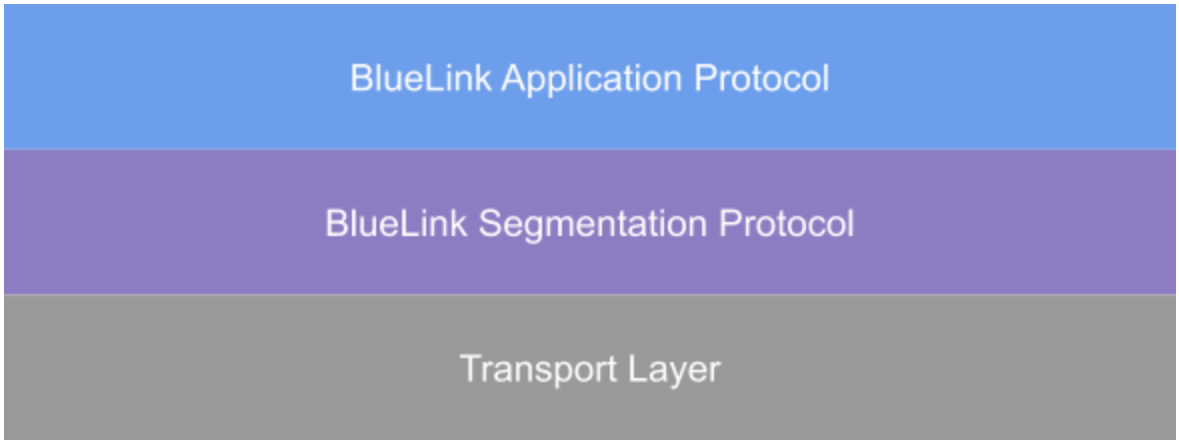
BlueLink Application Protocol

Note: please refer to https://our.internmc.facebook.com/intern/wiki/Oculus/Concept_Engineering/Projects/BlueLink/BlueLink_Protocol/ for the most up to date information.

This document describes the full specification of a BlueLink protocol implementation. **A BlueLink implementation (e.g. the Stella companion app) does NOT need to implement the full spec to be compliant.** Only the subset of the spec that is used by the client must be implemented.



Goals

Primary Goals

- [Stella] It must be transport agnostic: supporting BLE, SPI and possibly WiFi and their MTU sizes.
- [Stella] It must NOT require any additional copies on the MCU to translate or route data from one transport to the other.
- [Stella] Exposing additional resources on the SoC that should be exposed to the App must NOT require any changes to the MCU firmware or increase the MCU's code size. And vice-versa. -> To increase developer velocity and stay within the restrictive limits of the MCU.
- [Stella] The MCU should be able to route messages to the App and SoC without knowing their contents.
- [Stella] The app must be able to detect whether some functionality is supported on a potentially outdated version of the device.
- [Stella] It must be possible to implement with no allocations on the MCU.

- [Stella] It must be possible to handle/stream large messages WITHOUT imposing any full content buffering on the MCU where there is significantly less memory available than some of our message sizes require (e.g. images).
- [Stella] It must handle any link coming up and going down gracefully.
- [Stella] It must be CPU efficient -- zero copies necessary.
- [Stella] It must be bandwidth efficient -- limited overhead to the message size.
- [Stella] It must be evolvable:
    - New capabilities can come up
    - Old capabilities can be removed without any cost
    - Message schemas can evolve to include new fields or deprecate old ones
- [Stella] It must be simple enough to maintain multiple separate implementations:
    - Cross-platform C++ implementation (SoC and tools)
    - Minimal code-size, no allocation implementation in C on the MCU
    - iOS companion app in Swift
    - Android companion app in Kotlin

Secondary Goals

- [BlueBox] The BlueLink specification and implementation should be reusable for future or other projects and not be Stella-specific. As such, it should be extensible enough for our future BlueLink goals that might not seem necessary for Stella **while also NOT adding any unnecessary overhead for Stella**:
    - It should be possible to implement more dynamic systems based on this in the future without changing or affecting Stella.

Non-Goals

- This protocol doesn't handle any security/authentication/encryption. **That it assumed to be handled at a lower layer.**
- **No complicated mesh networking** needed for Stella, simple routing on the MCU will suffice.

Protocol

The BlueLink Application Protocol allows clients to use BlueBox across the device boundary. BlueBox and uBB provide an easy way to use BLAP, but it can also be implemented independently.

