Table of Contents

Table of Contents
FIDO 2.0: Client To Authenticator Protocol
Abstract

This specification describes an application layer protocol for communication between an external authenticator and another client/platform. This protocol can be run over a variety of transport protocols using different physical media. This specification defines requirements for such transport protocols, but does not specify the details of how such transport layer connections should be set up.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

This document was published by the FIDO Alliance as a Review Draft. This document is intended to become a FIDO Alliance Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

This is a Review Draft Specification and is not intended to be a basis for any implementations as the Specification may change. Permission is hereby granted to use the Specification solely for the purpose of reviewing the Specification. No rights are granted to prepare derivative works of this Specification. Entities seeking permission to reproduce portions of this Specification for other uses must contact the FIDO Alliance to determine whether an appropriate license for such use is available.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO Alliance, Inc. and its Members and any other contributors to the Specification are not, and shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED "AS IS" AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Table of Contents

1. Overview
2. Conformance
3. Protocol Structure
4. Authenticator API
   4.1 authenticatorMakeCredential
   4.2 authenticatorGetAssertion
   4.3 authenticatorCancel
   4.4 authenticatorGetInfo
5. Message encoding
   5.1 Commands
   5.2 Responses
6. Transport-specific Bindings
   6.1 USB
      6.1.1 Design rationale
      6.1.2 Protocol structure and data framing
      6.1.3 Concurrency and channels
      6.1.4 Message and packet structure
      6.1.5 Arbitration
         6.1.5.1 Transaction atomicity, idle and busy states.
         6.1.5.2 Transaction timeout
6.1.5.3 Transaction abort and re-synchronization
6.1.5.4 Packet sequencing
6.1.6 Channel locking
6.1.7 Protocol version and compatibility
6.1.8 HID device implementation
   6.1.8.1 Interface and endpoint descriptors
   6.1.8.2 HID report descriptor and device discovery
6.1.9 CTAPHID commands
   6.1.9.1 Mandatory commands
      6.1.9.1.1 CTAPHID_MSG
      6.1.9.1.2 CTAPHID_INIT
      6.1.9.1.3 CTAPHID_PING
      6.1.9.1.4 CTAPHID_ERROR
   6.1.9.2 Optional commands
      6.1.9.2.1 CTAPHID_WINK
      6.1.9.2.2 CTAPHID_LOCK
6.1.9.3 Vendor specific commands
6.2 Near Field Communication (NFC)
   6.2.1 Conformance
   6.2.2 Protocol
   6.2.3 Applet selection
   6.2.4 Framing
      6.2.4.1 Request
      6.2.4.2 Response
   6.2.5 APDU length
6.3 Bluetooth Smart / Bluetooth Low Energy (BLE)
   6.3.1 Conformance
   6.3.2 Pairing
   6.3.3 Link Security
   6.3.4 Framing
      6.3.4.1 Request from Client to Authenticator
      6.3.4.2 Response from Authenticator to Client
   6.3.4.3 Command, Status, and Error constants
6.3.5 GATT Service Description
   6.3.5.1 FIDO 2.0 Service
   6.3.5.2 Device Information Service
   6.3.5.3 Generic Access Service
6.3.6 Protocol Overview
6.3.7 Authenticator Advertising Format
6.3.8 Requests
6.3.9 Responses
6.3.10 Framing fragmentation
6.3.11 Implementation Considerations
   6.3.11.1 Bluetooth pairing: Client considerations
   6.3.11.2 Bluetooth pairing: Authenticator considerations
   6.3.11.3 Handling command completion
   6.3.11.4 Data throughput
   6.3.11.5 Advertising
   6.3.11.6 Authenticator Address Type
7. Bibliography
A. References
   A.1 Normative references

1. Overview

This section is non-normative.

This protocol is intended to be used in scenarios where a user interacts with a relying party (a website or native app) on some platform (e.g., a PC) which prompts the user to interact with an external authenticator (e.g., a smartphone).

In order to provide evidence of user interaction, an external authenticator implementing this protocol is expected to have a mechanism to obtain a user gesture. Possible examples of user gestures include: as a consent button, password, a PIN, a biometric or a combination of these.

Prior to executing this protocol, the client/platform (referred to as host hereafter) and external authenticator (referred to as authenticator hereafter) must establish a confidential and mutually authenticated data transport channel. This specification does not specify the details of how such a channel is established, nor how transport layer security must be achieved.

2. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words must, must not, required, should, should not, recommended, may, and optional in this specification are to be interpreted as described in [RFC2119].

3. Protocol Structure

This section is non-normative.

This protocol is specified in three parts:
- Authenticator API: At this level of abstraction, each authenticator operation is defined similarly to an API call - it accepts input parameters and returns either an output or error code. Note that this API level is conceptual and does not represent actual APIs. The actual APIs will be provided by each implementing platform.

- Message Encoding: In order to invoke a method in the authenticator API, the host must construct and encode a request and send it to the authenticator over the chosen transport protocol. The authenticator will then process the request and return an encoded response.

- Transport-specific Binding: Requests and responses are conveyed to external authenticators over specific transports (e.g., USB, NFC, Bluetooth). For each transport technology, message bindings are specified for this protocol.

This document specifies all three of the above pieces for external FIDO 2.0 authenticators.

4. Authenticator API

Each operation in the authenticator API can be performed independently of the others, and all operations are asynchronous. The authenticator may enforce a limit on outstanding operations to limit resource usage - in this case, the authenticator is expected to return a busy status and the host is expected to retry the operation later. Additionally, this protocol does not enforce in-order or reliable delivery of requests and responses; if these properties are desired, they must be provided by the underlying transport protocol or implemented at a higher layer by applications.

Note that this API level is conceptual and does not represent actual APIs. The actual APIs will be provided by each implementing platform.

The authenticator API has the following methods and data structures.

4.1 authenticatorMakeCredential

This method is invoked by the host to request generation of a new credential in the authenticator. It takes the following input parameters:

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Data type</th>
<th>Required?</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpd</td>
<td>String</td>
<td>Required</td>
<td>Identity of the relying party. See [FIDOPlatformApiReq]</td>
</tr>
<tr>
<td>clientDataHash</td>
<td>Byte Array</td>
<td>Required</td>
<td>Hash of the ClientData contextual binding specified by host. See [FIDOSignatureFormat]</td>
</tr>
<tr>
<td>accountInformation</td>
<td>AccountInfo</td>
<td>Required</td>
<td>Friendly UI details to be used by the authenticator when displaying the credential to the user for selection and usage authorization. See [FIDOWebApi] for AccountInfo type specification.</td>
</tr>
<tr>
<td>cryptoParameters</td>
<td>sequence of FIDOCredentialParameters</td>
<td>Required</td>
<td>A sequence of FIDOCredentialParameters structures, as specified in [FIDOWebApi].</td>
</tr>
<tr>
<td>blacklist</td>
<td>Sequence of Credentials</td>
<td>Optional</td>
<td>A sequence of Credential structures, as specified in [FIDOWebApi]. The authenticator is requested to return an error (see Section TBD) if it recognizes any of them.</td>
</tr>
<tr>
<td>extensions</td>
<td>FIDOExtensions</td>
<td>Optional</td>
<td>Parameters to influence authenticator operation. These parameters might be authenticator specific.</td>
</tr>
</tbody>
</table>

When such a request is received, the authenticator performs the following procedure:

1. If the blacklist parameter is present and contains a credential ID that is present on this authenticator, terminate this procedure and return error code TBD.
2. If the cryptoParameters parameter does not contain a valid AlgorithmIdentifier structure that is supported by the authenticator, terminate this procedure and return error code TBD.
3. Optionally, if the extensions parameter is present, process any extensions that this authenticator supports.
4. If the authenticator has a display, show the contents of the accountInformation and rpd parameters to the user. Request permission to create a credential. If the user declines permission, return an error code.
5. Generate a new cryptographic key pair for the algorithm specified.
6. Associate the newly-created key pair with the rpd.
7. Generate an attestation statement for the newly-created key using clientDataHash.

