The Qt Company

Embedded product planning and requirements guide

Introduction

There has never been a better time to build embedded products. A profusion of open-source software stacks, community resources, silicon components, and electronics assemblies have made it possible to develop polished, feature-rich products in a minimal amount of time.

Yet paradoxically, the abundance of technology choices without clear and definitive selection criteria can derail development into non-productive research projects and analysis paralysis. Worse yet, bets on technology losers can force engineering teams into market-delaying re-work exercises or early product obsolescence.

To simplify the information overload at the outset of your Internet of Things (IoT) or embedded project, we've examined the most important criteria to consider, put together a list of the most used technologies, and rated them in easy-to-compare categories.

We sincerely hope that this guide can reduce the effort and risk in building your next product.

Contents

OUR METHODOLOGY

How do you approach building a new embedded or IoT device? The legacy method for doing this was relatively straightforward: outline your requirements, design your hardware board, and develop software to run on it.

However, this method no longer works well today. Competition is fierce, market windows are short, and customer expectations are high. These pressures have two direct effects. One is that, rather than hardware, software now dictates product design. This is because software development requires significant time and resources, and can make or break a device's overall customer experience. The other is that companies must be nimbler than ever. They need to plan for constant change.

To develop modern embedded products, you need to take four very important steps.

4 important steps for developing modern embedded products

1

Set the scope.

Write your **market requirements document** to describe your product's feature set and usability, just as you would typically. Just make sure you have a serious discussion between senior management, marketing, support, and engineering about how far and wide that product's software might extend. Make sure you have talked to your customers and understood your competition. From all of this, you'll have developed some "future-proofing" factors that you should consider adding to the requirements.

2

Determine the process.

Understand the interaction between your software developers, UX designers, and hardware engineers. Ensure that the tools and workflows you develop support efficient and reliable product development as well as your team's ability to move fast and adapt quickly.

3

Select the software.

A product's software stack determines what it can do and what it can run on. Pick software that supports the widest variety of hardware with a credible support and maintenance plan. Choose software with the fewest external dependencies on libraries or components. Opt for software with industry standards, stable APIs and reusable software components.

4

Select the hardware.

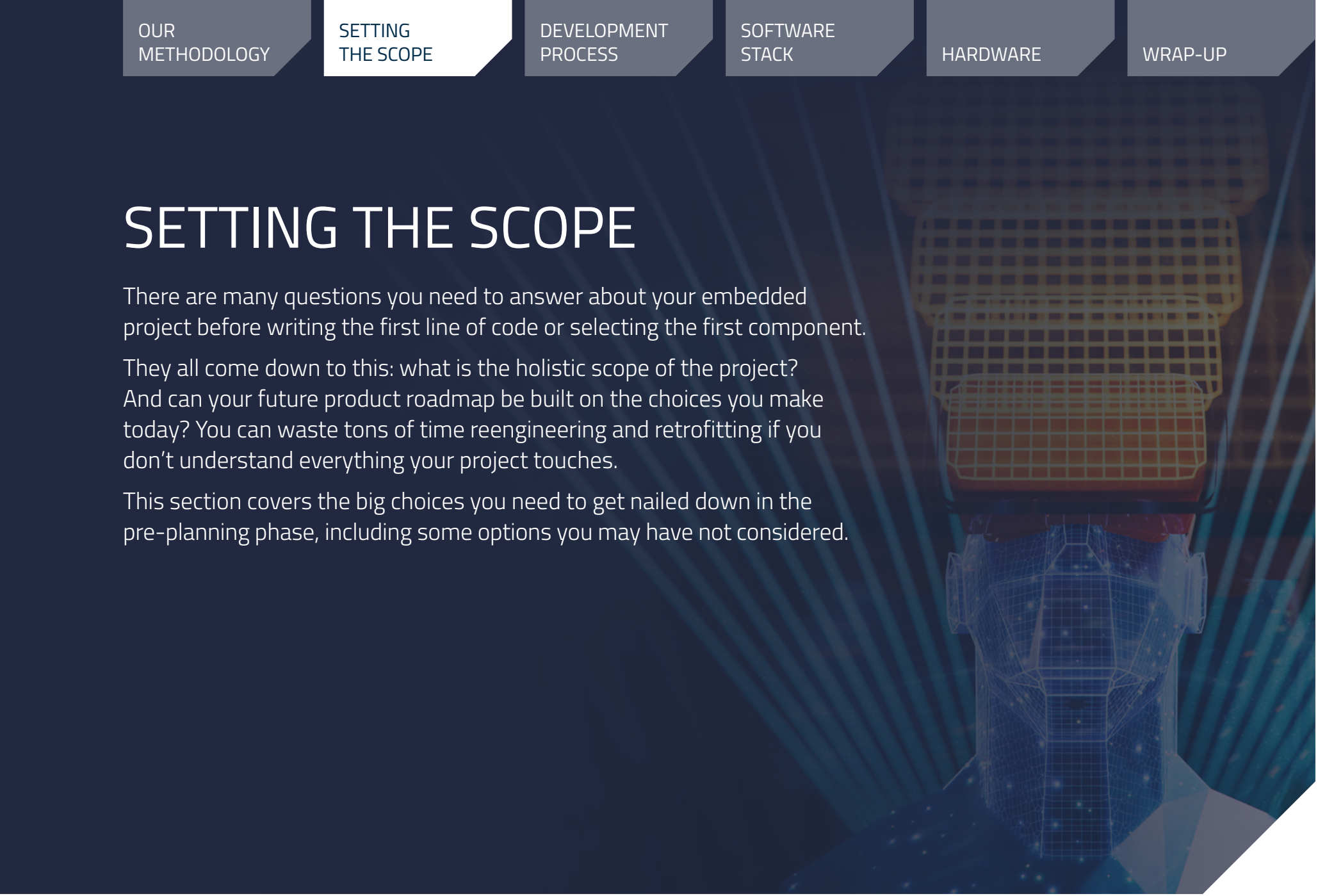
Pick the hardware platform that meets your requirements and software. Cost, reliability, and dependable delivery are traditional criteria; finding hardware platforms that can scale both up and down is a bonus.

SETTING THE SCOPE

There are many questions you need to answer about your embedded project before writing the first line of code or selecting the first component.

They all come down to this: what is the holistic scope of the project? And can your future product roadmap be built on the choices you make today? You can waste tons of time reengineering and retrofitting if you don't understand everything your project touches.

This section covers the big choices you need to get nailed down in the pre-planning phase, including some options you may have not considered.

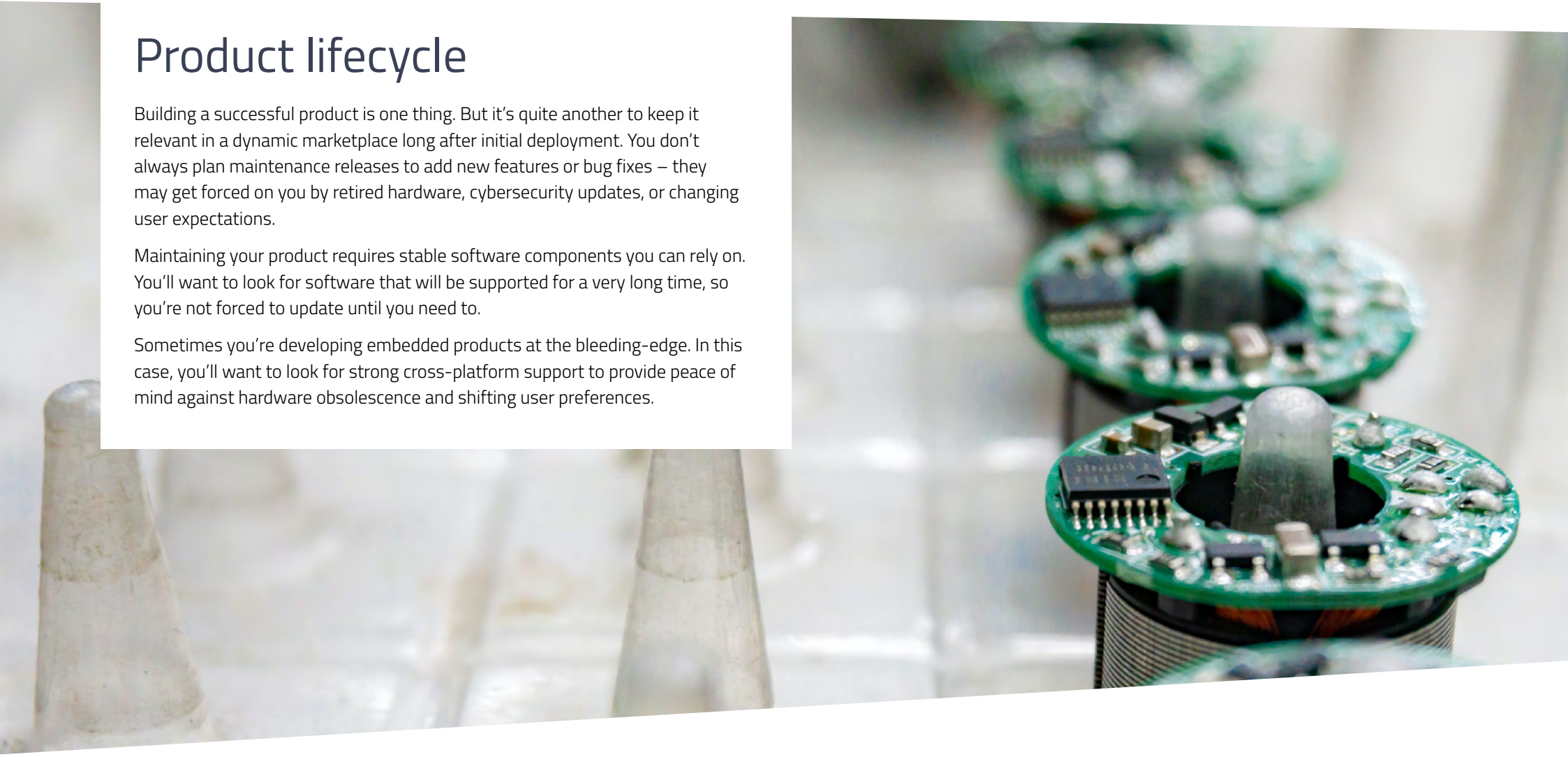


Product lifecycle

Building a successful product is one thing. But it's quite another to keep it relevant in a dynamic marketplace long after initial deployment. You don't always plan maintenance releases to add new features or bug fixes – they may get forced on you by retired hardware, cybersecurity updates, or changing user expectations.

