ALLSEEN
ALLIANCE

Programming IoT Applications Using AllJoyn.js

Brian Spencer
Engineer, Staff/Manager

Qualcomm Connected Experiences, Inc.

October, 2014

Agenda

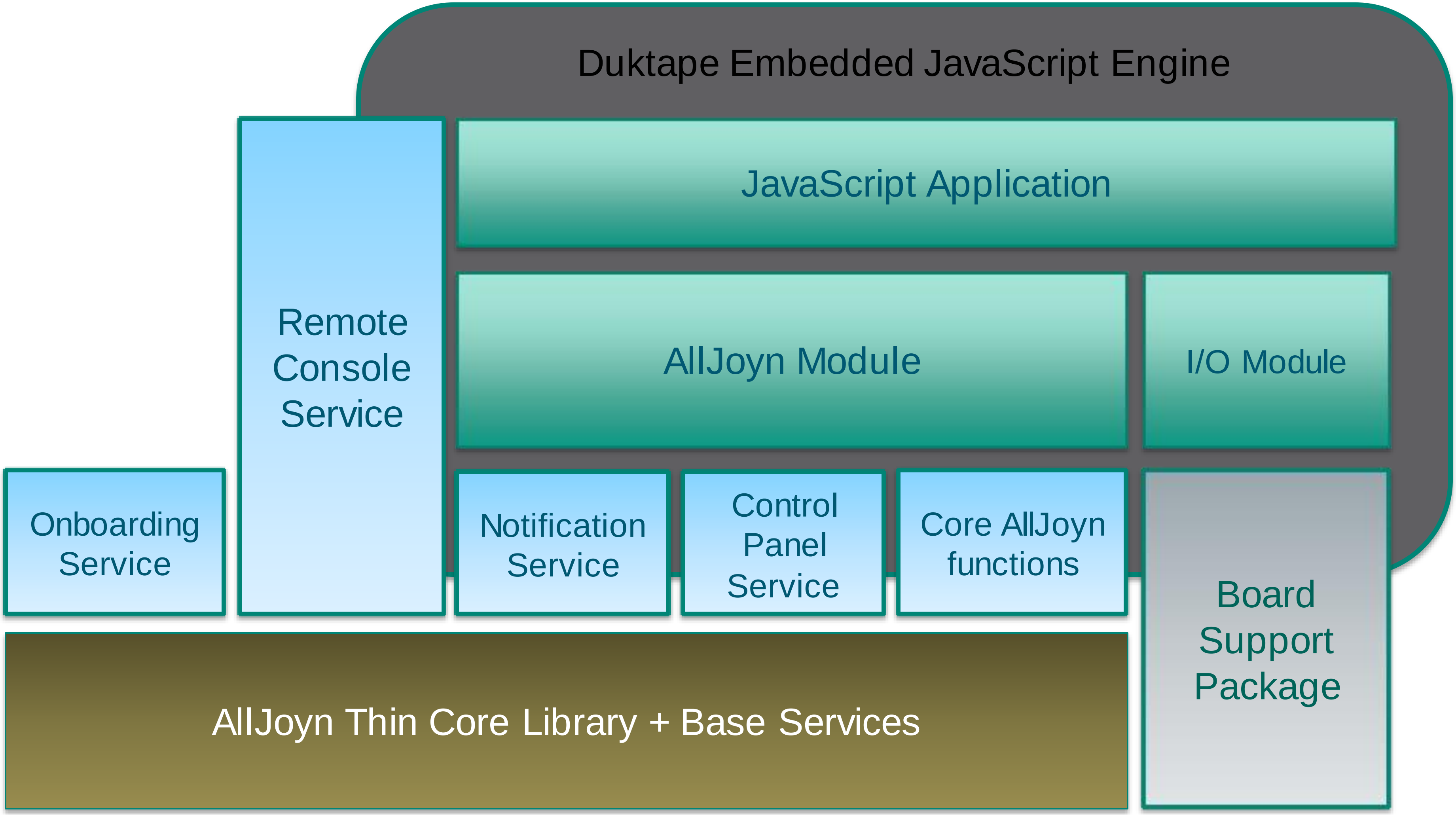
1. Overview
2. The AllJoyn Module
3. The Input/Out Module
4. Simulated I/O
5. Base Services
6. The AllJoyn.js console
7. Code samples

Overview and Architecture

What is AllJoyn.js?