On success, the authenticator must return the following structure in its response:

<table>
<thead>
<tr>
<th>Member name</th>
<th>Data type</th>
<th>Required?</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>credential</td>
<td>Credential</td>
<td>Required</td>
<td>A credential type and a byte string that must be used by the host to identify this key for future operations. From the perspective of the host, this is simply an opaque identifier for the key.</td>
</tr>
<tr>
<td>publicKey</td>
<td><em>BYTEARRAY</em></td>
<td>Required</td>
<td>The DER encoding of the SubjectPublicKeyInfo structure from [RFC5280] (Section 4.1.2.7) generated for the new credential.</td>
</tr>
<tr>
<td>rawAttestation</td>
<td><em>BYTEARRAY</em></td>
<td>Optional</td>
<td>The raw attestation statement. Its structure is opaque to the Platform/Client. See [FIDOKeyAttestation] for structure details.</td>
</tr>
</tbody>
</table>

4.2 authenticatorGetAssertion

This method is used by a host to request cryptographic proof of user authentication as well as user consent to a given transaction, using a previously generated credential that is stored by the authenticator. Provide it takes the following input parameters:

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Data type</th>
<th>Required?</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpd</td>
<td>String</td>
<td>Required</td>
<td>Identity of the relying party. See [FIDOPlatformApiReq]</td>
</tr>
<tr>
<td>clientDataHash</td>
<td><em>BYTEARRAY</em></td>
<td>Required</td>
<td>Hash of the ClientData contextual binding specified by host. See [FIDOSignatureFormat]</td>
</tr>
<tr>
<td>whitelist</td>
<td>Sequence of Credentials</td>
<td>Optional</td>
<td>A sequence of Credential structures, as specified in [FIDOWebApi]. The authenticator is requested to only generate a FIDOAssertion using one of them.</td>
</tr>
<tr>
<td>extensions</td>
<td>FIDOExtensions</td>
<td>Optional</td>
<td>Parameters to influence authenticator operation. These parameters might be authenticator specific.</td>
</tr>
</tbody>
</table>

When such a request is received, the authenticator performs the following procedure:

1. Locate all FIDO 2.0 credentials that are eligible for retrieval under the specified criteria:
   - If a whitelist is present and is non-empty, locate all mentioned credentials which are present on this authenticator. Discard any for credential which is not bound to the specified rpd.
   - If a whitelist is not present, locate all credentials which are present on this authenticator and which are bound to the specified rpd.
2. Optionally, if the extensions parameter is present, process any extensions that this authenticator supports.
3. Display all these credentials to the user, using their friendly name. If the authenticator stored an AccountInfo object during the creation of this credential, display the stored account information. Also, display the rpd of the requester (specified in the request) and ask the user to select a credential. If the user declines to select a credential or takes too long (as determined by the authenticator), terminate this procedure and
return an error code.
4. If the user selects a credential, sign the clientDataHash with it, using the structure specified in [FIDOSignatureFormat].

On success, the authenticator must return the following structure in its response:

<table>
<thead>
<tr>
<th>Member name</th>
<th>Data type</th>
<th>Required?</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>credential</td>
<td>Credential</td>
<td>Optional</td>
<td>Credential whose private key was used to generate the assertion. May be omitted if the whitelist has exactly one Credential.</td>
</tr>
<tr>
<td>authenticatorData</td>
<td>Byte Array</td>
<td>Required</td>
<td>Authenticator's raw contextual binding, as specified in [FIDOSignatureFormat].</td>
</tr>
<tr>
<td>signature</td>
<td>Byte Array</td>
<td>Required</td>
<td>Raw signature from the authenticator, as specified in [FIDOSignatureFormat].</td>
</tr>
</tbody>
</table>

4.3 authenticatorCancel

Using this method, the host can request the authenticator to cancel all ongoing operations return to a ready state. It takes no input parameters and returns success or failure.

4.4 authenticatorGetInfo

Using this method, the host can request that the authenticator report a list of all supported protocol versions (currently, "FIDO_2_0" is the only supported version) and extensions. This method takes no inputs.

On success, the authenticator must return:

<table>
<thead>
<tr>
<th>Member name</th>
<th>Data type</th>
<th>Required?</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>versions</td>
<td>Sequence of strings</td>
<td>Required</td>
<td>List of supported versions.</td>
</tr>
<tr>
<td>extensions</td>
<td>Sequence of strings</td>
<td>Optional</td>
<td>List of supported extensions.</td>
</tr>
<tr>
<td>aaguid</td>
<td>String</td>
<td>Optional</td>
<td>The claimed AAGUID.</td>
</tr>
</tbody>
</table>

5. Message encoding

Many transports (e.g., Bluetooth Smart) are bandwidth-constrained, and serialization formats such as JSON are too heavy-weight for such environments. For this reason, all encoding is done using the concise binary encoding CBOR [RFC7049].

Messages from the host to authenticator are called "commands" and messages from authenticator to host are called "replies". All values are big endian encoded.

5.1 Commands

All commands are structured as:

<table>
<thead>
<tr>
<th>Name</th>
<th>Length</th>
<th>Required?</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command Value</td>
<td>1 byte</td>
<td>Required</td>
<td>The value of the command to execute</td>
</tr>
<tr>
<td>Command Parameters</td>
<td>variable</td>
<td>Optional</td>
<td>CBOR [RFC7049] encoded set of parameters. Some commands have parameters, while others do not (see below)</td>
</tr>
</tbody>
</table>

The assigned values for commands and their descriptions are:

<table>
<thead>
<tr>
<th>Command Name</th>
<th>Command Value</th>
<th>Has parameters?</th>
</tr>
</thead>
<tbody>
<tr>
<td>authenticatorMakeCredential</td>
<td>0x01</td>
<td>yes</td>
</tr>
<tr>
<td>authenticatorGetAssertion</td>
<td>0x02</td>
<td>yes</td>
</tr>
<tr>
<td>authenticatorCancel</td>
<td>0x03</td>
<td>no</td>
</tr>
<tr>
<td>authenticatorGetInfo</td>
<td>0x04</td>
<td>no</td>
</tr>
</tbody>
</table>

Command parameters are encoded using a CBOR map (CBOR major type 5). The CBOR map must be encoded using the definite length variant.

Some commands have optional parameters. Therefore, the length of the parameter map for these commands may vary. For example, authenticatorMakeCredential may have 4, 5, or 6 parameters, while authenticatorGetAssertion may have 2, 3, or 4 parameters.

All command parameters are CBOR encoded following the JSON to CBOR conversion procedures as per the CBOR specification [RFC7049]. Specifically, parameters that are represented as DOM objects in the Authenticator API layers (formally defined in the Web API [FIDOWebApi]) are converted first to JSON and subsequently to CBOR.

