Table of Contents

Introduction

This is the documentation for Signal K specification master version available in the following formats:

- html
- pdf
- epub
- mobi

Getting Started in Using Signal K

You can start using Signal K by

- connecting to the demo server on the Internet with a browser
- installing either Node or Java server on any computer
- getting some hardware for your boat, such as a Raspberry Pi, suitable USB adapters for your boat's network (NMEA0183, Nmea 2000 or roll your own with I2C sensors) and installing Node or Java server
- purchasing a commercial Signal K gateway such as iKommunicate by Digital Yacht
- installing OpenPlotter, which includes a Signal K server

Once you have a server running (or you start by using the demo server) you can install some Signal K supporting mobile App such as

- NMEARemote by Zapfware (iOS)
- OceanIX (Android)

Getting Started in Developing with Signal K

- Example HTML5 applications
- Signal K JavaScript client
- iKommunicate Developer's Guide)

Signal K Data Model

Signal K defines two data formats, full and delta, for representing and transmitting data.

In additiong the 'sparse' format is the same as the full format, but doesn't contain a full tree, just parts of the full tree.

Full format

The simplest format is the full format, which is the complete Signal K data model sent as a JSON string. Abbreviated for clarity it looks like this:

```
{"vessels":{"9334562":{"navigation":{"courseOverGroundTrue":{"value":11.9600000381},"courseOverGroundMagnetic":{"value":93.0000000000},"more":"a lot more data here...","waterTemp":{"value":0.0000000000},"wind":{"speedAlarm":{"value":0.0000000000},"directionChangeAlarm":{"value":0.0000000000},"directionApparent":{"value":0.0000000000},"directionTrue": {"value":0.0000000000},"speedApparent":{"value":0.0000000000},"speedTrue": {"value":0.0000000000}}}}}}
```

Formatted for ease of reading:

```
{
  "vessels": {
    "9334562": {
      "version": "0.1",
      "name": "motu",
      "mmsi": "2345678",
      "source": "self",
      "timezone": "NZDT",
      "navigation": {
        "state": {
          "value": "sailing",
          "source": "self",
          "timestamp": "2014-03-24T00: 15: 41Z"
        },
        "headingTrue": {
          "value": 23,
          "source": {
            "pgn": "128275",
            "device": "/dev/actisense",
            "timestamp": "2014-08-15-16:00:05.538",
            "src": "115"
          },
          "timestamp": "2014-03-24T00: 15: 41Z"
        },
        "more": "a lot more data here...",
        "roll": {
          "value": 0,
          "source": "self",
          "timestamp": "2014-03-24T00: 15: 41Z"
        },
        "rateOfTurn": {
          "value": 0,
          "source": "self",
          "timestamp": "2014-03-24T00: 15: 41Z"
        }
      }
    }
  }
}
```

The message is UTF-8 ASCII text, and the top-level attribute(key) is always "vessels". Below this level is a list of vessels, identified by their MMSI number or a generated unique id. There may be many vessels, if data has been received from AIS or other sources. The format for each vessel's data uses the same standard Signal K structure, but may not have the same content, i.e. you won't have as much data about other vessels as you have about your own.

The values are always SI units, and always the same units for the same key. I.e. `speedOverGround` is always meters per second, never knots, km/hr, or miles/hr. This means you never have to send 'units' with data, the units are specific for a key, and defined in the data schema.

The ordering of keys is also not important, they can occur in any order. In this area Signal K follows normal JSON standards.

The full format is useful for backups, and for populating a secondary device, or just updating all data, a kind of 'this is the current state' message.

However sending the full data model will be wasteful of bandwidth and CPU, when the majority of data does not change often. So we want to be able to send parts of the model (i.e. parts of the hierarchical tree).

Sparse format

The sparse format is the same as the full format but only contains a limited part of the tree. This can be one or more data values.

```
{"vessels":{"self":{"navigation":{"position":{"latitude":{"value":-41.2936935424}}
}}}}

{"vessels":{"self":{"navigation":{"position":{"longitude":{"value":173.2470855712},
"source":"self","timestamp":"2014-03-24T00:15:41Z"}}}}}
```

or, more efficiently (and formatted):

```
{
  "vessels": {
    "self": {
      "navigation": {
        "position": {
          "latitude": {
            "value": -41.2936935424
          },
          "longitude": {
            "value": 173.2470855712
          }
        }
      }
    }
  }
}
```

