Application and Window Management

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, February 25, 2014

Table of Contents

About This Guide

This document describes application and window management on the QNX CAR platform.

All application developers should read this guide.

To find out about:	Go to:
Application and window management in the QNX CAR platform	*Introduction to Application and Window Management* (p. 9)
Application and window management in HTML5 HMI	*HTML5 HMI* (p. 11)
Application and window management in Qt5 HMI	*Qt5 HMI* (p. 19)
The PPS objects and system events that a custom Application window manager implementation must support	*Creating an application window manager: Requirements* (p. 25)

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`
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 use a forward slash (/) as a delimiter in all 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
Introduction to Application and Window Management

Application and Window Management describes the process of starting/stopping applications and explains how windows interact with the HMI.

In this release, the QNX CAR platform comes with separate HMI implementations, one of which is built with the Qt framework and the other with HTML5-related technologies. Both HMI versions provide the same screen-switching functionality through their taskbar. The implementations of application and window management provide a useful reference for writing an HMI-based window manager suitable for automotive systems.

Here is the summary of the key differences between the implementations of application and window management:

HTML5 HMI	Qt5 HMI
Application and window management is managed by a separate Navigator application.	Application and window management is managed by componments (Window manager, Application manager) of the Qt5 HMI.
Each tab on the application and management display area is associated with separate applications.	Each tab on the application and management display area is associated with different views of the same process.
Virtual keyboard runs in the Navigator application.	Virtual keyboard runs as a separate process external to the Qt5 HMI.

Application and window management uses the Persistent Publish/Subscribe (PPS) service to publish app information and to receive window data from certain apps, which helps it display those apps. To render app windows, applications use the Screen Graphics Subsystem. However, the mechanisms used to access these lower-level services differ between the two HMIs.

Note that you can also write a generic window manager (which might not even render any HMI components), and how to do so is explained in "Creating Your Own Application Window Manager" in the QNX SDK for Apps and Media documentation.

For information about PPS objects that are specific to application and window management (which can be found in `/pps/system/navigator/`), see the *PPS Objects Reference*. For details on Screen, see the *Screen Graphics Subsystem Developer's Guide*.

Chapter 2
HTML5 HMI

In the HTML5 HMI, application and window management is managed by the Navigator application. This application is implemented in JavaScript and uses WebWorks extensions to access the PPS and Screen services.

The Navigator display area includes seven tabs that each access an app or group of apps.

Here are the tabs on the Navigator display area:

- Push-to-Talk—activates Automatic Speech Recognition (ASR)
- Home—shows a consolidated view of several key apps
- Navigation—allows the user to interact with the currently active navigation engine
- Media Player—allows the user to play videos or music, or to listen to the radio
- Car Control—provides access to vehicle settings for audio, climate control, etc.
- Communications—provides access to email, text messages, contacts, and a telephone dial pad
- Apps Section—shows the complete set of available apps

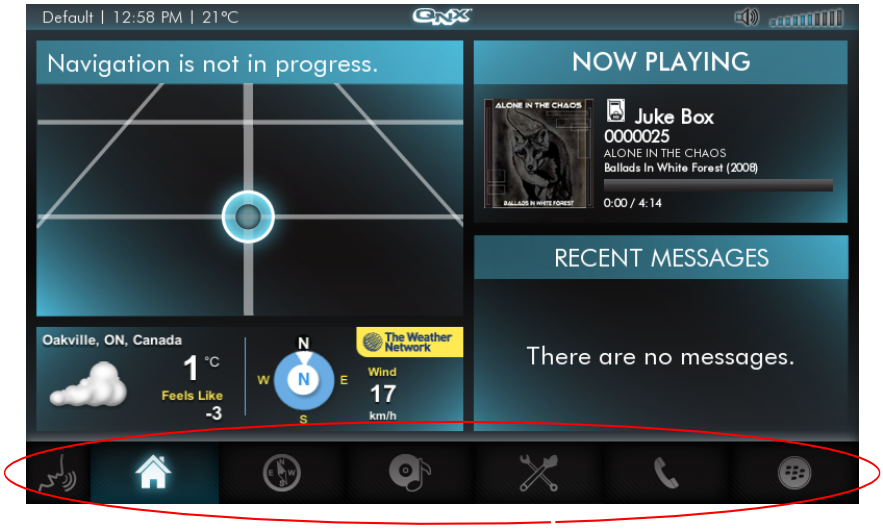

Figure 1: Home screen with Navigator taskbar

For more information about HTML5 and JavaScript support and the WebWorks extensions in the QNX CAR platform, see *HTML5 and JavaScript Framework*.