**EXAMPLE 1**

An AccountInfo DOM object defined as follows:

```js
var userAccountInformation = {
  rpDisplayName: "ACME",
  displayName: "John P. Smith",
  name: "johnpsmith@example.com",
  id: "109823723409872",
  imageUri: "https://pics.acme.com/00/p/aBjjjpqPb.png"
};
```

would be CBOR encoded as follows:

```
45 # map(5)
 6d 7270446973706c61794e616d65 # "rpDisplayName"
 64 41636d65 # "Acme"
 64 726973706c61794e616d65 # "displayName"
 7b 646973706c61794e616d65 # "name"
 6d 64af686e20502e20536d697468 # "johnpsmith@example.com"
 64 6e616d65 # "name"
 64 62 # id
```
For each command that contains parameters, the parameter map keys and value types are specified below:

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameter Name</th>
<th>Key</th>
<th>Value type</th>
</tr>
</thead>
<tbody>
<tr>
<td>authenticatorMakeCredential</td>
<td>rpId</td>
<td>0x01</td>
<td>UTF-8 encoded text string (CBOR major type 3).</td>
</tr>
<tr>
<td></td>
<td>clientDataHash</td>
<td>0x02</td>
<td>byte string (CBOR major type 2).</td>
</tr>
<tr>
<td>accountInformation</td>
<td></td>
<td>0x03</td>
<td>CBOR definite length map (CBOR major type 5).</td>
</tr>
<tr>
<td>cryptoParameters</td>
<td></td>
<td>0x04</td>
<td>CBOR definite length array (CBOR major type 4) of CBOR definite length maps (CBOR major type 5).</td>
</tr>
<tr>
<td>blacklist</td>
<td></td>
<td>0x05</td>
<td>CBOR definite length array (CBOR major type 4) of CBOR definite length maps (CBOR major type 5).</td>
</tr>
<tr>
<td>extensions</td>
<td></td>
<td>0x06</td>
<td>CBOR definite length map (CBOR major type 5).</td>
</tr>
<tr>
<td>authenticatorGetAssertion</td>
<td>rpId</td>
<td>0x01</td>
<td>UTF-8 encoded text string (CBOR major type 3).</td>
</tr>
<tr>
<td></td>
<td>clientDataHash</td>
<td>0x02</td>
<td>byte string (CBOR major type 2).</td>
</tr>
<tr>
<td>whitelist</td>
<td></td>
<td>0x03</td>
<td>CBOR definite length array (CBOR major type 4) of CBOR definite length maps (CBOR major type 5).</td>
</tr>
<tr>
<td>extensions</td>
<td></td>
<td>0x04</td>
<td>CBOR definite length map (CBOR major type 5).</td>
</tr>
</tbody>
</table>

**Example 2**

A DOM object that is a sequence of FIDOCredentialParameters defined as follows:

```javascript
var cryptoParams = [
  {
    type: "FIDO",
    algorithm: "ES256",
  },
  {
    type: "FIDO",
    algorithm: "RS256",
  }
];
```

would be CBOR encoded as:

```
82  # array(2)
a2  # map(2)
64  # text(4)
74797065  # "ES256"
6a  # text(10)
53636f70666443726564  # "ScopedCred"
69  # text(9)
616c6f7269746869  # "algorithm"
65  # text(5)
4553323536  # "RS256"
```

**Example 3**

The following is a complete encoding example of the `authenticatorMakeCredential` command (using same account and crypto parameters as above) and the corresponding `authenticatorMakeCredential_Response` response:

```
01  # authenticatorMakeCredential command
  a4  # map(4)
  01  # unsigned(1) -- rpId
  68  # "rpId"
  61636d652e636f6d65  # "acme.com"
  02  # unsigned(2) -- clientDataHash
  2420  # byte string(32)
  5a81483d96b0bc15ad19af7fa5662e14b275729fbc05579b18513e7f555001db1  # ...
  03  # unsigned(3) -- accountInformation
  04  # unsigned(4) -- cryptoParameters
  05  # "accountInformation"
  6d  # text(13)
  7270446973706c61794e616d65  # "rpDisplayName"
  64  # text(4)
  41636d65  # "Acme"
  6b  # text(11)
  646973706c61794e616d65  # "displayName"
  60  # text(13)
  4a666866e20502e20536d697468  # "John P. Smith"
  64  # text(4)
  6e616d65  # "name"
  76  # text(22)
  6a666866e70736e6974684065786973686976657273737970617274696e67  # "johnpsmith@example.com"
  62  # text(2)
  6c65666666666666  # "...
  6964  # "id"
  70  # unsigned(16)
  31303938323337323333343039383732  # "109823735409872"  
  68  # "timeStamp"
  966d61676555555e24  # "imageURL"
  7828  # text(40)
  68747470733a2f2f706963732e61636d652e636f6d623f30302f702f61426a6a6a6a6a707150622e706e67  # "https://pics.acme.com/00/p/abjjjpngPb.png"  
  04  # ...
  02  # map(4)
  64  # "authenticatorGetAssertion"
  74797065  # "ES256"
```
EXAMPLE 4

The following is a complete encoding example of the authenticatorGetAssertion command and the corresponding authenticatorMakeCredential_Response response:

```
02 a3
01 68
16366d526ea3666d6f
02 7640
53636770656443726564
6a
53636770656443726564
65
616067726974686d
65
453323536
64
74797065
6a
53636770656443726564
69
616067726974686d
65
525332356
authenticatorGetAssertion_Response response:
00 a3
01 64
74797065
6a
53636770656443726564
62
6964
7824
3844437341334442d45435332d2437
3442d41303542d6444444238323842
3636342
02 a5
63
67479
63
525341
63
616067
65
525332356
65
567874
20
6a
790106
6c0d52345b67785b695356b757657748
6848315c44b65674a48343255575049
496d60453269554d535664333257
5755486465f93546656576556763
567556634e747746df797341463635
505a7e14d4b856152d2d368146562
4b7e65387154742d6334795930456650
39393662383202466323373244a
52465674455262347172856343d78
5a487537562a473e4669762492868
464b7a5697394563644506744a7978
314e700453469625377f5445466335
59a476646473337642d236165396f
464f6835a66e68492753556b6664
44d051784f4865072d4214230d456
435359355374765f5a5f537146444f
4c79655768787878c6c2677334b39
797f513386674434a4b695771372d
45467a7461393052343635784741
4516f49434b6e327634e46d2715648
406c5746624d50372d838d4f4d7865
664d32373777
03 65
64
4354142
03 22
```

```
# authenticatorGetAssertion command
# map(3) # unsigned(1) -- rpId
# map(8) # "aawm.com"
# unsigned(2) -- clientDataHash
# map(64) # "TPMOO" # byte string
# "e3b044298f0e4c194c44d500a11c94bcf155f442d40" # base64
# status = success
# map(4) # unsigned(1) -- credential
# map(12) # "RS256"
# "id"
# "ScopedCred"
# map(5) # "ext"
# "false"
# "null"
```
5.2 Responses

All responses are structured as:

<table>
<thead>
<tr>
<th>Name</th>
<th>Length</th>
<th>Required?</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>1 byte</td>
<td>Required</td>
<td>The status of the response. 0x00 means success; all other values are errors. See table TBD for error values.</td>
</tr>
<tr>
<td>Response Data</td>
<td>variable</td>
<td>Optional</td>
<td>CBOR encoded set of values.</td>
</tr>
</tbody>
</table>

Response data is encoded using a CBOR map (CBOR major type 5). The CBOR map must be encoded using the definite length variant.