Mix and match of misc values are also valid:

```
{
  "vessels": {
    "self": {
      "navigation": {
        "courseOverGroundTrue": {
          "value": 11.9600000381
        },
        "position": {
          "latitude": {
            "value": -41.2936935424
          },
          "longitude": {
            "value": 173.2470855712
          },
          "altitude": {
            "value": 0
          }
        }
      }
    }
  }
}
```

This mix of any combination of values means we don't need to create multiple message types to send different combinations of data. Just assemble whatever you want and send it. When parsing an incoming message a device should skip values it has no interest in, or doesn't recognise. Hence we avoid the problem of multiple message definitions for the same or similar data, and we avoid having to decode multiple messages with fixed formats.

Delta format

While building the reference servers and clients it was apparent that a third type of message format was useful. This format specifically sends changes to the full data model. This was useful for a number of technical reasons, especially in clients or sensors that did not hold a copy of the data model.

The format looks like this (pretty printed):

```
{
  "context": "vessels.230099999",
  "updates": [
    {
      "source": {
        "pgn": "128275",
        "device": "/dev/actisense",
        "timestamp": "2014-08-15-16:00:05.538",
        "src": "115"
      },
      "values": [
        {
          "path": "navigation.logTrip",
          "value": 43374
        },
        {
          "path": "navigation.log",
          "value": 17404540
        }
      ]
    },
    {
      "source": {
        "device": "/dev/actisense",
        "timestamp": "2014-08-15-16:00:00.081",
        "src": "115",
        "pgn": "128267"
      },
      "values": [
        {
          "path": "navigation.courseOverGroundTrue",
          "value": 172.9
        },
        {
          "path": "navigation.speedOverGround",
          "value": 3.85
        }
      ]
    }
  ]
}
```

In more detail we have the header section:

```
{
  "context": "vessels.230099999",
    "updates": [
      ...data goes here...
    ]
}
```

The message can be recognised from the other types by the topmost level having "context" and "updates" rather than "vessels".

Context is a path from the root of the full tree. In this case 'vessels.230099999'. All subsequent data is relative to that location. The context could be much more specific, e.g. 'vessels.230099999.navigation', whatever is the common root of the updated data.

The 'updates' holds an array (JSON array) of updates, each of which has a 'source' and JSON array of 'values'.

```
{
  "source": {
    "device": "/dev/actisense",
    "timestamp": "2014-08-15-16:00:00.081",
    "src": "115",
    "pgn": "128267"
  },
  "values": [
    {
      "path": "navigation.courseOverGroundTrue",
      "value": 172.9
    },
    {
      "path": "navigation.speedOverGround",
      "value": 3.85
    }
  ]
}
```

The 'source' values is the same and applies to each of the 'values' items, which removes data duplication. It also allows rich data to be included with minimal impact on message size.

Each 'value' item is then simply a pair of 'relative path', and 'value'.

Message Integrity

Many messaging systems specify checksums or other forms of message integrity checking. In Signal K we assume a reliable transport will guarantee a valid message. This is true of TCP/IP and some other transports but not always the case. For other transports (eg RS232 serial) a specific extended data format will apply, which is suited to that transport. Hence at the message level no checksum or other tests need to be made.

Encoding/Decoding

The JSON message format is supported across most programming environments and can be handled with any convenient library.

On micro-controllers with limited RAM it is wise to read and write using streaming rather than hold the whole message in precious RAM. There is an implementation of Signal K JSON streaming on an Arduino Mega (4K RAM) in the related Freeboard project, which will be released in Signal K eventually.

Multiple Values for a Key

There are two use cases for multiple data:

- Multiple versions of a common device - eg two engines
- Multiple devices providing duplicate data - multiple values for the same signalk key from different sensors, eg COG from both compass and gps

Multiple versions of a common device

Consider the data point `temperature`. There are many versions of temperature: air, water, engineRoom, fridge, freezer, main cabin, etc. Some are well-known, and in common usage, some will be very vessel specific.

So we need a structure that provides a flexible way to hold lots of sub-items. The simple solution is an array of `temperature` objects however https://github.com/SignalK/specification/wiki/Arrays-are-Evil.