Maintaining your product requires stable software components you can rely on. You'll want to look for software that will be supported for a very long time, so you're not forced to update until you need to.

Sometimes you're developing embedded products at the bleeding-edge. In this case, you'll want to look for strong cross-platform support to provide peace of mind against hardware obsolescence and shifting user preferences.



Reusability and cross-platform

Good code is hard to write, so reuse it as often as possible on as many platforms as possible. Determine the processor architectures, the platforms, and the operating systems that your software stack needs to run on. Focus your software's external dependencies into a few well-controlled modules, and you'll find reusing it much easier.

Every software component may not run on every single combination, but you want to avoid inventing custom workarounds when possible. And while it's great to support open source, don't get forced into contributing to an open-source module your project uses just to ensure it supports your required hardware/software combination. Pick software that already has support for what you need.

WHEN TO ABSTRACT

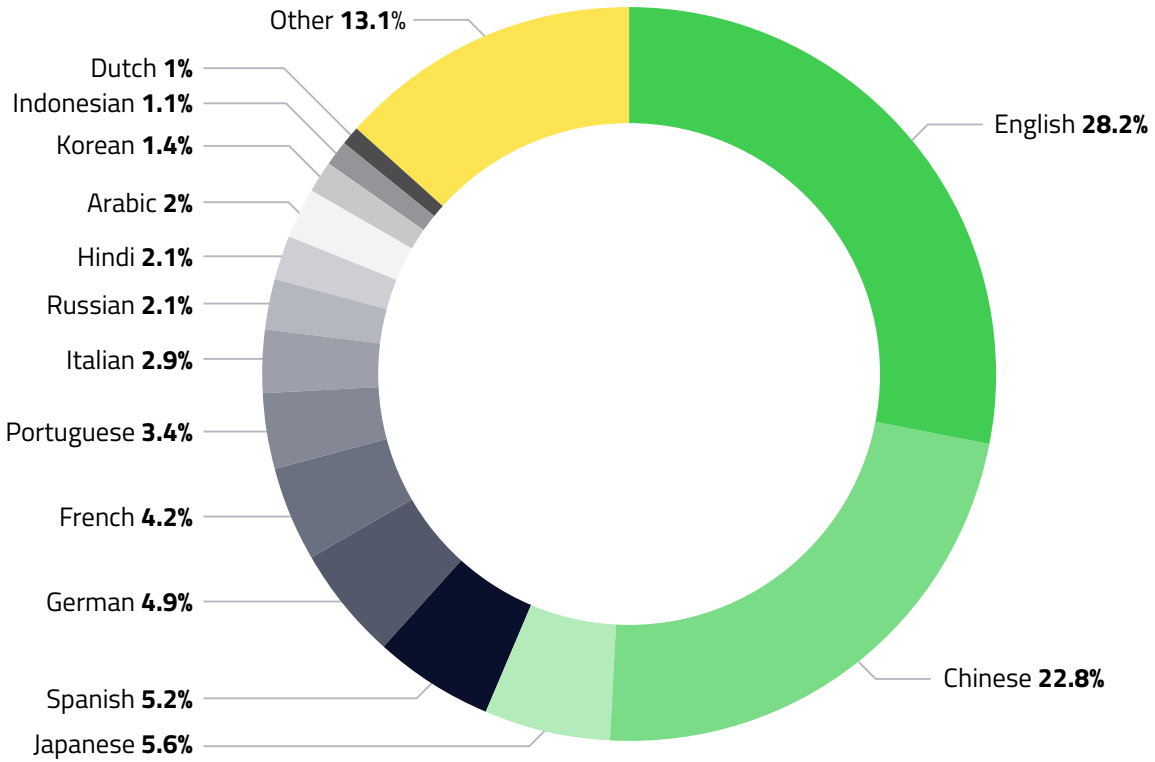
If your software doesn't use cross-platform compatible software from the start, you'll probably need to create abstraction layers to insulate your software from changing dependencies. Admittedly, this requires additional up-front work. However, a decent abstraction implemented early can save a huge amount of time compared to retrofitting the entire code base after-the-fact.

Everything doesn't need abstraction – focus on the areas that are most likely to change when swapping out hardware, OS, or third-party software components. Don't try to get it perfect, since no matter how well thought-out the design, your abstraction may not accommodate all future changes. It's okay if it needs slight tweaks when you change directions.

Internationalization

Are you going to restrict your product to a single language or region? Probably not, unless you can walk away from a global audience and the ability to grow your market share. Even if your initial release is in English only, make sure that you develop your code using languages, tools, and UI frameworks that make an internationalization (i18n) transition easy. You'll want string constants that can be transparently resourced for developer and translator ease, and UIs that are separated from the business logic.

Gross domestic product (GDP) by language



Latest available estimates: **Unicode technical note #13**.

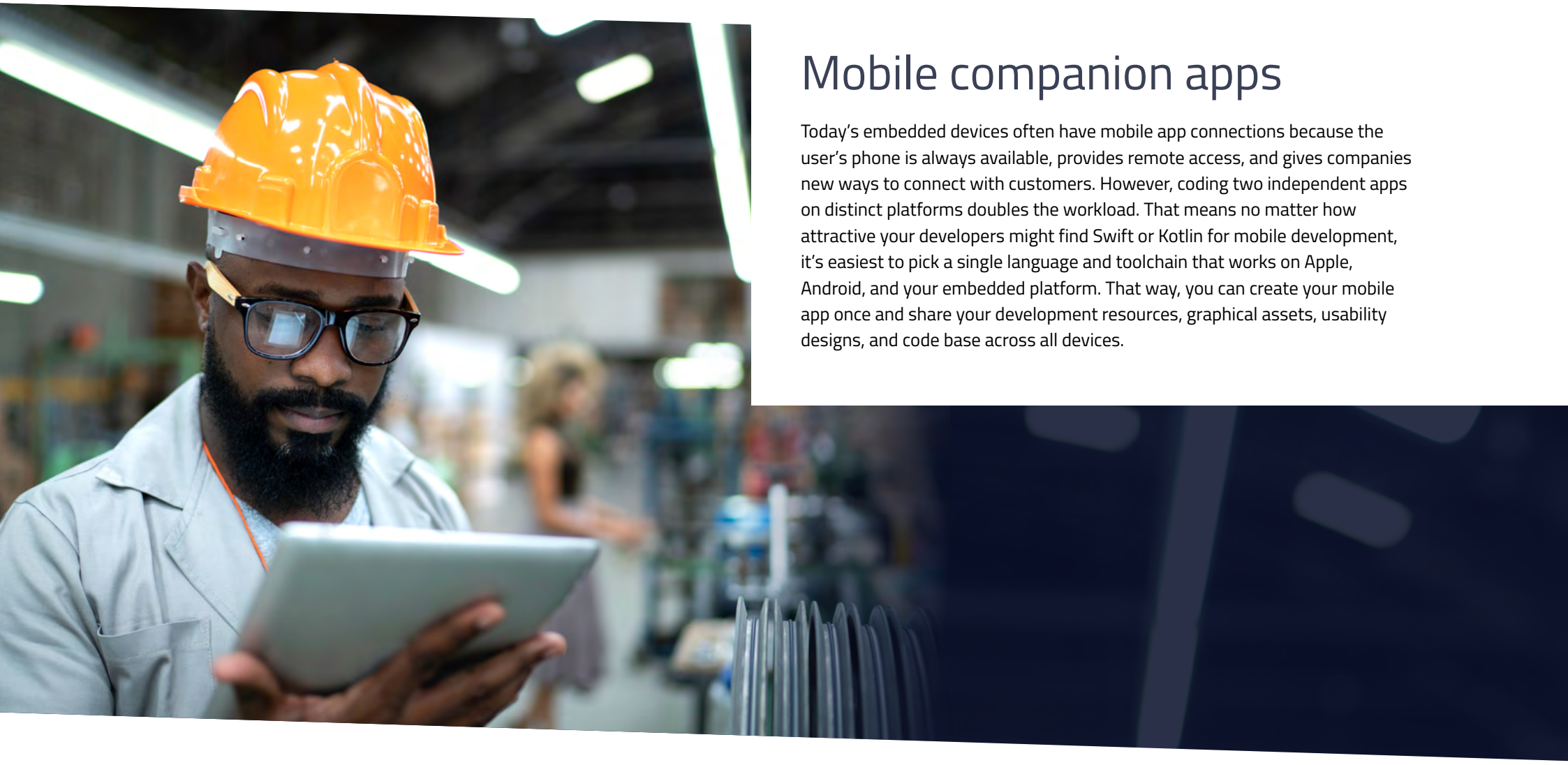
Connectivity, updatability, and security

It's a given that most embedded devices today need connectivity, especially to support over-the-air (OTA) updates for security patches, bug fixes, and feature updates. Decide whether updates should be driven by the device, your backend, your user, or a combination of the three. While many developers attempt to create their own OTA update scheme, there's enough complexity and consequence here to strongly consider using a commercial off-the-shelf solution instead. If you go the open-source route, make sure you pick a project that doesn't only support your hardware but also accounts for active maintenance in its roadmap. Also, be sure to plan for the extra storage your OTA solution requires – some of which will more than double your flash footprint.

