AllThingsTalk Binary Conversion Language (ABCL)

1.0.0

Intro

AllThingsTalk Binary Conversion Language (ABCL) 1.0.0 is a JSON-based, domain specific language, used for encoding and decoding AllThingsTalk asset data to and from binary payloads (e.g. LoraWAN payloads).

This document is modeled after https://states-language.net/spec.html

Table of contents

Structure of a conversion

The goal of ABCL is to provide an easy way of defining a schema for decoding freeform third party binary payloads.

A set of mappings in ABCL specific to a device type is called a Conversion. Conversions are encoded as JSON objects.

Example: Home alarm system

```
{
    "name": "alarm",
    "comment": "Home alarm system",
    "version": "1.0.0",
    "sense": [
       {"asset": "movement", "value": {"byte": 0, "type": "boolean"}}
    ],
    "actuate": [
       {"asset": "reset", "field": {"byte": 0, "type": "boolean"}}
    ]
}
```

In this example, we declare a Conversion used with a home alarm system device. This simple device contains two assets.

For device:

```
{
...
        "assets": [
                {"name": "movement", "is": "sensor", "profile": {"type": "boolean"}},
                {"name": "reset", "is": "actuator", "profile": {"type": "boolean"}}
        ]
}
```

First is a boolean sensor named “movement”, that changes its state to *True* when there’s movement in the house. From that point, “movement” stays *True*, until the second asset, a boolean actuator named “reset”, is actuated with command *True*. This resets the device, which then continues sending “movement” as *False* (heartbeat) until it senses movement again.

The payload that this device is sending (uplink payload) is a single byte. When it’s equal to *0x00*, the value of “movement” asset gets set to *False*. Otherwise, its value is set to *True*.

The payload that this device is receiving (downlink payload) is a single byte. It’s set to *0x01* when “reset” actuator is set to *True*. Otherwise, it’s set to *False*.

Top-level fields

A Conversion MUST[1] have a string field named **“name”**, whose value represents the name of the conversion and must be unique within the conversion system. For automatically generated names, using RFC 4122 UUIDs is recommended.

A Conversion MAY have a string field named **“comment”**, provided for human-readable description of the conversion.

A Conversion SHOULD have a string field named **“version”**, whose value indicates the minimum ABDCL version - using semantic versioning - that needs to be used to execute the conversion. If this value is not specified, the conversion system is allowed to make its best guess about the version, and convert appropriately.

A Conversion SHOULD have a list field named **“sense”**, which contains statements that need to be evaluated during “sensing” - when deserializing binary payloads received from the device into sensor and virtual asset values.

A Conversion SHOULD have a list field named **“actuate”**, which contains statements that need to be executed during “actuation” - when serializing data from actuator and configuration asset values into binary payloads that are sent to the device.

Statements

Statements are JSON objects that describe a single operation that needs to be performed in a conversion.

A statement MUST be either a *mapping* statement, a *control* statement, or a *comment* statement. Statements appear in *statement blocks*.

Statement blocks

Statement blocks are JSON lists whose elements are statements or statement blocks that need to be performed in order to complete the conversion.

A statement block MAY contain no elements. Making a statement block empty is the same as omitting it altogether - no statements get executed. This can be useful in code generation.

---

[1] In this document, keywords for indicating requirement levels, as specified in RFC 2119 will be used.

There are three types of statement blocks: *sense*, *actuate*, and *do*. *Sense* and *actuate* statement blocks are executed when sensing (receiving data), and actuating (sending data), and they are present in the home alarm example. *Do* is used wherever embedding a statement block - usually within control structures - is needed.

Mapping statements

Mapping statements are used for serializing and deserializing asset values and metadata to and from specific parts of binary payloads, special values, variables and constants.

The difference in mapping direction and available metadata fields make for a slight difference between mapping statements in sense block and mapping statements in actuate block, so these will be addressed separately.