- AllJoyn.js combines the AllJoyn Thin Core Library (AJTCL) and base services with “duktape” an open source small-footprint ECMAScript 5.0-compliant runtime engine.
  - For more information on “duktape” see www.duktape.org
- A set of JavaScript APIs provide an easy to use abstraction layer over the AllJoyn core, base services, and the device I/O peripherals.
- The combined implementation is targeted at microcontrollers having a minimum of 128K RAM (preferably 256K for “real applications”) and 500K Flash.
- Also designed to run on Linux, Windows, and OS X
- Includes a “console” service for installing scripts and debugging application code.

Alljoyn.js Architecture



Console Service

- An AllJoyn service that runs alongside the JavaScript app
  - Functionality is exposed as an AllJoyn interface
  - AllJoyn.js source tree includes a command line console client
- Provides remote access to running JavaScript application
  - OTA flashing of new JavaScript applications
  - Execute JavaScript code on target in real-time
  - Logging of output from JavaScript `print()` and `alert()` functions
  - Displays notifications from running JavaScript program

Programming model

- AllJoyn.js is 100% event driven.
  - No blocking calls
  - Write operations that cannot be buffered may introduce delays
- Functions registered with the AllJoyn object (AJ) are called when various AllJoyn events happen:
  - AllJoyn bus attachment events:

    `onAttach onDetach`
  - AllJoyn messages:

    `onSignal onMethodCall onPropSet onPropGet onPropGetAll`
- Functions can be registered with one-shot and interval timers

  `setTimeout clearTimeout setInterval resetInterval clearInterval`
- Functions can be registered to be called on input and output triggers
  - `setTrigger`

Debug output

- Duktape has two built-in functions for logging output to a debug console

  ```
  print()
  alert()
  ```

- In AllJoyn.js, the functionality of the above methods are identical except the output string is prefixed with “PRINT” or “ALERT”

NOTE: When the console client is connected to the running JavaScript, output is redirected to and displayed in the console application.

AllJoyn Module

Provides access to core AllJoyn functionality

onAttach

- Registers a function to be called when AlllJoyn.js application becomes attached to an AllJoyn Router

```
AJ.onAttach = function() {
    alert(‘Attached to the AllJoyn bus’);
}
```

- Functions that require an AllJoyn connection can now be called:
  - Initiate service discovery
  - Add match rules for signals
  - Launch a control panel

onDetach

- Registers a function to be called when AlllJoyn.js application becomes detached from an AllJoyn Router

```
AJ.onDetach = function() {
    alert(‘Detached from the AllJoyn bus’);
}
```

- Allows the application to delete stale objects and cleanup

Interface and Object definitions

- A definition is required for the interfaces and objects used by an AllJoyn.js application.
- These definitions supply essential information required to send and receive signals, make and handle method calls, and access properties.

```
AJ.interfaceDefinition[‘test.InterfaceA'] = {
    mySignal:{ type:AJ.SIGNAL, args:[’s’] },
    myProperty:{ type:AJ.PROPERTY, signature:’u’ },
    myMethod:{ type:AJ.METHOD, args:[‘i’, ‘i’], returns:[’i’] }
};

AJ.interfaceDefinition[‘org.example.Interface2’] = {
     /* signals, methods, and properties */
};

AJ.objectDefinition['/myApp'] = {
    interfaces:[‘test.InterfaceA’,‘org.examples.Interface2’]
};
```

Interface members

- Signals entries describe AllJoyn signal messages
  - The ‘type” property is set to value AJ.SIGNAL.
  - Optional “args” property to define the type signatures of the values that get carried in the signal. Signals with no “args” property carry no data.
  - A “description” property used to support Events & Actions usage.
- Method entries an AllJoyn describe method call messages
  - The “type” property is set to AJ.METHOD
  - Optional “args” and “returns” property that define the type signatures of the input values and output values of the method call.
- Property entries describe AllJoyn properties.
  - The “type” property is set to AJ.PROPERTY
  - Required “signature” property to specify the type signature for the value
  - Optional “access” property specifies read (“r”), write (“w”), or read/write “rw” access.
    - If not present the default access is read/write.

