QNX CAR™ Platform for Infotainment 2.1

Architecture Guide

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: Tuesday, September 16, 2014

Table of Contents

About This Guide

The *Architecture Guide* provides an overview of the QNX CAR platform and describes how its components work together.

To find out about:	See:
The architecture layers	*Top-Level Design* (p. 9)
Mechanisms that you can leverage when you extend or replace architectural components	*Common Mechanisms* (p. 13)
The automotive hardware boards supported by the platform	*Supported Boards* (p. 15)
Technologies provided by partners	*Ecosystem* (p. 17)
Optimizing the boot sequence and boot times	*Boot Time Optimization* (p. 19)
QNX Neutrino RTOS, OpenGL ES, and Software Update components	*Platform Layer* (p. 21)
Multimedia, Navigation, Automatic Speech Recognition, Radio, and the Mobile Device Gateway components	*Middleware Layer* (p. 29)
The HTML5 and Qt5 application runtime environments, application management, and the HMI Notification Manager.	*HMI Layer* (p. 49)

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL)`
Command options	`-lR`
Commands	`make`
Constants	`NULL`
Data types	`unsigned short`
Environment variables	***PATH***
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	**Ctrl–Alt–Delete**
Keyboard input	`Username`
Keyboard keys	**Enter**
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective** → **Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we typically use a forward slash (/) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

Chapter 1
Top-Level Design

The QNX CAR platform architecture consists of three layers: Platform, Middleware, and Human-Machine Interface (HMI).

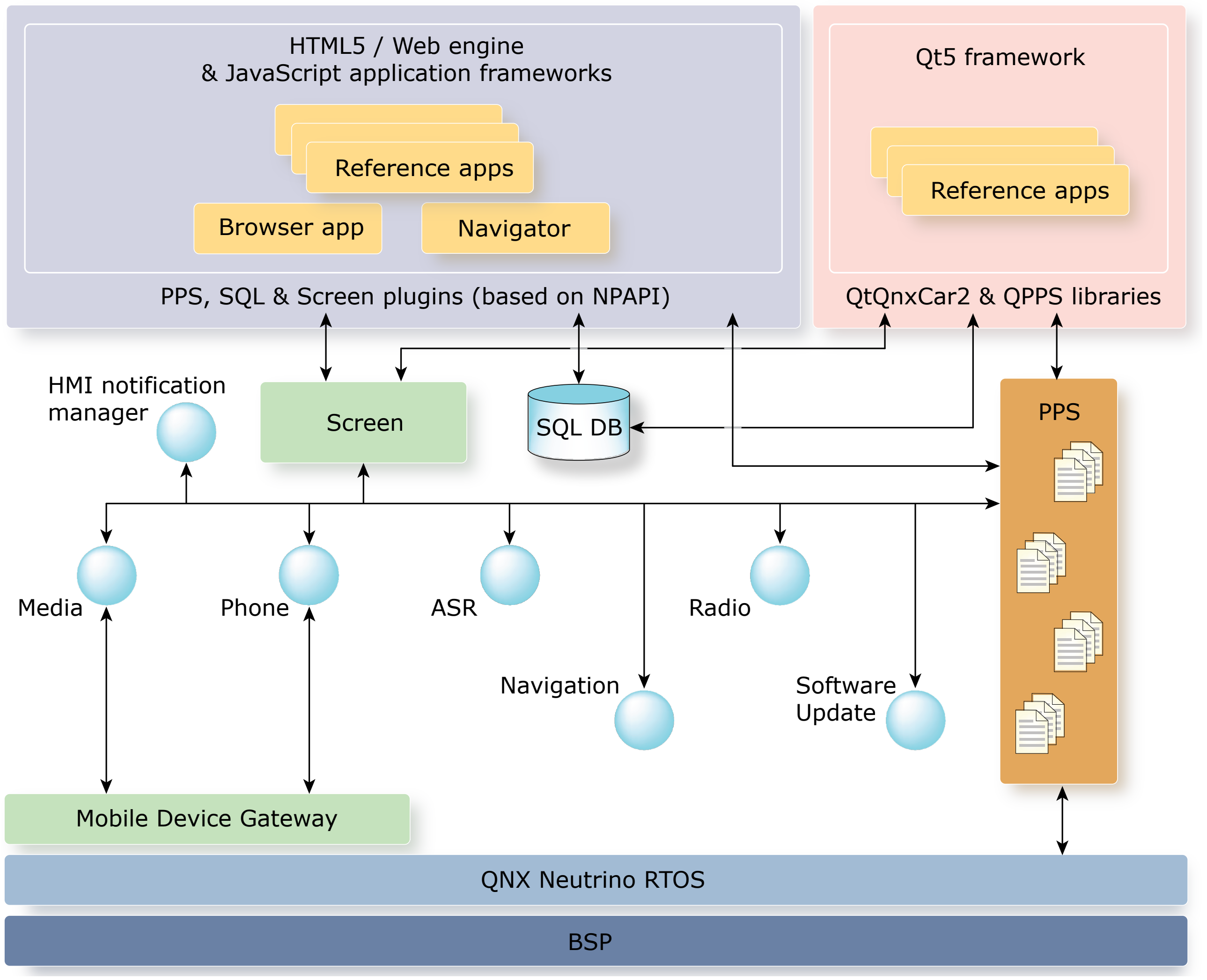

Figure 1: A high-level view of the QNX CAR architecture

Platform

The Platform layer includes the QNX Neutrino RTOS, the Board Support Package (BSP) appropriate for the hardware platform, and the Software Update component for managing system upgrades.

Middleware

The Middleware layer provides infotainment-related services to applications, such as multimedia, Bluetooth, navigation, and radio. The Automatic Speech Recognition (ASR) component provides services not only to the layer above but to the other middleware components as well. For example, ASR can be used to turn on the radio, initiate a Bluetooth phone call, or play a video track managed by the multimedia component. The Mobile Device Gateway provides interfaces to external components through Bluetooth, DLNA, Wi-Fi, and other mechanisms.

Meanwhile, the HMI Notification Manager reacts to system events such as navigation updates or text message arrivals by updating the HMI, based on a policy specified in a configuration file.

HMI

The HMI layer consists of several reference UI apps that provide controls for general subsystems such as the media player, navigation service, and climate controller.

This release includes both a Qt5 and an HTML5 version of the HMI. By default, the Qt5 HMI is shown but you can configure your QNX CAR system to launch the HTML5 HMI at startup, as explained in "Running the HTML5 HMI" in the *User's Guide*. The appearance and functionality of the entire HMI—the **Home** screen shown after the HMI loads, all reference UI apps, and the UI controls for switching screens and launching apps—is identical in the two versions. Note that the architecture supports running apps written with many UI technologies, including native (OpenGL ES), Qt, HTML5, and others, with either HMI version.

The HMI layer includes a browser engine based on the Torch browser. This engine is the runtime environment for the HTML5 HMI and HTML5 apps. Plugins based on the Netscape Plugin Application Programming Interface (NPAPI), which are loaded by the browser engine, provide apps with access to services from the Middleware layer. For example, the SQL plugin accesses databases that store contact information and messages read off Bluetooth devices. A plugin for the Persistent Publish/Subscribe (PPS) service supports persistent object storage and change notification.

To access the NPAPI plugins, HTML5 apps use Apache Cordova plugins implemented in JavaScript, which provide high-level APIs for performing tasks related to navigation, media playback, and so on. The reference apps in the HTML5 HMI use Cordova as well as other JavaScript frameworks. For more information on the HTML5 frameworks in the product, see the *HTML5 and JavaScript Framework* guide.

Qt5 apps can use the `QtQnxCar2` and `QPPS` libraries to access Middleware layer services. The first library exposes a set of C++ classes that provide high-level controls for using those services and also access to databases through SQL query support. The second library wraps the POSIX interface to the PPS objects used by those services with a replacement Qt interface, which allows client apps to use the Qt notification mechanisms of signals and slots. The reference apps in the Qt5 HMI use both these libraries, with the exception of Media Player, which also uses the special-purpose `QPlayer` library to perform media browsing and playback.

Chapter 2
Common Mechanisms

The QNX CAR platform includes defined mechanisms that you can leverage when you extend or replace architectural components.

Persistence

The PPS service provides persistence so that objects and their attributes can persist across reboots. For descriptions of all the objects that reside on the QNX CAR platform, see the *PPS Objects Reference*.

Interprocess communication

PPS provides services for components to communicate with each other in a loosely coupled fashion. For finer-grained, deterministic IPC, use the services provided by the OS itself. For details, see “Interprocess Communication (IPC)” in the QNX Neutrino RTOS *System Architecture* guide.

Logging

Native apps use the system `slogger` (or `slogger2`) facility. HTML5 apps use the JavaScript console.

Application packaging

Native and HTML5 apps must be packaged before being installed on the QNX CAR platform.

Application installation

Instructions for installing a packaged app are included in *Application and Window Management* (in your QNX SDK for Apps and Media package).

Chapter 3
Supported Boards

The QNX CAR platform is designed to run on popular automotive boards.

Supported boards include:

- Texas Instruments:
  - J5 Eco Rev 5
  - OMAP5 uEVM (ES2 Silicon)
- Freescale:
  - i.MX6Q SABRE Lite
- NVIDIA:
  - Jetson (Tegra 3)

  Our support for this board is experimental at this time. See the release notes for limitations and other details.

Chapter 4 Ecosystem

QNX Software Systems works with partners in its large ecosystem to offer the best possible foundation for building vehicle infotainment systems and to help OEMs and Tier-1 suppliers reduce costs and time to market.