Some of the WebWorks extensions that Navigator depends on are specific to Navigator. These extensions provide a public API for Navigator (and other system components) that includes functions, for example, to pause and resume an app. The Navigator WebWorks extensions are defined in:

path/html5/webworks/tools/BB10webworks-1.0.2.9/Framework/ext/navigator

The Navigator implementation is defined in:

path/html5/webworks/apps/Navigator/js

Prerequisite knowledge

To understand the HTML5 Navigator implementation, you must understand the JavaScript programming language and the WebWorks Software Development Kit (SDK).

Advanced knowledge of JavaScript is required to understand the reference implementation of Navigator and to develop new apps and services for the QNX CAR platform. For more information about JavaScript, see *http://www.w3schools.com/js/default.asp*.

WebWorks plays several roles in the platform software by providing:

- a packaging facility for HTML5 apps
- WebViews (described below)
- a mechanism for separating a public JavaScript API from a private implementation

A *WebView* is a view that is rendered by the web engine for displaying an app. Each HTML5 app has its own WebView. Trusted apps are run “in process”, sharing a web engine instance. Other apps are run “out of process”, protected from each other by process boundaries, each with their own web engine instance.

Window management

The Navigator uses WebWorks to manage WebViews and jScreen to manage windows.

Managing WebViews

One of the roles of the Navigator is to manage WebViews. These WebViews are set up by web applications that are run on the QNX CAR platform.

Currently, Cordova and WebWorks applications are supported. An application developer supplies the application JavaScript code and any required plugins or extensions. Cordova and WebWorks web applications are similar in structure and in how their layers interact with user interface components.

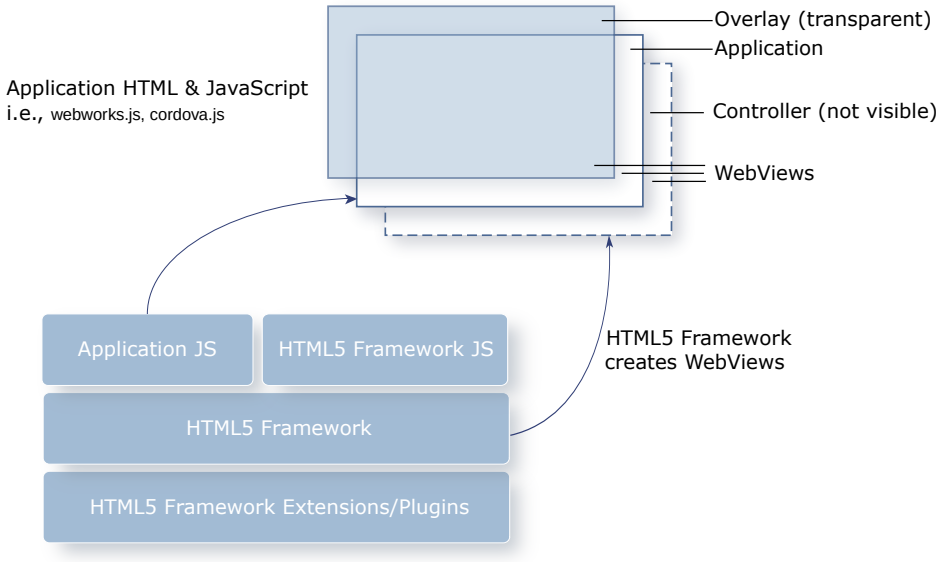

Figure 2: Web application structure and user interface components

The controller mechanism runs concurrently in an invisible WebView. The Controller, the first app that is launched, has access to the WebLauncher API. This allows the Controller to create other WebViews.

The app that is currently running is displayed in a visible WebView. A WebView can be visible, invisible or transparent. A transparent WebView is always available to allow notifications such as dialogs to appear over top of the application that is currently running.

The visual layout and the logical layers of the user interface are shown in the figure below. The status bar along the top and the taskbar along the bottom are always visible. Several WebViews can be active simultaneously, but only one WebView can be visible at a time.

