Building Flexible, Future-Proof Infotainment Systems

Tina Jeffrey, Automotive Product Marketing Manager, QNX Software Systems
Andy Gryc

Car infotainment systems are evolving from purpose-specific devices into connected, upgradeable platforms — as indeed, they must. For evidence, look no further than the issue of smartphone integration. New smartphones come to market almost every month, and new smartphone apps come out every hour, whereas infotainment systems must operate in the field for 10 years or more. So how can an infotainment system designed today work with phones or apps created tomorrow, or several years from now? There is, of course, no single connectivity solution that can address the problem. Thus, an infotainment system design should possess the underlying flexibility to accommodate the inevitable evolution of the mobile market. The alternative is early obsolescence.

A similar dilemma applies to apps running on the infotainment system itself. Even if a connected smartphone provides a significant portion of the system's user experience (UX), the system itself still needs to run a core set of applications. That way, it can deliver a satisfactory UX regardless of which smartphone is present, or even if no smartphone is present.

The question is, which application environment? A homegrown environment may make little sense, since app developers prefer to target the high-volume environments used in smartphones. On the other hand, smartphone app environments generally aren't designed to address the performance, reliability, and security requirements of the car. So how do you deliver built-in apps and still remain "automotive grade"?

Going native

To understand how these issues can be addressed, consider the software options available to infotainment system developers. For many, the tried-and-true approach is a native C/C++ toolkit such as EB GUIDE, Qt, or Crank Storyboard. Indeed, these toolkits often provide the best path to creating a quality UX: they generally boot faster, perform more responsively, and consume less memory than "virtual machine" environments such as Android or HTML5.

Native toolkits can also streamline product development. Some, for example, support state machines, which allow developers to build an entire HMI without writing code and which make the final HMI easier to test. Some toolkits also allow developers to take HMI components designed in a program like Photoshop and import them directly into the live system design, rather than spend days or weeks recreating the components in code.

The problem is, many native toolkits cannot support applications written in a popular app environment such as Android or HTML5. Is the solution, then, to use one of these environments not just for apps, but as the basis for the entire HMI? There is, in some cases, an argument for doing so.

Going mobile

HTML5, for example, offers many capabilities of a traditional HMI toolkit, including a rendering engine, content authoring tools, and a programming language; it also offers benefits that some native toolkits would be hard-pressed to match. For instance, HTML5 supports Cascading Style Sheets (CSS), which cleanly separate business logic from the HMI, making the HMI relatively easy to customize or re-skin. Moreover, HMTL5 can run on head units as well as on mobile phones, allowing developers to create a single HMI code base that works regardless of whether a car has a head unit (where the HMI runs in the car) or a headless, phone-assisted system (where the HMI runs on the phone). HTML5 also supports the notion of an “executable HMI specification,” where the automaker supplies the HMI prototype, coded in HTML5, and the tier 1 supplier takes care of connecting the HMI to any required services. This approach eliminates the tedious, error-prone process of recreating the entire HMI from screen printouts.

Despite these advantages, mobile app environments like HTML5 don’t always serve as the best foundation for a built-in HMI. In particular, no one wants the outside world, with its unpredictable web content and potential security threats, to threaten HMI behavior. And, as mentioned, such environments aren't in the same league as native toolkits when it comes to boot times, performance, and memory usage.

Does all this mean that automotive companies must choose between the benefits of native HMI toolkits and those of a mobile app environment? Not at all. Look, for example, at the accompanying photo, in which a head unit based on the QNX CAR Platform for Infotainment is running apps from a mix of mobile environments in an HMI built with a native toolkit. Components built with the various environments all appear on the same display, at the same time, with no visible separation between them.

Apps from a mix of mobile environments running in an HMI built with a native toolkit.

Blending output

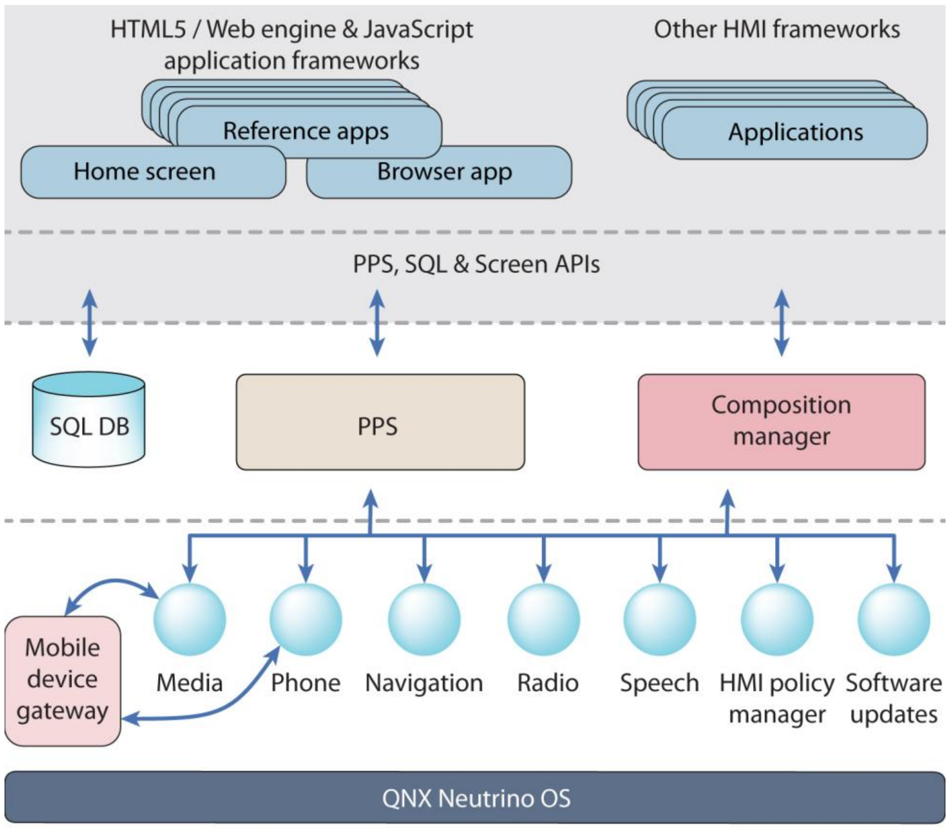
To combine these environments successfully, a software platform must support several key technologies. The first is graphical composition, which consolidates output from multiple application windows onto a single display. The windows may need to be tiled, overlapping, or blended — or some combination thereof. To perform this consolidation quickly and responsively, the platform's graphical framework should take advantage of hardware acceleration in the GPU. Properly implemented, graphical composition allows the user to interact with components created in different environments without having to manually switch environments or, indeed, seeing any noticeable change when moving from one component to another.