For each response message, the map keys and value types are specified below:

<table>
<thead>
<tr>
<th>Response Message</th>
<th>Member Name</th>
<th>Key</th>
<th>Value type</th>
</tr>
</thead>
<tbody>
<tr>
<td>authenticatorMakeCredential_Response</td>
<td>credential</td>
<td>0x01</td>
<td>CBOR definite length map (CBOR major type 5).</td>
</tr>
<tr>
<td></td>
<td>credentialPublicKey</td>
<td>0x02</td>
<td>byte string (CBOR major type 2).</td>
</tr>
<tr>
<td></td>
<td>rawAttestation</td>
<td>0x03</td>
<td>byte string (CBOR major type 2).</td>
</tr>
<tr>
<td>authenticatorGetAssertion_Response</td>
<td>credential</td>
<td>0x01</td>
<td>CBOR definite length map (CBOR major type 5).</td>
</tr>
<tr>
<td></td>
<td>authenticatorData</td>
<td>0x02</td>
<td>byte string (CBOR major type 2).</td>
</tr>
<tr>
<td></td>
<td>signature</td>
<td>0x03</td>
<td>byte string (CBOR major type 2).</td>
</tr>
<tr>
<td>authenticatorGetInfo_Response</td>
<td>versions</td>
<td>0x01</td>
<td>CBOR definite length array (CBOR major type 4) of UTF-8 encoded strings (CBOR major type 3).</td>
</tr>
<tr>
<td></td>
<td>extensions</td>
<td>0x02</td>
<td>CBOR definite length array (CBOR major type 4) of UTF-8 encoded strings (CBOR major type 3).</td>
</tr>
<tr>
<td></td>
<td>aaguid</td>
<td>0x03</td>
<td>CBOR UTF-8 encoded string (CBOR major type 3).</td>
</tr>
</tbody>
</table>

6. Transport-specific Bindings

6.1 USB

6.1.1 Design rationale

CTAP messages are framed for USB transport using the HID (Human Interface Device) protocol. We henceforth refer to the protocol as CTAPHID. The CTAPHID protocol is designed with the following design objectives in mind:

- Driver-less installation on all major host platforms
- Multi-application support with concurrent application access without the need for serialization and centralized dispatching.
- Fixed latency response and low protocol overhead
- Scalable method for CTAPHID device discovery

Since HID data is sent as interrupt packets and multiple applications may access the HID stack at once, a non-trivial level of complexity has to be added to handle this.

6.1.2 Protocol structure and data framing

The CTAP protocol is designed to be concurrent and state-less in such a way that each performed function is not dependent on previous actions. However, there has to be some form of “atomicity” that varies between the characteristics of the underlying transport protocol, which for the CTAPHID protocol introduces the following terminology:

- Transaction
- Message
- Packet

A transaction is the highest level of aggregated functionality, which in turn consists of a request, followed by a response message. Once a request has been initiated, the transaction has to be entirely completed before a second transaction can take place and a response is never sent without a previous request.

Request and response messages are in turn divided into individual fragments, known as packets. The packet is the smallest form of protocol data unit, which in the case of CTAPHID are mapped into HID reports.
6.1.3 Concurrency and channels

Additional logic and overhead is required to allow a CTAPHID device to deal with multiple "clients", i.e. multiple applications accessing the single resource through the HID stack. Each client communicates with a CTAPHID device through a logical channel, where each application uses a unique 32-bit channel identifier for routing and arbitration purposes.

A channel identifier is allocated by the FIDO authenticator device to ensure its system-wide uniqueness. The actual algorithm for generation of channel identifiers is vendor specific and not defined by this specification.

Channel ID 0 is reserved and $\text{FFFFFFFF}$ is reserved for broadcast commands, i.e. at the time of channel allocation.

6.1.4 Message and packet structure

Packets are one of two types, initialization packets and continuation packets. As the name suggests, the first packet sent in a message is an initialization packet, which also becomes the start of a transaction. If the entire message does not fit into one packet (including the CTAPHID protocol overhead), one or more continuation packets have to be sent in strict ascending order to complete the message transfer.

A message sent from a host to a device is known as request and a message sent from a device back to the host is known as response. A request always triggers a response and response messages are never sent ad-hoc, i.e. without a prior request message.

The request and response messages have an identical structure. A transaction is started with the initialization packet of the request message and ends with the last packet of the response message.

Packets are always fixed size (defined by the endpoint and HID report descriptors) and although all bytes may not be needed in a particular packet, the full size always has to be sent. Unused bytes should be set to zero.

An initialization packet is defined as

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>CID</td>
<td>Channel identifier</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>CMD</td>
<td>Command identifier (bit 7 always set)</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>BCNTH</td>
<td>High part of payload length</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>BCNTL</td>
<td>Low part of payload length</td>
</tr>
<tr>
<td>7</td>
<td>(s - 7)</td>
<td>DATA</td>
<td>Payload data (s is equal to the fixed packet size)</td>
</tr>
</tbody>
</table>

The command byte has always the highest bit set to distinguish it from a continuation packet, which is described below.

A continuation packet is defined as

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>CID</td>
<td>Channel identifier</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>SEQ</td>
<td>Packet sequence 0x00..0x7f (bit 7 always cleared)</td>
</tr>
<tr>
<td>5</td>
<td>(s - 5)</td>
<td>DATA</td>
<td>Payload data (s is equal to the fixed packet size)</td>
</tr>
</tbody>
</table>

With this approach, a message with a payload less or equal to (s - 7) may be sent as one packet. A larger message is then divided into one or more continuation packets, starting with sequence number 0, which then increments by one to a maximum of 127.

With a packet size of 64 bytes (max for full-speed devices), this means that the maximum message payload length is $64 - 7 + 128 \times (64 - 5) = 7609$ bytes.

6.1.5 Arbitration

In order to handle multiple channels and clients concurrency, the CTAPHID protocol has to maintain certain internal states, block conflicting requests and maintain protocol integrity. The protocol relies on each client application (channel) behaves politely, i.e. does not actively act to destroy for other channels. With this said, a malign or malfunctioning application can cause issues for other channels. Expected errors and potentially stalling applications should however be handled properly.

6.1.5.1 Transaction atomicity, idle and busy states.

A transaction always consists of three stages:

1. A message is sent from the host to the device
2. The device processes the message
3. A response is sent back from the device to the host

The protocol is built on the assumption that a plurality of concurrent applications may try ad-hoc to perform transactions at any time, with each transaction being atomic, i.e. it cannot be interrupted by another application once started.

The application channel that manages to get through the first initialization packet when the device is in idle state will keep the device locked for other channels until the last packet of the response message has been received. The device then returns to idle state, ready to perform another transaction for the same or a different channel. Between two transactions, no state is maintained in the device and a host application must assume that any other process may execute other transactions at any time.

If an application tries to access the device from a different channel while the device is busy with a transaction, that request will immediately fail with a busy-error message sent to the requesting channel.

6.1.5.2 Transaction timeout

A transaction has to be completed within a specified period of time to prevent a stalling application to cause the device to be completely locked out for access by other applications. If for example an application sends an initialization packet that signals that continuation packets will follow and that application crashes, the device will back out that pending channel request and return to an idle state.

6.1.5.3 Transaction abort and re-synchronization

If an application for any reason "gets lost", gets an unexpected response or error, it may at any time issue an abort-and-resynchronize command. If the device detects a SYNC command during a transaction that has the same channel id as the active transaction, the transaction is aborted (if
possible) and all buffered data flushed (if any). The device then returns to idle state to become ready for a new transaction.

### 6.1.5.4 Packet sequencing

The device keeps track of packets arriving in correct and ascending order and that no expected packets are missing. The device will continue to assemble a message until all parts of it has been received or that the transaction times out. Spurious continuation packets appearing without a prior initialization packet will be ignored.

### 6.1.6 Channel locking

In order to deal with aggregated transactions that may not be interrupted, such as tunneling of vendor-specific commands, a channel lock command may be implemented. By sending a channel lock command, the device prevents other channels from communicating with the device until the channel lock has timed out or been explicitly unlocked by the application.