So instead we simply put the individual temperature objects in as children of `temperature`

```
{
    "temperature": {
        "air": {
            "value": 26.7,
            "source": "vessels.self.sources.n2k.n2k1-12-0"
        },
        "water": {
            "value": 18.2,
            "source": "vessels.self.sources.n2k.n2k1-12-1"
        }
    }
}
```

And in `vessels.self.sources`

```
{
   "n2k":{
        "n2k1-12-0": {
            "timestamp": "2014-08-15-16: 00: 00.081",
            "source": {
                "label": "Outside Ambient Masthead",
                "bus": "/dev/ttyUSB1"
            },
            "value":"dump the raw n2k data here"
        },
        "n2k1-12-1": {
            "timestamp": "2014-08-15-16: 00: 00.081",
            "value": 18.2,
            "source": {
                "label": "Water Temperature",
                "bus": "/dev/ttyUSB1"
            }
        },
        "n2k2-201-0": {
            "timestamp": "2014-08-15-16: 00: 00.081",
            "value": 66.7,
            "source": {
                "label": "Engine Room",
                "bus": "/dev/ttyUSB2"
            }
        }

}
```

This scheme allows for both well-known keys `temperature.air` and vessel specific `temperature.aftFreezer`. It is also valid in the following form, but makes it more difficult to refer to the source if it maps to multiple signalk keys (eg NMEA 0183 RMC sentence https://github.com/SignalK/specification/wiki/Samples---NMEA-0183-RMC):

```
{
    "temperature": {
        "air": {
            "value": 26.7,
            "source": "n2k1-12-0",
            "n2k1-12-0": {
                "timestamp": "2014-08-15-16: 00: 00.081",
                "value": 26.7,
                "source": {
                    "label": "Outside Ambient Masthead",
                    "bus": "/dev/ttyUSB1"
                }
            }
        },
        "water": {
            "value": 18.2,
            "source": "n2k1-12-1",
            "n2k1-12-1": {
                "timestamp": "2014-08-15-16: 00: 00.081",
                "value": 18.2,
                "source": {
                    "label": "Water Temperature",
                    "bus": "/dev/ttyUSB1"
                }
            }
        }
    }
}
```

It maintains the primary requirement that a given data value have a fixed and unique uri, but gives flexibility in the structure and complexities of data. It also fulfils the requirement for discovery of data keys, vessel specific sources, and provides the ability to navigate the structure in a consistent progamatical way.

Multiple devices providing duplicate data

It is quite possible for a key value to come from more than one device. eg position (lat/lon) could come from several gps enabled devices, and multiple depth sounders are not uncommon. We need a consistent way to handle this.

All the incoming values may well be valid in their own context, and it is feasible that all of them may be wanted, for instance, displaying depth under each hull on a catamaran.

Hence discarding or averaging is not a solution, and since signalk is unable to derive the best way to handle multiple values it must always fall to a default action, with human over-ride when needed.

The solution presented below has flaws. See https://github.com/SignalK/specification/issues/48 for discussion.

In signal K we can leverage the above method and simply store all the devices in the tree under the main item, and have the main items `source` reference the options. Lets consider this for `courseOverGroundTrue`

If its the first value for the key, it becomes the default value and looks like this:

```
{
  "vessels": {
    "self": {
      "navigation": {
        "courseOverGroundTrue": {
          "value": 102.29,
          "source": "vessels.self.sources.n2k.actisense-115-129026"
        }
      },
      "sources": {
        "n2k": {
          "actisense-115-129026": {
            "value": 102.29,
            "bus": "/dev/actisense",
            "timestamp": "2014-08-15-16: 00: 01.083",
            "src": "115",
            "pgn": "129026"
          }
        }
      }
    }
  }
}
```

It has come from device `vessels.self.sources.n2k.actisense-115-129026` , where further details can be found.

If another value with different source arrives, we add the source with a unique name, so both values are in there - if its our preferred source (from persistent config) we auto-switch to it, otherwise we just record it. It look like this:

```
{
  "vessels": {
    "self": {
      "navigation": {
        "courseOverGroundTrue": {
          "timestamp": "2014-08-15-16: 00: 01.083",
          "value": 102.29,
          "source": "vessels.self.sources.n2k.actisense-115-129026"
        }
      },
      "sources": {
        "n2k": {
          "actisense-115-129026": {
            "value": 102.29,
            "bus": "/dev/actisense",
            "timestamp": "2014-08-15-16: 00: 01.083",
            "src": "115",
            "pgn": "129026"
          },
          "actisense-201-130577": {
            "value": 102.29,
            "bus": "/dev/actisense",
            "timestamp": "2014-08-15-16: 00: 00.085",
            "src": "201",
            "pgn": "130577"
          }
        }
      }
    }
  }
}
```

Rules

Now simple rules can apply to obtain the default, or any specific value:

- The implementation must ensure that the `key.value` holds an appropriate value. This will be easy if there is only one, and will probably be user configured if more.
- If the `source` value is `string` then it is a reference key to the source object, and can be a relative or absolute signalk key.
- The `source` (as a reference string) also provides a mechanism to handle deprecated keys.
- If the `source` value is a `json object` then it holds meta data on the source of the value.
- Alternate sources must be discovered manually, or by implementation specific meta-data.