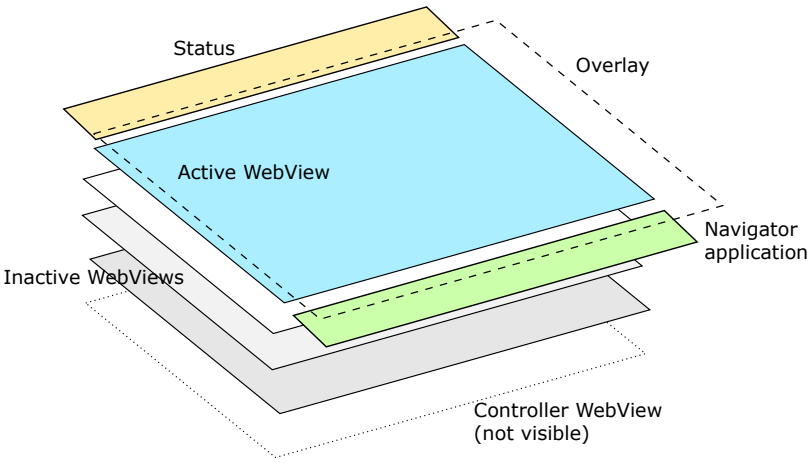

Figure 3: HTML5 Navigator layout

Managing Windows

The Navigator is also responsible for the placement and appearance of application windows. The Navigator uses the jScreen extension to communicate with Screen. The Navigator has the role of the Screen window manager and therefore, has the ability to manage their properties.

The Navigator deals with the application windows' z-order, transparency, positioning on the physical display, and scaling through Screen API functions. For more information on Screen, see *Screen Graphics Subsystem Developer's Guide*.

Once started, applications communicate directly with Screen from within their own context. Applications manage their own windows through the Screen API.

Application management

Managing the application life cycle (starting and stopping applications) is achieved through the launcher service by way of the Persistent Publish/Subscribe (PPS) service.

The HTML5 Navigator uses the JPPS extension to access PPS.

The figure below shows how an application is launched in the HTML5 Navigator:

1. A touch or ASR event triggers the Navigator to launch an application.
2. The Navigator communicates through its JPPS API to access PPS..
3. The Launcher reads the PPS object and begins launch procedures.
4. The Launcher asks the authorization manager to check permissions to launch the application.
5. When it receives authorization, the Launcher completes the application launch.

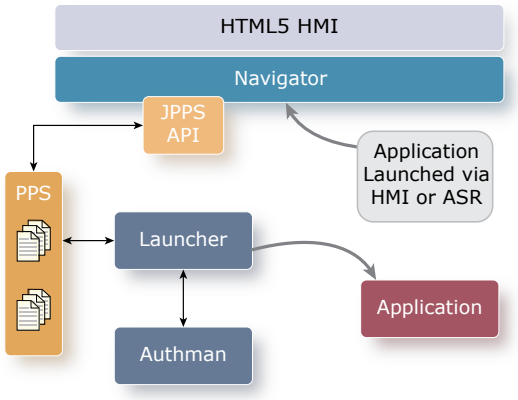

Figure 4: Step-by-step view of how HTML5 Navigator launches an application

Stopping an application follows a very similar flow to launching an application. The same libraries and the Launcher is used to terminate the application. The stopping of an application can be triggered by a touch or ASR event, or by the application itself.

Reference implementation

The reference implementation of the HTML5 Navigator consists of seven JavaScript files.

These files are located in this target image directory:

`/apps/Navigator.devTest_Navigator__a4514a37/native/js/`

The following table gives the key responsibilities of each file:

Table 1: Navigator implementation files

File	Key Responsibilities
`Applications.js`	`Navigator.Applications` provides functions that operate on apps, including *start*, *stop*, *show*, and *hide*.
`EventDispatcher.js`	`Navigator.EventDispatcher` provides functions for objects to register listeners for events. This class is a base class for `Navigator.Applications`, `Navigator.Status`, `Navigator.Tabs`, and `Navigator.Voice`.
`HNM.js`	`Navigator.HNM` registers PPS listeners for: • `navigatorhnmstatus` • `navigatorhnmnotification`
`index.js`	Defines properties for an app's WebView and layout. Registers listeners for the following events (defined in `Navigator.Applications`): • `E_APP_STARTED` • `E_APP_STOPPED` • `E_APP_GROUP_COMPLETE` • `E_APP_ERROR` • `E_APP_COVER_CHANGE` Registers PPS listeners for:

File	Key Responsibilities
	• `navigatorstartrequest` • `navigatorstoprequest`
`Navigator.js`	Sets up infrastructure to permit inheritance from JavaScript objects.
`Tabs.js`	`Navigator.Tabs` provides functions that operate on tabs, including *create*, *remove*, and *highlight*.
`Voice.js`	`Navigator.Voice` provides functions for updating the HMI based on the activity of the voice subsystem, including *select*, *cancel*, and *addItem*.