This feature is optional and has not to be considered by general CTAP HID applications.

### 6.1.7 Protocol version and compatibility

The CTAPHID protocol is designed to be extensible, yet maintaining backwards compatibility to the extent it is applicable. This means that a CTAPHID host shall support any version of a device with the command set available in that particular version.

### 6.1.8 HID device implementation

This description assumes knowledge of the USB and HID specifications and is intended to provide the basics for implementing a CTAPHID device. There are several ways to implement USB devices and reviewing these different methods is beyond the scope of this document. This specification targets the interface part, where a device is regarded as either a single or multiple interface (composite) device.

The description further assumes (but is not limited to) a full-speed USB device (12 Mbit/s). Although not excluded per se, USB low-speed devices are not practical to use given the 8-byte report size limitation together with the protocol overhead.

#### 6.1.8.1 Interface and endpoint descriptors

The device implements two endpoints (except the control endpoint 0), one for IN and one for OUT transfers. The packet size is vendor defined, but the reference implementation assumes a full-speed device with two 64-byte endpoints.

**Interface Descriptor**

```
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bNumEndpoints</td>
<td>2</td>
<td>One IN and one OUT endpoint</td>
</tr>
<tr>
<td>bInterfaceClass</td>
<td>0x03</td>
<td>HID</td>
</tr>
<tr>
<td>bInterfaceSubClass</td>
<td>0x00</td>
<td>No interface subclass</td>
</tr>
<tr>
<td>bInterfaceProtocol</td>
<td>0x00</td>
<td>No interface protocol</td>
</tr>
</tbody>
</table>
```

**Endpoint 1 descriptor**

```
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bmAttributes</td>
<td>0x03</td>
<td>Interrupt transfer</td>
</tr>
<tr>
<td>bEndpointAddress</td>
<td>0x01</td>
<td>1, OUT</td>
</tr>
<tr>
<td>bMaxPacketSize</td>
<td>64</td>
<td>64-byte packet max</td>
</tr>
<tr>
<td>bInterval</td>
<td>5</td>
<td>Poll every 5 millisecond</td>
</tr>
</tbody>
</table>
```

**Endpoint 2 descriptor**

```
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bmAttributes</td>
<td>0x03</td>
<td>Interrupt transfer</td>
</tr>
<tr>
<td>bEndpointAddress</td>
<td>0x81</td>
<td>1, IN</td>
</tr>
<tr>
<td>bMaxPacketSize</td>
<td>64</td>
<td>64-byte packet max</td>
</tr>
<tr>
<td>bInterval</td>
<td>5</td>
<td>Poll every 5 millisecond</td>
</tr>
</tbody>
</table>
```

The actual endpoint order, intervals, endpoint numbers and endpoint packet size may be defined freely by the vendor and the host application is responsible for querying these values and handle these accordingly. For the sake of clarity, the values listed above are used in the following examples.

#### 6.1.8.2 HID report descriptor and device discovery

A HID report descriptor is required for all HID devices, even though the reports and their interpretation (scope, range, etc.) makes very little sense from an operating system perspective. The CTAPHID just provides two "raw" reports, which basically map directly to the IN and OUT endpoints. However, the HID report descriptor has an important purpose in CTAPHID, as it is used for device discovery.

For the sake of clarity, a bit of high-level C-style abstraction is provided

```c
// HID report descriptor
const uint8_t HID_ReportDescriptor[] = {
    HID_UsagePage (FIDO_USAGE_PAGE),
    HID_Usage (FIDO_USAGE_CTAPHID),
    HID_Collection (HID_Application),
    HID_Usage (FIDO_USAGE_DATA_IN),
    HID_LogicalMin (0),
    HID_LogicalMaxS (0xff),
    HID_ReportSize (8),
};
```
A unique Usage Page is defined for the FIDO alliance and under this realm, a CTAPHID Usage is defined as well. During CTAPHID device discovery, all HID devices present in the system are examined and devices that match this usage pages and usage are then considered to be CTAPHID devices.

The length values specified by the HID_INPUT_REPORT_BYTES and the HID_OUTPUT_REPORT_BYTES should typically match the respective endpoint sizes defined in the endpoint descriptors.

6.1.9 CTAPHID commands

The CTAPHID protocol implements the following commands.

6.1.9.1 Mandatory commands

The following list describes the minimum set of commands required by an CTAPHID device. Optional and vendor-specific commands may be implemented as described in respective sections of this document.

6.1.9.1.1 CTAPHID_MSG

This command sends an encapsulated CTAP message to the device. The semantics of the data message is defined in the CTAP/CBOR data encoding specification.

**Request**

<table>
<thead>
<tr>
<th>CMD</th>
<th>CTAPHID_MSG</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCNT</td>
<td>1..(n + 1)</td>
</tr>
<tr>
<td>DATA</td>
<td>CTAP command byte</td>
</tr>
<tr>
<td>DATA+1</td>
<td>n bytes of CBOR encoded data</td>
</tr>
</tbody>
</table>

**Response at success**

<table>
<thead>
<tr>
<th>CMD</th>
<th>CTAPHID_MSG</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCNT</td>
<td>1..(n + 1)</td>
</tr>
<tr>
<td>DATA</td>
<td>CTAP status code</td>
</tr>
<tr>
<td>DATA+1</td>
<td>n bytes of CBOR encoded data</td>
</tr>
</tbody>
</table>

6.1.9.1.2 CTAPHID_INIT

This command synchronizes a channel and optionally requests the device to allocate a unique 32-bit channel identifier (CID) that can be used by the requesting application during its lifetime. The requesting application generates a nonce that is used to match the response. When the response is received, the application compares the sent nonce with the received one. After a positive match, the application stores the received channel id and uses that for subsequent transactions.

To allocate a new channel, the requesting application shall use the broadcast channel CTAPHID_BROADCAST_CID. The device then responds the newly allocated channel in the response, using the broadcast channel.

**Request**

<table>
<thead>
<tr>
<th>CMD</th>
<th>CTAPHID_INIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCNT</td>
<td>8</td>
</tr>
<tr>
<td>DATA</td>
<td>8-byte nonce</td>
</tr>
</tbody>
</table>

**Response at success**

<table>
<thead>
<tr>
<th>CMD</th>
<th>CTAPHID_INIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCNT</td>
<td>17 (see note below)</td>
</tr>
<tr>
<td>DATA</td>
<td>8-byte nonce</td>
</tr>
<tr>
<td>DATA+8</td>
<td>4-byte channel ID</td>
</tr>
<tr>
<td>DATA+12</td>
<td>CTAPHID protocol version identifier</td>
</tr>
<tr>
<td>DATA+13</td>
<td>Major device version number</td>
</tr>
<tr>
<td>DATA+14</td>
<td>Minor device version number</td>
</tr>
<tr>
<td>DATA+15</td>
<td>Build device version number</td>
</tr>
<tr>
<td>DATA+16</td>
<td>Capabilities flags</td>
</tr>
</tbody>
</table>

The protocol version identifies the protocol version implemented by the device. An CTAPHID host shall accept a response size that is longer than the anticipated size to allow for future extensions of the protocol, yet maintaining backwards compatibility. Future versions will maintain the
response structure to this current version, but additional fields may be added.
The meaning and interpretation of the version number is vendor defined.
The following device capabilities flags are defined. Unused values are reserved for future use and must be set to zero by device vendors.

| CAPABILITY_WINK | Implements the WINK function |

6.1.9.1.3 CTAPHID_PING

Sends a transaction to the device, which immediately echoes the same data back. This command is defined to be a uniform function for debugging, latency and performance measurements.