Connections require software security, and this security must be built-in, not bolted-on. Software should be protected from the worst when it comes to uploaded files, user input, and network connections. Ensure your software engineers are trained in cybersecurity best practices and work with cybersecurity experts when needed. Hardware needs security scrutiny too: don't populate ports or components that were specifically added for development, debugging, or testing on your production systems, and make sure that you're taking advantage of hardware locks and security features like Secure Core or TrustZone.

LOOK FOR THESE OTA FEATURES

- Authentication and encryption measures to prevent OTA misuse
- Patch verification, image validation, and retry mechanisms for reliable delivery
- Integrated build tools for automatic patch generation
- Diagnostic reporting to track in-field software versions and update attempts
- Difference computing and compression to minimize package size
- Delivery throttling and variant segmenting to reduce server load
- Rollback capability to handle bad deployment emergencies



Mobile companion apps

Today's embedded devices often have mobile app connections because the user's phone is always available, provides remote access, and gives companies new ways to connect with customers. However, coding two independent apps on distinct platforms doubles the workload. That means no matter how attractive your developers might find Swift or Kotlin for mobile development, it's easiest to pick a single language and toolchain that works on Apple, Android, and your embedded platform. That way, you can create your mobile app once and share your development resources, graphical assets, usability designs, and code base across all devices.

Cloud and beyond

There are, of course, many other capabilities you can consider for your IoT or embedded product. Here are several.

Cloud-powered.

Divide your device's feature implementation between the edge and the cloud, letting you add instant features to every product in the field and offload portions of your product's logic and resources to a scalable alternative.

Integrated artificial intelligence (AI).

Collect data about hardware faults, log information, and usage patterns, and use machine learning to build products with better UIs and smarter behavior.

Digital twins.

Gather data on your device's states and status, allowing you to **virtualize your physical device** on a desktop or mobile.

Blockchain.

Add protection with a verifiable, unchangeable, and trusted record of transactions distributed across systems.

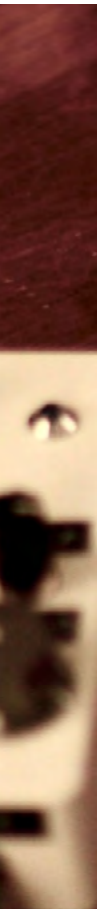
Did you know?

Although **blockchain** is typically used for currency, it can also be useful for embedded devices whenever trusted data storage is critical:

- **Recording** sensor data for compliance and regulatory review
- **Validating** messages for third-party authentication
- **Logging** tamper-proof data for process oversight
- **Storing** manufacturing information for certification
- **Sharing** calibration records for measurement certainty

DEVELOPMENT PROCESS

The goal at a project's outset should be to develop workflows that shorten development time without sacrificing (or even improving) quality. This includes streamlining the workflow between four main functions: UX design, software development, hardware engineering, and quality assurance (QA). Also consider better tools, cooperative design, and agile methodologies.

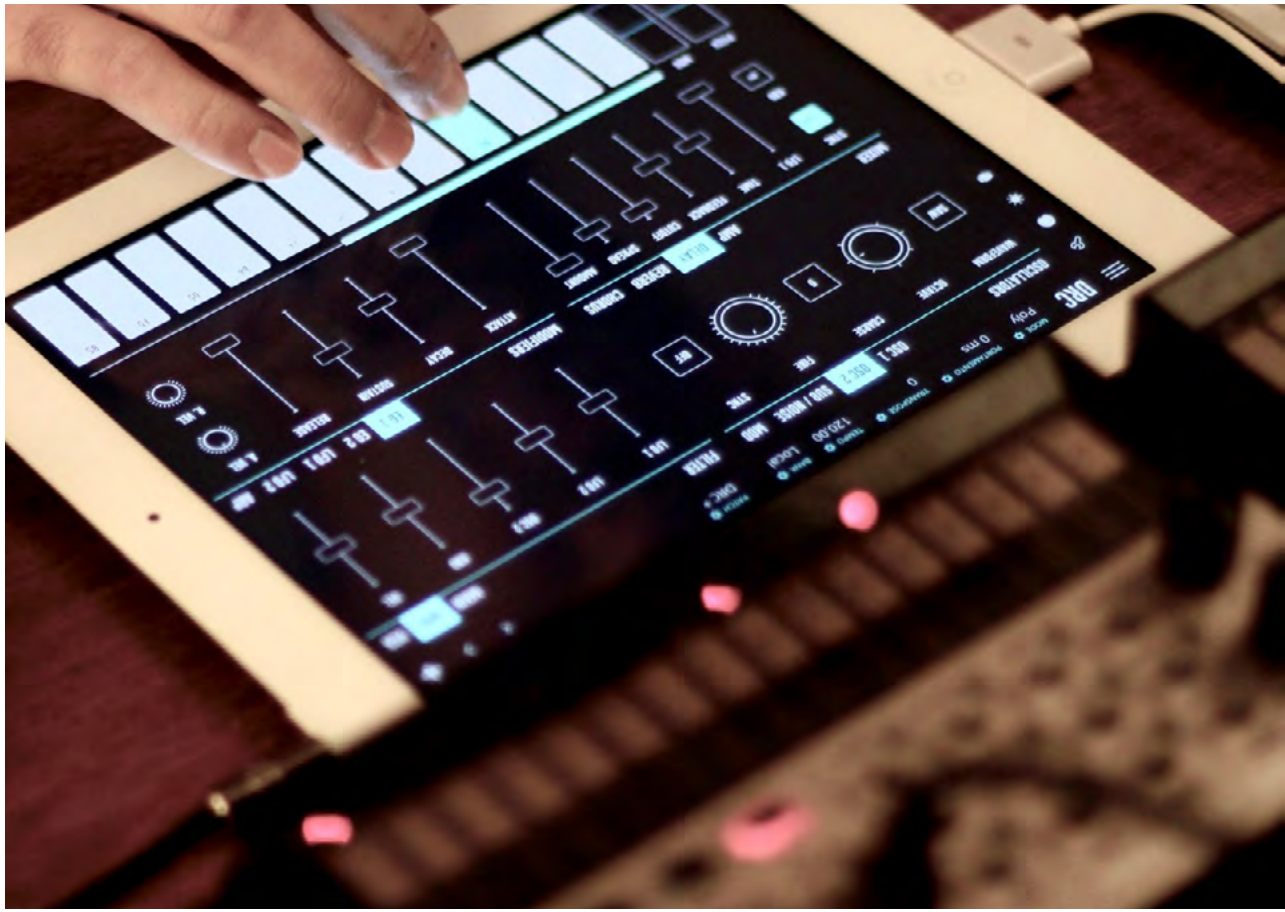


Software development

You want your software development toolchain to be easy to learn yet efficient to use, so new developers pick it up quickly while experienced ones churn out quality code. Your tools should generate optimal results – small binaries, fast executables, and a resource-light run-time. While many open-source options are available and excellent, don't be afraid to spend money on tools if they help streamline your development – it's worth the investment.

Rapid but stable tools

Ensure that you pick tools in active development so that your team can get access to new programming paradigms, more efficient methodologies, support for new hardware, and the ever-important bug fixes. And while you want active third-party tools, be certain the software tools you rely on aren't changing so fast or unpredictably that it introduces extra friction into your development. Sourcing most of your tools from the same place can minimize distribution and compatibility issues among your team.



Building

The build process is the unsung hero of software development: it enables consistent, reliable, and tracked software builds. While almost every build system will have custom components, try to conform your build tools and scripts as much as possible to industry standards and popularly used tools. People have already solved the problems your team has – don't be afraid to leverage them.

Testing

Lacking an adequate test framework can kill a project. The more you can rely on your selected tool suite to help provide your test scaffolding and continuous integration (CI) needs, the less you will need to manually create. Another important benefit of using existing commercial or open-source tools is that these tools will be feature-rich and properly maintained compared to your home-grown solution. Don't scrimp on testing since it's much better to find problems before integration – and before customers do.

UX design and UX workflow

User experience (UX) is the most definitive way to differentiate your product today; people become loyal to a brand when they have a positive experience. The best user interfaces combine the talents of both designers and developers; they also have a framework that supports both skillsets to work simultaneously on a project.

An iterative workflow between design and development helps build better products faster by avoiding wasted time in several areas:

- **reimplemented design screens**
- **design errors from misunderstood specifications or incomplete hand-offs**
- **stifled usability from rigid processes of traditional UX workflows**

Designers need to be able to create, visualize, and prototype the all-important user experience using artwork developed with their preferred tools. You want these tools to support a clean separation of user interface (UI) and business logic so that designers can test and integrate improved UI designs throughout the software development process.

