HTML5-Hardware Communication with PPS Messaging

Andy Gryc, Senior Product Marketing Manager, Automotive
Kerry Johnson, Senior Product Marketing Manager, Automotive
QNX Software Systems Limited
agryc@qnx.com, kjohnson@qnx.com

Introduction

Human-Machine Interfaces (HMIs) developed with HTML5 reside in a high-level, virtualized environment, and they work well in this environment. This fact does not preclude their needing to access hardware, however. In mobile devices, for example, they need to retrieve the device orientation and, if there are GPS or accelerometer chips, information these chips provide for applications that use geo-location. In-vehicle systems need to retrieve even more information from low-level components such as the CAN bus, GPIO pins, and I^2C and SPI devices.

Writing specific interfaces to communicate between the HMI and each low-level service is a costly—and likely unsustainable—proposal. A better approach is to use an HMI-agnostic, asynchronous messaging model such as Persistent Publish/Subscribe (PPS). A service for pushing out changes and receiving notifications, PPS provides a simple and effective way for the HMI to communicate with low-level components and the vehicle hardware.

Drawing on our experience building in-vehicle systems, we will describe how a PPS messaging model facilitates communication between an HTML5 HMI and low-level components.

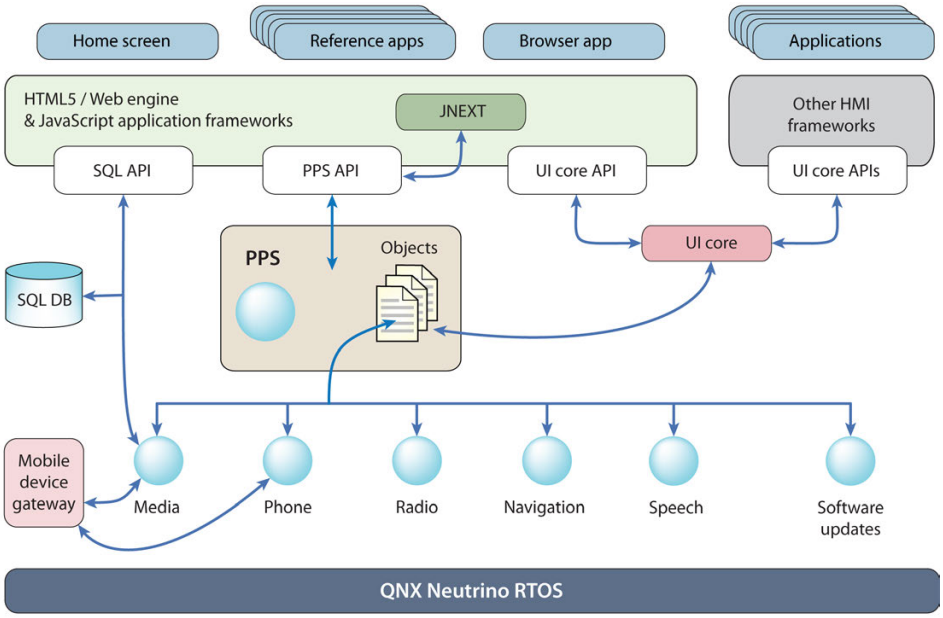

Figure 1. A high-level view of the QNX CAR 2 platform architecture showing how PPS is used to communicate between the HMI and lower-level components

The problem

A problem facing developers of almost all but the simplest systems is the increasing diversity and complexity of components at all levels. On the one hand, an automotive infotainment system integrate many devices and services—from multimedia players to virtual mechanics; on the other, pressures such as release schedules, budgets and re-usability needs have made HMI development in native C/C++ code prohibitively expensive and too time-consuming. HTML5 and its ancillary technologies (CSS3, JavaScript, AJAX, JSON, etc.) offer an excellent, non-proprietary solution for building rich, device-agnostic HMIs. Unfortunately, like other high-level HMI technologies, HTML5 doesn't offer a simple solution for communicating between the HMI layer and the many low-level components found in today's systems.