To see all the entries, use the REST api or subscribe to the parent object. A given device may choose to subscribe to a specific entry in the object, allowing multiple displays of the key, or users of the various values. The 'list' verb used in a query message can provide available keys.

Unique names

The identifier for each device should be unique within the server, and possibly be constructed as follows:

```
n2k: producerid-sourceid-pgn (producer id from server configuration, others from n
2k data) - NOTE: will change, currently under discussion.
nmea0183: producerid-talkerid-sentence (like n2k)
signalk: any valid string matching regex [a-zA-Z0-9-]. eg alphabet, hyphens, and 0
 to 9
```

(The nmea0183 talker id is not in the schema as I write this, it will be added shortly)

Metadata

The Use Cases

Let's assume we have engine1.rpm as a key/value in Signal K. We want to display it on our dashboard, and monitor alarms for temp, oil, rpm etc.

We can drop a generic dial gauge on our dash and display rpm, but it can't know maxRpm, or alarms unless its an engine-specific gauge, and knows where to look in the Signal K schema. So we will end up with a profusion of role specific gauges to maintain. We also have non standard key names for max, min, high, low, etc. which pollute the schema.

Currently the Signal K server has a set of specific alarm keys. These grow over time and are becoming awkward. The server can only monitor these specific keys at present as there is no mechanism for arbitrary alarm definition.

Metadata for a Data Value

Each data key should have an optional `.meta` object. This holds data in a standard way which enables the max/min/alarm and display to be automatically derived.

```
{
  "displayName": "Tachometer, Engine 1",
  "shortName": "RPM",
  "warnMethod": "visual",
  "warnMessage": "any text",
  "alarmMethod": "sound",
  "alarmMessage": "any text",
  "zones": [
    {"lower":0.0,"upper":500,"state":"alarm", "message":"Stopped or very slow Rpm"
},
    {"lower":500,"upper":3000,"state":"normal", "message":""},
    {"lower":3000,"upper":3500,"state":"warn", "message":"Approaching maximum rpm"
},
    {"lower":3500,"upper":9999,"state":"alarm", "message":"Exceeding maximum rpm"}
  ]
}
```

Since the settings object is always the same, the tachometer can now limit its range, and display green, yellow, and red sectors. The generic gauge can now perform this role, with correct labels etc.

The alarms problem is also improved, as the server can run a background process to monitor any key that has a `.meta` object, and raise a generic alarm event. By recursing the tree the alarm monitoring can find the source (engine1), giving the alarm context. See [[Alarm Handling]]

The alarms functionality then becomes generic, and grows with the spec. This is may be the case for other functionality also.

Default Configuration

Other than a few standard keys it is unlikely that the `.meta` can have global defaults, as it is very vessel specific (e.g. a sail boat will have speeds from 0-15kts, a ski boat will have 0-50kts). So the values will have to be configured by the user on the individual vessel as required.

It is probably possible to have profiles that set a range of default `.meta`, e.g. sail vessel, or motor vessel, and if two vessels have the same engine, then the engine profiles will also tend to be the same.

Alarm Management

An alarm watch is set by setting the `meta.zones` array appropriately. A background process on the server checks for alarm conditions on any attribute with a `meta.zones` array. If the keys value is within a zone the server sets an alarm key similar to `vessels.self.notifications.[original_key_suffix]`, eg an alarm set on `vessels.self.navigation.courseOverGroundTrue` will become `vessels.self.notifications.navigation.courseOverGroundTrue`.

The object found at this key should contain the following:

```
{
  "message": "any text",
  "state": "[normal|alert|warn|alarm|emergency]"
}
```

Other Benefits

The common profiles should be exportable and importable. This would allow manufacturers or other users to create profiles for specific products or use cases, which could then be imported to a vessel.