Commands

Commands are a set of BlueLink built-ins that are meant to setup any services/topics that need to go through the system and keep both ends of the link in sync. **This set of commands is meant to be non-exhaustive and grow in the future; however a BlueLink-compliant implementation does NOT have to support all commands to function.**

The only required command should be `connect()` which is used to initially establish a connection.

Command Requests

- **A BlueLink implementation only needs to send the commands it needs to use and set up the features it wants to use.**
- A BlueLink implementation needs to handle the possibility that the command sent is unsupported by the remote implementation.

```
union CommandRequestUnion {
  /// Listed below...
}

table CommandRequest {
  token:uint16; // Token used to match the response.
  request:CommandRequestUnion (required); // Request payload.
}
```

Command Responses

- A BlueLink implementation **should respond to all incoming command requests** with a corresponding command response.
- The `token` field must match the same `token` value specified in the incoming command request in order to match requests and responses together.
- The `result` field must match a value in the BlueBox API Result enum:
  - `OK`: If the command was successful, the `result` should be set to `OK`.
  - `NOT_IMPLEMENTED`: **If the command is not supported, the `result` should be set to `NOT_IMPLEMENTED`.**
  - `MALFORMED_BUFFER`: If the command payload was malformed and could not be decoded, the `result` should be set to `MALFORMED_BUFFER`.
  - `VERSION_NOT_SUPPORTED`: The client's BlueLink version is incompatible with the server's BlueLink version.
  - Other: If the command is supported, but was unsuccessful due to some other reason, the `result` should be set to some error code depending on the failure

type and the command specified. See specification for each command you plan to support below.

- **The `response` field need only be set if the command was successful.**

```
union CommandResponseUnion {
  /// Listed below…
}

table CommandResponse {
  token:uint16; // Token matching the corresponding request.
  result:bluebox.api.Result; // Error code.
  response:CommandResponseUnion; // Optional response payload.
}
```

Command List

Connect **[WIP]**

After a physical link is established between two BlueLink endpoints, a BlueLink session is established via a 3-way handshake. Upon reception of a `ConnectionRequest`, the receiving BlueLink endpoint shall reply with a `ConnectionResponse` if, at a minimum, communication with the remote version is allowed. **It is implementation defined when the remote will respond with a `ConnectionResponse`.** For example, a remote may wait until everything is registered on its end before completing a connection handshake. On acceptance, the endpoint receiving the ConnectResponse responds with a final `ConnectionAck` if the remote endpoint's version is also allowed.

Note: Either endpoint can complete this handshake, but a session is not established until a ConnectAckResponse is sent or received. If both endpoints happen to initiate the handshake, then the second handshake shall be still completed, but communications are not stalled by the completion of the second.

```
struct ConnectionRequest {
  version:Version;
  mtu_size:uint32;
  session:SessionId;
}
```

```
struct ConnectionResponse {
  state:ConnectionState;
```

```
  mtu_size:uint32;
  session:SessionId;
}
```

```
struct ConnectionAck {
  state:ConnectionState;
}
```

Disconnect **[WIP]**

Disconnecting automatically unregisters all child nodes, publishers, subscriptions, service servers and service clients. This can be used as a shorthand instead of sending multiple unregistrations.

```
table DisconnectRequest {}
```

```
table DisconnectResponse {}
```

Register Node

A node definition consists of *one* thing that need to be provided on registration:

- A **unique** node `name`.

If successful, the response will hold:

- A unique `node_id` for you to use if you wish to unregister it or register other resources in the future.

Failure can be characterized in the `CommandResponse`'s `result` field as follows:

- `NODE_NAME_NOT_UNIQUE`: The `name` was not unique across the system.
- `NODE_NAME_NOT_VALID`: The `name` included invalid characters.

```
table RegisterNodeRequest {
  name:string; // Unique node name.
}
```

```
table RegisterNodeResponse {
  node_id:NodeId; // To be used for future un/registration purposes.
}
```

Unregister Node

Unregistering a node only requires the `node_id` from registration.
If successful, the response will be empty.
Failure can be characterized in the `CommandResponse`'s `result` field as follows:

- `NODE_NOT_FOUND`: The `node_id` was not found.

Unregistering a node automatically unregisters all child publishers, subscriptions, service servers and service clients. This can be used as a shorthand instead of sending multiple unregistrations.

```
table UnregisterNodeRequest {
  node_id:NodeId; // From RegisterNodeResponse.
}
```

```
table UnregisterNodeResponse {}
```

Register Publisher

A topic definition consists of *three* things that need to be provided on registration:

- A parent `node_id`.
- A **unique** topic `name`.
- A message `type`.