Technologies provided by partners include:

Feature	Partners
Voice recognition	• AT&T • Nuance
Terminal mode	• RealVNC
Bluetooth connectivity	• Sybase iAnywhere • Cybercom
Software Update	• Red Bend
Navigation	• Elektrobit
Integrated apps	• Pandora • TuneIn • The Weather Network
Multimedia cookies	• Texas Instruments • Freescale
DSP radio interfaces	• Texas Instruments
Components and frameworks	• jQuery • Sencha

Chapter 5 Boot Time Optimization

Every system has its own set of requirements to meet at boot time. The platform is shipped with many boot-time optimizations already implemented, but it is designed so that you can optimize system startup so that it best meets your goals. By implementing some simple techniques at various points in the boot sequence, you can make the OS and applications load, initialize, and launch more quickly or in a specific sequence.

In the QNX CAR platform, the boot sequence has been optimized using several different techniques. These optimizations focus on the following goals:

Early splash screen and camera

The system loads the image filesystem (IFS) and begins executing the build script as soon as possible. The build script then launches the Screen Graphics Subsystem and the graphical app as early as possible. To accomplish these tasks, we have optimized the initial program loader (IPL) to reduce the IFS size and have reordered the sequence in which services are launched in the boot script.

Early audio

This optimization uses the same techniques as early splash screen and camera, except that audio is started as early as possible in the boot sequence.

Early HMI display

To display the HMI as early as possible, the system uses the same techniques as the first two optimizations while reducing the HMI's dependencies to what is strictly necessary. This work has led to the development of the *Boot Manager*, a service that manages HMI dependencies and instructs the System Launch and Monitor (SLM) service to launch processes based on the state of the system and the HMI requirements. The Boot Manager allows the HMI to come up before all the apps are instantiated.

Early playing of last audio

This optimization is based on a new multimedia service (`mm-player`) that saves its state at shutdown and restores it at power-up. To meet the goal of quickly resuming audio playback at the same track and position when the system was shut down, the `mm-player` service and its dependencies are carefully placed in the SLM configuration file so that they are run at the earliest point possible.

For more information on optimization techniques and on the system's boot sequence, see the *Boot Optimization Guide*.

Chapter 6
Platform Layer

The platform layer includes the QNX Neutrino RTOS, the BSP applicable to the target hardware, and the Software Update component.

QNX Neutrino RTOS

The QNX Neutrino RTOS is a full-featured, robust operating system that scales down to meet the constrained resource requirements of realtime embedded systems. Its true microkernel design and modular architecture enable customers to create highly optimized and reliable systems with low total cost of ownership.

Microkernel

Microkernel architecture gives the QNX Neutrino RTOS the reliability to run life- and safety-critical applications in nuclear power plants, hospitals, space stations—wherever a rock-solid, dependable system is a must.

The QNX Neutrino RTOS is so reliable because it is a true microkernel operating system. Every driver, protocol stack, filesystem, and application runs in the safety of memory-protected user space, *outside the kernel*. If a component happens to fail, it can be automatically restarted without affecting any other components or the kernel. No other commercial OS offers this degree of protection.

Instant Device Activation

Instant Device Activation (IDA) allows in-vehicle systems to perform intelligently even before the OS is operational.

Code is directly linked into the startup component of the boot loader. In this way it can perform all the necessary functions, such as responding to external events, meeting critical 50-millisecond startup response times, accessing hardware, and storing data for use by the full driver. For example, a system can be configured to provide immediate response to power-mode messages transmitted over the CAN bus.

IDA allows system developers to manage data from the CAN bus without adding costly hardware. Conventional OS implementations often take several seconds to boot up from a cold or low-power state, requiring auxiliary communications processors to meet timing and response requirements. Using QNX IDA technology, this problem can be solved in software, eliminating hardware components and decreasing bill of material (BOM) costs.

Networking

The QNX Neutrino RTOS supports IPv4/IPv6 over Ethernet as well as Wi-Fi 802.11 and provides the standard complement of network services, including DNS, DHCP, `inetd`, firewall, FTP, TFTP, HTTP, Telnet, PPP, NFS, and NTP. Since the OS supports POSIX APIs, you can easily incorporate other open-source networking components (e.g., Asterisk for VoIP).

As part of our vehicle reference implementations, QNX Software Systems provides a full Wi-Fi access point that can be used in conjunction with a Bluetooth-capable mobile phone to provide an Internet gateway, which can then be used throughout the cab of the vehicle.

As a distributed operating system, the QNX Neutrino RTOS uses an underlying networking approach known as Transparent Distributed Processing (TDP, also known as Qnet). All Qnet-connected nodes can share devices and OS resources. For example, every node on a network can transparently communicate with a Bluetooth-connected phone, even if the node doesn't have a Bluetooth interface. This provides overall cost savings by reducing the memory footprint and by removing additional software costs.

The QNX concept Porsche provides another example. This car has a multimedia library that runs on the head unit. The two Rear Seat Entertainment (RSE) units are able to share this library. On the head unit, the library is accessed through a POSIX path such as `/db/mmlibrary.db`. When accessed from the two RSE units, the path for this resource would be `/net/headunit/db/mmlibrary.db`. The simplicity of TDP allows the underlying source code to remain the same. Only the resource path changes, even when accessing resources across the network. Note that the name change can be avoided by using a symbolic link.

Fault detection

Keeping components isolated through memory protection lets the OS do the hard work of finding stray pointers and crashes, often finding difficult bugs that can elude a system deployed on a monolithic operating system.

We also provide a powerful *instrumented kernel* that traces all system activity, including interrupts, thread scheduling, and interprocess communications. You can use the system profiler (included in the QNX Momentics Tool Suite) to visualize this trace log to help diagnose system problems and optimize the software. For details, see “Analyze Your System with Kernel Tracing” in the *IDE User's Guide*.

High Availability

The modular, microkernel architecture of the QNX Neutrino RTOS enables faults to be isolated, down to the driver level. Together with the High Availability framework's “smart watchdog”, this architecture helps your system recover from faults automatically. This approach enables you to develop truly self-healing systems.

Our High Availability technology provides:

Instant fault notification

The watchdog automatically detects process faults, triggering recovery procedures. Heartbeating technology can detect nonfatal errors.

Customized failure recovery

Using the High Availability framework library, your system can tell the watchdog what recovery actions to take if an error occurs.

Instant reconnections

The High Availability framework includes a client-recovery library that lets your system instantly reestablish broken connections if a component fails.

Postmortem analysis

If a process faults, the High Availability framework can generate a full memory dump for analysis.

Resilience to internal failures

The watchdog employs a self-monitoring “guardian” that can take over if the watchdog itself fails.

When the watchdog detects component failures, it notifies the system and then manages recovery.

Adaptive Partitioning

Adaptive Partitioning is a technology that can budget CPU cycles for a process or group of processes to create a system whose parts are all protected against resource starvation.

Task or process starvation is a fundamental concern for any embedded system. Services provided by lower-priority threads—including diagnostic services that protect the system from software faults or denial-of-service attacks—can be starved of CPU cycles for unbounded periods, compromising system availability.

Adaptive Partitioning guarantees that all partitions get their budgeted share of CPU time to ensure your system runs correctly under all conditions.

Adaptive Partitioning reserves CPU cycles for a process (or group of processes) and applies a dynamic scheduling algorithm to assign CPU cycles from partitions that aren't using them to partitions that can benefit from extra processing time. Partition budgets are enforced only when the CPU is running to capacity. Adaptive Partitioning thus lets systems run at capacity, enforcing partitioning budgets only when processes in more than one partition compete for cycles.

Resource manager framework

The resource manager framework lets you integrate new technologies and services through a standard POSIX interface that all applications can use.

A resource manager is a user-level server program that accepts messages from other programs and, optionally, communicates with hardware. The resource manager framework can be used for any requirement, from adding a new device type (driver level) to a whole software subsystem, such as a voice-recognition engine.

Since the QNX Neutrino RTOS is a distributed, microkernel system with virtually all nonkernel functionality provided by user-installable programs, a clean and well-defined interface is required between client programs and resource managers. With this framework, resource managers have a common, POSIX interface—*open()*, *read()*, *write()*, and *close()*—to provide a wrapper around the underlying technology. Once a resource manager is implemented, it instantly becomes network-distributed and available locally.

OpenGL ES

OpenGL ES is subset of the OpenGL cross-platform API designed for 2D and 3D graphics on embedded systems, including consoles, phones, appliances, and vehicles. It consists of a well-defined subset of desktop OpenGL, creating a flexible and powerful low-level interface between software and graphics acceleration.

OpenGL ES includes profiles for floating- and fixed-point systems and for the EGL specification for portably binding to native windowing systems. The benefits of OpenGL ES include:

- “raw” 3D graphics support, accelerated by the GPU
- standardized low-level interface across different GPUs
- the ability to create customized vertex and fragment shaders for specialized effects
- superior performance (all other frameworks eventually call OpenGL ES for rendering)

Software Update

The Software Update (SWU) feature for the QNX CAR platform provides a mechanism for safely updating your platform software.

The feature uses these components:

- a software update daemon (`swud`)
- plugins for the update daemon that provide platform-specific functionality
- a partial-shutdown utility (`downsize`) that prepares the system for a software update
- an HMI application that allows the user to initiate a software update

When you connect a USB mass-storage device to your system and the device contains a software update package, which consists of a *manifest file* and a *delta file*, the following occurs:

1. The `mcd` utility detects the manifest file (which has a filename ending in `.manifest`) in the software update package and notifies `swud`.
2. The `swud` daemon validates the update package by parsing the manifest file and the delta file (whose full path is specified in the manifest file). If the update package is *not* valid (e.g., the base version stated is wrong), `swud` logs an error (to `slogger` and to the HMI) and then stops the update process.
3. If the update package is valid, `swud` notifies the HMI, which displays the pending update in the software update application.
4. When ready, the user initiates the software update from the HMI.
5. The `swud` service coordinates with the `downsize` utility to terminate all processes except those needed to perform the update (e.g., `screen`, disk drivers).
6. After the nonessential processes have been terminated, `swud` updates the system based on the contents of the delta file.
7. The system is rebooted and the newer version of the system now runs.