This may also have possibilities for race control or charter management. For instance a limit on lat/lon would raise an 'Out of Bounds' email on a charter vessel.

A lot of the current max/min/alarm values could be removed to simplify and standardise the spec.

Permissions Model

The permissions model for Signal K is based on the UNIX file permissions model. This was first developed in the late 1970's and is still perfectly suited to the internet today, so its got to be a pretty sound model!.

So we adapted it for Signal K. See http://www.tutorialspoint.com/unix/unix-file-permission.htm

Each key in Signal K has an optional `_attr` value.

```
"vessels": {
    "self":{
             //the usual signal k keys, navigation, environment, etc

       "_attr":{                 // filesystem specific data, eg security, possibl
y more later
                "_mode": 640,         // unix style permissions, often written in
`owner:group:other` form, `-rw-r-----`
                "_owner" : "self",    // owner, surprisingly. The user who created
 the item, sometimes a virtual user like 'self'
                "_group": "self"      // group
             }
           }
        }
```

By default the `vessels.self` key has the above `_attr`. This effectively means that only the current vessels 'owner' can read and write from this key or any of its sub-keys. It also allows users in group `self` to read the data. This provides a way to give additional programs or users read-only access. In the above case an external user connecting from outside the vessel and requesting vessel data would receive `{}`, eg nothing.

Note:keys beginning with `_` are always stripped from signal k messages

Since the above is a default, Signal K devices that lack the resources to implement security should always be installed behind a suitable gateway that can provide security. Again, the simplest security is the default read-write only within the local vessel (typically the current network). This makes a basic implementation as simple as possible.

The permissions apply recursively to all sub-keys, unless specifically overwritten. You can only provide a **narrowing** change in permissions, eg less than the parent directory. In the above case if the permissions for `vessels.self.navigation.position` were set to `"_mode" : 644`, it would have no effect as access is blocked at the `vessels.self` key. The `vessels.self` _attr must now also be `"_mode" : 644`, and all its other subkeys explicitly set to `"_mode" : 640`

Hence setting complex permissions are likely beyond the typical user. For this reason we believe there should be a choice of default permission 'templates' for the signal K tree. Users would select their preference from a config screen. A paranoid user may prefer the above setup, another may chose to allow basic data similar to AIS (position, cog, speed, etc), and others may expose much more.

Templates also allow sharing of data for specific uses or needs, like a social group, or a marina.

Exposing everything (`"_mode" : 666`) would be dangerous - it would potentially allow external users to gain control of the vessels systems, however it is useful for demos and software development. All signal K implementations should always consider the potential danger of such permissions, and protect users if possible.

The implementation of proper security is the responsibility of the Signal K software implementation provider.

By manipulating the `_attr` values for the Signal K keys, and creating suitable users and groups a sophisticated and well proven security model for vessel data can be created.

Ports, Urls and Versioning

Short Names

- `self` refers to the current vessel. Normally used in `vessels.self...`.

Ports

The Signal K HTTP and WebSocket services SHOULD be found on the usual HTTP/S ports (80 or 443). The services SHOULD be found on the same port, but may be configured for independent ports and MAY be configured for ports other than HTTP/S.

A Signal K server MAY offer Signal K over TCP or UDP, these services SHOULD be on port 55555[1].

If an alternate port is needed it SHOULD be an arbitrary high port in the range 49152–65535[2].

URL Prefix

The Signal K applications start from the `/signalk` root. This provides some protection against name collisions with other applications on the same server. Therefore the Signal K entry point will always be found by loading `http(s)://«host»:«port»/signalk`.

Versioning

The version(s) of the Signal K API that a server supports SHALL be available as a JSON object available at `/signalk`:

```
{
    "endpoints": {
        "v1": {
            "version": "1.1.2",
            "signalk-http": "http://192.168.1.2/signalk/v1/api/",
            "signalk-ws": "ws://192.168.1.2:34567/signalk/v1/stream"
        },
        "v3": {
            "version": "3.0",
            "signalk-http": "signalk/v3/api/",
            "signalk-ws": "ws://192.168.1.2/signalk/v3/stream",
            "signalk-tcp": "tcp://192.168.1.2:34568"
        }

    }
}
```

This response is defined by the `discovery.json` schema. In this example, the server supports two versions of the specification: `1.1.2` and `3.0` . For each version, the server indicates which transport protocols it supports and the URL that can be used to access that protocol's endpoint; in the example, the `1.1.2` REST endpoint is located at `http://192.168.1.2/signalk/v1/api/` . Clients should use one of these published endpoints based on the protocol version they wish to use.