Type signatures – basic types

- Type signatures are strings that describe how the AllJoyn framework sends arguments and property values over the network.
  - AllJoyn.js uses signatures in the interface definitions to automatically convert between the ECMAScript object types and the representations required on the network.
- Number mappings
  - The signed and unsigned integer type signatures (‘a’, ‘q’, ‘i’, ‘u’, ‘t’, etc.) all map to ECMAScript numbers. 64-bit integer values may lose precision in this translation.
  - The ‘d’ (double) signature maps exactly to an ECMAScript number.
- Strings
  - The string signature types (‘s’, ‘g’, ‘o’) all map to ECMAScript strings.
- Byte arrays (signature ‘ay’) map to the duktape buffer type.
  - A buffer type can be indexed like an array to access the individual bytes.

Type signatures – container types

- Array signature types
  - Any other signature type prefixed by an ‘a’ , e.g., ‘as’ is an array of strings, ‘au’ is an array of 32-bit unsigned integers.
  - Map to ECMAScript arrays. The elements are mapped according the rule applicable to the element signature type.
- Structure signature types
  - A sequence other signature types inside parentheses, e.g., ‘(ssi)’ is a structure comprising two strings and a 32-bit signed integer.
  - Map to ECMAScript arrays to preserve order. The structure elements are mapped according the rule applicable to each element signature type.
- Dictionary signature types
  - Like arrays of structures with curly-braces instead of parentheses, e.g., ‘a{s(ss)}’ is a dictionary where the keys are strings and elements are a pair of strings.
  - Dictionaries map cleanly to ECMAScript objects.

Type signatures – variants

- Variants are a powerful feature for specifying data types at runtime.
  - The network representation of a variant includes the type signature of the value.
  - Variants are specified with the signature type ‘v’ and can replace any other signature string in a signature type, e.g., ‘av’ is an array of variants, ‘(yyv’) is a structure with two bytes and a variant.
  - When receiving a variant value, the type signature is available in the AllJoyn message so AllJoyn.js has all the information needed to do the correct type mapping following the rules described earlier.
  - When sending a variant value, the signature must be specified in the application. This is done by wrapping the value in an object where the property name is the required signature.
- Variant signature syntax examples:

```
{ ‘s’:”Hello World” }
{ ‘(ddd)’:[ x, y, z ]  }
{ ‘ai’:[1, 2, 3, 4, 5, 6, 7, 8, 9] }
```

Service Discovery

- There are two ways to discover a service:
  - By interface which is the primary use case
  - By name which is mainly for app-to-app and legacy use cases.
- AJ.findService()
  - Takes two arguments: an interface name and the callback function to be called when the service has been found.
  - If the second argument is omitted, discovery of specified interface is canceled.
  - If the required service is discovered, the callback function is called with a service object that provides information about the discovered service.
- AJ.findServiceByName()
  - Take three arguments: the service name, a service description and the callback function to be called when the service is found.
  - If the second and third arguments are omitted, discovery of the specified name is canceled.
  - The service description is an object with three properties: “path”, “interface”, and “port”.

Service Object

- A service object describes a connected remote service.
  - Passed as the argument to discovery and “onPeerConnected” callback functions
  - Represents an active session to a remote service
- Properties of a service object
  - “path” is the object path for the service on the remote device
  - “interfaces” is an array of the interfaces implemented by the remote service
  - “dest” is the unique AllJoyn bus name for the service endpoint
- Functions defined on a service object
  - “method” returns a method object for calling a method on the service
  - “signal” returns a signal object for sending a signal to a service
  - “getProp”, “setProp”, and “getAllProps” functions to access properties on the service
- Service disposal
  - When the application no longer holds a reference to the service, object sessions with the remote service are automatically cleaned up.