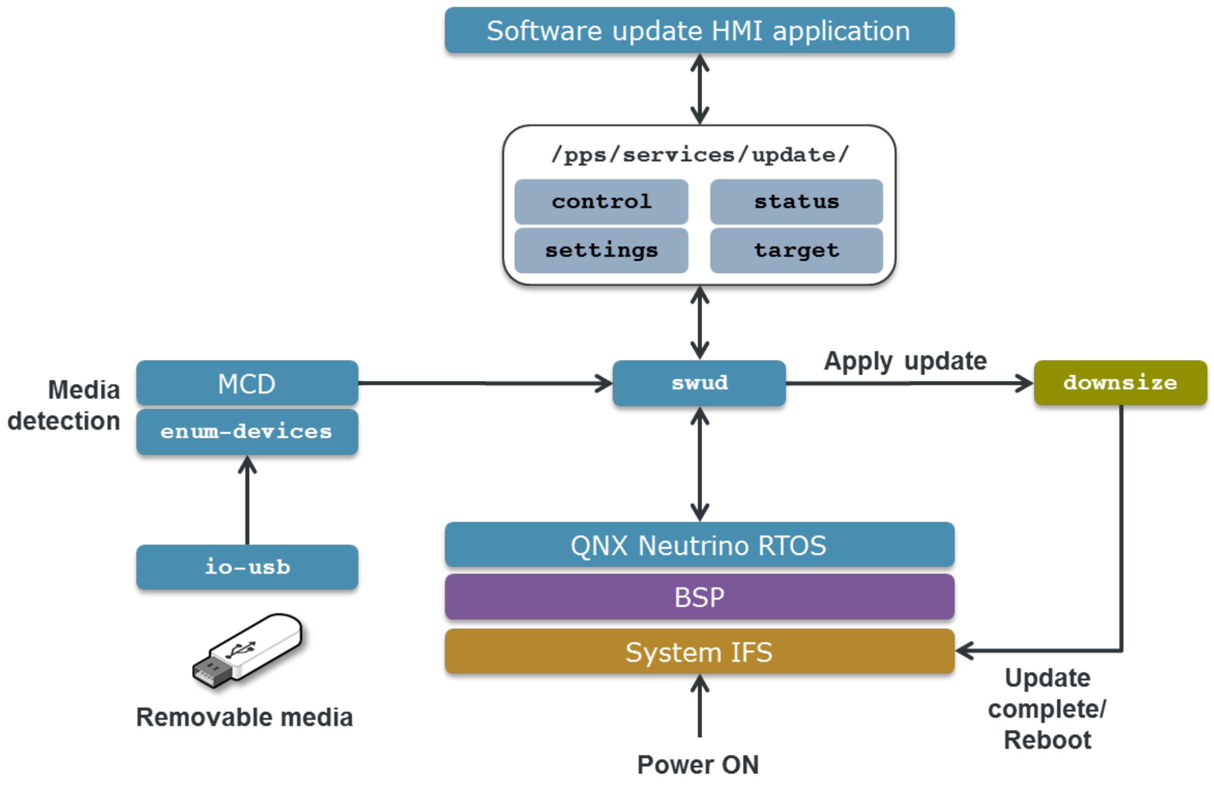

Figure 2: Software Update components detect and validate the software update package, downsize the system processes, and reboot the system to apply the update

The unique microkernel architecture of the QNX Neutrino RTOS makes software updates easy and reliable. For example, if the user obtains a new smartphone that requires a new device driver, the Software Update mechanism ensures that the QNX CAR system uses the new driver with the new phone but an older driver with an older phone. This allows the system to be updated for new devices yet reduces the validation effort so that only the new devices require revalidation.

For more information on initiating software updates, see Software Updates in the *User's Guide*. For a detailed explanation of swud and how to generate a delta file, see Software Updates in the *System Services Reference*.

Chapter 7
Middleware Layer

The Middleware layer provides infotainment-related services to applications. The major components of the layer include Multimedia, Navigation, ASR, Radio, and the Mobile Device Gateway.

Multimedia

Besides providing access to media devices, the multimedia subsystem reads and interprets metadata, converts audio/video streams, and manages playlists. The subsystem also provides the business logic for retrieving album art, directing track playback, detecting media sources, and presenting media to the user for selection.

The media browsing and playback engine is decoupled from the media rendering engine, allowing various HMI components to interact with iPods, USB sticks, and other media devices. The multimedia design is shown in this diagram:

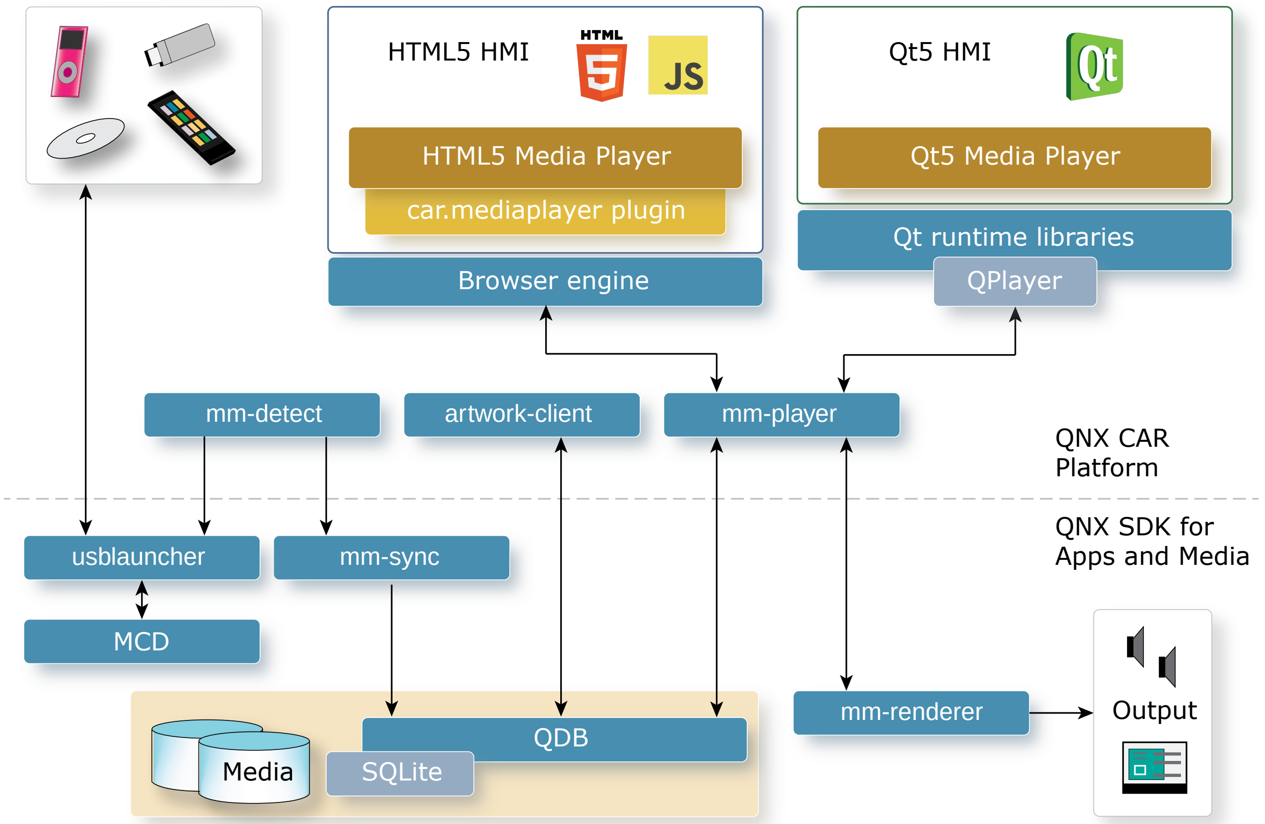

Figure 3: Architecture of media browsing and playback subsystems in HTML5 and Qt5 HMIs

HMI interaction with media playback services

Both the HTML5 and Qt5 versions of the HMI include the Media Player app but they provide separate mechanisms to this application for sending requests to `mm-player`, which is the back-end media browsing and playback service. In the HTML5 HMI, Media Player communicates with `mm-player` by sending requests and receiving media information through the `car.mediaplayer` plugin, which runs within the *browser engine* (p. 54). In the Qt5 HMI, Media Player makes calls to functions in the `QPlayer` library, which talks to `mm-player`. The `mm-player` service uses `QDB` to retrieve media information from databases and uses `mm-renderer` to control the media flow during playback.

Multimedia detection and synchronization binaries

The platform includes utilities that can detect when the user attaches a device and then respond to this action by synchronizing the metadata describing the device's media content to an embedded database. The synchronization ensures that the database has up-to-date media information before any tracks can be played.

Multimedia detection and synchronization relies on the following binaries:

`/bin/usblauncher`

This service enumerates USB devices, launches drivers for communicating with those devices, and publishes their hardware information in PPS.

`/usr/sbin/mcd`

The `mcd` utility (media content detector or MCD) mounts filesystems of attached devices so their contents can be read by other programs.

`/usr/sbin/mm-detect`

This service calls `mm-sync` to initiate media synchronization, notifies the Media Player about device attachments and detachments, and provides synchronization status updates.

`/usr/sbin/mm-sync`

This service synchronizes the information of media files on mass storage devices with the contents of QDB databases, which can then be read by the Media Player.

`/usr/sbin/qdb`

The QNX Database Server (QDB) utility manages the embedded databases that store multimedia information.

`/usr/sbin/artwork_client_car2`

This media utility uploads artwork found on devices, such as album cover art and thumbnail graphics, to QDB databases.

Multimedia browsing and playback binaries

Through the multimedia browsing and playback service, applications can discover media content on attached devices, control the playback of audio and video tracks or playlists, and retrieve the current playback status and notifications of status changes.

The following binaries manage media browsing and playback:

`/usr/sbin/mm-player`

Browsing and playback commands sent from the HMI Media Player are processed by the `mm-player` service. This program uses device-specific libraries to explore media filesystems, retrieve metadata from tracks and playlists, and initiate media flow for playback. It also interacts with `mm-renderer` to direct the media flow to the specified outputs.