In Home alarm system example, both of the following are mapping statements:

```
{"asset": "movement", "value": {"byte": 0, "type": "boolean"}}

{"asset": "reset", "field": {"byte": 0, "type": "boolean"}}
```

Locations

Locations are JSON keys in mapping statements that identify types of data sources and data destinations in a given mapping on which *selectors* in their values should be used.

```
{
    "sense": [
        {
            "asset": "movement",
            "value": {"byte": 0},
            "at": {"byte": 1, "bytelength": 4},
           "comment": "Movement mapping"
       }
    ],
    "actuate": [
        {
            "const": 3,
            "field": {"byte": 0},
           "comment": "First byte always needs to be set to 3"
       },
       {
            "asset": "reset",
            "field": {"byte": 1},
       }
    ]
}
```

In this example, we can see all available locations in *sense* and *actuate* statement blocks.

Sense block

Mapping statements in sense block MUST contain a string field **“asset”**, whose value is a *selector* that identifies the asset by name.

Mapping statements in sense block MUST contain a field **“value”**, whose value is a *selector* that identifies the data that will be stored as the new value of the asset.

Mapping statements in sense block MAY contain a field **“at”**, whose value is a *selector* that identifies the data that will be stored as the timestamp at which the data was received. If “at” is not supplied, the timestamp will be set to the time of conversion execution.

Mapping statements in sense block MAY have a string field named “comment”, provided for human-readable description of the given mapping.

Actuate block

Mapping statements in actuate block MUST contain a field **“field”**, whose value is a *selector* that identifies the location in payload (or a special variable) to which asset or constant data needs to be stored.

Mapping statements in actuate block MUST contain either a **“const”** field or an **“asset”** field.

The “asset” field’s value MUST be a valid JSON string.

The “const” field’s value MUST be a valid JSON value. If bytearray constants are needed, one should simply used backslash escapes, as described in json.org string section.

```
{"const": "\x01\xab", "comment": "This encodes a two element bytearray, [0x01, 0xAB]"}
```

Constants should be used in cases where payload needs to be filled in with payload format specific data regardless of asset values - e.g. you might want to prefix all your payloads with a magic number. Otherwise, you’ll probably want to use an asset.

Mapping statements in actuate block MAY have a string field named “comment”, provided for human-readable description of the given mapping.

Paths

Asset mappings (mappings that contain asset locations) MAY contain a JSON string field “path”, whose value is a JSON Path used to select a specific field from asset’s object value. For example, in actuate block,

```
{"asset": "a", "path": "x", "field": {"byte": 0}}
```

would set payload's first byte to a.x. So, if a was {"x": 3}, first byte would become 3.

Control statements

Control statements are used for executing control logic that MAY lead to executing more statements. *Switch* is the only available control statement in this version of ABDCL.

Example

```
{"switch": {"byte": 0}, "on": [
    {"case": 0, "do": [
       {"asset": "movement", "value": {"byte": 1}}]},
    {"case": 1, "do": [
       {"asset": "temperature", "value": {"byte": 1}}]},
    {"case": "$default", "do": [
       {"asset": "error", "const": "invalid byte 0"}]}]}
```

In this example payload's first byte is selected, and its value is used to determined if the contents of the second byte will be mapped to the asset named "movement" (if first byte is 0), or to "temperature" (if first byte is 1). If its none of those, asset "error" gets set to "invalid byte 0".

Control statement MAY have a string field named "comment", provided for human-readable description of the conversion.

Control statements MUST have a JSON object field named "switch" that specifies the *selector* that's going to be evaluated, and its value tested in *cases*.

Control statements MUST have a JSON array field named "on" that contains a list of *cases* that *switch* value will be tested on.

Control statement *on* list MAY contain zero or more *case statements*.

Control statement *on* list MAY contain a *comment statement*.

Case statements

Case statement MUST have a JSON object field named "case", whose value is a *selector* whose value is tested with the *switch* selector's value in outer switch control statement. If its equal, *do* statement block is executed.

Case statement MUST have a JSON list field named "do", whose value is a *do statement block* that is executed if case and switch match.

Case statement MAY have a string field named “comment”, provided for human-readable description of the conversion.