Accepting remote connections

- To explicitly accept or reject a connection from a remote peer, register the “onPeerConnected” callback function.
  - Return *true* to accept the connection or *false* to reject it

```
AJ.onPeerConnected = function(peer) {
    connectedPeer = peer; // Save the service object
    return true;          // Accept the connection
}
```

  - The argument to the callback function is a service object.
    - Use the service object to send signals and make method calls to the connected peer.
- When no “onPeerConnected” callback registered:
  - AllJoyn.js will automatically accept all connections
    - Note: The Application will not have access to a service object that is needed to send signals or make method calls to the remote peer.

Invoke a method on a remote service

- A service object has all the information needed interact with a remote service Bus Object methods.
  - The “method” function returns a method object.
  - The application just has to specify the method name.
  - AllJoyn.js can usually figure out which interface to use.

```
var myMethod = svc.method(‘myMethod’);
```

  - If the method name is ambiguous – or just for clarity make the interface explicit:

```
var myMethod = svc.method({ ‘myMethod’:’test.InterfaceA’ };
```

- Making a method call
  - Call the ‘call’ function on the method object passing the required arguments:

```
myMethod.call(1, 2);
```

  - Set a callback function to handle the reply from the method call:

```
myMethod.call(1, 2).onReply = function(val){ alert(“result = “, val) };
```

Method call replies

- There are several reasons a method call can fail.
  - A timeout occurred because the service did not respond quickly enough.
    - The timeout is generated internally by the AllJoyn framework and turned into an error reply.
  - The return value is an error reply.
    - This might be an error reply from the service.
    - Or might be internally generated by the AllJoyn framework.
  - Applications should always check if the reply was an error reply.

```
myMethod.call(1, 2).onReply = function() {
    if (this.isErrorReply) {
      alert(“Method call returned error: “, this.error);
    } else {
      print(“Method call was succesfull”);|
    }
}
```

Handling a method call

- Incoming method calls from remote services are all passed to a single callback function registered by the application:

```
AJ.onMethodCall = function() {
   print(“Object path: “, this.path);
   print(“Interface: ”, this.iface);
   print(“Member: ”, this.member);
   print(“Arguments: “, JSON.stringify(arguments));
}
```

  – The number of arguments and values depend on the method being called.

  – The “this” object carries information about the method member and interface.

- Method calls generally need a reply even when there are no reply arguments send back to the caller.

```
AJ.onMethodCall = function()  {
    if (this.member == “myMethod”) {
       this.reply(args[0] * args[1]);
    }
}
```

Rejecting a method call

- There are several ways to respond to a method call to cause a failure.
  - Ignore a method call.
    - The sender will eventually get a timeout.
  - Send an error response
    - This is the “correct” way to respond to a method call that has invalid arguments or that cannot be processed due to resources or other conditions

```
AJ.onMethodCall = function()  {
    if (this.member == “myMethod”) {
        if (busy) {
            this.errorReply(“Too busy right now – try later”);
        } else {
            this.reply(args[0] * args[1]);
        }
    }
}
```

  - Throw an error from the onMethodCall function.
    - AllJoyn.js will turn an unhandled exception into a reject error reply.
    - This also handles exceptions that get thrown for other reasons.

Sending a signal

- Method calls are made to remote BusObjects; conversely, signals are sent by local BusObjects.
  - The AllJoyn.js service object has the destination information needed to send a signal, however the application must specify which local BusObject is sending the signal
- To send a signal, the application creates an AllJoyn.js signal object. There are two ways to do this:
  - The signal is sent to the specific service identified by the service object.

