Real-Time Embedded User Interfaces

Justin Ireland
August 2010

Introduction

Modern appliances and electronic devices are becoming increasingly sophisticated. Advanced feature sets require advanced configuration options. Multi level menu systems offer more granular control to the users but at the cost of usability. Unfortunately, most user interfaces are not advanced enough to solve the usability issues and therefore the interface actually becomes the weak link.

Physical interfaces such as button panels or remote controls have many disadvantages. Panel controls require a dedicated display for feedback and physical access to a fixed location. IR remote controls offer only one-way communication with limited feedback and can become difficult to manage for multiple devices.

Many different 3rd party control systems are available to develop fully customizable user interfaces for controlling multiple types of devices. These control systems are essentially software development platforms which provide a very high level of customization and development options. However, the highly technical nature of these platforms makes it cost prohibitive to implement in many situations.

We propose a new type of solution to address these issues and bring simplified management and control to a broader market through embedded virtual interfaces.

Design Concept

An embedded virtual interface is a Graphical User Interface designed for a specific appliance, served from an embedded system controller connected directly to the appliance. See the illustration below.
The embedded system controller maintains a proprietary link with the appliance at all times and presents a virtual interface to multiple clients using HTTP. The GUI is built with standard web technologies designed to display within any compatible browser on any computing device. Building and embedding the GUI in this way has two distinct advantages:

- **Universal Access** - By using web standards (HTML/CSS/javascript) and ensuring a high level of cross-browser compatibility we allow universal access from any device or operating system, regardless of manufacturer. The software does not have to be precompiled for a specific operating system. This allows us to maximize accessibility with minimal development costs.
- **Ease of Integration** - Since the GUI is included with the appliance there is no need for customized development that would be necessary with a 3rd party control system.

**Potential Issues**

Though this concept solves some problems it does present a few technical hurdles that must be overcome.

**How to implement real-time communication between the appliance and embedded system controller?**

As a manufacturer we have complete control over how our appliance communicates with external systems which makes this a less trivial issue. We are able to broadcast events as they occur. For example, when a user performs an action that changes the state of an appliance, that state change is automatically broadcast out to external devices that may be listening. The embedded system controller is serially connected to the appliance and listening for these types of changes. When a response is heard it immediately processes that response to understand what changes have occurred.

A key consideration for this method is that some responses may not be initiated from the GUI but instead from the physical interface. The appliance still needs to broadcast the event regardless of where the change was
initiated from and the embedded system controller needs to be able to receive and process that response independent of its own requests. This may require a unique communication protocol between appliance and embedded system controller where various appliance states can be deciphered from a simple message.

Real-time communication becomes a bit more complicated if you do not have control over the appliance behavior. For example, we have access to the appliance firmware so we can ensure all events are broadcast as they occur. But not every appliance is designed to communicate this way. Some appliances will only respond when queried. Therefore, in order for the embedded system controller to maintain a near real-time state model of the appliance it must constantly ask for updates of every possible state. This poses some challenges that could make it unstable to implement.

**How to implement real-time communication between the embedded system controller and web clients?**

Web pages typically rely on user interaction to post and retrieve data. However, a real-time interface cannot rely on user interaction for updates. Additionally, if multiple users can access the same appliance simultaneously then it is important that the information displayed in their interface is always up to date. Otherwise users may unknowingly act on “expired” information.

For example, say we have two users attempting to control an appliance simultaneously. User A is accessing an embedded GUI through a browser from a remote location. User B has physical access to the appliance through a keypad. User B pushes a button that changes the state of the appliance. But User A may not be aware of this change until his GUI refreshes to reflect this new state change. If the GUI does not refresh before User A interacts then problems may occur as User A is acting on “expired” information.

To avoid this we need to make sure the browser client is updated to reflect changes in real-time.

Traditional web interfaces rely on a full page load to update the contents of a page. However, this is highly inefficient as it requests all page content from the server to be resubmitted regardless of whether or not any changes have occurred. This wastes bandwidth and CPU cycles. Handling multiple clients this way could easily overload the limited resources of the embedded system controller. **AJAX** was developed for this purpose. AJAX allows individual elements to be updated without having to reload an entire page. However, AJAX alone doesn’t solve the problem of getting real-time updates.
In the last few years COMET has become a popular method to handle real-time bidirectional communication between an HTTP server and multiple clients. However, COMET has never been standardized which makes it difficult to implement and work with different browsers. Some of the more popular COMET methods involve long-polling where the client opens an HTTP request and the server keeps it open until an event occurs. The server responds and closes the connection requiring the client to repeat the process. Though this method works, it does have some limitations and scalability issues. Websockets were designed specifically to address the issues with COMET. The basic premise of websockets is that it creates a dedicated TCP socket connection between the server and a client. Updates are sent across the socket connection only when they occur so you can achieve real-time communication with the least amount of resources necessary. Websockets were originally introduced as part of the HTML5 specification but have since been separated into its own specification. As of this writing, Websockets is still being specified by the W3C and the IETF.