With HTML5 becoming the HMI technology of choice, the problem facing system architects, then, is finding or devising a light-weight messaging model that bridges the gap between the HTML5 layer and disparate low-level components. Further, since a) many systems must be able to expand to include new devices and technologies, and b) many HMIs must accommodate applications using different HMI technologies, such as Elektrobit GUIDE and Qt, this messaging model must be open. That is, it must be able to integrate these new components and technologies, easily and efficiently.

Persistent Publish/Subsribe

To understand how PPS can simplify the design of embedded applications that must support a wide range of devices and software components, as well as communicate with a sophisticated HMI, we need to look at some of the details of how PPS works.

An Object-based System

The QNX implementation of PPS is an object-based service with publishers and subscribers in a loosely coupled messaging architecture. Any PPS client can be a publisher only, a subscriber only, or both a publisher and a subscriber, as required by the implementation.

PPS objects are integrated into the file system pathname space, and publishing is asynchronous. Publishers modify objects and their attributes, and write them to the filesystem. When any publisher changes an object, the PPS service informs clients subscribed to that object of the change. PPS clients can subscribe to multiple objects, and PPS objects can have multiple publishers as well as multiple subscribers. Thus, publishers with access to data that applies to

Binary or human-readable objects?

A PPS service can be designed to use either binary or human-readable objects.

We chose to use human-readable objects and attributes for PPS, considering that the benefits to development and debugging outweigh the cost of the larger objects.

Human-readable objects allow debugging from the command-line using simple filesystem utilities, such as `cat` for subscribe and `echo` for publish. For example:

```
cat /pps/media/PlayCurrent
cat /pps/media/.all?wait
```

or:

```
echo
"attr::value">>/pps/objectfilename
```

Similarly, debugging information, including PPS object and attributes, can be retrieved by a simple program that subscribes to an object and prints.

different object attributes can use one object to communicate their information to all that object's subscribers.

PPS clients must know which PPS objects are of interest. If they are publishers, they must know what to publish and when; if they are subscribers, they must know to which objects they must subscribe, and which object attributes interest them. PPS clients do not have to manage errors, or buffers beyond what they need for *open()*, *read()* and *write()* POSIX API calls, confirming that they can make sense of what they read, and determining if they want their reads to be blocking or non-blocking.

Since PPS leverages the services of standard POSIX file systems, it can work with any programming language or application environment. A component written in one language can communicate with components written in any other language. No special knowledge of the other components is required.

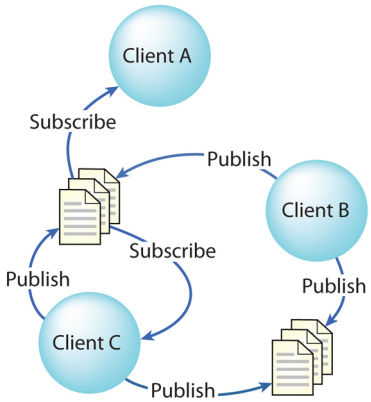

Figure 2. PPS clients and objects. Client A is only a subscriber; client B is only a publisher, and Client C is both a publisher and a subscriber.

Persistence

A Persistent Publish/Subscribe service can maintain data across reboots. Persistence is a characteristic determined by the system designer, and is set for individual attributes.

When PPS is running it maintains its objects in memory, but saves persistent objects to appropriate storage, either on demand or at shutdown. It restores these objects on startup, either immediately, or on first access. Of course, the underlying persistent storage depends on a reliable file system and on storage media, such as a hard disk, NAND or NOR flash.

As well as ensuring data persistence across reboots, PPS can simplify startups. With many other messaging models, if a client comes up after the server, it must request up-to-date data from the server, in case something changed between the times the server and the client started up. This is also true if a client loses contact with a server, and it is true for each and every client on the system. With PPS, however, the service restores its persistent objects on startup and maintains them as they change. No matter when a client starts or reconnects it needs only to read these objects to acquire current data.

