
- **Keep the good parts.** Try to preserve your application's business logic, if possible.
- **Learn** about the hardware and OS interactions your application has and how can they be handled in Linux.
- **Do not migrate and refactor together.** Do one after the other, but never simultaneously. ■

Get in touch

With numerous migration projects under our belt, KDAB is your partner for modernizing your product. If you have any questions or would like to discuss your migration project, we look forward to hearing from you via *info@kdab.com*

About the KDAB Group

The KDAB Group is the world's leading software consultancy for architecture, development and design of Qt, C++ and OpenGL applications across desktop, embedded and mobile platforms. KDAB is the biggest independent contributor to Qt and is the world's first ISO 9001 certified Qt consulting and development company. Our experts build runtimes, mix native and web technologies, solve hardware stack performance issues and porting problems for hundreds of customers, many among the Fortune 500. KDAB's tools and extensive experience in creating, debugging, profiling and porting complex applications help developers worldwide to deliver successful projects. KDAB's trainers, all full-time developers, provide market leading, hands-on, training for Qt, OpenGL and modern C++ in multiple languages.

www.kdab.com