`/usr/sbin/mm-renderer`

The `mm-renderer` service manages the flow of media content from one input to one or many outputs. A media flow can be directed to an audio or video output device for playback or to a file for recording.

Navigation

The QNX CAR platform includes the Elektrobit (EB) *street director* navigation software. Both the HTML5 and Qt5 HMI versions include a navigation front end that accesses the EB navigation engine through the platform's native navigation service.

The native navigation service can communicate with any navigation engine running on the system. This design allows you to install and use software from multiple vendors and reconfigure your system to use a particular navigation engine without impacting your HMI apps. For more information, see "Navigation Engine" in the *System Services Reference*.

The navigation subsystem design for the two HMI versions is shown this diagram:

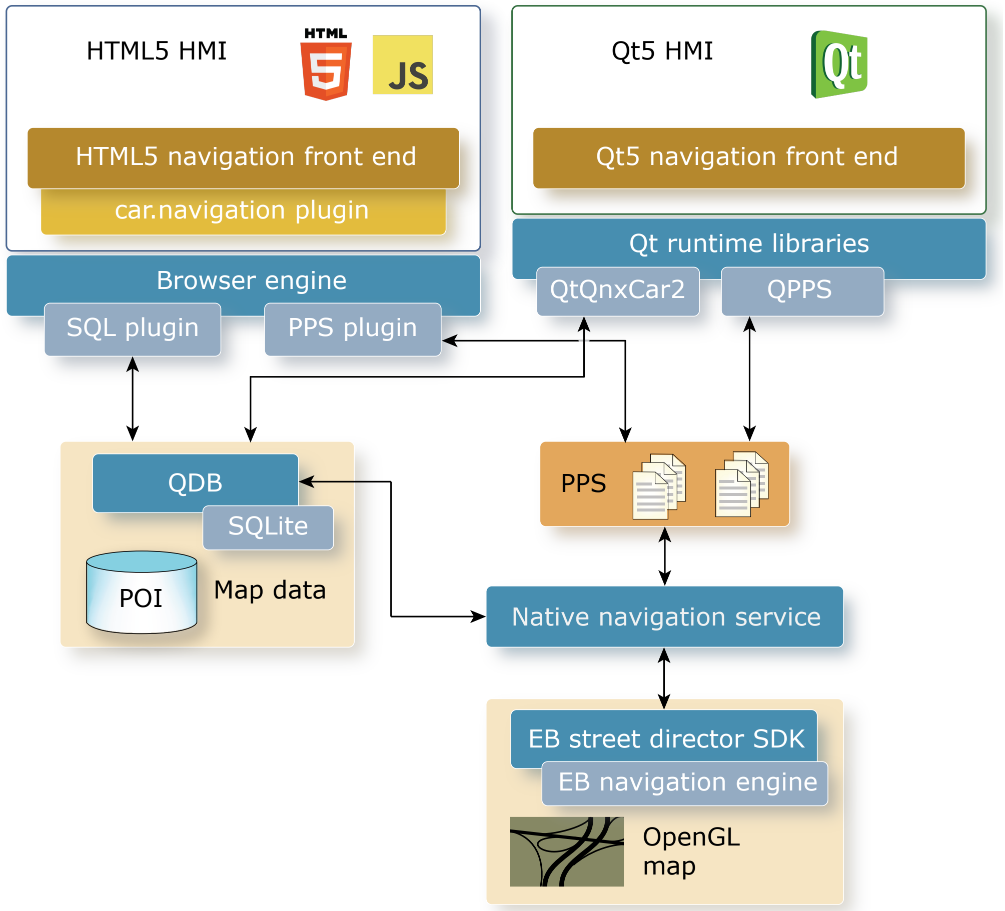

Figure 4: Architecture of navigation subsystems in HTML5 and Qt5 HMIs

The native navigation service publishes information to a PPS object, making details about the route available to other applications. For example, trip information can be made available to the weather application, which can then display the weather forecast for the estimated arrival time at the destination. The `car.navigation` plugin provides

HTML5 applications with navigation status, which it reads from PPS. For Qt5 applications, the `QtQnxCar2` and `QPPS` libraries wrap the PPS objects and provides functions for reading navigation information.

When the user directs the navigation system to a Point of Interest (POI) or asks to search a city, province, state, or country for a particular POI, the HMI writes to the navigation control PPS object by using either the JavaScript plugin or the Qt-supporting libraries, depending on which HMI version is in use.

When the native navigation service reads a request from the control PPS object, it forwards the request to the EB SDK. Once the EB SDK returns the results, the native navigation service updates the same PPS object to notify the sender of the request about the request's completion. The service also writes the results to a database, so subscribers that receive the notification can then read the results from the database. In the HTML5 HMI, subscribers access the database through the SQL plugin, which runs within the *browser engine* (p. 54). In the Qt5 HMI, subscribers access the database through the `QtQnxCar2` library, which runs within the Qt runtime environment.

Automatic Speech Recognition (ASR)

The ASR subsystem provides speech-recognition services to other components. Interfaces hide the existence of third-party speech-recognition software so that vendors can be replaced without affecting the rest of the system.

The ASR subsystem uses application-specific conversation modules to provide speech/prompting handling throughout the system. Conversation modules are decoupled from the speech-recognition provider so the same modules will work for multiple ASR vendors. This architecture allows customers to easily add or remove functionality, including adaptations for downloadable applications.

The platform uses various modules to manage human-machine interactions:

Prompt modules

Manage machine-to-human interactions, which can be either visual (onscreen) or audible.

Audio modules

Provide a hardware abstraction layer to hide the details of how audio is captured.

Recognition modules

Provide an abstraction layer to hide the details of the speech-to-text engine, allowing ASR services to be substituted transparently.

Conversation modules

Define how to handle human-to-machine speech commands.

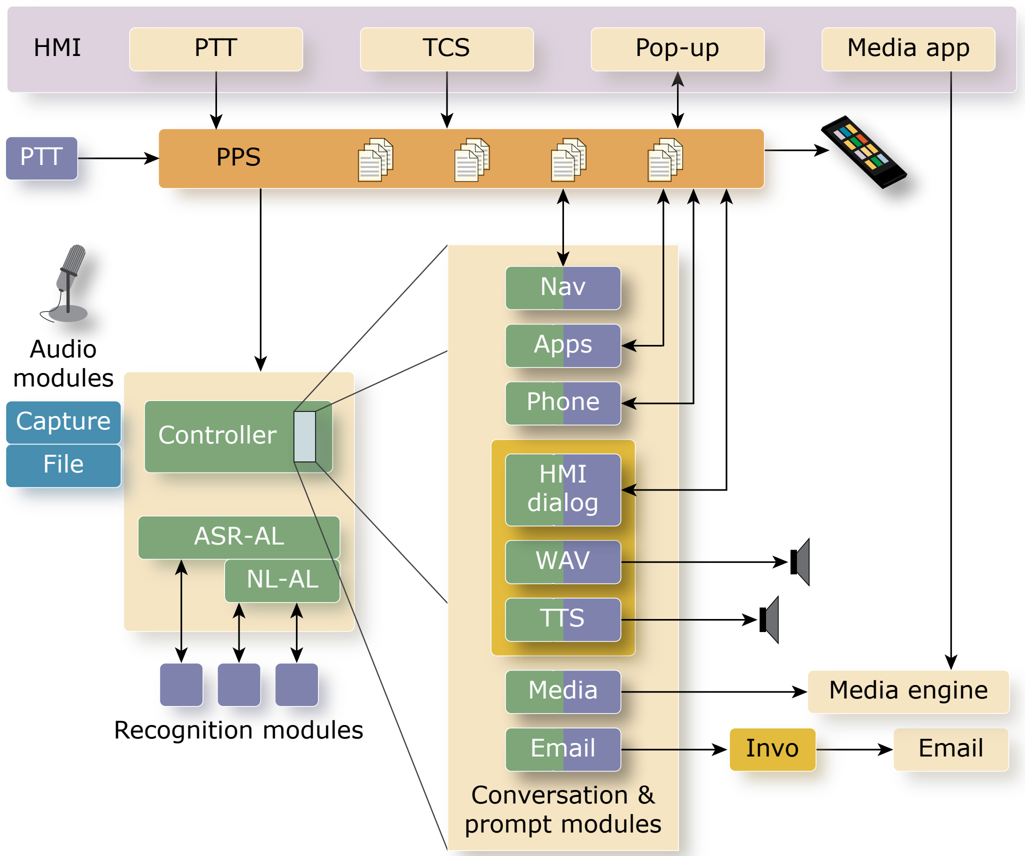

Figure 5: ASR subsystem architecture

The ASR components are pluggable modules. With this architecture, you can reconfigure the ASR system to:

- add or remove conversation modules to modify the speech flow
- adjust the prompt module to change how the user receives responses
- change the recognition module to switch speech-to-text services
- specify different audio modules to capture audio from different sources

For more information, see the following:

- "Automatic Speech Recognition (ASR)" in the *User's Guide*
- *Automatic Speech Recognition Developer's Guide*

ASR modules

The ASR uses various modules to perform tasks such as audio capture and import and to provide prompt services.

ASR services are launched through PPS when the user activates the **Push-to-Talk** tab on the HMI taskbar. These services use different ASR modules.