Abstracting services

To combine environments, the platform must also provide an abstraction layer that enables applications created with a variety of tools and languages to interact with system services. For instance, in a service abstraction layer based on publish/subscribe messaging, data objects allow applications to access services such as the multimedia engine, database engine, voice recognition engine, vehicle buses, connected smartphones, Bluetooth profiles, hands-free calling, and contact databases. These data objects can consist of multiple attributes, each providing access to a specific feature such as the frequency of the current radio station or the RPM of the engine. System services publish these objects and modify their attributes; other programs can then subscribe to the objects and receive updates whenever the attributes change.

Ideally, this messaging layer is programming-language independent, allowing programs written in a variety of programming languages (C, C++, HTML5, Java, JavaScript, etc.) to intercommunicate, without requiring any special knowledge of one another. Thus, an app in a high-level environment like

HTML5 can easily access services provided by a device driver or other low-level service written in C or C++.



Using persistent publish/subscribe messaging to implement an abstraction layer between high-level applications and system services

Containing apps

Applications from the mobile world can help enrich and extend the infotainment UX. Nonetheless, it's important to ensure the security (and hence the safety) of the car by protecting it from the "wild west" of mobile apps. The system's software platform must therefore isolate such apps in a container to prevent malicious or poorly coded apps from impacting the vehicle, even accidentally.

Keeping it fresh

To stay current, an infotainment system must support over-the-air (OTA) software updates. This requirement will only grow in importance as cars become more connected to fast-evolving cloud services and mobile devices. Ideally, the OTA implementation will use the car's built-in modem. It could also use a smartphone connection, in which case it should use a technology like NFC to simplify the task of pairing the phone and car, as many consumers find conventional Bluetooth pairing difficult and time-consuming.

For practical and economic reasons, OTA updates should consume as little time and network bandwidth as possible. Ideally, then, an infotainment system will support fine-grained updates in which the system downloads only new or modified software components. A publish/subscribe architecture makes such updates easier to implement, as it provides loose, flexible connections between software components, enabling almost any component to be upgraded or replaced without affecting the components that it communicates with. A microkernel OS also simplifies fine-grained updates by enabling device drivers, virtual machines, file systems, networking stacks, and other system-level services to run as separate, dynamically upgradeable processes.

All that being said, keeping an infotainment system relevant cannot simply be a matter of “throwing” apps at the problem. The app paradigm, where the driver must consciously switch from one app to another, only creates distraction in the car. Thus, popular streaming music services like Pandora or Slacker should be seamlessly integrated into the radio user interface; the same goes for point-of-interest or location-based service apps, which should be integrated into the navigation system.

The ideal auto app, then, isn't an app at all, but a plug-in. With a plug-in architecture, the car’s natural interfaces can be extended to include new content and new features in a way that makes apps much easier to understand and interact with.

About QNX Software Systems

QNX Software Systems Limited, a subsidiary of BlackBerry, is a leading vendor of operating systems, development tools, and professional services for connected embedded systems. Global leaders such as Audi, Cisco, General Electric, Lockheed Martin, and Siemens depend on QNX technology for vehicle infotainment units, network routers, medical devices, industrial automation systems, security and defense systems, and other mission- or life-critical applications. Founded in 1980, QNX Software Systems Limited is headquartered in Ottawa, Canada; its products are distributed in more than 100 countries worldwide. Visit www.qnx.com and facebook.com/QNXSoftwareSystems, and follow @QNX_News on Twitter. For more information on the company's automotive work, visit qnxauto.blogspot.com and follow @QNX_Auto.

www.qnx.com

© 2014 QNX Software Systems Limited. QNX, QNX CAR, Momentics, Neutrino, and Aviage are trademarks of BlackBerry Limited, which are registered and/or used in certain jurisdictions, and are used under license by QNX Software Systems Limited. All other trademarks belong to their respective owners.