System Scalability

With PPS, publisher and subscriber do not know each other; their only connection is an object that has a meaning and purpose for both of them. This messaging model gives developers great flexibility when designing a system: they can, if necessary, delay decisions on module connection points and data flow until runtime. Because such decisions are neither hardcoded nor directly linked, they can be adapted as situations or requirements evolve; they can even change dynamically as the system runs.

The loosely-coupled PPS messaging model also simplifies the integration of new software components. Since publisher and subscriber do not have to know each other, developers adding components need only to determine what these new

components should publish, and what data they need other PPS clients to publish. No fine-tuning of APIs is required, and system complexity does not increase as components are added.

QNX CAR 2 application platform

The QNX CAR 2 application platform offers an ideal use-case for the Persistent Publish/Subscribe messaging model. From the user's perspective, key capabilities of this platform include:

- HMI: HTML5-based HMI designed to support easy branding, reskinning and personalization
- Information and entertainment: multimedia (audio and video); AM, FM and HD radio; streaming Pandora and TuneIn internet radio; Weather Network integrated weather reporting; Apple and DLNA support for phone- and home-based media
- Automotive interfaces: climate control and virtual mechanic
- Navigation, handsfree, speech, social networking

As can be seen from the above list, and not surprisingly for an automotive infotainment platform, the QNX CAR 2 platform supports myriad applications and a wide range of low-level software components directly connected to hardware devices. It thus requires a simple and scalable messaging model to handle communications between many disparate components.

JavaScript PPS wrapper sample

The JNEXT code sample shows some basic interactions between the HMI and the PPS service. They are adapted from the QNX CAR 2 application platform.

```
// initialize the object
var PPS_COMMANDS = "/pps/services/bluetooth/control";
var commandPPS;
commandPPS = new JNEXT.PPS();
commandPPS.init();

// assign the event handler to listen for changes

commandPPS.onChange = function(e) {
       // code to execute on change event
}

// open object for read/write

commandPPS.open(PPS_COMMANDS, JNEXT.PPS_RDWR);

//read object and retrieve attributes
commandPPS.read();
var myPPSData = commandPPS.ppsObj;  //gets PPS object in
key:value format

//write JSON style object, which consist of key:value pairs to
PPS
commandPPS.write( {
       msg:"key",
       dat:”Message One”
} );
```

The HMI for the QNX CAR 2 platform uses HTML5 with a JavaScript framework that includes the Sencha and jQuery JavaScript libraries. The system architecture is designed to support easy integration of other HMI technologies such as Adobe AIR and Elektrobit GUIDE. The HTML5 and Cascading Style Sheets (specifically, CSS3) facilitate migration of applications to and from the in-vehicle system and mobile devices (smartphones and tablets).

The PPS messaging handles communications between most system components and the HMI. Since PPS messaging is technology- and language-agnostic, only a small number of APIs is needed to provide the interfaces between the HTML5 HMI and the underlying components. Specifically:

- a PPS API handles communications between the HMI and the PPS service
- an SQL API interfaces with local media databases

UI core APIs handle communications between the HTML5 layer and a user-interface core component, and between this component and other HMI technologies, such as Adobe AIR and Elektrobit GUIDE