Request

<table>
<thead>
<tr>
<th>CMD</th>
<th>CTAPHID_PING</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCNT</td>
<td>0..n</td>
</tr>
<tr>
<td>DATA</td>
<td>n bytes</td>
</tr>
</tbody>
</table>

Response at success

<table>
<thead>
<tr>
<th>CMD</th>
<th>CTAPHID_PING</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCNT</td>
<td>n</td>
</tr>
<tr>
<td>DATA</td>
<td>N bytes</td>
</tr>
</tbody>
</table>

6.1.9.1.4 CTAPHID_ERROR

This command code is used in response messages only.

<table>
<thead>
<tr>
<th>CMD</th>
<th>CTAPHID_ERROR</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCNT</td>
<td>1</td>
</tr>
<tr>
<td>DATA</td>
<td>Error code</td>
</tr>
</tbody>
</table>

The following error codes are defined

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR_INVALID_CMD</td>
<td>The command in the request is invalid</td>
</tr>
<tr>
<td>ERR_INVALID_PAR</td>
<td>The parameter(s) in the request is invalid</td>
</tr>
<tr>
<td>ERR_INVALID_LEN</td>
<td>The length field (BCNT) is invalid for the request</td>
</tr>
<tr>
<td>ERR_INVALID_SEQ</td>
<td>The sequence does not match expected value</td>
</tr>
<tr>
<td>ERR_MSG_TIMEOUT</td>
<td>The message has timed out</td>
</tr>
<tr>
<td>ERR_CHANNEL_BUSY</td>
<td>The device is busy for the requesting channel</td>
</tr>
</tbody>
</table>

6.1.9.2 Optional commands

The following commands are defined by this specification but are optional and does not have to be implemented.

6.1.9.2.1 CTAPHID_WINK

The wink command performs a vendor-defined action that provides some visual or audible identification a particular authenticator device. A typical implementation will do a short burst of flashes with a LED or something similar. This is useful when more than one device is attached to a computer and there is confusion which device is paired with which connection.

Request

<table>
<thead>
<tr>
<th>CMD</th>
<th>CTAPHID_WINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCNT</td>
<td>0</td>
</tr>
<tr>
<td>DATA</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Response at success

<table>
<thead>
<tr>
<th>CMD</th>
<th>CTAPHID_WINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCNT</td>
<td>0</td>
</tr>
<tr>
<td>DATA</td>
<td>N/A</td>
</tr>
</tbody>
</table>

6.1.9.2.2 CTAPHID_LOCK

The lock command places an exclusive lock for one channel to communicate with the device. As long as the lock is active, any other channel trying to send a message will fail. In order to prevent a stalling or crashing application to lock the device indefinitely, a lock time up to 10 seconds may be set. An application requiring a longer lock has to send repeating lock commands to maintain the lock.

Request
6.1.9.3 Vendor specific commands

A CTAPHID may implement additional vendor specific commands that are not defined in this specification, yet being CTAPHID compliant. Such commands, if implemented must have a command in the range between CTAPHID_VENDOR_FIRST and CTAPHID_VENDOR_LAST.

6.2 Near Field Communication (NFC)

6.2.1 Conformance

Please refer to [ISOIEC-7816-4-2013] for APDU definition.

6.2.2 Protocol

The general protocol between a FIDO 2.0 client and an authenticator over NFC is as follows:

1. Client sends an applet selection command
2. Authenticator replies with success
3. Client sends a command for an operation (MakeCredential or GetAssertion)
4. Authenticator replies with response data or error

6.2.3 Applet selection

A successful Select allows the client to know that the applet is present and active. A client shall send a Select to the authenticator before any other command.

The FIDO 2.0 AID consists of the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RID</td>
<td>0xA000000647</td>
</tr>
<tr>
<td>AC</td>
<td>TBD</td>
</tr>
<tr>
<td>AX</td>
<td>0x0001</td>
</tr>
</tbody>
</table>

The command to select the FIDO 2.0 applet is:

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data In</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0xA4</td>
<td>0x04</td>
<td>0x0C</td>
<td>0x08</td>
<td>AID</td>
<td>TBD (version string length)</td>
</tr>
</tbody>
</table>

In response to the applet selection command, the FIDO authenticators shall reply with its version string in the successful response. In this writing, the version string is "TBD", hence a successful response to the applet selection command would consist of the following bytes:

0xXX..XX9000

6.2.4 Framing

Conceptually, framing defines an encapsulation of FIDO 2.0 commands. In NFC, this encapsulation is done in an APDU following [ISOIEC-7816-4-2013]. Fragmentation, if needed, is discussed in the following paragraph.

6.2.4.1 Request

Requests APDU shall have the following format:

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data In</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80</td>
<td></td>
<td>0x00</td>
<td>0x00</td>
<td>Variable</td>
<td>Command parameters</td>
<td>Variable</td>
</tr>
</tbody>
</table>

6.2.4.2 Response

Response shall have the following format in case of success:

<table>
<thead>
<tr>
<th>Data</th>
<th>Status word</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;9000&quot; - Success</td>
<td>For other values, see [ISOIEC-7816-4-2013]</td>
</tr>
</tbody>
</table>

6.2.5 APDU length

Length fields (Lc and Le) can be short (encoding a length up to 255) or extended (encoding a length up to 65535).

Some responses may not fit into a short APDU response. For this reason, FIDO 2.0 authenticators must respond in the following way:
6.3 Bluetooth Smart / Bluetooth Low Energy (BLE)

6.3.1 Conformance

Authenticator and Client devices using BLE shall conform to Bluetooth Core Specification 4.0 or later [BTCORE].

Bluetooth (tm) SIG specified UUID values shall be found on the Assigned Numbers website [BTASSNUM].

6.3.2 Pairing

BLE is a long-range wireless protocol and thus has several implications for privacy, security, and overall user-experience. Because it is wireless, BLE may be subject to monitoring, injection, and other network-level attacks.

For these reasons, Clients and Authenticators must create and use a long-term link key (LTK) and shall encrypt all communications. Authenticator must never use short term keys.

Because BLE has poor ranging (i.e., there is no good indication of proximity), it may not be clear to a FIDO Client with which BLE Authenticator it should communicate. Pairing is the only mechanism defined in this protocol to ensure that FIDO Clients are interacting with the expected BLE Authenticator. As a result, Authenticator manufacturers should instruct users to avoid performing Bluetooth pairing in a public space such as a cafe, shop or train station.

One disadvantage of using standard Bluetooth pairing is that the pairing is "system-wide" on most operating systems. That is, if an Authenticator is paired to a FIDO Client which resides on an operating system where Bluetooth pairing is "system-wide", then any application on that device might be able to interact with an Authenticator. This issue is discussed further in Implementation Considerations.

6.3.3 Link Security

For BLE connections, the Authenticator shall enforce Security Mode 1, Level 2 (unauthenticated pairing with encryption) before any FIDO 2.0 messages are exchanged.

6.3.4 Framing

Conceptually, framing defines an encapsulation of FIDO 2.0 raw messages responsible for correct transmission of a single request and its response by the transport layer.

All requests and their responses are conceptually written as a single frame. The format of the requests and responses is given first as complete frames. Fragmentation is discussed next for each type of transport layer.