Comment statement

```
{"comment": "this is a comment"}
```

Comment statement MUST have a string field named “comment”.

It can be used wherever statements can be used. It is provided for human-readable descriptions and has no effect on execution.

Selectors

Selectors are JSON values. They are used to “select” data from a given location type or “select” data used in a control statement.

Selectors MAY have a string field named “comment”, provided for human-readable description of the conversion.

Const selectors

Const selectors are constant values described in the section above (actuate block).

Examples

```
3

"hello"

{"a": "dict"}

"\x01\x02"

["list", "items"]
```

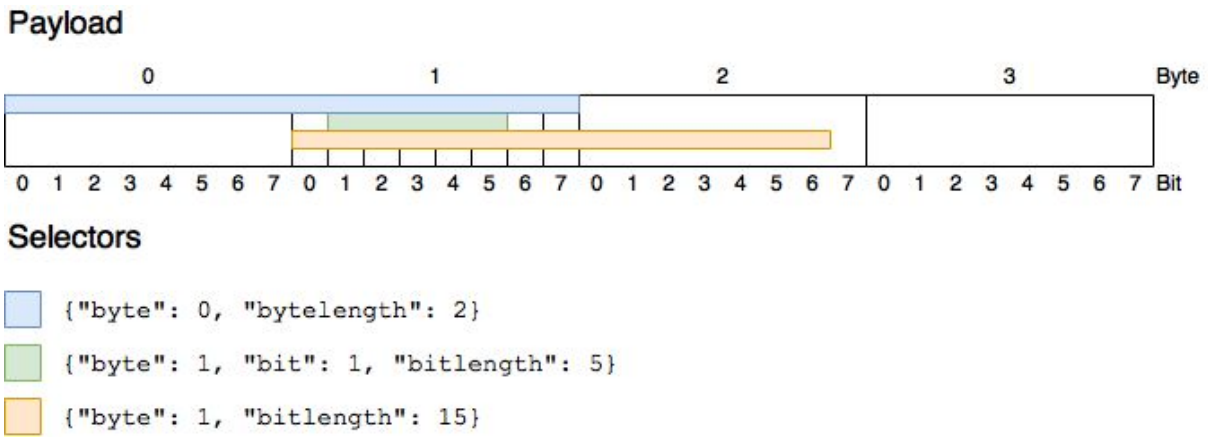
Payload selectors

Payload selectors are JSON objects that describe locations and lengths of payload chunks, that need to be read using the supplied format and extracted from the payload.

Examples

```
{"byte": 1, "bytelength": 3, "type": "integer", "signed": true}

{"byte": 3, "bit": 2, "bitlength": 8, "type": "string"}

{"byte": 4, "bytelength": 4, "byteorder": "big", "type": "number"}

{"byte": 0, "bytelength": 4, "type": "datetime", "format": "epoch"}

{"byte": 1, "bytelength": 3, "type": "integer", "signed": false}

{"byte": 7, "bytelength": 3, "type": "integer", "format": "bcd8421"}
```

Payload bit and byte order is always big-endian.



byte, bytelength, bit, bitlength

Payload selector MUST have an integer field named “byte”, whose value represents the starting byte from which the chunk is going to be selected.

Payload selector MAY have an integer field named “bytelength”, whose value represents the length of the chunk in bytes, starting from and including the byte indexed by “byte” field. It defaults to 1 (one).

Payload selector MAY have an integer field named “bit”, whose value represents the starting bit in the starting byte from which the chunk is going to be selected. It defaults to 0 (zero).

Payload selector MAY have an integer field named “bitlength”, whose value represents the length of the chunk in bits, starting from and including the bit indexed by “bit” field.
Payload selector MUST NOT contain both the “bytelength” and “bitlength”.

type, byteorder, format, signed

Payload selector SHOULD contain a string field named “type”, whose value represents the type of the value that’s going to be read from the chunk. It defaults to “integer”.