    ```
    var mySignal = svc.signal('/myApp', 'mySignal');
    ```

  - The signal is broadcast to all services on the bus. This form is rarely used unless the signal is specified as sessionless.

    ```
    var mySignal = AJ.signal('/myApp', 'mySignal');
    ```

- To send the signal, call the `send` function with an argument list.

  ```
  mySignal.send("hello world");
  ```

Handling a signal

- Just like a method call but there is no reply to send.
- Incoming signals from remote services are all passed to a single callback function registered by the application.

```
AJ.onSignal = function() {
   print(“Object path: “, this.path);
   print(“Interface: ”, this.iface);
   print(“Member: ”, this.member);
   print(“Arguments: “, JSON.stringify(arguments));
}
```

  - The number of arguments and values depend on the signal definition.
  - The “this” object carries information about the signal member and interface

Setting and getting service properties

- AllJoyn.js provides APIs for getting and setting specific properties.
  - These are just special cases of method calls.

```
svc.setProp(“myProperty”, 42).onReply = function() {
 if (this.isErrorReply) {
     alert(“Property was not set: “, this.error);
  } else {
     print(“Property was successfully set”);
  }
}

svc.getProp(“myProperty”).onReply = function(val) {
   if (!this.isErrorReply) { print(“Value is “, val) }
}
```

- To get all properties implemented by an interface:

```
svc.getAllProps(“test.InterfaceA”).onReply = function(props) {
   printf(“Properties “, JSON.stringify(props));
}
```

Handling property set/get requests

- The application registers callbacks to handle property access requests.
  - These are special case method call handlers so they must call `reply()`.

```
var storedValue = 0;
AJ.onPropSet = function(iface, prop, value) {
    if (prop == ‘myProperty') {
        storedValue = value;
        this.reply();
    }
}
AJ.onPropGet = function(iface, prop) {
    if (prop == ‘myProperty')  {
        this.reply(storedValue);
    }
}
AJ.onGetAllProps = function(iface) {
    if (iface == “test.InterfaceA”) {
        this.reply({ myProperty:storedVal });
    }
}
```

Persistent Storage APIs

- Store function for writing JavaScript objects to non-volatile storage.

  ```
  AJ.store(“mySavedState”, myState);
  ```

  - The object is encoded as a JSON string.
  - On embedded MCUs, stores objects in Flash.
  - On Linux, Windows, etc., write the objects to a file.

- Load function for reading JavaScript object out of non-volatile storage.

  ```
  var myState = AJ.load(“mySavedState”);
  ```

- Also provides access to AllJoyn config service parameters:

  ```
  print(AJ.load(“DeviceName”));
  print(AJ.load(“SoftwareVersion”));
  ```

Input/Output Module

Provides abstraction layer for timers and I/O functions

One-shot and Interval Timers

- Similar to APIs provided by most browsers
  - Times are specified in milliseconds
  - Call `setTimeout` to set up a one-shot timer
  - Call `setInterval` to set up an interval timer
  - Functions return a handle that can be used to cancel or modify the timer.

```
var tick = setInterval(function() { alert(“tick”) }, 1000);
resetInterval(tick, 60 * 1000);
clearInterval(tick);

function wakeUp() {
    alert(“Time to wake up!);
}
setTimeout(wakeUp, 7 * 60 * 60 * 1000);
```

- Application can have multiple timers running concurrently.

General purpose I/O pins

- Provides a hardware-independent abstraction layer for GPIO and other input/output peripherals
  - Pins are labeled pin[0] through pin[N]
  - Multiplexed pin functions can be queried at runtime
  - Pins can be configured to any function supported by the hardware.
  - Pin information includes properties for physical pin number, datasheet id, schematic id, and a free-form description.
  - To enumerate information for all the pins on a device:

```
for (var i = 0; i < IO.pin.length; ++i) {
    print(IO.pin[i].info.description, “ “, IO.pin[i].functions));
};
```

Configuring I/O Pins

- The I/O module currently has the following functions for configuring pins:

  ```
  digitalIn(), digitalOut(), analogIn(), analogOut()
  ```

  – Functions for other pin functions are not yet implemented.