Pulling off a smooth UX workflow within your company requires the right corporate culture. **Read this whitepaper** for an in-depth discussion about how your UX is critical to your project's success and how to develop a UX-supportive culture within your organization.

Creating design prototypes that can run on real target hardware from an early stage will eliminate some unnecessary inter-departmental feedback loops. Running UI designs on actual embedded hardware lets designers confirm real-life visual qualities such as screen size, aspect ratio, and color depth. More importantly though, this provides immediate feedback on performance. Since UI design is driven by mobile phones that often have higher performance than their embedded equivalents, designers can understand when their UIs are overkill or underperforming so they can tweak them before the design is done.

Integrated development environment

An integrated development environment (IDE) is where programmers spend the majority of their time coding, testing, and debugging. But a good IDE needs to do much more.

Integration

Look for fully-integrated tools that can support the entire software development process – designing, writing, building, debugging, profiling, testing, localizing, and deploying. IDEs that work with real targets, as well as some form of target emulation, let development continue even when there isn't enough prototype hardware to go around.

GUI design

Designers create a product's graphical user interface (GUI) with WYSIWYG (what you see is what you get) tools to build screens with controls and visual elements. GUI design should be directly supported by your IDE, both on a workstation and a target. This will be a big time-saver, especially during prototyping and usability fine-tuning. It's also a necessity to find an environment that allows designers to build UIs without special training.

Asset import

Look for IDEs that can integrate with designer tools (such as Adobe Photoshop, Autodesk 3ds Max, Blender, Figma, Maya, Modo, or Sketch) so artists can get those assets directly into your project. If the tools can also import UI actions, animations, and interactions, it lets these UI elements be specified by the design team rather than reimplemented by developers.

Project management

Version control and build systems may not be glamourous, but they are absolute necessities for reliable software and a productive development environment. An IDE that has out-of-the-box integration with your preferred revision control system and build tools makes engineers' lives easier and more efficient.

Expansion

A good plug-in ecosystem helps the IDE support your company's workflow and tools, and also lets developers adapt it to their individual preferences and development timesavers. Plug-ins for static code analysis and performance metrics give developers critical insights into their software to both fix and optimize it.

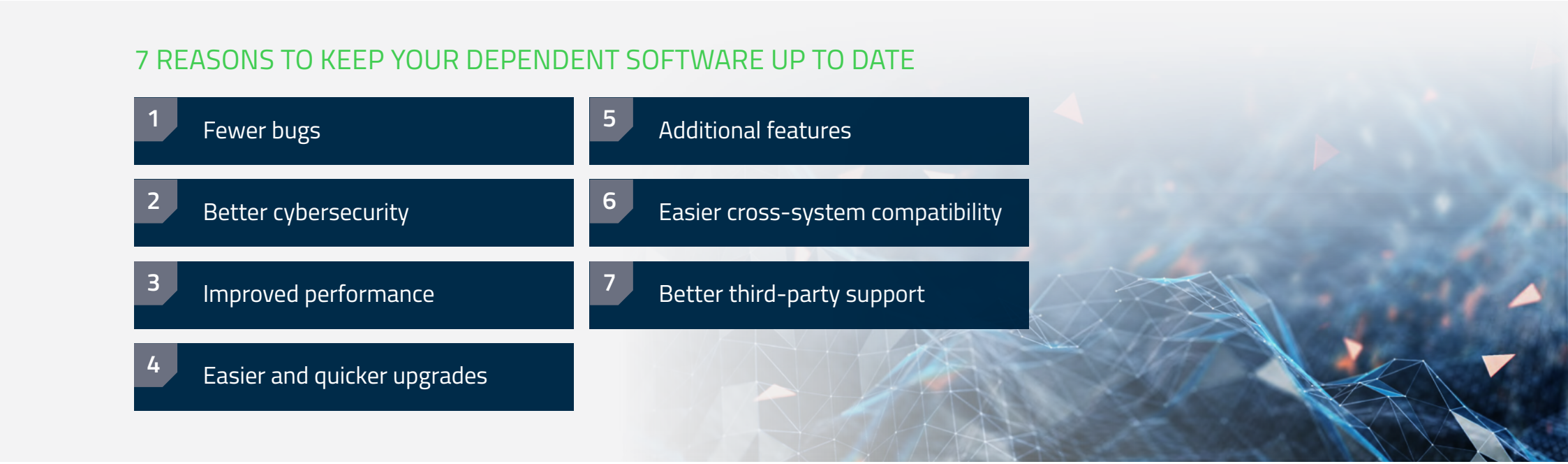
Maintenance

How are you going to manage product maintenance releases? We're not talking about your own bugs and feature updates but rather changes in the external software you use. All dependencies you bring into your product – operating system, drivers, libraries, frameworks, and tools – are changing at their own pace in ways that you can't control. These can introduce problems like broken application programming interfaces (APIs), conflicting application binary interfaces (ABIs), incompatible toolchains, and deprecated software.

Consider incorporating software building blocks into your application that provide long-term support, bug fixes, and cybersecurity patches even on older releases. Because of this, you'll want your third-party inclusions to have a strong and thriving developer community.

7 REASONS TO KEEP YOUR DEPENDENT SOFTWARE UP TO DATE

1. Fewer bugs
2. Better cybersecurity
3. Improved performance
4. Easier and quicker upgrades
5. Additional features
6. Easier cross-system compatibility
7. Better third-party support



Community and documentation

Finally, look for big ecosystems. How widely supported your software tools are can make a huge difference in solving tough problems, finding already available and compatible libraries, or hiring new talent.

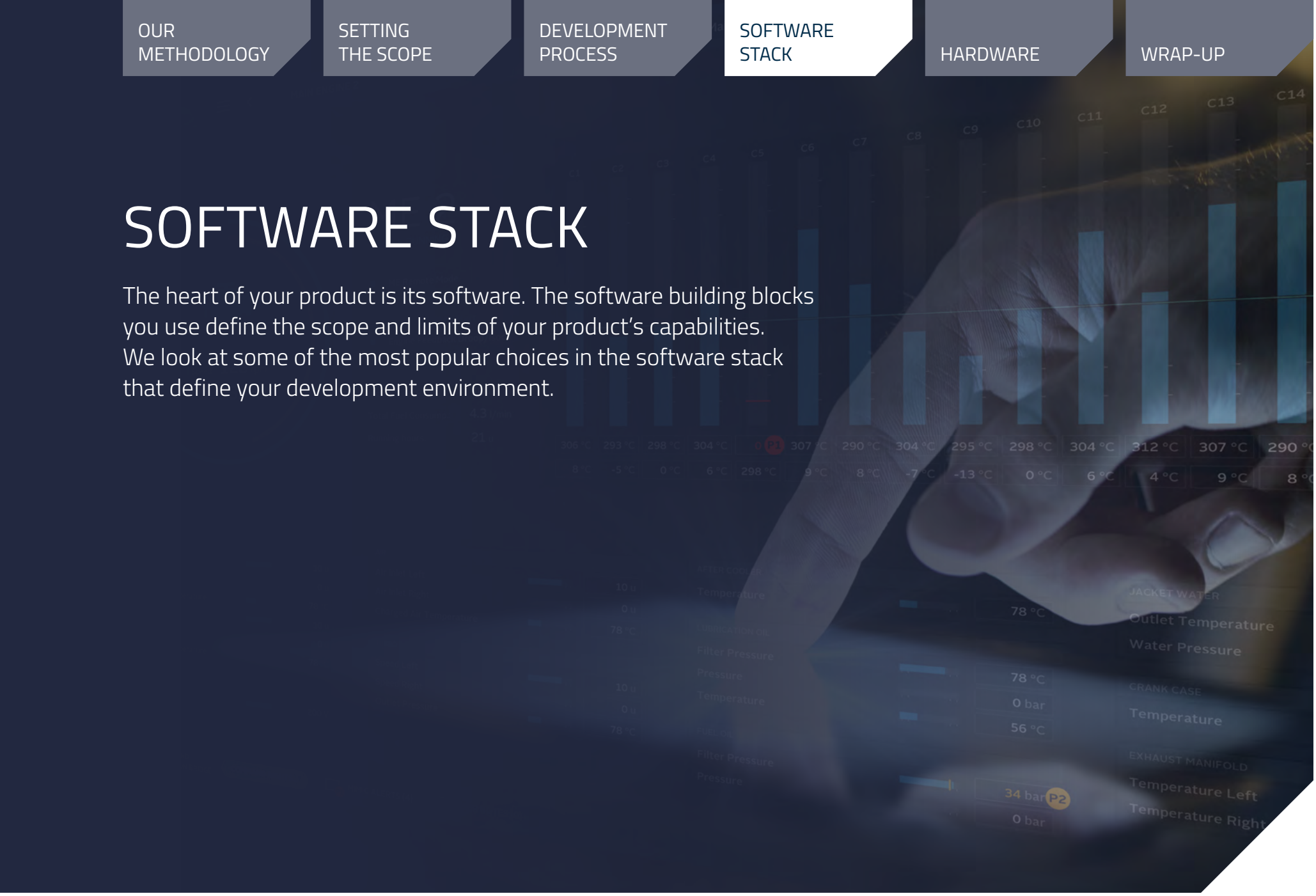
Look for solutions with deep support in multiple areas: dedicated online conversations, stack overflow keywords, git sample code repos, and job search categories. The more plug-ins, applications, command-line tools, or third-party developers you have for your favorite toolchain, the more chance there is that someone has created the solution you need. It lets you quickly find help, whether that's detailed docs, API examples, online tips – even hiring contractors.