If successful, the response will hold:

- A unique `publisher_id` for you to use if you wish to unregister in the future.
- A unique `topic_id` for you to use when publishing messages.

Failure can be characterized in the `CommandResponse`'s `result` field as follows:

- `NODE_NOT_FOUND`: The `node_id` was not found.
- `TOPIC_NAME_NOT_UNIQUE`: The `name` was not unique across the system.
- `TOPIC_NAME_NOT_VALID`: The `name` included invalid characters.

```
table RegisterPublisherRequest {
  node_id:NodeId; // From RegisterNodeResponse.
  name:string; // Unique topic name.
  type:string; // Namespaced message type name.
}
```

```
table RegisterPublisherResponse {
  publisher_id:PublisherId; // To be used for future unregistration
```

```
purposes.
  topic_id:TopicId;  // To be used for published messages.
}
```

Unregister Publisher

Unregistering a publisher only requires the `node_id` and `publisher_id` from registration.
If successful, the response will be empty.
Failure can be characterized in the `CommandResponse`'s `result` field as follows:

- `NODE_NOT_FOUND`: The `node_id` was not found.
- `PUBLISHER_NOT_FOUND`: The publisher_id was not found.

```
table UnregisterPublisherRequest {
  node_id:NodeId; // From RegisterNodeResponse.
  publisher_id:PublisherId; // From RegisterPublisherResponse.
}
```

```
table UnregisterPublisherResponse {}
```

Register Subscription

A topic definition consists of *three* things that need to be provided on registration:

- A parent `node_id`.
- A **registered** topic `name`.
- A message `type`.

If successful, the response will hold:

- A unique `subscription_id` for you to use if you wish to unregister in the future.
- A unique `topic_id` for you to use when publishing messages.

Failure can be characterized in the `CommandResponse`'s `result` field as follows:

- `NODE_NOT_FOUND`: The `node_id` was not found.
- `TOPIC_NOT_FOUND`: The `name` was not found.
- `TOPIC_NAME_NOT_VALID`: The `name` included invalid characters.
- `TOPIC_NAME_AMBIGUOUS`: The `name` is ambiguous (only applicable if a shortened `name` was provided).
- `TOPIC_TYPE_MISMATCH`: The `type` doesn't match what was registered by a publisher.

```
table RegisterSubscriptionRequest {
  node_id:NodeId; // From RegisterNodeResponse.
  name:string; // Registered topic name.
```

```
  type:string; // Namespaced message type name.
}
```

```
table RegisterSubscriptionResponse {
  subscription_id:SubscriptionId; // To be used for future unregistration
purposes.
  topic_id:TopicId; // To be used for published messages received.
}
```

Unregister Subscription

Unregistering a publisher only requires the `node_id` and `subscription_id` from registration.
If successful, the response will be empty.
Failure can be characterized in the `CommandResponse`'s `result` field as follows:

- `NODE_NOT_FOUND`: The `node_id` was not found.
- `SUBSCRIPTION_NOT_FOUND`: The `subscription_id` was not found.

```
table UnregisterSubscriptionRequest {
  node_id:NodeId; // From RegisterNodeResponse.
  subscription_id:SubscriptionId; // From RegisterSubscriptionRequest.
}
```

```
table UnregisterSubscriptionResponse {}
```

Register Service Server

A service definition consists of *four* things that need to be provided on registration:

- A parent `node_id`.
- A **unique** service `name`.
- A service `request_type`.
- A service `response_type`.

If successful, the response will hold:

- A unique `service_server_id` for you to use if you wish to unregister in the future.
- A unique `service_id` for you to use handle incoming service requests.

Failure can be characterized in the `CommandResponse`'s `result` field as follows:

- `NODE_NOT_FOUND`: The `node_id` was not found.
- `SERVICE_NAME_NOT_UNIQUE`: The `name` was not unique across the system.
- `SERVICE_NAME_NOT_VALID`: The `name` included invalid characters.

```
table RegisterServiceServerRequest {
  node_id:NodeId; // From RegisterNodeResponse.
  name:string; // Unique service name.
  request_type:string; // Namespaced request type name.
  response_type:string; // Namespaced response type name.
}
```

```
table RegisterServiceServerResponse {
  service_server_id:ServiceServerId; // To be used for future
unregistration purposes.
  service_id:ServiceId; // To be used for service requests and responses.
}
```

Unregister Service Server

Unregistering a service server only requires the `node_id` and `service_server_id` from registration.
If successful, the response will be empty.
Failure can be characterized in the `CommandResponse`'s `result` field as follows:

- `NODE_NOT_FOUND`: The `node_id` was not found.
- `SERVICE_SERVER_NOT_FOUND`: The `service_server_id` was not found.

```
table UnregisterServiceServerRequest {
  node_id:NodeId; // From RegisterNodeResponse.
  service_server_id:ServiceServerId; // From RegisterServiceServerResponse.
}
```

```
table UnregisterServiceServerResponse {}
```

Register Service Client

A service definition consists of *four* things that need to be provided on registration:

- A parent `node_id`.
- A **registered** service `name`.
- A service `request_type`.
- A service `response_type`.