- When a pin is configured as a digital input pin. the application must specify if the pin is `pullUp`, `pullDown`, or `openDrain`.

  ```
  var button = IO.digitalIn(IO.pin[2], IO.pullUp);
  ```

- A trigger function can be set on a digital input pin. The trigger function can be configured to be called when the pin state changes.

  ```
  button.setTrigger(IO.risingEdge, function(){print(“button up”)});
  ```

  – To disable a previously set trigger:

  ```
  button.setTrigger(IO.disable);
  ```

Setting and reading digital pins

- Digital input and output pins have a level property that can be set and read. An optional initial value can be provided for digital output pins.

```
var led = IO.digitalOut(IO.pin[2], 1);
led.level = 0;
led.level = 1;
```

- Digital output pins also have a `toggle` function that changes the level value from 0 to 1 or 1 to 0, depending on the current state.

```
led.toggle();
```

- Digital output pins that support Pulse Width Modulation (pwm) can be configured with a duty cycle and a frequency.
  - The duty cycle is a value in the range 0.0 to 1.0.
  - The frequency is in Hz.

```
led.pwm(0.5, 200);
```

Setting and reading analog pins

- Analog input pins have a value property that can be read.

```
var temperature = IO.analogIn(IO.pin[8]);
print(“Temperature is “, temperature);
```

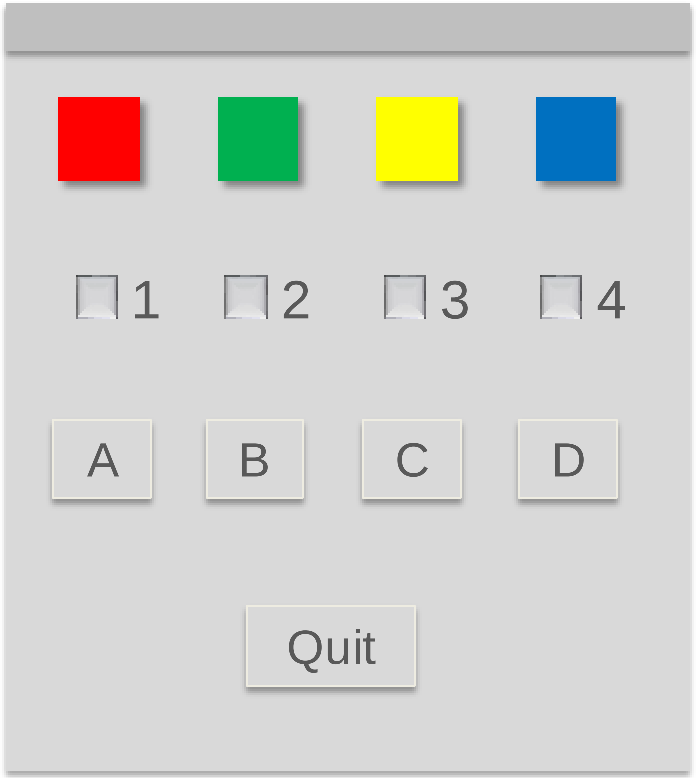
- Analog output pins have a value property that can be set.

```
var speed = IO.analogOut(IO.pin[6]);
speed.value = 0;
```

Simulated I/O

Simulated I/O for development

- AllJoyn.js includes a simple I/O simulator that makes is easy to prototype applications on a Linux or Windows desktop.
- The UI is written in Python and can be easily extended or enhanced.
  - Out of the box, the UI provides simulated LEDs, digital input pins, and push buttons.
- The Linux and Windows builds automatically look for and connect to simio.py.



Base Services

Base service integration

- AllJoyn.js currently integrates with four base services:
  - Onboarding – gets a device onto a Wi-Fi network
  - Configuration – sets up authentication credentials, friendly name, etc.
  - Notification – send text messages for human consumption
  - Control Panel – a generic UI toolkit
- Onboarding and Configuration are mostly transparent to JavaScript applications.
  - Some parameters from the Configuration service can be read and set
- AllJoyn.js implements APIs to support Notification and Control Panel services.