ESSENTIALS FOR YOUR SOFTWARE STACK COMPONENTS

- Availability of source code
- No vendor lock-in
- Low total cost of ownership
- Quick time-to-market
- Thriving developer community
- Well-supported tool ecosystem

SOFTWARE STACK

The heart of your product is its software. The software building blocks you use define the scope and limits of your product's capabilities. We look at some of the most popular choices in the software stack that define your development environment.



OS

The component of your software stack that is perhaps the most influential is the operating system. Your choice of OS dictates the software you can incorporate into your system. It also can make some tasks easy while making others fiendishly difficult. (Although nowhere near as difficult as going bare metal and doing it all yourself!) Here's our take on eight of the most popular operating systems in the embedded market.

	Embedded Linux	Android	QNX Neutrino RTOS	GreenHills INTEGRITY	Wind River VxWorks	Amazon FreeRTOS	webOS	Windows for IoT
Ease of development	★★★★	★★★★★	★★★	★★	★★★	★★	★★★★	★★★★
Efficiency	★★★	★★	★★★★	★★★★	★★★★	★★★★★	★★	★★★
Deterministic behavior (real-time)	★★★	★	★★★★★	★★★★★	★★★★★	★★★★	★★	★★
API	POSIX	POSIX	POSIX	POSIX	POSIX	FreeRTOS	POSIX	UWP
Connectivity	★★★★★	★★★	★★★	★★★★	★★★	★★	★★★★★	★★★★★
Graphics	★★★★	★★★★★	★★★★	★★	★★	★	★★★★	★★★★★
Hardware support	★★★★★	★	★★★	★★	★★★★	★	★★	★★★★
Open source	Yes	Yes	No	No	No	Yes	Yes	No
Community	★★★★★	★★★★	★★★	★★	★★	★★	★★	★★★
Licensing pricing	$	$	$$$$	$$$$	$$$	$	$	$$$$$
Customization and hardening cost	$$$	$$$$	$$	$$	$$	$$$	$$$	$

★ is the lowest ranking while ★★★★★ is the highest. Please refer to technology acronyms in Appendix A

The charts in this eBook represent a quick and easy assessment that highlights software and hardware strengths in a way that is less sensitive to marketplace dynamics and captures the essence of each component. (With the rapid pace of development, there's no way a comprehensive guide can be kept fully up to date.) Typical use cases for these charts include large (industrial, medical, autonomous), medium (infotainment, white goods, kiosk), and small applications (handheld devices, IoT, wearables). If multiple options for a component are available, we list stats for the most capable variant. We've selected some of the most popular hardware choices but check with your supplier for their full range of products.

Containers and hypervisors

Today's powerful processors have brought desktop and cloud solutions to embedded, including two powerful methods to subdivide your product's software architecture: containers and hypervisors.

Containers can **accelerate embedded project development**: making tool setup instantaneous and consistent, standardizing testing frameworks, and maintaining multiple independent tool/library configurations. Containers can also be an interesting building block when **used on the target** by making for straightforward device provisioning, easy hardware vendor updates, and excellent testing from the same codebase.

Hypervisors are exclusively about helping the target by providing protection, sandboxing, and independent operation. They allow you to **run multiple operating systems (OSes) simultaneously** on one chip. As an example, you might run your system-critical or safety-critical features on an OS like the QNX Neutrino RTOS or GreenHills INTEGRITY RTOS, the product's main UI on a second OS like Embedded Linux, and downloadable third-party apps on a third OS like Android or WebOS. Hypervisors also make a critical difference when building a device with functional safety requirements.

Language

Building your application on top of an OS will require a programming language. Every programming language has strengths and weaknesses that find their way into the development process and impact the development of your embedded application. These constraints include the types of GUIs that each language can support. Here we explore some of the embedded industry's leading candidates.

	C	C++	Java	Python / MicroPython	JavaScript/ HTML5/CSS	Rust
Strengths	Embedded, bare-metal, IoT	Embedded, bare-metal, standalone apps, IoT	Web, cloud	Cloud, data science	Web	Embedded, bare-metal
Ease of development	★★★	★★	★★★	★★★★★	★★	★★★★
Expressive power	★	★★★★	★★	★★★★★	★★★	★★★★
Ease of maintenance	★★★★	★★★★	★★★★	★★★	★	★★★★
Longevity	★★★★★	★★★★	★★★	★★★	★	★
Runtime efficiency	★★★★★	★★★★★	★★★	★★	★	★★★★
Library/module availability	★★★★	★★★★	★★★	★★★★	★★★★	★★
Low-level interface	★★★★★	★★★★★	★★	★★★	★	★★★★
Connectivity support	★★	★★★ / ★★★★★[1]	★★★★	★★★★★	★★★★★	★★
Graphics support	★★★★	★★★★★	★★★	★★★	★★★★★	★
Developer community	★★★	★★★	★ / ★★★★[2]	★★★★	★★★★★	★★

★ is the lowest ranking while ★★★★★ is the highest. Please refer to technology acronyms in Appendix A

1 Provides high level of connectivity and networking support when platform libraries included
2 Java Developer community exclusive of Android (low) and inclusive of Android (high)

GUI framework

One of the most important pieces in your technology stack is your graphical user interface (GUI). The GUI framework that your application uses can affect more than just what the application looks like, but also what capabilities you can implement, how fast you can bring your product to market, and what platforms you can support. Whatever you can do with someone else's prebuilt and pre-tested code is something you don't have to design, code, and test yourself.

Cross-platform

A key characteristic of your GUI framework is its cross-platform capability. How many platforms does it support, and does it support them all consistently? You'll need this if you ever switch hardware, need to support multiple OSes, or share source between your embedded product and your mobile or desktop companion apps.

Growth and change are inevitable, so don't pick just based on today's platform requirements. Look for GUI frameworks that support a wide variety of desktop OSes (Linux, MacOS, Windows), mobile OSes (Android, iOS), and embedded OSes (QNX, INTEGRITY, Linux/Wayland, Linux/X11, UWP, VxWorks) – as well as numerous hardware architectures and platforms.

STRONG CROSS-PLATFORM GUIS HELP YOUR PRODUCT IN SEVERAL WAYS

TIME TO MARKET.
A tool that has broad pre-packaged capabilities, many programmer-oriented features, and an active, supportive community can make developers dramatically more productive and get products to market faster.

UNIFIED EXPERIENCE.
Good tooling can provide the same experience to users regardless of the platform they use, letting them use all platforms consistently.

CODE REUSE.
Consistency of features, APIs, build environments, and asset management between platforms allows developers to easily move code between embedded, mobile, and desktop solutions.

EASE OF DEVELOPMENT.
Strong cross-platform toolchains allow developers to use their preferred environments - where they're more productive – rather than forcing them to develop or build on each supported target.

Components

Every GUI framework provides support for more than just widgets; look for variety here too. Look for frameworks with a plethora of component options like 2D and 3D design and visualization, embedded browsers, peripheral support, common network protocols, internationalization, full multimedia support, database integration, sensor access, charting – whatever pre-built pieces can help off-load development from your staff.