The server must only return valid URLs and should use IANA standard protocol names such as `http` . However, a server may support unofficial protocols and may return additional protocol names; for example, the response above indicates the server supports a `signalk-tcp` stream over TCP at on port `34568` .

A server may return relative URIs that the client must resolve against the base of the original request.

Streaming WebSocket API: /signalk/v1/stream

Initiates a WebSocket connection that will start streaming the server's updates as Signal K delta messages. You can specify the contents of the stream by using a specific URL:

- ws://hostname/signalk/v1/stream?subscribe=self
- ws://hostname/signalk/v1/stream?subscribe=all
- ws://hostname/signalk/v1/stream?subscribe=none

With no query parameter the default is `self`, which will stream the data related to the `self` object. `all` will stream all the updates the server sees and `none` will stream only the heartbeat, until the client issues subscribe messages in the WebSocket stream.

If a server does not support some streaming options listed in here it must respond with http status code `501 Not Implemented`.

See Subscription Protocol for more details.

Connection Hello

Upon connection a 'hello' message is sent as follows:

```
{
  "version": "1.1.2",
  "timestamp": "2015-04-13T01:13:50.524Z",
  "self": "123456789"
}
```

REST/HTTP API: /signalk/v1/api/

Note the trailing slash in the path.

The base URL MUST provide a Signal K document that is valid according to the full Signal K schema specification. The contents SHOULD be all the current values of the data items the server knows.

If the path following the base is a valid Signal K path `GET` MUST retrieve the Signal K branch named by the path; e.g. `/signalk/v1/api/vessels/123456789/navigation/speedThroughWater` returns

```
{
    "value": 2.55,
    "source": {
        "type": "NMEA0183",
        "src": "VHW",
        "label": "signalk-parser-nmea0183"
    },
    "timestamp": "2015-08-31T05:45:36.000Z"
}
```

Streaming WebSocket API: /signalk/v1/stream

Initiates a WebSocket connection that will start streaming the server's updates as Signal K delta messages. You can specify the contents of the stream by using a specific URL:

- ws://hostname/signalk/v1/stream?subscribe=self
- ws://hostname/signalk/v1/stream?subscribe=all
- ws://hostname/signalk/v1/stream?subscribe=none

With no query parameter the default is `self`, which will stream the data related to the `self` object. `all` will stream all the updates the server sees and `none` will stream only the heartbeat, until the client issues subscribe messages in the WebSocket stream.

If a server does not support some streaming options listed in here it must respond with http status code `501 Not Implemented`.

See Subscription Protocol for more details.

Connection Hello

Upon connection a 'hello' message is sent as follows:

```
{
  "version": "1.1.2",
  "timestamp": "2015-04-13T01:13:50.524Z",
  "self": "123456789"
}
```

Discovery and Connection Establishment

Service Discovery

A Signal K server SHOULD advertise its services over mDNS/Bonjour. The server MUST use the service types

- `_signalk-http._tcp` for http API
- `_signalk-ws._tcp` for WebSocket
- `_signalk-https._tcp` for HTTPS API
- `_signalk-wss._tcp` for secure WebSocket

Furthermore a server SHOULD advertise its web interface with normal Bonjour convention `_http._tcp` and `_https._tcp`.

A sample Bonjour record output, dumped using avahi-discover:

```
Service data for service 'signalk-http (2)' of type '_signalk-http._tcp' in domain
 'local' on 4.0:
    Host 10-1-1-40.local (10.1.1.40),
    port 8080,
    TXT data: [
        'vessel_uuid=urn:mrn:signalk:uuid:6b0e776f-811a-4b35-980e-b93405371bc5',
        'version=v1.0.0',
        'vessel_name=urn:mrn:signalk:uuid:6b0e776f-811a-4b35-980e-b93405371bc5',
        'vessel_mmsi=urn:mrn:signalk:uuid:6b0e776f-811a-4b35-980e-b93405371bc5',
        'server=signalk-server',
        'path=/signalk'
        ]

Service data for service 'signalk-ws (2)' of type '_signalk-ws._tcp' in domain 'lo
cal' on 4.0:
    Host 10-1-1-40.local (10.1.1.40),
    port 3000,
    TXT data: [
        'vessel_uuid=urn:mrn:signalk:uuid:6b0e776f-811a-4b35-980e-b93405371bc5',
        'version=v1.0.0',
        'vessel_name=urn:mrn:signalk:uuid:6b0e776f-811a-4b35-980e-b93405371bc5',
        'vessel_mmsi=urn:mrn:signalk:uuid:6b0e776f-811a-4b35-980e-b93405371bc5',
        'server=signalk-server',
        'path=/signalk'
        ]
```