Notifications – simple

- To create a notification with a message in the default language:

  ```
  var notif = AJ.notification(<urgency>, “Hello World”);
  ```

  Urgency is one of the following constants:

  ```
  AJ.notification.Emergency (=0)
  AJ.notification.Warning   (=1)
  AJ.notification.Info      (=2)
  ```

- To send the notification:

  ```
  notif.send(<time-to-live>);
  ```

  Time-to-live is the number of seconds that the notification will remain deliverable. *The notification service requires this to be at least 30 seconds and no more than 12 hours (4320 seconds).*

- The creation and send operations are separated out so that additional properties can be set on the notification before it is sent.

Notifications – customized

- Additional properties can be set on the notification prior to calling send.
- Explicitly set the text to specify multiple languages for the notification.

```
notif.text = {
    en:"Hello World",
    sp:"Hola Mundo“
 };
```

- Associate an icon with the notification.

```
notif.iconUrl = "http://url/to/icon";
notif.iconPath = “/notif/icon”;          // Object path on notif sender
```

- A notification can be canceled by the sender after it has been sent.

```
notif.cancel();
```

Control Panel Service

- The Control Panel service allows a headless application to expose a simple control panel built from a set of simple widgets.
  - A generic application running on a handset, tablet, or other device can render the UI without knowing anything about the device being controlled.
- The JavaScript application creates a control panel, adds a top-level container widget and then adds various widgets that define the UI.

```
var cp = AJ.controlPanel();
var root = cp.containerWidget();
var rate = root.propertyWidget(cp.SLIDER, 500, "Flash rate:");
rate.range = { min:20, max:1000, increment:50, units:"msec" };
var led = IO.digitalOut(IO.pin[1]);
var blinky = setInterval(function(){led.toggle()}, rate.value);
rate.onValueChanged = function(val){resetInterval(blinky, val)}
AJ.onAttach = function() { cp.load(); }
```

Property Widgets

- Property widgets are used for setting and getting property values
  - The widget definition specifies the preferred UI rendering, options are:
    - SLIDER
    - CHECK_BOX
    - SPINNER
    - RADIO_BUTTON
    - SLIDER
    - TIME_PICKER
    - DATE_PICKER
    - NUMBER_PICKER
    - KEYPAD
    - ROTARY_KNOB
    - TEXT_VIEW
    - NUMERIC_VIEW
    - EDIT_TEXT

Property Widget Range and Choice

- Property widgets that have numeric values can specify ranges.
  - A range is an object that has “min” and “max” properties.
  - An optional “increment” property provides additional information for the UI renderer.
  - An optional “units” property provides a label the UI renderer can attach to the displayed value.

```
var flow = root.propertyWidge(cp.ROTARYKNOB, 0, ‘Sprinker flow rate’);
flow.range = { min:0, max:100, increment:5, units:’litres per minute’ };
```

- Property widgets with discrete values can specify choices.
  - The choices are numbered 0 through N.

```
var color = root.propertyWidget(cp.RADIO_BUTTON, 0, ‘Color picker’);
color.choices = [ "red", "orange", "yellow", "green", "blue” ];
```

Tracking Property Widget Changes

- An “onValueChanged” callback function can be registered on any property widget.
  - This function is called whenever a property value is set either locally or remotely from a Control Panel controller application.
- Enable/disable
  - When set to false, the “enable” value on a property widget tells the renderer to disable or gray out the property in the UI.
- Writeable
  - When set to false, the “writeable” value on a property widget tells the renderer that the value can no longer be set.

Using the Console Application

Console Application

- The Console application is a standalone AllJoyn application that communicates with an AllJoyn service running alongside the JavaScript application
- If called with a JavaScript file, the Console application connects to installs an new application into a running AllJoyn.js instance.
  - The previous application is overwritten.
  - If there are errors running the script, they are output to the console.
- If called without a JavaScript file, the Console application connects to a running AllJoyn.js.
- In either case, after connecting to the AllJoyn.js instance, any input is sent to the JavaScript interpreter.
  - This allows real-time interaction with the running JavaScript program.