Language

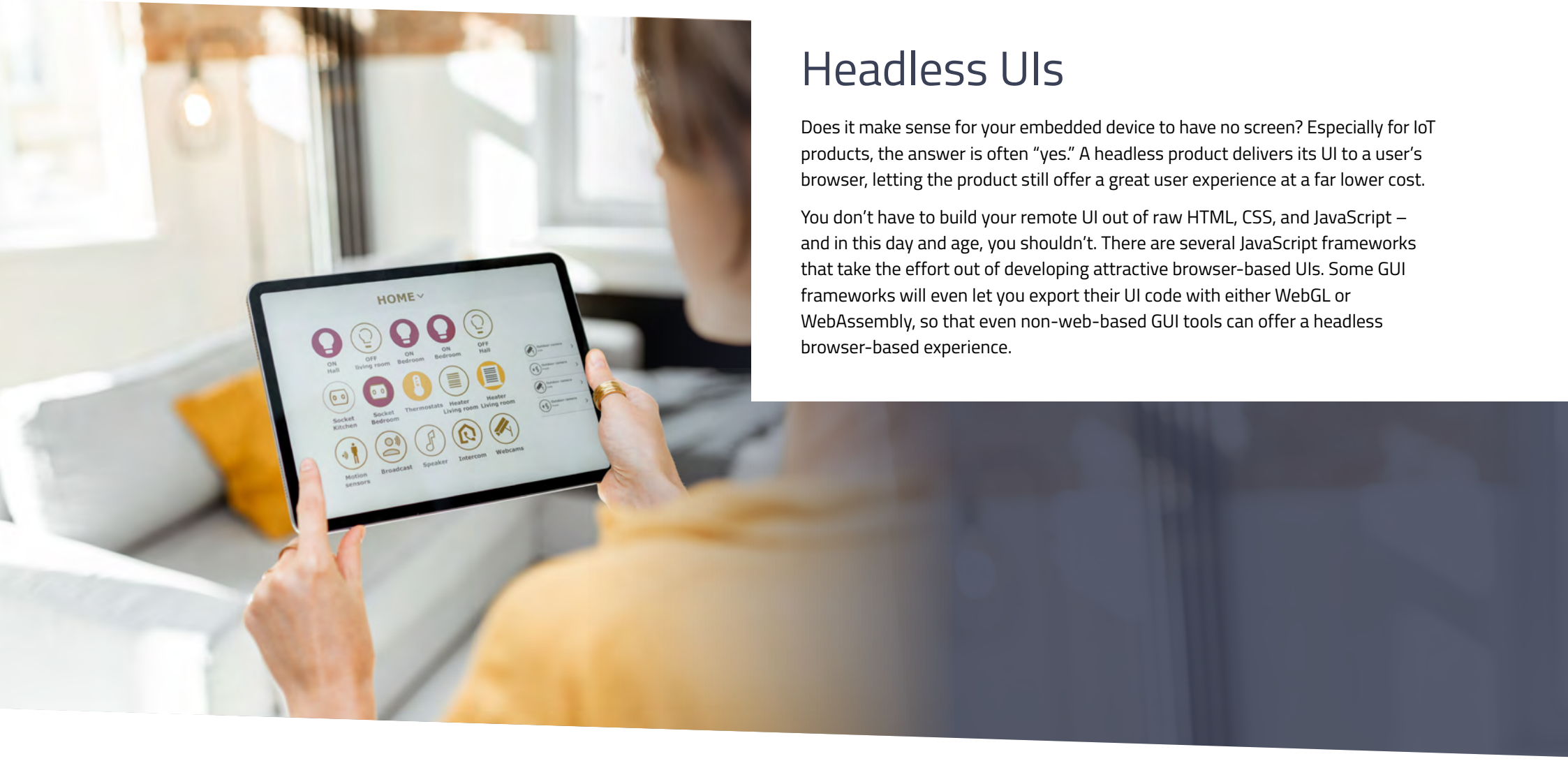
Frameworks can provide APIs for different languages, which lets your development team write in the language in which they're most productive. It also gives developers the flexibility to pick the best language for the job. For example, if your framework supports both C++ and Python, the team can choose C++ when performance is paramount and Python when development speed is the priority.

Development support

Look for GUI frameworks that either provide their own IDE, or have plug-ins for popular IDEs like VSCode, Visual Studio, or Eclipse. It's essential your UI be easy to design, build, test, and debug. A specialized development environment is also key.

LOOK AND FEEL:
OS, FRAMEWORK, OR CUSTOM

There will always be debates on the merits of your product's look and feel. Should it use OS-specific widgets, controls provided by your framework, or a custom look? Unfortunately, a platform-provided UI isn't always available – many embedded products use Linux, which has no default UI. Not to mention that if your product runs on multiple OSes, each platform will look different from the other. We feel that picking a UI that's universally understood, clean, and attractive is usually the best overall compromise – whether custom built by you or provided by the framework. It will help you maintain a consistent and branded look and feel across your entire product line.



Headless UIs

Does it make sense for your embedded device to have no screen? Especially for IoT products, the answer is often "yes." A headless product delivers its UI to a user's browser, letting the product still offer a great user experience at a far lower cost.

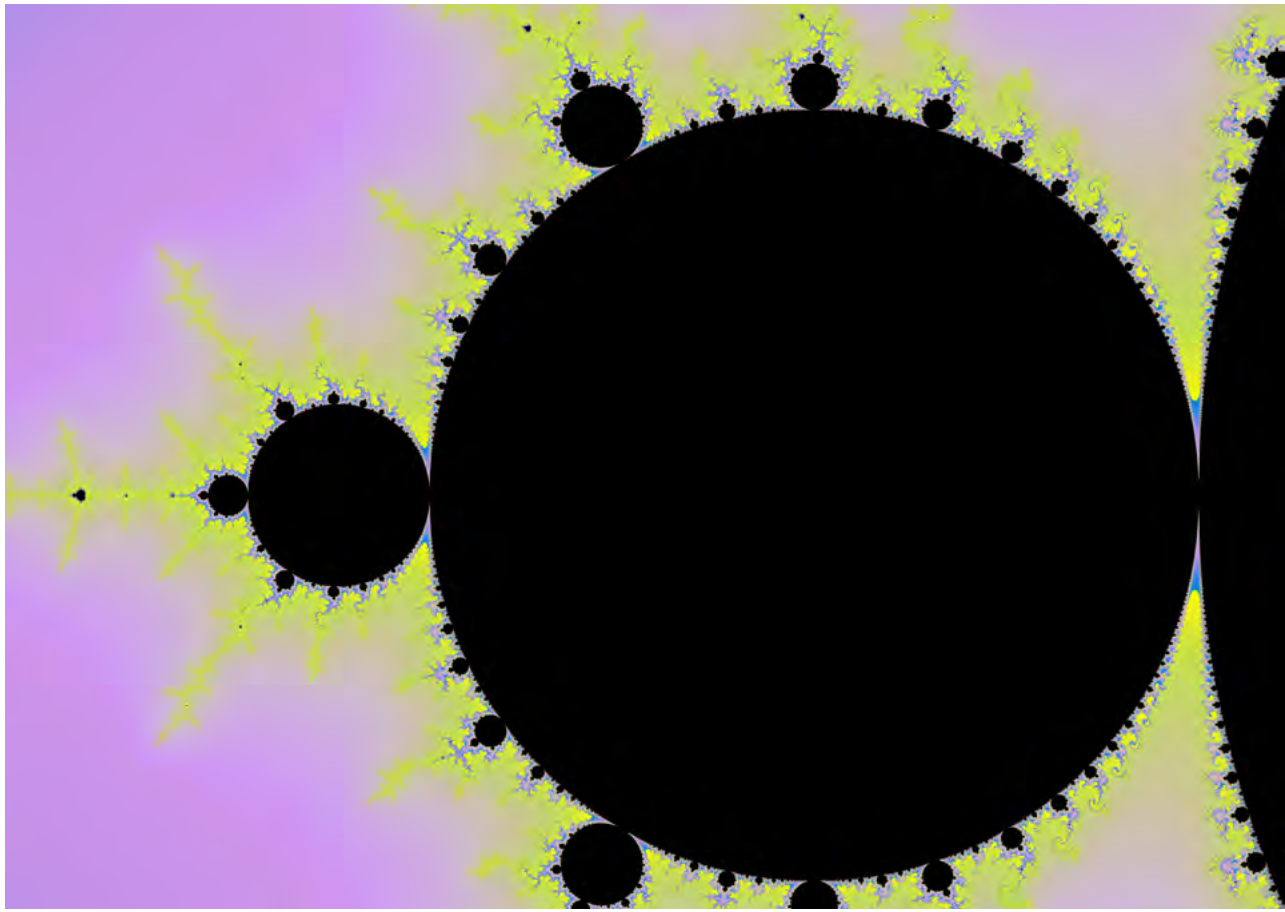
You don't have to build your remote UI out of raw HTML, CSS, and JavaScript – and in this day and age, you shouldn't. There are several JavaScript frameworks that take the effort out of developing attractive browser-based UIs. Some GUI frameworks will even let you export their UI code with either WebGL or WebAssembly, so that even non-web-based GUI tools can offer a headless browser-based experience.

Libraries

Application software needs a lot more than just a GUI. Your product might need lots of pre-built and pre-tested libraries: network protocols, JSON and/or XML parsing, machine learning, matrix math, fast Fourier transforms (FFTs), random number generators, cryptography, multi-threading support, image processing, regular expressions, database support, and much more.

Most of these things aren't simple, and the more complex the code, the more you should depend on software that has stood the test of time than try to write it yourself. You may be able to get many pre-packaged libraries already included with your GUI framework. You may also need to incorporate a number of well-supported external libraries like Boost, Apache Portable Runtime (APR), NumPy, TensorFlow, or many others – depending on your implementation language.

Regardless of what features you require, look for the smallest set of libraries that can comprehensively meet your needs with compatible and easily comprehensible APIs and excellent documentation.



HARDWARE

Now that we've investigated the software requirements, it's time to investigate the hardware. Many of the product features or roadmap futures will solidify your hardware choices. For example, if the product needs fluid 3D animations, you had better pick something with hardware-accelerated graphics. If it needs a fast mobile connection, you'll need either WiFi ac, USB, or both. If it has an interaction model that includes gestures, touch, or voice, you'll need to include the necessary hardware and sensors.

Let's look at the kinds of hardware choices we need to make.

Product line and scalability

The chips behind an embedded system can have a tremendous range of capability, from 3 GHz 64-bit processors with 16 cores and hardware-accelerated multi-display 4K graphics processing units (GPUs) down to tiny 8-bit micros with four pins of input/output (I/O).

At this point, you should have an idea of the range of your product's current and future requirements. Pick an initial hardware choice that can encompass the breadth of your product scope within a compatible family of parts. It's best to consider systems and architectures that have both high- and low-end options. Pick components you can reuse across multiple products in a product line to take advantage of the bigger volumes that this will give you. And if your industry builds products with long lifespans, make sure that the parts you pick will be available for years to come.

That said, be mindful of the minimal processing, speed, and memory requirements that your software stack will demand from the hardware. Don't let downward cost pressure trap you into choosing underpowered hardware, since a forced hardware migration late in the development cycle will be even costlier.