If successful, the response will hold:

- A unique `service_client_id` for you to use if you wish to unregister in the future.
- A unique `service_id` for you to use handle incoming service requests.

Failure can be characterized in the `CommandResponse`'s `result` field as follows:

- `NODE_NOT_FOUND`: The `node_id` was not found.
- `SERVICE_NOT_FOUND`: The `name` was not found.
- `SERVICE_NAME_NOT_VALID`: The `name` included invalid characters.
- `SERVICE_NAME_AMBIGUOUS`: The `name` is ambiguous (only applicable if a shortened `name` was provided).
- `SERVICE_TYPE_MISMATCH`: The `request_type` and/or `response_type` don't match what was registered by a service server.

```
table RegisterServiceClientRequest {
  node_id:NodeId; // From RegisterNodeResponse.
  service:string; // Registered service name.
  request_type:string; // Namespaced request type name.
  response_type:string; // Namespaced response type name.
}
```

```
table RegisterServiceClientResponse {
  service_client_id:ServiceClientId; // To be used for future
unregistration purposes.
  service_id:ServiceId; // To be used for service requests and responses.
}
```

Unregister Service Client

Unregistering a service client only requires the `node_id` and `service_client_id` from registration.
If successful, the response will be empty.
Failure can be characterized in the `CommandResponse`'s `result` field as follows:

- `NODE_NOT_FOUND`: The `node_id` was not found.
- `SERVICE_CLIENT_NOT_FOUND`: The `service_client_id` was not found.

```
table UnregisterServiceClientRequest {
  node_id:NodeId; // From RegisterNodeResponse.
  service_client_id:ServiceClientId; // From RegisterServiceClientResponse.
}
```

```
table UnregisterServiceClientResponse {}
```

Future Commands

In the future we expect to have additional commands, that do not need to necessarily be supported.

- **Get Time**: To request the time in the remote time domain for time synchronization purposes.
- **Query**: To request a list of available resources (nodes/services/topics/etc.). This should be paginated for performance and latency, as the response size can be unbounded.

Invoking and Handling a Service

A service can be exposed by simply registering a service server (see: Register Service Server). Once a service client is registered (see: Register Service Request), it can send a serialized service request FlatBuffer payload that matches the specified `request_type` and that is **preceded by the following serialized struct**:

```
struct ServiceRequestHeader {
  service_client_id:ServiceClientId; // From a
RegisterServiceClientResponse.
  token_id:TokenId; // Token used to match the response.
  size:uint32; // Payload will directly follow.
}
```

A service server responds to a request with the associated serialized service response FlatBuffer payload that matches the specified `response_type` and that is **preceded by the following serialized struct**:

```
struct ServiceResponseHeader {
  service_client_id:ServiceClientId; // From the corresponding
ServiceRequest.
  token_id:TokenId; // Token matching the corresponding request.
  size:uint32; // Payload will directly follow.
}
```

Publishing a Message to a Topic

A topic can be exposed by simply registering a publisher (see: Register Publisher). **Only once a subscription is registered (see: Register Subscription), will the publisher start sending messages across the wire.** The message payload will be a serialized FlatBuffer matching the specified `type` and will be **preceded by the following serialized table**:

```
struct TopicMessageHeader {
  topic_id:TopicId; // Topic ID returned from a RegisterPublisherResponse.
  seq:uint16; // Message sequence number.
  nanoseconds:uint64; // Message timestamp in nanoseconds in origin time
domain.
  size:uint32; // Payload will directly follow.
}
```

IDs and Tokens

The IDs used to represent all resources after registration are simply defined as `uint16`. The protocol only requires that they're unique across a BlueLink session across all resource types (topics, services, nodes, publishers, subscriptions, service servers, service clients). **The generation of IDs is implementation defined.**

The tokens used to represent all request/response relationships are simply defined as `uint16`. The protocol only requires that they're unique across active requests of for the specific `service_id` or all active commands.**The generation of IDs is implementation defined.**