6.3.4.1 Request from Client to Authenticator

Request frames must have the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>CMD</td>
<td>Command identifier</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>HLEN</td>
<td>High part of data length</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>LLEN</td>
<td>Low part of data length</td>
</tr>
<tr>
<td>3</td>
<td>s</td>
<td>DATA</td>
<td>Data (s is equal to the length)</td>
</tr>
</tbody>
</table>

Supported commands are PING and MSG. The constant values for them are described below.

The data format for the MSG command is defined in the Message Encoding section of this document.

6.3.4.2 Response from Authenticator to Client

Response frames must have the following format, which share a similar format to the request frames:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>STAT</td>
<td>Response status</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>HLEN</td>
<td>High part of data length</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>LLEN</td>
<td>Low part of data length</td>
</tr>
<tr>
<td>3</td>
<td>s</td>
<td>DATA</td>
<td>Data (s is equal to the length)</td>
</tr>
</tbody>
</table>

When the status byte in the response is the same as the command byte in the request, the response is a successful response. The value indicates an error, and the response data contains an error code as a variable-length, big-endian integer. The constant value for ERROR is described below.

Note that the errors sent in this response are errors at the encapsulation layer, e.g., indicating an incorrectly formatted request, or possibly an error communicating with the Authenticator’s FIDO 2.0 message processing layer. Errors reported by the FIDO 2.0 message processing layer itself are considered a success from the encapsulation layer’s point of view, and are reported as a complete MSG response.

Data format is defined in the Message Encoding section of this document.

6.3.4.3 Command, Status, and Error constants

The COMMAND constants and values are:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PING</td>
<td>0x81</td>
</tr>
<tr>
<td>KEEPALIVE</td>
<td>0x82</td>
</tr>
<tr>
<td>MSG</td>
<td>0x83</td>
</tr>
<tr>
<td>ERROR</td>
<td>0xbf</td>
</tr>
</tbody>
</table>
The KEEPALIVE command contains a single byte with the following possible values:

<table>
<thead>
<tr>
<th>Status Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCESSING</td>
<td>0x01</td>
</tr>
<tr>
<td>TUP_NEEDED</td>
<td>0x02</td>
</tr>
<tr>
<td>RFU</td>
<td>0x00, 0x03-0xFF</td>
</tr>
</tbody>
</table>

The ERROR constants and values are:

<table>
<thead>
<tr>
<th>Error Constant</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR_INVALID_CMD</td>
<td>0x01</td>
<td>The command in the request is unknown/invalid</td>
</tr>
<tr>
<td>ERR_INVALID_PAR</td>
<td>0x02</td>
<td>The parameter(s) of the command is/are invalid or missing</td>
</tr>
<tr>
<td>ERR_INVALID_LEN</td>
<td>0x03</td>
<td>The length of the request is invalid</td>
</tr>
<tr>
<td>ERR_INVALID_SEQ</td>
<td>0x04</td>
<td>The sequence number is invalid</td>
</tr>
<tr>
<td>ERR_REQ_TIMEOUT</td>
<td>0x05</td>
<td>The request timed out</td>
</tr>
<tr>
<td>NA</td>
<td>0x06</td>
<td>Value reserved (HID)</td>
</tr>
<tr>
<td>NA</td>
<td>0x0a</td>
<td>Value reserved (HID)</td>
</tr>
<tr>
<td>NA</td>
<td>0x0b</td>
<td>Value reserved (HID)</td>
</tr>
<tr>
<td>ERR_OTHER</td>
<td>0x7f</td>
<td>Other, unspecified error</td>
</tr>
</tbody>
</table>

6.3.5 GATT Service Description

This profile defines two roles: FIDO Authenticator and FIDO Client.

- The FIDO Client shall be a GATT Client
- The FIDO Authenticator shall be a GATT Server

The following figure illustrates the mandatory services and characteristics that shall be offered by a FIDO Authenticator as part of its GATT server:

![Figure 1] Mandatory GATT services and characteristics that must be offered by a FIDO Authenticator. Note that the Generic Access Service (GAS) is not present as it is already mandatory for any BLE compliant device.

The table below summarizes additional GATT sub-procedure requirements for a FIDO Authenticator (GATT Server) beyond those required by all GATT Servers.

<table>
<thead>
<tr>
<th>GATT Sub-Procedural</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Characteristic Value</td>
<td>Mandatory</td>
</tr>
<tr>
<td>Notifications</td>
<td>Mandatory</td>
</tr>
<tr>
<td>Read Characteristic Descriptors</td>
<td>Mandatory</td>
</tr>
<tr>
<td>Write Characteristic Descriptors</td>
<td>Mandatory</td>
</tr>
</tbody>
</table>

The table below summarizes additional GATT sub-procedure requirements for a FIDO 2.0 Client (GATT Client) beyond those required by all GATT Clients.

<table>
<thead>
<tr>
<th>GATT Sub-Procedural</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discover All Primary Services</td>
<td>(*)</td>
</tr>
<tr>
<td>Discover Primary Services by Service UUID</td>
<td>(*)</td>
</tr>
<tr>
<td>Discover All Characteristics of a Service</td>
<td>(**)</td>
</tr>
<tr>
<td>Discover Characteristics by UUID</td>
<td>(**)</td>
</tr>
<tr>
<td>Discover All Characteristic Descriptors</td>
<td>Mandatory</td>
</tr>
<tr>
<td>Read Characteristic Value</td>
<td>Mandatory</td>
</tr>
<tr>
<td>Notification</td>
<td>Mandatory</td>
</tr>
<tr>
<td>Read Characteristic Descriptors</td>
<td>Mandatory</td>
</tr>
<tr>
<td>Write Characteristic Descriptors</td>
<td>Mandatory</td>
</tr>
</tbody>
</table>

(*): Mandatory to support at least one of these sub-procedures.

(**): Mandatory to support at least one of these sub-procedures.

Other GATT sub-procedures may be used if supported by both client and server.
Specifcics of each service are explained below. In the following descriptions: all values are big-endian coded, all strings are in UTF-8 encoding, and any characteristics not mentioned explicitly are optional.

### 6.3.5.1 FIDO 2.0 Service

An Authenticator shall implement the FIDO 2.0 Service described below. The UUID for the FIDO 2.0 GATT service is TODO:0x????, it shall be declared as a Primary Service. The service contains the following characteristics:

<table>
<thead>
<tr>
<th>Characteristic Name</th>
<th>Mnemonic</th>
<th>Property</th>
<th>Length</th>
<th>UUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIDO 2.0 Control Point</td>
<td>fido2ControlPoint</td>
<td>Write</td>
<td>Defined by Vendor (20-512 bytes)</td>
<td>TBD</td>
</tr>
<tr>
<td>FIDO 2.0 Status</td>
<td>fido2Status</td>
<td>Notify</td>
<td>N/A</td>
<td>TBD</td>
</tr>
<tr>
<td>FIDO 2.0 Control Point Length</td>
<td>fido2ControlPointLength</td>
<td>Read</td>
<td>2 bytes</td>
<td>TBD</td>
</tr>
<tr>
<td>FIDO 2.0 Service Revision</td>
<td>fido2ServiceRevision</td>
<td>Read</td>
<td>Defined by Vendor (20-512 bytes)</td>
<td>0x2A2B</td>
</tr>
</tbody>
</table>

fido2ControlPoint is a write-only command buffer.

fido2Status is a notify-only response attribute. The Authenticator will send a series of notifications on this attribute with a maximum length of (ATT_MTU-3) using the response frames defined above. This mechanism is used because this results in a faster transfer speed compared to a notify-read combination.

fido2ControlPointLength defines the maximum size in bytes of a single write request to fido2ControlPoint. This value shall be between 20 and 512.

fido2ServiceRevision defines the revision of the FIDO 2.0 Service. The value is a UTF-8 string. For this version of the specification, the value fido2ServiceRevision shall be FIDO 2.0 Rev 1 or in raw bytes: 0x46494402032230525E620331.