SoC/CPU

The central processing unit (CPU) of your system on a chip (SoC) is probably the most important, most impactful, and most expensive hardware component in an embedded system. It can also lock you into decisions for your entire product family. Here's a table of some of the most capable and popular off-the-shelf SoC eval boards.

	AMD	Intel	NXP	Renesas	TI
Eval board	Kontron D3713-V4 mITX	Avnet BOX NUC6CAYS AJL	NXP i.MX 8 QuadMax MEK	R-Car H3 Starter Kit	TI AM572x EVM
SoC*	AMD Ryzen V1807B	Intel Apollo Lake (Celeron J3455)	NXP i.MX8	R-CAR H3	AM5728
Memory	4 DDR4, 2 DIMM (up to 32GB), ECC option	2GB onboard DDR (up to 8GB), 32GB eMMC	LPDDR4 (x64), 32GB eMMC	384K sys RAM, DDR4, 80MB onboard flash, 8GB eMMC, microSD	2GB DDR3L, 4GB eMMC, microSD
Peripherals	HDMI, 4 x DP++, 4K, Vega GPU, PCIe, SATA, audio, GigE, USB 3.1+2.0	HDMI, Intel HD 500 GPU, audio, GigE, PCIe, SATA, USB 3.0 + 2.0, WiFi ac, BT,	MIPI, LVDS, 4K, PMIC, GPU, PCIe, audio, GigE, USB 3.0, CAN, up to 4 HDMI displays	HDMI, PowerVR GPU, LVDS, WiFi, BT, audio, Eth 10/100, USB 2.0	7" capacitive touch screen, HDMI, audio, PowerVR GPU, GigE, SATA, MiniPCIe, USB 3.0
Architecture	x86-64	x86-64	ARM-64	ARM-64	ARM-32
Processing power	V1807B x 4 core @ 3.8GHz	J3455 x 4 core @ 1.5GHz	A72 x 2 core @ 1.6 GHz; A53 x 4 core @ 1.2 GHz; M4 x 2 core @ 266 MHz	A57 x 4 core @ 1.5 GHz; A53 x 4 core @ 1.2 GHz	A15 x 2 core @ 1.5GHz, M4 x 2 core
Use cases	Large	Large	Large	Large	Medium
Power draw	★	★	★★	★★	★★★
Longevity	★★★	★★★	★★★★★	★★★★★	★★★★★
Unique features	Up to 4 displays @ 4K	NUC is mini-PC, but SoC is embedded- compatible	Accel, gyro, pressure, light sensors	Automotive, EAVB, 440 pin expansion	Camera option, 2 x C66x DSPs

★ is the lowest ranking while ★★★★★ is the highest. Please refer to technology acronyms in Appendix A

* SoCs may include additional features that are unexposed by an eval board.

SBC

Maybe your product can be made without a custom circuit board. If so, there are a wide variety of single-board computer (SBC) designs available, as outlined below.

	Broadcom	Intel	Nvidia	Rockchip	TI	Qualcomm
SBC	Raspberry Pi 4 Model B	Radxa Rock Pi X	Jetson TX2	Pine64 ROCKPro64	BeagleBoard X-15	Arduino Yún 2
SoC*		Intel Cherry Trail	Tegra X2	Rockchip RK3399 SOC	Sitara AM5728	QC Atheros AR9331
Memory	8GB RAM, microSD	4GB RAM, 32GB flash, microSD	8GB RAM, 32GB eMMC	LPDDR4 (up to 4GB), eMMC, microSD	2GB RAM, 4GB flash, microSD	64MB RAM, 16MB flash, microSD
Peripherals	µHDMI, 4K, MIPI, VideoCore VI GPU, WiFi ac, GigE, USB 3.0+2.0, BT 5.0	Intel Gen8 GPU, HDMI, audio, WiFi ac, GigE, USB 3.0+2.0, BT 4.2	HDMI 2.0, 4K, GP10B GPU, GigE, MIPI, 6 cameras, USB 3.0+2.0	4K, MIPI, Mali GPU, audio, GigE, USB 3.0+2.0,	HDMI, PowerVR GPU, audio, GigE, SATA, USB 3.0+2.0	Audio, 100Eth, WiFi b/g/n, USB 2.0
Architecture	ARM-32	x86-64	ARM-64	ARM-64	ARM-32	MIPS32
Processing power	A72 x 4 core @ 1.5GHz	Atom x 4 core @ 1.44GHz	Denver 2 x 2 core @ 2.0GHz, A57 x 4 core @ 2.0 GHz	A72 x 2 core @ 2.0GHz, A53 x 4 core @ 2.0GHz	A15 x 2 core @ 1.5GHz; M4 x 2 core @ 212 MHz	24K @ 400MHz
Use cases	Medium	Large	Large	Medium	Medium	Small
Power draw	★★★	★	★★★	★★★	★★★	★★★★★
Longevity	★★★★	★★	★★	★★★★	★★★	★★
Unique features	Expansion ecosystem	Windows compatible	256 CUDA cores for AI or comp vision, video encoder/decoders	Wifi ac + BT and touch panel optional modules, big.LITTLE arch for low power	2 TMS DSPs, 4 x PRU for low-latency I/O control	ATmega32U4 micro on board

★ is the lowest ranking while ★★★★★ is the highest. Please refer to technology acronyms in Appendix A

* SoCs may include additional features that are unexposed by an eval board.

RAM

Following the CPU, RAM is often the second most expensive component on an embedded board. If your embedded device doesn't have the luxury of abundant random-access memory (RAM), you carry the risk that your application eventually may not fit into available memory. To discover these show-stopper conditions early, run each of the biggest memory consumers in your system (OS, graphical UI, database, etc.) on your target hardware and independently under a test environment that mirrors your real runtime environment as close as you're able. Carefully track their memory high-water marks and add them all up with a sizable slop factor.

You need good tools to help you understand memory usage during the application's lifecycle so you can pinpoint the biggest memory-hogging components to focus on. With some preliminary data that follows your use patterns, it also may guide you in setting memory-impacting application configurations in the way that they'll eventually need to be set for production. This may include RAM set aside for purposes like graphics display buffers, audio buffers, or disk caches.

MEMORY TOOLS

It's not just other people's software – don't forget to test your app to make sure that it isn't the cause of memory problems. Tools like **Cppcheck**, **heob**, **GammaRay**, or **valgrind** can be invaluable to perform static and runtime analysis of your code and find excessive memory allocations or memory leaks.

Flash

Most boards can use microSD cards or another external flash interface, allowing engineering staff to develop with plenty of spare disk space yet optimize for the needed capacity when development is over. A bigger challenge with flash is often its speed, and any disk-bound applications will suffer unless they're using reasonably fast flash.

In secure digital (SD) card lingo (whether standard, mini, or micro), the class or video class designation indicates how many MB/second can be written; a class 10 card can write 10MB/sec, and a V90 card can write 90MB/sec. It can be useful to have multiple microSD cards of different classes available. This allows simple measurements of the impact of flash speed by swapping out different memory cards, letting you determine the best cost versus speed tradeoff.

Having said this, if your evaluation board has on-board flash, you'll want to maximize your use of it – it will almost always provide better redundance and performance than an SD card.

FORMATTING YOUR FLASH

Flash file systems can employ very different strategies and trade-offs to manage device wear leveling, error detection/correction, and crash resilience. Try testing your app under different file systems to see which one performs best, especially if your app relies on any of the following behaviors:

- Fast bootup speeds
- High-speed bulk reads
- High-frequency continual writes
- Huge numbers of small files
- Performant disk compression
- Speedy directory scans

MCU

Today's microcontroller units (MCUs) are very capable and getting more so – a perfect fit if you don't need maximum horsepower. Because these chips generally don't have a memory management unit (MMU), you'll need to either run your application bare metal or pick an OS that is able to support a non-MMU chip like Azure RTOS ThreadX or Amazon FreeRTOS. You'll also need to be especially careful with RAM and Flash use. Here we look at microcontrollers with a variety of capabilities and architectures.

	Microchip	NXP	Renesas	ST	TI
Eval board	PIC32MZ Embedded Graphics with Stacked DRAM Starter Kit	i.MX RT 1170 EVK	RH850/ D1M1A	32F769I-DISC0 discovery kit	C2000 Delfino MCU F28379D LaunchPad
MCU	PIC32MZ DA	i.MX RT 1176 DVMAA	RH850 G3M	STM32F7	TMS320-F28379D
Memory	32MB SDRAM, 4MB flash, microSD	64MB RAM, 336MB flash, SD card	4MB RAM, 5MB flash, DDR2, ECC option	532KB RAM, 2MB flash, microSD	204KB RAM, 1MB flash
Peripherals	5.0" WVGA, 2D GPU, audio, Eth 10/100, USB 2.0, CAN, 6 x SPI, ADC	5.5" 720p display, OpenVG GPU, MIPI, audio, USB, CAN, GigE, EAVB, SIM	2D GPU, sprites, LVTTL vid in, audio, 2 x I2C, 6 x CSI, 3 x CAN, Eth 10/100, EAVB	4" touch display, 2D GPU, MIPI, audio, USB, 4 x I2C, 6 x SPI, 3 x CAN, Eth 10/100	USB, ADC/DACs, PWMs, CAN
Architecture	MIPS32	Cortex-M7 + M4	RH850	Cortex-M7	C28X
Processor speed	252MHz	M7 @ 1GHz, M4 @ 400MHz	240MHZ	216MHz	200MHz x 2
Temperature rating	–40 to 125°C	–40 to 125°C	-40 to +150 °C	–40 to +105 °C	–40 to 125°C
Power draw	★★★★	★★★	★★★★	★★★★	★★★★
Longevity	★★★★	★★★	★★★	★★★	★★★★
Unique features	Display through MEB expansion module, crypto option avail, Raspberry Pi header	Mag + accel sensors, Arduino header	Automotive	opt WiFi module, Arduino header	2 CLA FPUs, Trig + Complex math units, BoosterPack header, no graphics support

★ is the lowest ranking while ★★★★★ is the highest. Please refer to technology acronyms in Appendix A

User I/O

Your user interface and input/output (I/O) mechanisms are one of the biggest areas of product differentiation, but they can also drive cost and complexity into your product. Here's a table of factors to consider in planning how your devices talk to your users.