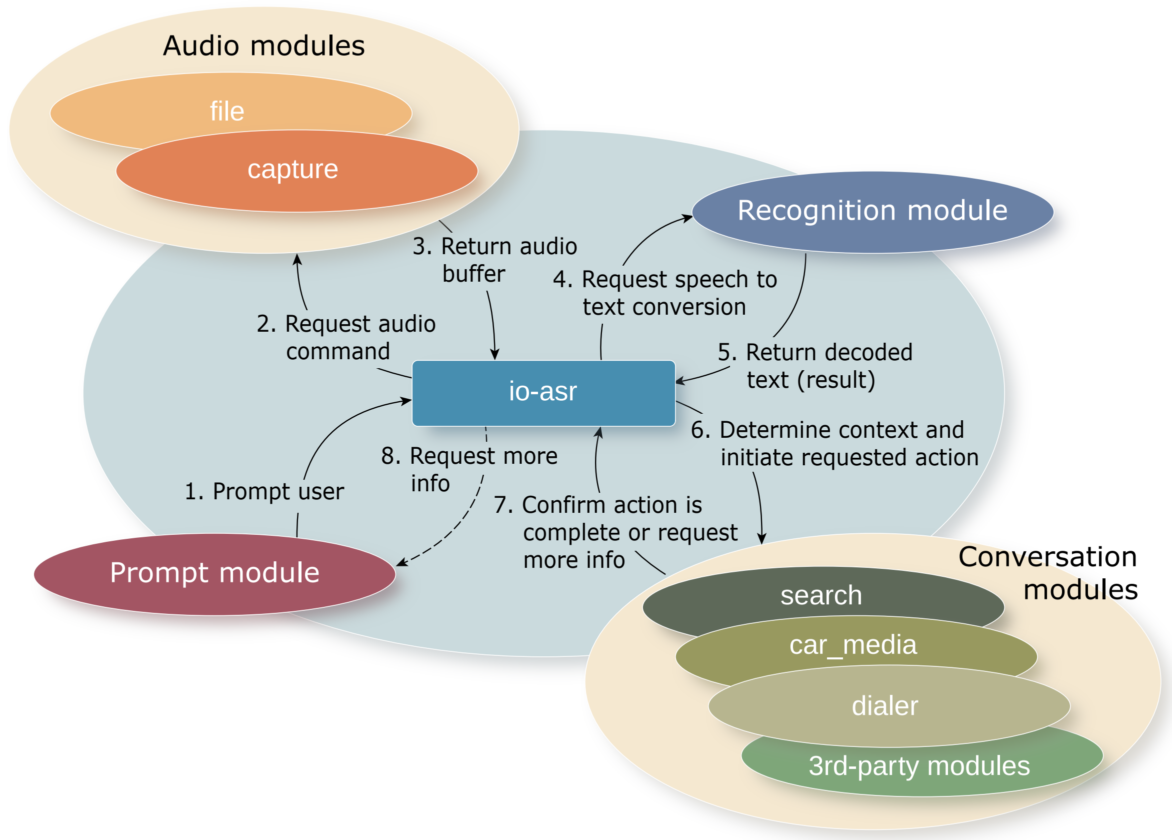

Figure 6: A typical ASR sequence to manage speech commands

Audio capture module

The audio capture module detects the spoken command, including the beginning and end of sentences, and then forwards the audio stream to the third-party recognition modules.

Audio file module

The audio file module imports audio from a file, which it can save for future use. This module is used primarily for testing.

Recognition modules

The recognition module converts a spoken command (*utterance*) to text. It collects the audio sample, passes it to a third-party *recognizer* for processing, and converts the vendor-specific result data (*dictation*) to the format required by the ASR. The ASR service then passes this result on to the Natural Language Adaptation Layer (NLAL). The NLAL uses the grammar provided by the conversation module to produce *intent* information, which it adds to the data in the original result structure.

For example, the recognition module would take the utterance “search media for Hero” and create a results structure with the dictation as follows:

- Result Type: Dictation
- Utterance: “search media for Hero”
- start-rule: search#media-search
- confidence: 600

From this dictation, the NLAL would add *intent* information to the structure:

- Result Type: Intent
- Utterance: “search media for Hero”
- Start-rule: search-media
- Confidence: 600
- Intent entries: 2
- Field: `search-type`, value: `media`
- Field: `search-term`, value: `"Hero"`

Confidence is a value from 0 to 1000 (0 means no confidence; 1000 means complete confidence).

- The conversation modules need the intent fields to understand the meaning of an utterance; without them a conversation is impossible.
- Some recognition modules can produce intent results directly, bypassing the NLAL. The intents that are extracted by the NLAL are predicated by a grammar that must be provided with the conversation module.

Conversation modules

The conversation modules are responsible for:

- determining the domain (e.g., navigation, phone)
- determining whether the conversation is complete or whether another recognition pass is required
- creating and/or modifying PPS objects as required

Apps such as Media Player and Navigation subscribe to PPS objects for changes. For example, if the user presses the **Push-to-Talk** tab and says “play Arcade Fire”, the recognition modules parse the command. The Media conversation module then activates the media engine, causing tracks from the desired artist to play.

Prompt modules

Used primarily by conversation modules, prompt modules provide audio and visual prompt services. Specifically, these modules provide notification of nonspeech responses to onscreen notifications (e.g., selecting an option or canceling a command). Prompts come from prerecorded WAV files or from Text-To-Speech (TTS).

ASR and TTS integration

The ASR and TTS components, libraries, and configuration files manage ASR conversations, enable modules to communicate with each other, and allow control of various component settings.

ASR modules are linked to a particular service variant, depending on the ASR technology used to perform the speech recognition. For instance, in the current release of the platform, the service is `io-asr-generic`. If Nuance technology is used, the service is called `io-asr-nuance`.

The following ASR modules are `io-asr-`*service_name* runtime dependencies:

`/usr/sbin/io-asr-generic`

Manages ASR conversations. This binary isn't tied to a particular recognition technology. If you use this binary, you can specify the recognition module as a plug-in module. However, if you use a technology-specific module binary, such as `io-asr-nuance`, you won't be able to switch recognition modules without recompiling.

`/lib/libasr-core.so.1`

An ASR library that enables inter-module communication.

`/etc/asr-car.cfg`

The recognition module's configuration file that lets you configure:

- audio capture characteristics
- TTS synthesis settings
- recognition module attributes
- module ID mapping

`/opt/asr`

A directory containing resources required by `io-asr-generic` and its modules.

Prompt modules

`asr-offboard_tts-prompt.so`

Prompting service for playing text strings or files as audio. These modules are configurable in the `asr-car.cfg` configuration file.

`asr-wav_file-prompt.so`

Prompting service for `.wav` file playback.

Audio modules

`capture`

Audio capture capability. You can configure this in the `audio/capture` section in the `asr-car.cfg` configuration file.

`file`

Import and save audio files. You can configure this in the `audio/file` section in the `asr-car.cfg` configuration file.

Recognition modules

`asr-vocon3200-recognition.so`

Interpretation of captured audio signals to deduce the words and sentences spoken.

Conversation modules

Conversation modules are independent of each other.

`search`

Used by other modules to handle search-related commands. For example, the navigation subsystem uses this module to process instructions to navigate to a destination or search for a point of interest.

Provides the capabilities for various conversation modules, including:

- app launching
- navigation
- internet and local media search
- weather queries

`car-media`

Processes voice commands for performing media playback actions.

To support these commands, the ASR subsystem uses different back-end plugins in the different HMI versions. By default, the platform is configured to use the plugin for the `mm-player` service (which works with the HTML5 HMI). If you want to use media voice commands when running the Qt5 HMI, you need to use the `mm-control` plugin. You can tell the ASR subsystem to load this plugin instead of the one for `mm-player` by modifying the ASR configuration file, as explained in “Using mm-control to process voice commands” in the *User's Guide*.

`dialer`

Process voice-dialing commands.

Third parties implementing ASR can add additional conversation modules to extend ASR capabilities (e.g., calendar).

Radio

The Radio Integration Module provides a control and status application that serves as a front end to a DSP-based radio.

This release includes a reference radio supplied by Texas Instruments for the TI Jacinto 5 platform. The radio module communicates with the HMI via the PPS service. In addition, the module provides the control interface to the DSP to support tuning, band selection, scanning, Radio Data System (RDS), and more.

Supported radio features include:

- AM, FM, and HD radio
- HD radio metadata
- station presets for each band
- seek and scan functionality
- simulation mode for boards without an antenna
- TI radio tuner on Jacinto 5
- configurable profiles for different regions
- radio control through PPS

Rearview camera

Overview

The QNX CAR platform supports a video feed from a camera attached to the reference board. This camera is intended for use as a rearview (backup) camera in the vehicle.

This release supports cameras on the following boards:

- Jacinto 5 Eco
- i.MX6D
- i.MX6Q SABRE Lite

When the platform detects any cameras attached to the hardware, it automatically launches the rearview camera software component. The default state of this component is to stay paused and not render the video feed. It can be triggered to open a window and render the video feed from the camera to the screen by:

- a user command through the HMI
- a change notice sent from the vehicle hardware (e.g., the vehicle is put in reverse)

For more information about using the rearview camera, see "Rearview Camera" in the *User's Guide*.

PPS objects

The QNX CAR platform uses the `/pps/qnxcar/sensors` PPS object's *cameraRearviewActive* attribute to learn if a rearview camera is attached and active.

For application and window management, the rearview camera software component uses the following PPS objects:

- `/pps/system/navigator/command`
- `/pps/system/navigator/windowgroup`

Mobile Device Gateway

The Mobile Device Gateway component allows passengers to link mobile devices to the QNX CAR platform. Supported interfaces include Bluetooth, USB, DLNA, MirrorLink, 3G, LTE, and Wi-Fi. Each connection has associated PPS objects that apps can use to interact with the device. By providing this level of indirection, apps are freed from having to know the details of the physical connection.

Android interfaces

The platform can connect with Android devices through Bluetooth (supported profiles include A2DP/AVRCP, HFP, MAP, PAN, PBAP, and SPP), DLNA, and MirrorLink, where available.

Bluetooth

The platform supports connections to external devices over Bluetooth—it can initiate a Bluetooth connection or accept a connection request.

Supported Bluetooth profiles include:

- HFP (Hands-Free Profile)
- MAP (Message Access Profile)
- PBAP (Phone Book Access Profile)
- SPP (Serial Port Profile)
- A2DP/AVRCP (Audio Profiles)

The QNX CAR platform uses the Cybercom blueGO application software framework, which is a wrapper around Sybase iAnywhere's Bluetooth protocol stack and profiles. While the profiles listed above are preintegrated and tested for this release, the Bluetooth stack supports many other profiles that can also be integrated.