Connection Establishment

Using the information above a web client or http capable device can discover and connect to a Signal K server using the following process:

- Listen for Signal K services using Bonjour/mDns.
- Use the Bonjour record to find the REST api interface `signalk-http`
- Make a GET call to (eg `http://10.1.1.40:8080/signalk` from above)
- And get the endpoints json

```
{
    "endpoints": {
        "v1": {
            "version": "1.1.2",
            "signalk-http": "http://192.168.1.2/signalk/v1/api/",
            "signalk-ws": "ws://192.168.1.2:34567/signalk/v1/stream"
        }
     }
 }
```

- Make further REST calls for more specific data, or open a websocket connection and subscribe to data

Subscription Protocol

Subcription protocol is currently available only on the Java server.

Introduction

By default a Signal K server will provide a new WebSocket client with a delta stream of the `vessels.self` record, as updates are received from sources. E.g. `/signalk/v1/stream` will provide the following delta stream, every time the log value changes .

```
{
  "context": "vessels",
   "updates": [{
      "source": {
        "pgn": "128275",
        "device": "/dev/actisense",
        "timestamp": "2014-08-15-16:00:05.538",
        "src": "115"
      },
      "values": [
        {
          "path": "navigation.logTrip",
          "value": 43374
        },
        {
          "path": "navigation.log",
          "value": 17404540
        }]
     }
     ]
}
```

> Below we refer to WebSockets, but the same process works in the same way over any transport. E.g. for a raw TCP connection the connection causes the above message to be sent, and sending the subscribe messages will have the same effect as described here.

This can be a lot of messages, many you may not need, especially if `vessel.self` has many sensors, or other data sources. Generally you will want to subscribe to a much smaller range of data.

First you will want to unsubscribe from the current default (or you may have already connected with `ws://hostname/signalk/v1/stream?subscribe=none`). To unsubscribe all create an `unsubscribe` message and send the message over the websocket connection:

```
{
  "context": "vessels.self",
  "unsubscribe": [
    {
      "path": "*",
    }
  ]
}
```

To subscribe to the required criteria send a suitable subscribe message:

```
{
  "context": "vessels.self",
  "subscribe": [
    {
      "path": "navigation.speedThroughWater",
      "period": 1000,
      "format": "delta",
      "policy": "ideal",
      "minPeriod": 200
    },
    {
      "path": "navigation.logTrip",
      "period": 10000
    }
  ]
}
```

- `path=[path.to.key]` is appended to the context to specify subsets of the context. The path value can use jsonPath syntax.

The following are optional, included above only for example as it uses defaults anyway:

- `period=[millisecs]` becomes the transmission rate, e.g. every `period/1000` seconds. Default=1000
- `format=[delta|full]` specifies delta or full format. Default: delta
- `policy=[instant|ideal|fixed]` . Default: ideal
  - `instant` means send all changes as fast as they are received, but no faster than `minPeriod` . With this policy the client has an immediate copy of the

current state of the server.
    - `ideal` means use `instant` policy, but if no changes are received before `period`, then resend the last known values.eg send changes asap, but send the value every `period` millisecs anyway, whether changed or not.
    - `fixed` means simply send the last known values every `period`.
- `minPeriod=[millisecs]` becomes the fastest message transmission rate allowed, e.g. every `minPeriod/1000` seconds. This is only relevant for policy='instant' to avoid swamping the client or network.

You can subscribe to multiple data keys multiple times, from multiple apps or devices. Each app or device simply subscribes to the data it requires, and the server and/or client implementation may combine subscriptions to avoid duplication as it prefers on a per connection basis. At the same time it is good practice to open the minimum connections necessary, for instance one websocket connection shared bewteen an instrument panel with many gauges, rather then one websocket connection per gauge.

When data is required once only, or upon request the `subscribe/unsubscribe` method should not be used. If the client is http capable the REST api is a good choice, or use `get/list/put` messages over websockets or tcp.