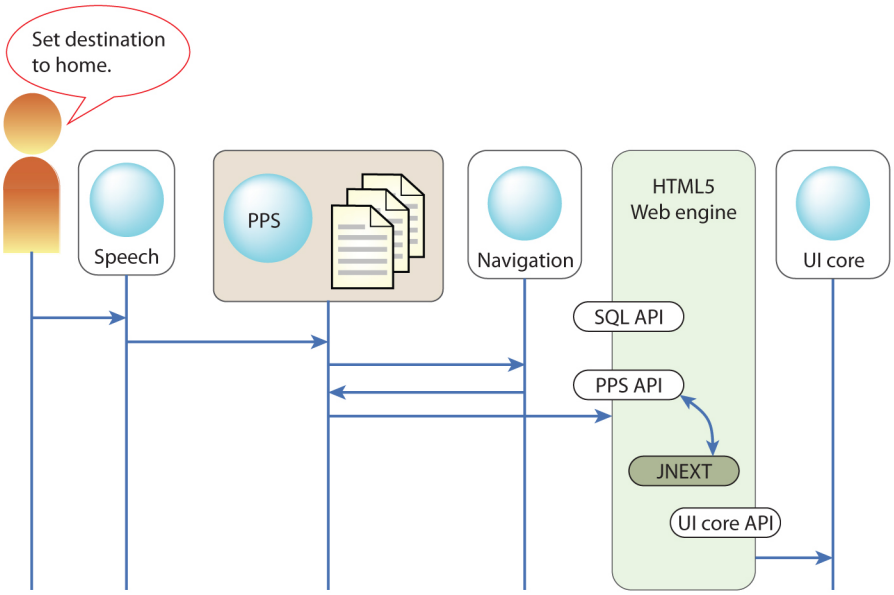

Figure 3. A use case showing communication between different QNX CAR 2 components: a) The driver speaks a command. b) The speech service processes the statement and publishes PPS objects. c) The navigation service, having subscribed for navigation-related PPS objects, receives an update, processes the requests and publishes PPS objects. d) The HTML5 / Web Engine is updated; it then renders the required user interface elements, and uses the UI core to display the result.

Communication between the HTML5 HMI and the hardware is handled with JavaScript wrapper classes and JNEXT. C/C++ programs interface directly with the vehicle hardware, and read and write the relevant PPS objects. To access this information from the HMI, JavaScript in the HTML5 HMI can call a JavaScript wrapper class to communicate with a JavaScript PPS class. The wrapper class exposes a natural class-based JavaScript API to other callers. Internally, the wrapper makes calls to an instance of a PPS class. The PPS class uses JNEXT (or some other mechanism) to allow the JavaScript to call into the native code and read and write relevant PPS objects.

As the diagram below shows, the overall architecture for the QNX CAR 2 application platform is both simple and flexible. Because the PPS messaging model is loosely coupled the system architecture is very flexible. If a new component or device is added, very little work is required; the new component must publish relevant data, and subscribe to the relevant PPS objects, while existing components must do the same for this new component if they need to communicate with it. Similarly, even changing the HMI technology would not cause great disruption in the underlying layers. All that would be required would be a change to the relevant API.

Conclusion

HTML5 is fast becoming not just a popular HMI technology, but the preferred environment for delivering rich, flexible user interfaces. Our experience with the QNX CAR 2 application platform has shown us that HTML is no longer just the standard for presenting web content, but a viable technology for HMIs for all sorts of applications—connected and not connected, using browser-based or HTML5 engine-only environments. We have also seen how a PPS messaging model facilitates communication between the many QNX CAR platform components, technologies and devices. Equally important, PPS enables a flexible architecture that requires relatively little work to integrate new devices and technologies.

About QNX Software Systems

QNX Software Systems is the leading global provider of innovative embedded technologies, including middleware, development tools, and operating systems. The component-based architectures of the QNX® Neutrino® RTOS, QNX Momentics® Tool Suite, and QNX Aviage® middleware family together provide the industry's most reliable and scalable framework for building high-performance embedded systems. Global leaders such as Cisco, Daimler, General Electric, Lockheed Martin, and Siemens depend on QNX technology for vehicle telematics and infotainment systems, industrial robotics, network routers, medical instruments, security and defense systems, and other mission- or life-critical applications. The company is headquartered in Ottawa, Canada, and distributes products in over 100 countries worldwide.

www.qnx.com