The front-end resource manager for Bluetooth is `io-bluetooth`, which offers a POSIX-compliant API and provides:

- low-level access to the Bluetooth radio chip (typically via a serial interface such as UART, USB, or I2S)
- support for Bluetooth profiles
- profile concurrency and control

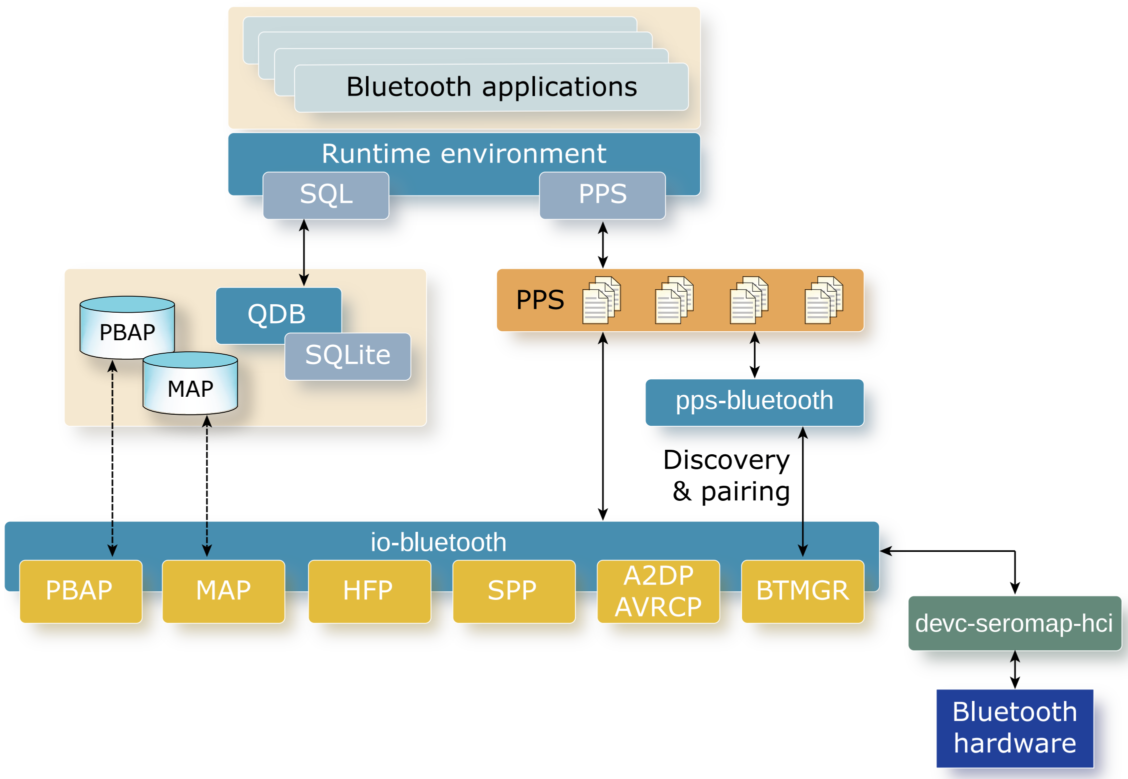

Figure 7: The QNX CAR architecture for Bluetooth

The PPS abstraction layer for Bluetooth provides an application interface for activities such as pairing, profile management, status updates, and device list management.

A Bluetooth connection can be initiated either by the vehicle or by the mobile device. When requesting the connection, the developer selects a Bluetooth profile. After the connection has been established, the system creates PPS objects that `io-bluetooth` uses to manage the connection according to the permissions and other parameters in the selected profile.

DLNA

Support for Digital Living Network Alliance (DLNA) is preintegrated, providing streaming capabilities from smartphones or tablets.

Apple iOS interfaces

The QNX CAR platform provides a *Made for iPod* interface for Apple iOS products, allowing users to synchronize media metadata and to browse and play media on the device. The iPod Out protocol is also supported.

The iOS interface supports various Apple models, generations, and firmware releases. The iPod interface software conforms to the relevant sections of the “Apple iPod Accessory Interface Protocol Specification Release”. Automotive suppliers will use this interface software in conjunction with hardware that complies with the “Apple iPod/iPhone Hardware Specification Release". These specifications are available from Apple.

Since the QNX iPod interface is intended to work with Apple iPods and iPhones that comply with these specifications, a high level of compatibility can be expected when using iPod Out control through a touchscreen or other external USB HID device or in the scenario when the iOS device takes over the entire screen.

The focus here is on driver and iAP protocol. To complete the implementation, automotive suppliers need to add:

- video capture to scale video appropriately and render as a display layer
- interface from video capture into composition manager

If and when Apple moves to digital video, we will be able to provide a complete framework for iPod Out.

MTP device interface

The QNX CAR platform provides an interface for devices that use the Microsoft Media Transfer Protocol (MTP). Users of the platform can synchronize media metadata and can browse and play media that is stored on these devices.

MTP devices maintain their own filesystems. With the MTP device interface, applications can access media on these devices in the same manner as they access media on USB devices.

For more information, see the *Multimedia Renderer Developer's Guide*.

MirrorLink

The QNX CAR platform relies on the RealVNC Mobile Solution for the VNC and other MirrorLink-specific protocols.

The platform supports the *MirrorLink* technology standard (version 1.1) to enable MirrorLink apps on a supported smartphone to work with the car's HMI. For the current list of MirrorLink Certified phones (also called *server devices*), see the following page at the Car Connectivity Consortium (CCC) site (using the **Servers** search filter):

MirrorLink™ Certified Product Listing

For more information, see the MirrorLink entry in the *System Services Reference*.

The Screen software component is used to scale and combine the VNC graphics output within the UI environment. For more information, see the *Screen Graphics Subsystem Developer's Guide*.

Network interfaces

The QNX CAR platform supports connections to mobile devices over Ethernet, Wi-Fi, 3G, and LTE.

The `io-pkt` resource manager is the low-level interface to networking/connectivity options such as wired Ethernet, Wi-Fi, and other radio devices (3G/LTE). The `io-pkt` resource manager integrates low-level drivers and the TCP/IP stack as well as a number of technologies for managing network connections.

PPS is used for controlling the various components contained in `io-pkt`. Overall network management includes:

- APIs for network control planes
- coordination of system-wide network configuration (preferred/default network interfaces, routes, and resolver configuration)
- general networking information details and device information

USB

USB connections are handled in the same way as Bluetooth and the networking interfaces—through a combination of low-level support (`io-usb`) and PPS management/control. Integration of CDC-ECM (USB Ethernet dongle), CDC-NCM (for MirrorLink), and Android Accessory Protocol can be handled through this component.

Chapter 8
HMI Layer

The HMI layer includes reference UI apps, application management services, and runtime components that support apps written in HTML5, Qt5, and other supported HMI technologies.

You can write your own automotive apps and showcase them on the *QNX App Portal* to allow Tier-1 suppliers and OEMs to download and evaluate them. For information on doing this, see *Showcasing Your Apps on the QNX App Portal*.

Application support

In this release, the QNX CAR platform comes with distinct HTML5 and Qt5 versions of the HMI.

Both versions provide identical UI controls for launching applications written in HTML5, Qt, and other HMI technologies. The applications run independently of the HMI, as shown in this diagram:

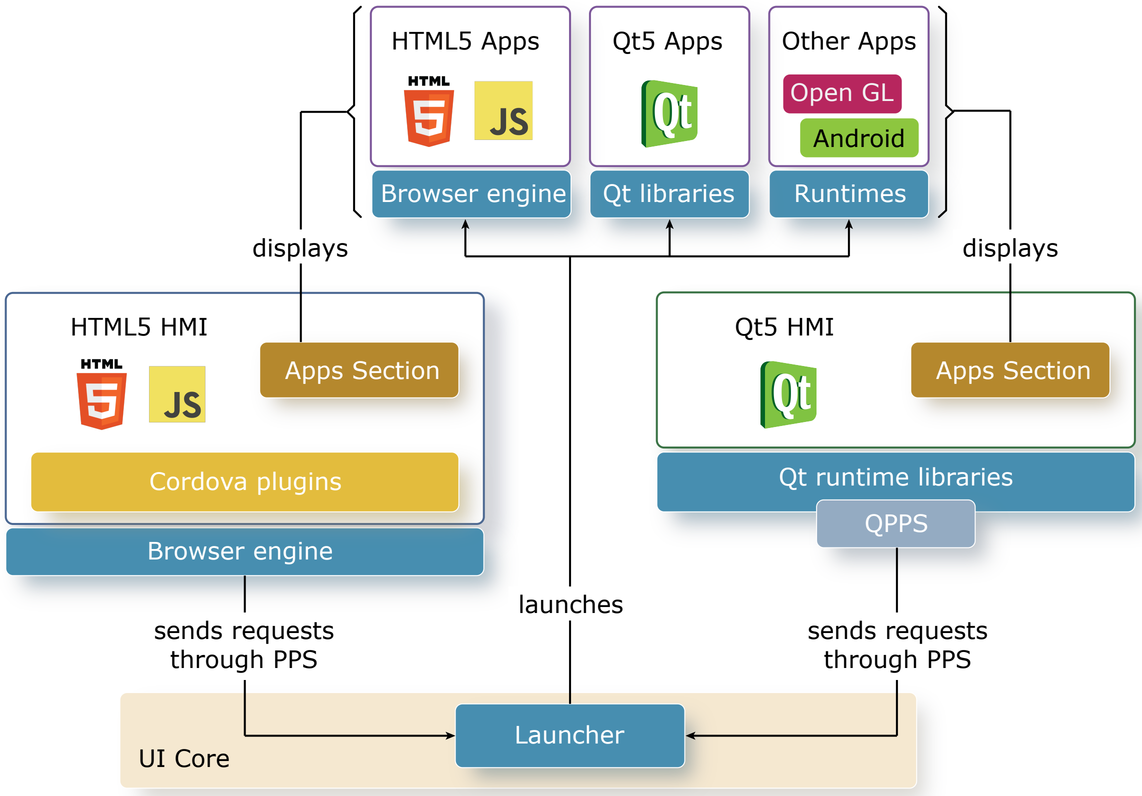

Figure 8: Application support in HTML5 and Qt5 versions of the HMI

The HMI shows any installed applications in the **Apps Section** screen. When the user requests to launch an app by tapping its icon in this screen, the HMI forwards the request through PPS to the Launcher service. The different HMI versions use different mechanisms to read from and write to PPS. The HTML5 HMI uses Cordova plugins, which run in the browser engine; the Qt5 HMI uses the QPPS library.