Available types are:

- **“integer”**: an integer whose bytelength is specified with “bytelength” and its sign by “signed”. The “format” can also be specified, currently just “bcd8421” which will parse data as binary coded decimal.
- **“number”**: an IEEE 754 floating point number. Its precision is determined by “bytelength”. A bytelength of 2 indicates half precision, a bytelength of 4 indicates single precision, and a bytelength of 8 indicates double precision.
- **“string”**: a UTF-8 string.
- **“boolean”**: a boolean value. If all bits in a chunk are zero, the value is *false*. Otherwise, it’s true.
- **“datetime”**: seconds since UNIX epoch. The chunk is read as a signed integer and converted to seconds.

Payload selector MAY contain a string field named “byteorder”. Supported byte orders are “big” and “little” (endian). They designate the byte order in which the selected chunk should be read. Default byte order is “big”.

Payload selector MAY contain a string field named “format”. Supported formats are type specific and are described for each type above.

Payload selector MAY contain a boolean field named “signed”. When used with “integer” type, it designates if the integer is signed or not. It defaults to *true*.

Calculations

Payload selectors MAY contain a string field “calculation”, whose value is a calculation that supports basic math operations and functions (+, -, *, /, ** for exponentiation, log, sqrt). The selected payload chunk is passed to the calculation via variable “val”, and externally, the output of the calculation becomes the final output value of the selector.

Example

```
{"asset": "a", "value": {"byte": 0, "calculation": {"val * 2"}}
```

When this mapping is executed with payload *0x02*, the first byte is read, multiplied by two, and only then stored as the value of asset “a”: 4.

Special selectors

Special selectors are JSON strings prefixed with a dollar sign (*$*), e.g. *“$asset”*. They are used for accessing the available metadata, or data available only in specific contexts. If using a raw string with the same value as one of the specified special selector names is needed (i.e. using *“$payload”* verbatim, not as the reference to the full payload) the dollar sign can be escaped with another dollar sign: *“$$payload”*.

$asset

Selects the current asset’s name. It can be used in *Actuate Block* to identify the asset that triggered the actuation conversion.

```
...
{"asset": "$asset", "field": {"byte": 1, "type": "boolean"}},
...
```

The snippet above is a bit artificial, but it does show what *$asset* can do - for any actuated actuator (it’s assumed that all of systems actuators send boolean commands), it will store its value into payload’s second byte. The code around it might be used to set the first byte to indicate which asset was actually actuated to the receiving device.

Also, *$asset* works very well with switches in actuate block, and its value will be described when we get to control blocks.

$default

Marks a default value / option. When used with switch control block’s case, it represents a case that will be executed when no other cases match.

$payload

Full contents of the received payload in *sense* block as a bytearray.

$payloadLength

Length of the received payload in *sense* block in bytes.

$meta[Port | RSSI | SNR | Timestamp | Latitude, Longitude]

Meta special variables are optionally populated by network service providers or other data suppliers.

$timestamp and $metaTimestamp

Seconds from UNIX epoch, marking the start of the conversion ($timestamp) and the timestamp optionally provided by the network service provider ($metaTimestamp). The final timestamp output is governed by the output type and format specified in mappings.

```
{"asset": "timestamp", "value": "$timestamp", "type": "datetime"}
```

Will set the state for asset named “timestamp” to current ISO 8601 datetime string.

Composite selectors

Composite selectors are used for building JSON objects or JSON lists out of simpler selectors.

They are similar to templates in that whenever a selector (payload, const or special selector) is found in a composite selector, it’s used to inject a value it selects into the outer composite selector. The way this works should become more obvious in the following examples.

Object composite selector

```
{"asset": "gps", "value": {
    "x":  {"byte":  8, "bytelength": 4, "type": "number"},
    "y":  {"byte": 12, "bytelength": 4, "type": "number"},
    "z":  {"byte": 16, "bytelength": 4, "type": "number"}}}
```

In a *sense block*, this sets the value of asset named “gps” to a JSON object, with fields x, y, z, set to listed payload chunks.

Object composite selector is a JSON object value assigned to a location, whose keys don’t indicate that the given JSON object is actually a selector (for example, a payload selector).