Chapter 3
Qt5 HMI

The Qt5 HMI is based on version 5.2 of the Qt framework and uses the QPPS library to access PPS and talks directly to Screen through its C API.

The application and window management display area of the Qt5 HMI includes seven tabs. Each tab is associated to a different view of the same process. The core view is the display area with the status bar and taskbar.

Here are the tabs on the application and window management display area:

- Push-to-Talk—activates Automatic Speech Recognition (ASR)
- Home—shows a consolidated view of several key apps
- Navigation—allows the user to interact with the currently active navigation engine
- Media Player—allows the user to play videos or music, or to listen to the radio
- Car Control—provides access to vehicle settings for audio, climate control, etc.
- Communications—provides access to email, text messages, contacts, and a telephone dial pad
- Apps Section—shows the complete set of available apps

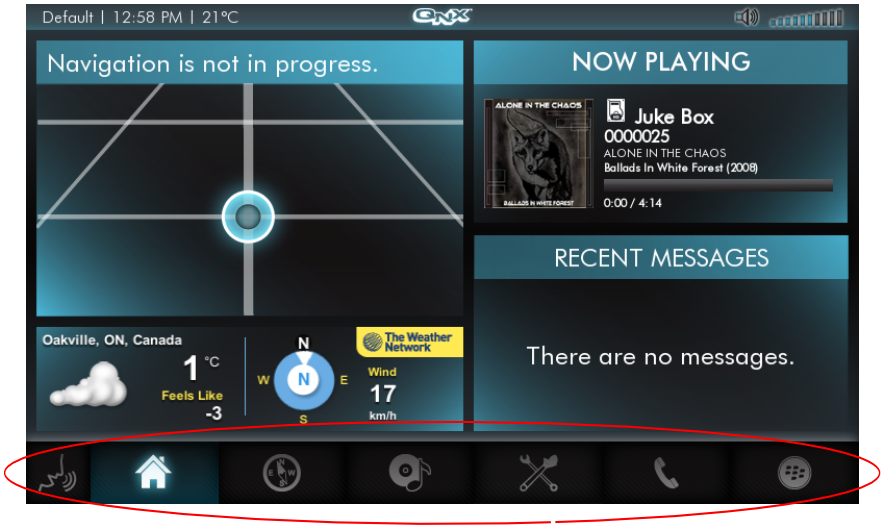

Figure 5: Home screen (Core view) with application and window management taskbar

Prerequisite knowledge

To understand the Qt5 application and window management, you must be proficient with C++ and understand the design and concept of the Qt framework, which provides a toolset for writing UI-based native applications on embedded platforms. For more information about Qt, see the *Digia Qt Project* website.

If you want to customize the Qt5 application and window management, or write a replacement using the same technologies, you must know how to use the Qt development tools, particularly *Qt Creator*.

Window management

To manage windows, the Qt5 Window manager processes Screen events sent by applications and uses the data in these events when communicating with Screen through its API.

The Qt5 HMI consists of several different layers. These layers are essentially separate Screen windows running in the same process. The layers of the Qt5 HMI are shown in the figure below:

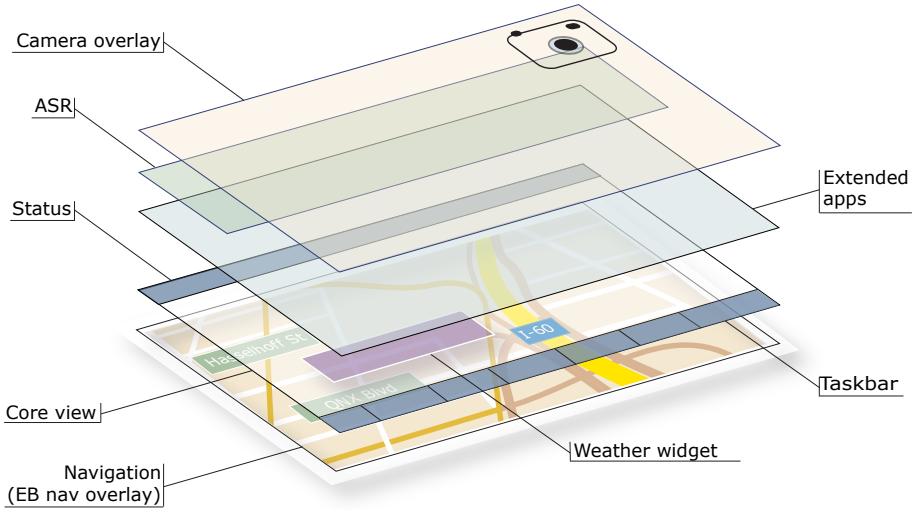

Figure 6: Qt5 HMI layers

In the Core view, the tabs trigger different views that render to the same window. While executing, extended apps are unaware of the activity of other extended apps and the Core view. Each extended app or Core view continues to render its own HMI component, whether or not it's currently displayed. It's up to the Qt5 Window manager to decide how, when, and where to display the windows of the extended apps. At any time, either one extended app or the Core view can be visible.