For information on how the HMI manages application windows, see the *Application and Window Management* guide. For instructions on writing a generic window manager, see "Creating Your Own Application Window Manager" in the QNX SDK for Apps and Media documentation.

Launcher

Launcher starts the selected app after it receives the launch request through PPS. HTML5 apps are run in the *browser engine* (p. 54). Qt apps are run in separate processes, each of which links in its own copy of any required Qt libraries (e.g., `QtQnxCar2`, `QPPS`). Thus, all Qt applications run independently of the HMI and of each other. This is also the case for apps written in other supported HMI technologies, such as Open GL.

The Launcher service is part of the UI Core set of services, which manages the application lifecycle and the visual display, as explained in "*Application Management* (p. 57)".

HTML5 application framework

The HTML5 application framework provides the components needed to support apps that are written with web technologies and that access native services.

Using this framework, developers can create and deploy web applications—built with technologies such as HTML5, CSS3, and JavaScript—that use plugins to access device hardware and native services (just like native C/C++ applications). The plugins are written with Apache Cordova, an open-source framework for developing mobile apps.

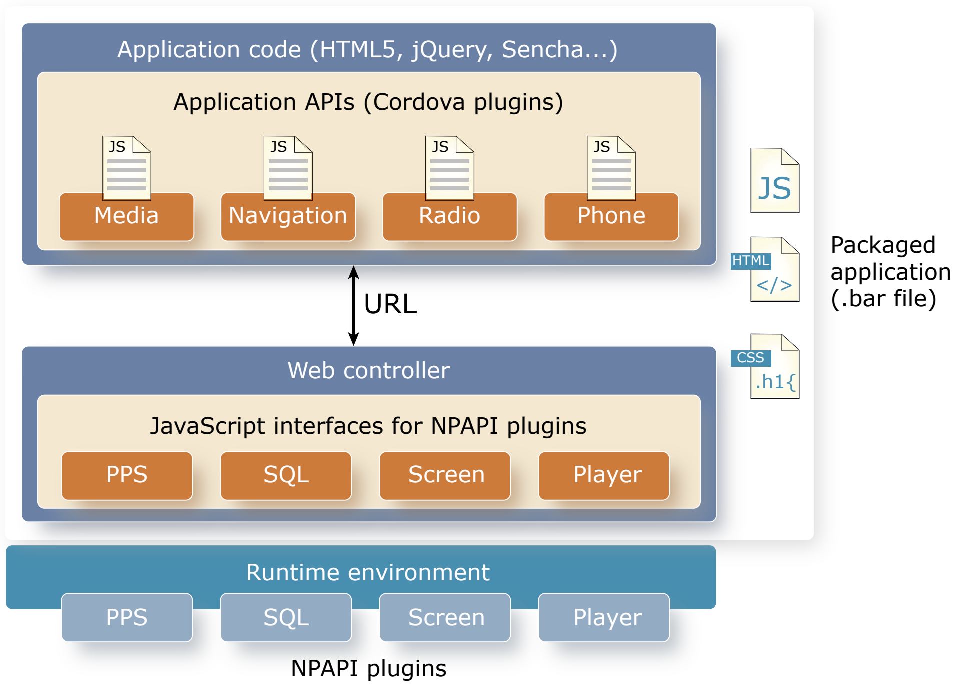
Application code (HTML5, jQuery, Sencha...)
Application APIs (Cordova plugins)
JS
Media
JS
Navigation
JS
Radio
JS
Phone
JS
HTML
</>
CSS
.h1{
Packaged application (.bar file)
URL
Web controller
JavaScript interfaces for NPAPI plugins
PPS
SQL
Screen
Player
Runtime environment
PPS
SQL
Screen
Player
NPAPI plugins

Figure 9: Architecture of the HTML5 application framework

As *Figure 9: Architecture of the HTML5 application framework* (p. 52) illustrates, the Cordova plugins provide JavaScript interfaces for accessing middleware services such as media, navigation, radio, and phone. The app code uses these interfaces to send requests and read results. The requests are sent through URLs to the Web Controller, which executes the plugin code. In the Web Controller, the Cordova plugins talk to plugins based on the Netscape Plugin Application Programming Interface (NPAPI), which access native services. The Web Controller exposes the interfaces for the NPAPI plugins but their code runs in the native layer of the browser. The browser supports the following NPAPI plugins:

PPS plugin

Provides the HTML5 domain with access to the full PPS API.

SQL plugin

Provides QDB database access, including a complete API for opening, querying, and modifying databases.

Screen plugin

Used by the Navigator application, this plugin provides window and buffer management and supports the event notification system. Navigator can use this plugin to set properties for window, context, and display, and to handle events received from the low-level graphics components.

Player plugin

Provides access to the `mm-player` media browsing and playback engine.

Developers can add NPAPI plugins as required.

Sample application

The Communications app uses the `qnx.message` Cordova plugin to read email and text messages and the `qnx.bluetooth.pbap` plugin to read address book information found on Bluetooth devices. When the devices are paired with your system, their message contents and address information gets stored in QDB databases. The Cordova plugins talk to the SQL plugin in the browser engine, which then accesses the databases. To manage phone calls on paired devices, the app uses the `qnx.phone` plugin, which in turn uses the PPS plugin to communicate through PPS with the Bluetooth services (`pps-bluetooth` and `io-bluetooth`).

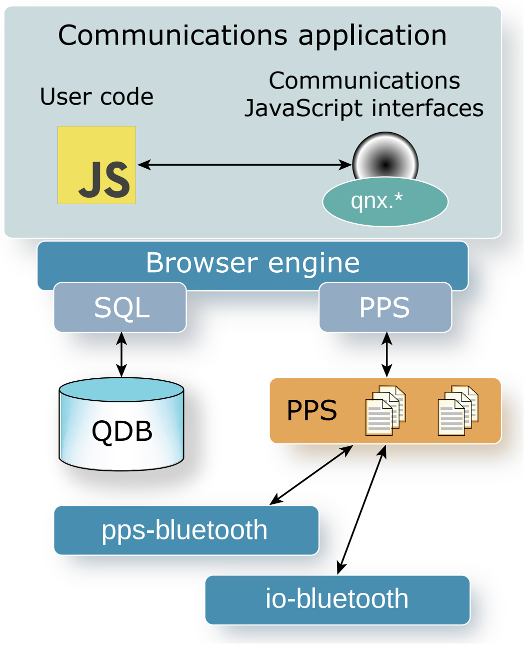

Figure 10: The reference Communications app

Browser engine

The browser engine provides the runtime environment for HTML5 apps. The engine supports many features, including canvas, WebSocket, session storage, offline apps, worker threads, DOM improvements, audio and video tags, and WebGL.

The QNX CAR platform provides a multiprocess architecture that allows system developers to partition the UI into a set of core and sandboxed apps. With this architecture, multiple WebViews (or windows) can either share a common engine instance or run in their own engine instance. Each WebView can be implemented with a separate JavaScript application framework (e.g., jQuery Mobile or Sencha Touch).

By running multiple WebViews in a single engine instance, overall memory footprint can be reduced. However, when all apps share the same engine, they're not isolated from each other. Bad behavior in one app can impact all other apps that share the same engine instance. This mode would typically be used for a set of well-tested apps that are deployed together as a bundle (e.g., core apps shipped from the manufacturer). Or, a single app can be run in its own private engine instance. This provides isolation at the expense of increased memory footprint.

Based on *WebKit*, the browser engine provides support for HTML5 (and related) standards and technologies, including CSS3 and the JavaScript language and associated standards, such as AJAX, JavaScript Object Notation (JSON), and XML. We have optimized WebKit in a number of ways:

- improved user interaction (e.g., complex touch-event handling, smooth zooming/scrolling, fat-finger touch target detection), performance, and battery life for mobile devices
- enhanced user operations such as fast scrolling and zoom (e.g., zooming in on a webpage) to reduce RAM utilization
- enhanced JavaScript execution to improve performance and reduce CPU utilization and unnecessary battery drain
- reduced power consumption (e.g., by throttling background threads)
- support for multimodal input (e.g., trackpad, keyboard, and virtual keyboard)
- improved overall speed (e.g., by selective image down-sampling)

Qt application model

QNX CAR 2.1 includes the runtime binaries for version 5.2 of the Qt framework as well as several Qt-compliant libraries that allow HMI apps to access middleware services.