**Proof Of Concept**

With a theoretical concept we set out to develop a proof of concept. For this we decided to create a very simple GUI that could be accessed through a browser to control an appliance.

**Appliance**

For the appliance we chose our SW-HDMI-4. This is a very basic 4 port HDMI switch with an RS232 interface. This interface allows us to send commands and receive updates over RS232.
The protocol is very basic and provides an easily parsable format for machine interpretation. The commands used are:

<table>
<thead>
<tr>
<th>Command</th>
<th>Syntax</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PW&lt;n&lt;CR&gt;</td>
<td>n=0 for Power OFF</td>
<td>Turn appliance on/off</td>
</tr>
<tr>
<td></td>
<td>n=1 for Power ON</td>
<td></td>
</tr>
<tr>
<td>PW&lt;CR&gt;</td>
<td>Response is PW&lt;n&lt;CR&gt; as described above</td>
<td>Query power status</td>
</tr>
<tr>
<td>CO&lt;n,m&lt;CR&gt;</td>
<td>n=1 for connection output 1. The SW-HDMI-4 only has a single output so this value will always be 1. m=1-4 for the input.</td>
<td>Select the source input to be routed to the output</td>
</tr>
<tr>
<td>CO&lt;CR&gt;</td>
<td>Response is CO&lt;n,m&lt;CR&gt; as described above</td>
<td>Query which display is connected to the specified output</td>
</tr>
<tr>
<td></td>
<td>n=1 for connection output 1. m=1-4 for the input</td>
<td></td>
</tr>
<tr>
<td>BO&lt;n&lt;CR&gt;</td>
<td>n=0 for Boost OFF</td>
<td>Turn signal Boost on/off</td>
</tr>
<tr>
<td></td>
<td>n=1 for Boost ON</td>
<td></td>
</tr>
<tr>
<td>BO&lt;CR&gt;</td>
<td>Response is BO&lt;n&lt;CR&gt; as described above</td>
<td>Query boost status</td>
</tr>
</tbody>
</table>

This SW-HDMI-4 is configured to broadcast all events over RS232. So, for example, if a user pushes the input button on the front panel to change the input source to #3 then the appliance would broadcast “CO1,3” over the serial connection.

**Embedded System Controller**

For the embedded system controller we decided to use a [Lantronix Matchport AR](#). The Matchport features several GPIO lines and an HTTP server running on a custom embedded linux kernel. To get the Matchport fully operational we used the development board connected to the SW-HDMI-4 via RS232 and ethernet side to the LAN.
Software
There were several processes we had to develop for the Matchport to get a functioning prototype.

First we had to establish a serial link between the Matchport and the SW-HDMI-4. This link would be setup automatically on boot up and would stay open as long as both devices were on. Upon initialization the Matchport would query the switch for its current configuration which it would use to create a state model.

The matchport would then begin listening for responses from the SW-HDMI-4. Responses are received and processed and the internal state model is updated to reflect the changes. If any changes occur it would trigger an update to any connected HTTP clients. The communication between server and clients uses a custom websocket connection. Messages are passed back and forth between server and client using JSON.
For the interface we developed a simple point and click GUI as pictured below.

![Image of the GUI](image)

**Model:** SW-HDMI-4  
**Description:** HDMI 4-Port Switch  
**Connection:** RS-232

The Power and Boost buttons can be clicked to toggle the setting on/off and clicking any of the green circles selects that numbered input. The light green circle indicates the active source input.

**Demonstration**  
A video demonstration of the working prototype can be seen here:

[http://www.youtube.com/watch?v=5ZuJIrkg3E4](http://www.youtube.com/watch?v=5ZuJIrkg3E4)

Notice in the video that there is (nearly) zero latency between synchronization of the GUI and the switch either from actions occurring at the switch or the GUI.

**Conclusion**  
Our working concept demonstrates the ability to provide a real-time web enabled virtual interface for non web enabled appliances. Using AJAX and the latest HTML5 technologies (websockets) we can achieve push notification with efficient resource utilization and near zero latency. As of this writing, websockets is still being developed and is only available in Firefox, Chrome and Safari beta builds but we expect to see it rolled out to stable releases soon, followed by other mobile versions.

With a bit more work we can adapt this system into a modular framework allowing rapid development of advanced virtual user interfaces for other appliances.