When active, the video feed from the rearview camera always becomes the top HMI layer and thus, replaces the display of all other apps. When the user pushes the **Push-to-Talk** button in the taskbar and starts an ASR session, the visual prompt for a voice command is always displayed over the other apps (except the rearview camera feed if it's active).

Next, any active extended app is displayed in the HMI midsection, which is the area between the status bar along the top and the taskbar along the bottom. Although these last two HMI components are rendered one layer lower, at the same level as the core apps (e.g., Home, Media Player, Car Control), the status bar and taskbar are always visible (except when the rearview camera feed is active). If no extended app is running, the HMI midsection shows the active core view.

Finally, when navigation is active, the animation of the map to reflect the driver's turn-by-turn progress as they approach their destination is rendered on the bottom HMI layer. This way, the map display is kept up to date on the navigation information feed in the **Home** screen or whenever the user presses the **Navigation** button and accesses the **Navigation** screen.

The Qt5 Window manager is responsible for the placement and appearance of application windows. The Qt5 Window manager uses the Screen API to communicate with Screen. The Qt5 Window manager has the role of the Screen window manager and therefore, has the ability to manage their properties.

The Qt5 Window manager deals with the application windows' z-order, transparency, positioning on the physical display, and scaling through Screen API functions. For more information on Screen, see *Screen Graphics Subsystem Developer's Guide*.

Once started, applications communicate directly with Screen from within their own context. Applications manage their own windows through the Screen API.

Application management

Managing the application life cycle (starting and stopping applications) in Qt5 HMI is achieved through the Launcher API (QtQnxCar2 library) and the QPPS library.

The Qt5 Application manager uses the QPPS library to access PPS.

The figure below shows how an application is launched in the Qt5 Application manager:

1. A touch or ASR event triggers the Qt5 Application manager to launch an application.
2. The Qt5 Application manager communicates through the Launcher API (QtQnxCar2 library).
3. The Launcher API uses the QPPS library to access PPS..
4. The Launcher reads the PPS object and begins launch procedures.
5. The Launcher asks the authorization manager to check permissions to launch the application.
6. When it receives authorization, the Launcher completes the application launch.

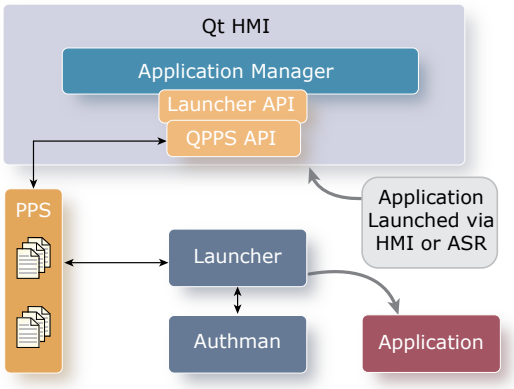

Figure 7: Step-by-step view of how Qt5 application manager launches an application

Stopping an application follows a very similar flow to launching an application. The same libraries and the Launcher is used to terminate the application. The stopping of an application can be triggered by a touch or ASR event, or by the application itself.

Chapter 4
Creating an Application Window Manager: Requirements

You can replace the application and window management with a custom implementation, provided that it supports the current external interfaces.

Your replacement implementation must:

- publish to the same PPS objects
- subscribe to the same PPS objects and handle state changes appropriately
- register with the platform for the same events

Application window manager interfaces

Application window manager publishes to the following PPS objects:

- `/pps/system/navigator/appdata`
- `/pps/system/navigator/command`
- `/pps/services/launcher/control`

Application window manager subscribes to the following PPS objects:

- `/pps/system/navigator/applications/applications`
- `/pps/services/app-launcher`
- `/pps/services/launcher/control`

Application window manager listens for the following system events:

- `navigatorappstarted`
- `navigatorappstopped`
- `navigatorapperror`
- `navigatorstartrequest`
- `navigatorstoprequest`
- `voicestate`
- `voiceresult`

Index