Object composite selectors CAN be multilevel.

Escaping

If a field needs to be named the same as one of the fields specified in language (like “byte”), it can be escaped with **&** (ampersand). If a field starting with **&** is needed, double escapes (&&) work.

Full example

For payload 0xABCD,

```
{"asset": "complex", "value": {
    "&byte": {"a": 4},
    "&&payloadLength": "$payloadLength",
    "x": {"byte": 1}}}
```

will set asset named “complex” to

```
{
    "byte": {"a": 4},
    "&payloadLength": 2,
    "x": -51
}
```

List composite selector

List composite selector functions similarly to object composite selector, except that it creates a list (which is denoted by a pair of square brackets surrounding its content).

For payload 0xABCD,

```
{"asset": "pair", "value": [{"byte": 0}, {"byte": 1}]}
```

will set asset named “pair” to

```
[171, 205]
```

Appendix A: Conversion examples

Containers


```
{
    "name": "containers",
    "version": "1.0.0",
    "sense": [
       {"switch": {"byte": 4}, "on": [
           {"case": 1, "comment": "Binary sensor", "do": [
              {"asset": "1", "value": {"byte": 5, "type": "boolean"}}]},
           {"case": 2, "comment": "Tilt sensor", "do": [
              {"asset": "2", "value": {"byte": 5, "type": "boolean"}}]},
           {"case": 3, "comment": "Push button", "do": [
              {"asset": "3", "value": {"byte": 5, "type": "boolean"}}]},
           {"case": 4, "comment": "Door sensor", "do": [
              {"asset": "4", "value": {"byte": 5, "type": "boolean"}}]},
           {"case": 5, "comment": "Temp sensor", "do": [
              {"asset": "5", "value": {"byte": 8, "bytelength":4, "type": "number"}}]},
           {"case": 6, "comment": "Light sensor", "do": [
              {"asset": "6", "value": {"byte": 8, "bytelength":4, "type": "number"}}]},
           {"case": 7, "comment": "Temp sensor", "do": [
              {"asset": "7", "value": {"byte": 5, "type": "boolean"}}]},
           {"case": 8, "comment": "Accelerometer", "do": [
              {"asset": "8", "value": {
                  "x":  {"byte":  8, "bytelength": 4, "type": "number"},
                  "y":  {"byte": 12, "bytelength": 4, "type": "number"},
                  "z":  {"byte": 16, "bytelength": 4, "type": "number"}}}]},
           {"case": 9, "comment": "GPS", "do": [
              {"asset": "9", "value": {
                  "latitude":  {"byte": 8, "bytelength": 4, "type": "number"},
                  "longitude":  {"byte": 12, "bytelength": 4, "type": "number"},
                  "altitude":  {"byte": 16, "bytelength": 4, "type": "number"}}}]},
           {"case": 10, "comment": "Pressure sensor", "do": [
              {"asset": "10", "value": {"byte": 8, "bytelength":4, "type": "number"}}]},
           {"case": 11, "comment": "Humidity sensor", "do": [
              {"asset": "11", "value": {"byte": 8, "bytelength":4, "type": "number"}}]},
           {"case": 12, "comment": "Loudness sensor", "do": [
              {"asset": "12", "value": {"byte": 8, "bytelength":4, "type": "number"}}]},
           {"case": 13, "comment": "Air quality sensor", "do": [
              {"asset": "13", "value": {"byte": 7, "bytelength":2, "type": "integer"}}]},
           {"case": 14, "comment": "Battery sensor", "do": [
              {"asset": "14", "value": {"byte": 7, "bytelength":2, "type": "integer"}}]},
           {"case": 15, "comment": "Integer sensor", "do": [
              {"asset": "15", "value": {"byte": 7, "bytelength":2, "type": "integer"}}]},
           {"case": 16, "comment": "Number sensor", "do": [
              {"asset": "16", "value": {"byte": 8, "bytelength":4, "type": "number"}}]}
       ]}
    ]
}
```