Example Console Interaction

```
Found script console service: :Zp5SKg6r.4
Joined session: 841438313
JSON.stringify(AJ)
Eval: JSON.stringify(AJ);
Eval result=0:
{"interfaceDefinition":{“test.DoorBell":{"ding_dong":{"type":1}}},"objectDefinition":{"/Door
Bell":{"interfaces":[“test.DoorBell"]}},"config":{"linkTimeout":10000,"callTimeout":10000},"
METHOD":0,"SIGNAL":1,"PROPERTY":2,"defaultLanguage":"en"}
JSON.stringify(IO);
Eval: JSON.stringify(IO);
Eval result=0: Eval result=0:
{"pin":[{"id":0},{"id":1},{"id":2},{"id":3},{"id":4},{"id":5}],"openDrain":2,"pullUp":4,"pul
lDown":8,"risingEdge":1,"fallingEdge":2}
IO.pin[0].info.description
Eval: IO.pin[0].info.description;
Eval result=0: Red LED
IO.pin[0].functions
Eval: IO.pin[0].functions;
Eval result=0: digitalOut
2+3
Eval: 2+3;
Eval result=0: 5
alert("Hello world")
Eval: alert("Hello world");
Hello world
Eval result=0: undefined
```

Code Samples

Send a notification on GPIO interrupt

```
var pbA = IO.digitalIn(IO.pin[8], IO.pullDown);
var pbB = IO.digitalIn(IO.pin[9], IO.pullDown);

AJ.onAttach = function()
{
    pbA.setTrigger(IO.fallingEdge, function() {
        AJ.notification(1, "Button A pressed").send(200); });

    pbB.setTrigger(IO.risingEdge, function() {
        AJ.notification(0, "Button B released").send(200); });
}

AJ.onDetach = function()
{
    pbA.setTrigger(IO.disable);
    pbB.setTrigger(IO.disable);
}
```

Controlling LED flash rate

```
var cp = AJ.controlPanel();

var c1 = cp.containerWidget(cp.VERTICAL, cp.HORIZONTAL);
var rate = c1.propertyWidget(cp.SLIDER, 500, "Flash rate:");
rate.range = { min:20, max:1000, increment:50, units:"msec" };

var led = IO.digitalOut(IO.pin[0]);

var blinky = setInterval(function(){led.toggle();}, rate.value);

rate.onValueChanged = function(val) { resetInterval(blinky, val); }

AJ.onAttach = function() { cp.load(); }
```

Doorbell – push button side

```
AJ.interfaceDefinition[‘test.DoorBell'] = {
    ding_dong:{ type:AJ.SIGNAL }
};

AJ.objectDefinition['/pushbutton'] = {
   interfaces:['org.allseen.DoorBell']
};

var pb = IO.digitalIn(IO.pin[8], IO.pullDown);

AJ.onAttach = function()
{
    AJ.findService(‘test.DoorBell', function(svc) {
        var dingdong = svc.signal('/pushbutton', 'ding_dong');
        pb.setTrigger(IO.fallingEdge, function() { dingdong.send() });
    });
}

AJ.onDetach = function() { pb.setTrigger(IO.disable) }
```

Doorbell - bell side

```
AJ.interfaceDefinition[‘test.DoorBell'] = {
   ding_dong:{ type:AJ.SIGNAL }
};

AJ.objectDefinition['/DoorBell'] = {
    interfaces:['org.allseen.DoorBell']
};

AJ.onSignal = function()
{
    if (this.member == 'ding_dong') {
        IO.system('aplay DoorBell.wav');
    }
}
```

Thank You

Follow Us On

- For more information on AllSeen Alliance, visit us at: allseenalliance.org & allseenalliance.org/news/blogs