	Touch screen 2D	Touch screen 3D	Audio	Voice	Physical controls	Haptic controls	Gestures
Hardware	Screen, capacitive plate	Screen, capacitive plate, graphics processing unit (GPU)	Speakers, digital analog converter (DAC)	Mic, analog digital converter (ADC), speakers, DAC, voice assistant interface	Rotary knobs, switches, buttons, sliders, light emitting diode (LED) indicators	Rotary knobs, switches, motors	Camera, proximity/ pressure sensors, IR camera, lidar
Pricing	$$$$$	$$$$$	$	$$$$	$	$$	$$$$
Communication richness	★★★★	★★★★★	★★	★★★★	★	★½	★
Speed of users to understand interface intent	★★★★★	★★★★★	★★	★★	★★★	★★★	★★
Ability to be self-explanatory	★★★★	★★★★	★★★	★★	★★★	★★★	★
Ability to use without training	★★	★	★★★★★	★★	★★★★★	★★★★	★★
Ease of internationalization	★★★★★	★★★★★	★★	★★	★★★★	★★★★	★★★
Hygiene for multi-user use	★	★	★★★★★	★★★★★	★	★	★★★★★

★ is the lowest ranking while ★★★★★ is the highest. Please refer to technology acronyms in Appendix A

Graphics

Your choice of graphics processing unit (GPU) is nearly always built-in to your SoC, so your graphics performance usually comes down to the quality of your graphics framework and display pipeline. Unless, of course, you're designing your own hardware. If this is the case, you must still be very careful to pick CPU and GPU combinations that have good existing OS and driver support. Creating your own graphics drivers is not at all recommended (and may not even be possible).

Every unique application generates different demands on the GPU, and real-world benchmarks are the only way to guarantee any particular metrics are met. In absence of a side-by-side specific comparison, our preferred graphical API standards are (in order): Vulkan, DirectX, OpenGL, OpenGL ES.

There can be notable performance differences between graphical API standards in terms of frame rate, GPU and CPU load, battery life, and excess heat. It's great if your GUI framework gives you the option to pick which underlying standard it uses to do its rendering. That allows your GUI code to stay independent of graphical standards. You can make measurements for your specific application and take advantage of performance gains if you need to.

Although 3D GPUs are the most common, if you're creating a standard UI with 2D elements, a 2D or 2.5D GPU might provide all the acceleration you need. With bit blit engines, layering and compositors, anti-aliasing, and font glyph rendering, these GPUs can speed the process of translating application rendering instructions into user-visible pixels.

Displays

Could your product stand out with a distinctive display? Quite probably. Here are a variety of ways to wow your customers.

	TFT LCD	PMOLED	AMOLED	QLED	E-paper
Description	Thin film transistor liquid crystal display	Passive-matrix organic light-emitting diode	Active-matrix organic light-emitting diode	Quantum light emitting diode	Electronic paper
Readability in bright light	★★★	★★★★	★★★★	★★★★	★★★★★
Readability in darkness	★★★★	★★★★★	★★★★★	★★★★★	★
Update speed	★★★★★	★★	★★★★★	★★★★★	★
Color reproduction	★★★★	★★★	★★★★★	★★★★★	★
Contrast	★★★	★★★★★	★★★★★	★★★★★	★★★★
Viewing angle	★★★	★★★★	★★★	★★★★	★★★★★
Power consumption	★	★★	★★★★	★★★	★★★★★
Durability	★★★★★	★★	★★★	★★★★	★★★★★
Size	1" – 100"	0.5" – 6"	1" – 18"	TV sizes	1" – 10"
Pricing	$$	$$$	$$$$$	?	$
Notes	Currently dominates display technology. Reliant on backlight to be visible under most conditions (very bright backlights can be used in sunlight).	Cheaper to manufacture than AMOLED, but higher current draw and slower to refresh. High current leads to quicker degradation.	Great color gamut and readability, although RGB components degrade unevenly and are subject to burn-in.	Advantages of AMOLED but eliminates color instability. Not yet available in embedded applications although planned by Samsung.	Usually monochromatic, but some color variants available. Power consumed only when image changes.

★ is the lowest ranking while ★★★★★ is the highest. Please refer to technology acronyms in Appendix A

WRAP-UP

Hopefully this guide will be useful in the planning process of your embedded or IoT device. While we've tried to cover most of the basics, new products become available all the time and no single guide can contain all the possible choices you might make. Our consultants and engineers have experience in helping companies build automotive, avionics, medical, industrial, consumer electronics, mobile apps, games, and much more. If you need some advice on what should go into your next project, we're happy to lend a hand.

QT EXPERIENCE

Although this is intended to be an impartial guide that should be useful regardless of your component selection, our development of the Qt software portfolio incorporates the many lessons we've learned from our experience helping companies plan products. If you'd like more information about it, you might be interested in these links:

Download Qt | **Getting started** | **Product overview** | **Resource center** | **Blog**

Appendix A – Acronyms, Initialisms, and Abbreviations

2D – 2 dimensional

3D – 3 dimensional

4K – 4K resolution

ABI – Application binary interface

ADC – Analog digital converter

AI – Artificial intelligence

AMOLED – Active-matrix organic light-emitting diode

API – Application programming interface

APR – Apache portable runtime

BT - Bluetooth

CAN – Controller area network

CI – Continuous integration

CLA – Carry-lookahead adder

CPU – Central processing unit

CSI – Camera serial interface

CSS – Cascading style sheets

CUDA – Compute unified device architecture (Nvidia)

DAC – Digital analog converter

DDR – Double data rate

DDR2 – Double data rate 2

DDR3L – Double data rate 3 low voltage

DDR4 – Double data rate 4

DIMM – Dual in-line memory module

DP++ – DisplayPort dual-mode

DSP – Digital signal processor

E-paper – Electronic paper

EAVB – Ethernet audio visual bridging

ECC – Error correcting code

eMMC – Embedded multi-media card

Eth – Ethernet

Eval – Evaluation

FFT - Fast Fourier transform

FPU – Floating-point processing unit

GB – Gigabyte (one billion bytes)

GDP – Gross domestic product

GHz – Gigahertz

GigE – Gigabyte Ethernet

GPU – Graphics processing unit

GUI – Graphical user interface

HDMI – High-definition multimedia interface

HTML5 – Hypertext markup language 5

IDE – Integrated development environment

LCD – Liquid crystal display

LED – Light emitting diode

LPDDR4 – Low-power double data rate 4

LVDS – Low-voltage differential signaling

LVTTL – Low-voltage transistor-transistor logic

MB – Megabyte (one million bytes)

MCU – Microcontroller unit

MHz - Megahertz

Mic – Microphone

microSD – Micro secure digital

MiniPCIe – Mini peripheral component interconnect express

MIPI – Mobile industry processor interface

MMU – Memory management unit

NUC – Next unit of computing (Intel)

OpenGL – Open graphics library

OpenGL ES – Open graphics library embedded systems

OS – Operating system

OTA – Over the air

PCIe – Peripheral component interconnect express

PMIC – Power management integrated circuit

PMOLED – Passive-matrix organic light-emitting diode

POSIX – Portable operating system interface

PRU – Programmable real-time unit

PWM – Pulse width modulator

QA – Quality assurance

QLED – Quantum light emitting diode

QML – Qt modeling language

RAM – Random access memory

RGB – Red green blue

RTOS – Real-time operating system

SATA – Serial AT attachment

SBC – Single board computer

SD – Secure digital

SDRAM – Synchronous dynamic random-access memory

SoC – System on a chip

SPI – Serial peripheral interface

TFT – Thin film transistor

TI – Texas Instruments

TMS – TMS320 series DSP (TI)

TV – Television

UI – User interface

USB – Universal serial bus

UWP – Universal Windows platform (Microsoft)

UX – User experience

V90 – SD association video speed class 90 (90MB/sec)

WebGL – Web graphics library

WiFi – IEEE 802.11 wireless protocol

WiFi ac – IEEE 802.11ac-2013 or WiFi 5

WYSIWYG - What you see is what you get

µHDMI – Micro high-definition multimedia interface

ABOUT THE QT COMPANY

The Qt Company is responsible for all Qt activities, including product development, commercial and open source licensing together with the Qt Project under the open governance model. Together with our licensing, support and services capabilities, we operate with the mission to work closely with developers to ensure that their Qt projects are deployed on time, within budget, and with a competitive advantage.

The Qt Company's goal is to provide desktop, embedded, and mobile developers and companies with the most powerful cross-platform UI and application framework. Together with its licensing, support and services capabilities, The Qt Company operates with the mission to work closely with developers.

www.qt.io

The Qt Company