The `get/list/put` messages work in the same way as their `GET/PUT` REST equivalents, returning a json result for the requested path.

Use Cases and Proposed Solutions

Local boat individual instruments

A gauge-type display for just one or a few data items for the 'self' vessel should be able to specify that it only wants those items for the self vessel.

This can be achieved by a default WebSocket connection `/signalk/v1/stream?subcribe=none`, then sending a JSON message:

```
{
  "context": "vessels.self",
  "subscribe": [
    {
      "path": "environment.depth.belowTransducer",
    },
    {
      "path": "navigation.speedThroughWater",
    }
  ]
}
```

The JSON format is also viable over a simple TCP or serial transport, and is therefore supported as the primary subscription method.

Map display with all known vessel positions & directions, served over 3G cellular connection

```
{
  "context": "vessels.*",
  "subscribe": [
    {
      "path": "navigation.position",
      "period": 120000,
      "policy": "fixed"
    },
    {
      "path": "navigation.courseOverGround",
      "period": 120000,
      "policy": "fixed"
    }
  ]
}
```

The result is a delta message of the Signal K data with just position and courseOverGround branches for all known vessels, sent every 2 minutes (120 seconds) even if no data has been updated.

Position of a certain vessel, immediately it changes, but once per minute at most

```
{
  "context": "vessels.230029970",
  "subscribe": [
    {
      "path": "navigation.position",
      "minPeriod": 60000,
      "policy": "instant"
    }
  ]
}
```

The result will be delta position messages for vessel 230029970, broadcast whenever it changes, but with minimum interval of 60 seconds. Messages are delayed to meet the minimum interval with newer messages overriding the previous message in the buffer.

Background and Design Rationale

In Signal K every datapoint should have a predictable and unique uri (address). What we want to maintain is to know that vessels.self.data.temp is always at that uri, and what the json form is. So if its an array thats workable. If its a json object with many instance keys, each which has a arbitrary name and the same internal structure that works too.

In fact the two forms represent the same data but have different uris and thats the crux. Essentially the first is data.item[collection], where data.item[1] is instance 1, eg the second (0 based) item in the array.

This is no different from data.item as the json object, and data.item.1 as the instance, with the name '1'.

From a code perspective its similar too, the object just has an array of keys. But with objects data.temp.instanceName.value is reliable and always the same.

Does that apply for data.temp[1].value? eg how do you reliably get data.temp.air.value with arrays?

In signalk or java or js, if I have two values in the array, and add one, then remove the first, suddenly the subscriptions are all wrong. The temp[1] did point to the second (0 based) object in the array, but after removing the first its now temp[0]. Subscriptions to temp[1] are now broken.

For an object temp[air] always gets the temp.air. Adding or removing other keys does not affect 'air'.

The array problem can be overcome by programming - but basically thats just fixing a problem that can be easily avoided by not using arrays.

This is a quick start for any-one that would like to contribute. Its roughly from technically unskilled to skilled, top to bottom. Dont be afraid to ask for help. Each task will probably start with a new thread for more details on the Google groups (https://groups.google.com/forum/#!forum/signalk). Be patient, civil, and persistent :-)

Completely unskilled at boat electronics:

- Join https://groups.google.com/forum/#!forum/signalk - as the user base grows, so does awareness.
- Tell others, spread the word
- Fly a Signal K flag from your boat
- If you have special skills (eg motors, batteries, navigation, etc) help us extend the Signal K protocol by identifying what we need to cover.
- Ask manufacturers about Signal K support
- Ask questions about what you dont understand, and collate the answers for us to put on the website.

Can do own installs, handyman, but not IT skilled.

- Try an install of Raspberry Pi and WIFI, document exactly how you did it, so others can follow.

Website or documentation skills

- Help us maintain the website, and improve the documents

Good computer skills, but not programming

- Download and try the java server (https://github.com/SignalK/signalk-server-java) and node server (https://github.com/SignalK/signalk-server-node) and the various apps and clients. Help test and identify issues, help improve documents so others can follow easier.
- Help with User manuals!

Systems engineer

- Help other users, help with scripts, develop and maintain install processes, managing our web sites, etc.
- Examples:
  - Create Debian packages of the Signal K software for easy installation to Raspbian

Software developer

- Download and test/fix our stuff, add improvements, join the team and help code, develop support in your own software.

Microprocessors

- Improve our Arduino stuff, add your own, incorporate Signal K into your products.