Developers can write apps that use the Qt framework to specify their GUIs and use the Qt-compliant libraries to access many middleware services. These services include but aren't limited to:

- PPS
- ASR
- navigation
- multimedia
- vehicle sensors
- Bluetooth profiles
- application launching

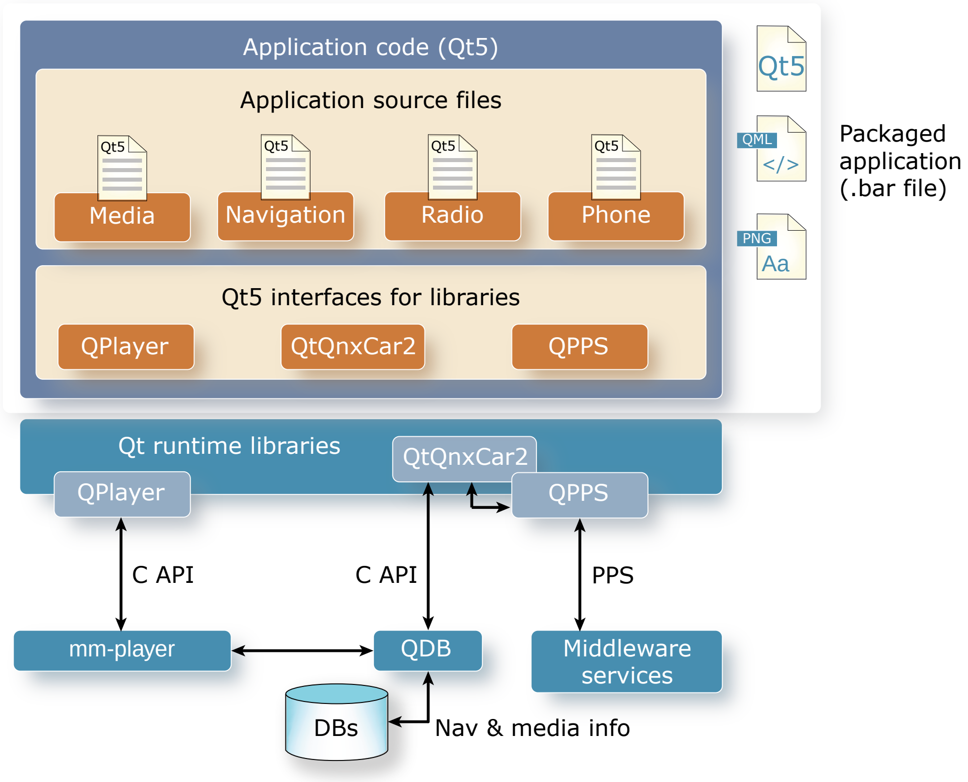

Figure 11: Qt application model

As *Figure 11: Qt application model* (p. 55) illustrates, Qt apps access middleware services through Qt5 interfaces in their source files. These interfaces are exposed by

Qt libraries and consist of C++ classes that manage services such as media, navigation, radio, and phone. Each app links in its own copy of any required Qt libraries, which talk directly to middleware services (e.g., `mm-player`, `QDB`) through their C APIs or PPS. The platform ships with these Qt-compliant libraries:

QPlayer

Integrates apps with the platform's media browsing and playback engine, `mm-player`, by providing a Qt5 API that wraps the `mm-player` C library.

QtQnxCar2

Provides controls for configuring all other middleware services (e.g., navigation, vehicle sensors, ASR) and for accessing QDB databases.

QPPS

Provides a Qt5 API for reading from and writing to PPS objects, replacing the POSIX API for PPS. The `QtQnxCar2` library uses the QPPS library, but apps can also use the QPPS library directly.

You can find full details on these libraries in the *Qt Development Environment* guide.

Application Management

The Application Management service (also known as UI Core) controls the application lifecycle: starting, switching, activating, sleeping, and terminating. It also provides a means of restricting the services that applications can use. UI Core includes Screen, a component that allows multiple applications built with disparate technologies to share the same physical display real estate.

The figure below shows the main UI Core components: the application installer, the authorization manager (`authman`), Screen, and Launcher.

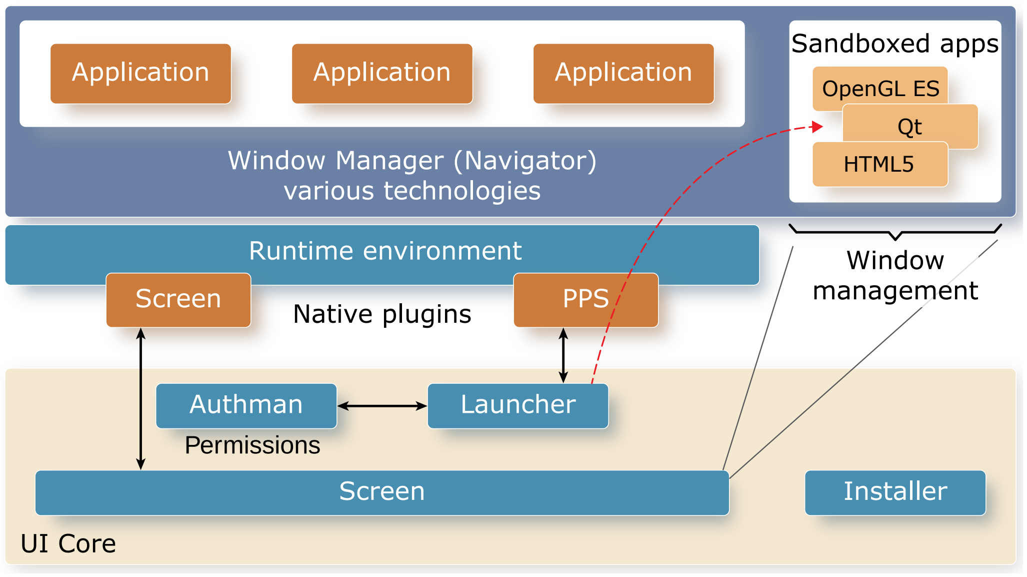

Figure 12: UI Core components

The application installer unpackages the application, validates its signature, and installs the application on the QNX CAR platform.

Launcher enables any application to launch any other application in any UI environment (subject to system permissions).

The `authman` module controls which APIs and system services can be used by each application, enforcing the security model that the system developers have defined. This authorization ensures that downloaded applications can't use interfaces they aren't authorized to use. See *Application and Window Management* for details.

The Screen module integrates multiple graphics and UI technologies into a single scene that can be rendered on the display. Developers can create a separate pane for the output of each rendering engine (such as HTML5, Qt, Video, or OpenGL ES). Each frame buffer can be transformed (scaling, translation, rotation, alpha blend, etc.) to build the final display. Whenever possible, Screen uses GPU-accelerated operations to build the display as optimally as possible, falling back on software only when the hardware can't satisfy a request.

As an example, consider the screenshot of the Torch browser application below:

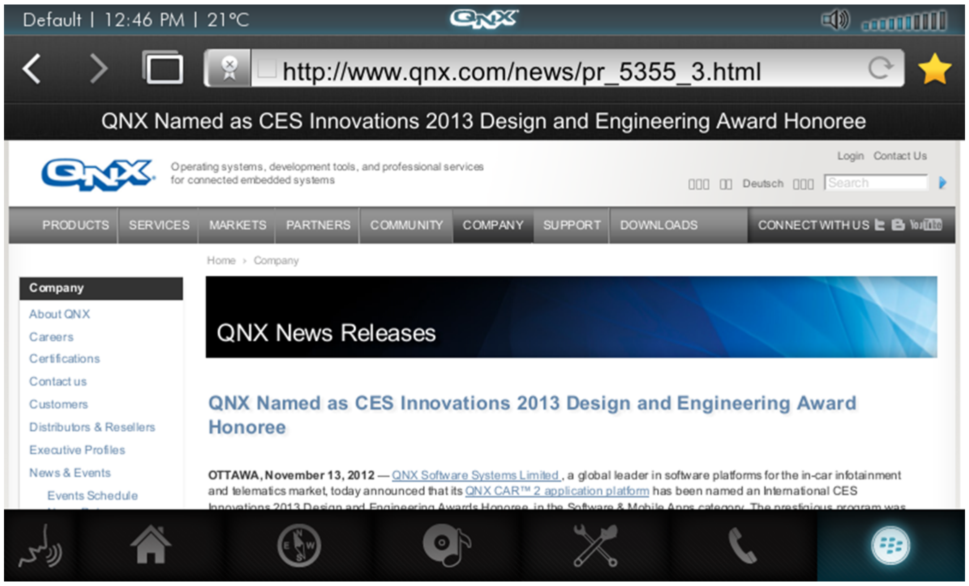
Default | 12:46 PM | 21°C
http://www.qnx.com/news/pr_5355_3.html
QNX Named as CES Innovations 2013 Design and Engineering Award Honoree
Operating systems, development tools, and professional services for connected embedded systems
Login Contact Us
Deutsch
Search
PRODUCTS SERVICES MARKETS PARTNERS COMMUNITY COMPANY SUPPORT DOWNLOADS CONNECT WITH US
Home › Company
Company
About QNX
Careers
Certifications
Contact us
Customers
Distributors & Resellers
Executive Profiles
News & Events
Events Schedule
QNX News Releases
QNX Named as CES Innovations 2013 Design and Engineering Award Honoree

Figure 13: Torch browser application

What appears to be a single homogeneous display is actually made up of the following:

- HTML5 browser chrome (title bar, forward/backward buttons, address bar)
- natively rendered image
- HTML5 navigator tabs

Screen provides a lightweight *composited* windowing system. Unlike traditional windowing systems that arbitrate access to a single buffer associated with a display, a composited windowing system combines individual off-screen buffers holding window contents into one image, which is associated with a display.

For more information about Screen, see the *Screen Graphics Subsystem Developer's Guide*.

HMI Notification Manager

The HMI Notification Manager handles asynchronous, multimodal events based on predefined priorities.

The manager appraises incoming events, applies appropriate rules specified in its configuration file, and then notifies all HMI subscribers via PPS. Like a window manager, the HMI Notification Manager decides when and how events should get processed, based on their priority, and determines whether or not to notify the user via the HMI. But unlike a window manager, it also responds to low-level system services using various input modalities and can manage various outputs in addition to a video display (e.g., you could use the manager to handle audio streams).

Suppose an event such as an incoming phone call occurs. The HMI Notification Manager will notify the Navigator application, which will automatically switch to the appropriate screen for the user to deal with the event, and then automatically switch back to the previous screen when that event has finished (e.g., the phone calls ends). Based on priorities set in the configuration file, the HMI Notification Manager can prioritize competing events—an incoming call versus a low-fuel alert—and apply the appropriate policy for handling them.

For more information, see the *HMI Notification Manager* guide.

Index