The fido2ServiceRevision Characteristic may include a Characteristic Presentation Format descriptor with format value 0x19, UTF-8 String.

### 6.3.5.2 Device Information Service

An Authenticator shall implement the Device Information Service [BTDIS] with the following characteristics:

- Manufacturer Name String
- Model Number String
- Firmware Revision String

All values for the Device Information Service are left to the vendors. However, vendors should not create uniquely identifiable values so that Authenticators do not become a method of tracking users.

### 6.3.5.3 Generic Access Service

Every Authenticator shall implement the Generic Access Service [BTGAS] with the following characteristics:

- Device Name
- Appearance

### 6.3.6 Protocol Overview

The general overview of the communication protocol follows:

1. Authenticator advertises the FIDO 2.0 Service.
2. Client scans for Authenticator advertising the FIDO 2.0 Service.
3. Client performs characteristic discovery on the Authenticator.
4. If not already paired, the Client and Authenticator shall perform BLE pairing and create a LTK. Authenticator shall only allow connections from previously bonded Clients without user intervention.
5. Client reads the fido2ControlPointLength characteristic.
6. Client registers for notifications on the fido2Status characteristic.
7. Client writes a request (e.g., an enroll request) into the fido2ControlPoint characteristic.
8. Authenticator evaluates the request and responds by sending notifications over fido2Status characteristic.
9. The protocol completes when either:
   - The Client unregisters for notifications on the fido2Status characteristic, or:
   - The connection times out and is closed by the Authenticator.

### 6.3.7 Authenticator Advertising Format

When advertising, the Authenticator shall advertise the FIDO 2.0 service UUID.

When advertising, the Authenticator may include the TxPower value in the advertisement (see [BTXPLAD]).

The advertisement may also carry a device name which is distinctive and user-identifiable. For example, “ACME Key” would be an appropriate name, while “XJS4” would not be.

The Authenticator shall also implement the Generic Access Profile [BTGAP] and Device Information Service [BTDIS], both of which also provide a user-friendly name for the device that could be used by the Client.

It is not specified when or how often an Authenticator should advertise, instead that flexibility is left to manufacturers.

### 6.3.8 Requests

Clients should make requests by connecting to the Authenticator and performing a write into the fido2ControlPoint characteristic.

### 6.3.9 Responses

Authenticators should respond to Clients by sending notifications on the fido2Status characteristic.
Some Authenticators might alert users or prompt them to complete the test of user presence (e.g., via sound, light, vibration). Upon receiving any request, the Authenticators shall send KEEPALIVE commands every \( k_{\text{KeepAliveMillis}} \) milliseconds until completing processing the commands. While the Authenticator is processing the request the KEEPALIVE command will contain status \( \text{PROCESSING} \). If the Authenticator is waiting to complete the Test of User Presence, the KEEPALIVE command will contain status \( \text{TUP_NEEDED} \). While waiting to complete the Test of User Presence, the Authenticator may alert the user (e.g., by flashing) in order to prompt the user to complete the test of user presence. As soon as the Authenticator has completed processing and confirmed user presence, it shall stop sending KEEPALIVE commands, and send the reply.

Upon receiving a KEEPALIVE command, the Client shall assume the Authenticator is still processing the command; the Client shall not resend the command. The Authenticator shall continue sending KEEPALIVE messages at least every \( k_{\text{KeepAliveMillis}} \) to indicate that it is still handling the request. Until a client-defined timeout occurs, the Client shall not move on to other devices when it receives a KEEPALIVE with \( \text{TUP_NEEDED} \) status, as it knows this is a device that can satisfy its request.

### 6.3.10 Framing fragmentation

A single request/response sent over BLE may be split over multiple writes and notifications, due to the inherent limitations of BLE which is not currently meant for large messages. Frames are fragmented in the following way:

A frame is divided into an initialization fragment and one or more continuation fragments.

An initialization fragment is defined as:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>CMD</td>
<td>Command identifier</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>HLEN</td>
<td>High part of data length</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>LLEN</td>
<td>Low part of data length</td>
</tr>
<tr>
<td>3</td>
<td>0 to (maxLen - 3)</td>
<td>DATA</td>
<td>Data</td>
</tr>
</tbody>
</table>

where \( \text{maxLen} \) is the maximum packet size supported by the characteristic or notification.

In other words, the start of an initialization fragment is indicated by setting the high bit in the first byte. The subsequent two bytes indicate the total length of the frame, in big-endian order. The first \( \text{maxLen} - 3 \) bytes of data follow.

Continuation fragments are defined as:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>SEQ</td>
<td>Packet sequence 0x00..0x7f (high bit always cleared)</td>
</tr>
<tr>
<td>1</td>
<td>0 to (maxLen - 1)</td>
<td>DATA</td>
<td>Data</td>
</tr>
</tbody>
</table>

where \( \text{maxLen} \) is the maximum packet size supported by the characteristic or notification.

In other words, continuation fragments begin with a sequence number, beginning at 0, implicitly with the high bit cleared. The sequence number must wrap around to 0 after reaching the maximum sequence number of 0x7f.

Example for sending a PING command with 40 bytes of data with a \( \text{maxLen} \) of 20 bytes:

<table>
<thead>
<tr>
<th>Frame</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{810028} [17 bytes of data]</td>
</tr>
<tr>
<td>1</td>
<td>{00} [19 bytes of data]</td>
</tr>
<tr>
<td>2</td>
<td>{01} [4 bytes of data]</td>
</tr>
</tbody>
</table>

Example for sending a ping command with 400 bytes of data with a \( \text{maxLen} \) of 512 bytes:

<table>
<thead>
<tr>
<th>Frame</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{810180} [400 bytes of data]</td>
</tr>
</tbody>
</table>

### 6.3.11 Implementation Considerations

#### 6.3.11.1 Bluetooth pairing: Client considerations

As noted in the Pairing section, a disadvantage of using standard Bluetooth pairing is that the pairing is "system-wide" on most operating systems. That is, if an Authenticator is paired to a FIDO Client which resides on an operating system where Bluetooth pairing is "system-wide", then any application on that device might be able to interact with an Authenticator. This poses both security and privacy risks to users.

While Client operating system security is partly out of FIDO's scope, further revisions of this specification may propose mitigations for this issue.

#### 6.3.11.2 Bluetooth pairing: Authenticator considerations

The method to put the Authenticator into Pairing Mode should be such that it is not easy for the user to do accidentally especially if the pairing method is Just Works. For example, the action could be pressing a physically recessed button or pressing multiple buttons. A visible or audible cue that the Authenticator is in Pairing Mode should be considered. As a counter example, a silent, long press of a single non-recessed button is not advised as some users naturally hold buttons down during regular operation.

Note that at times, Authenticators may legitimately receive communication from an unpaired device. For example, a user attempts to use an Authenticator for the first time with a new Client: he turns it on, but forgets to put the Authenticator into pairing mode. In this situation, after connecting to the Authenticator, the Client will notify the user that he needs to pair his Authenticator. The Authenticator should make it easy for the user to do so, e.g., by not requiring the user to wait for a timeout before being able to enable pairing mode.

#### 6.3.11.3 Handling command completion

It is important for low-power devices to be able to conserve power by shutting down or switching to a lower-power state when they have satisfied a Client's requests. However, the FIDO 2.0 protocol makes this hard as it typically includes more than one command/response. This is especially true if a user has more than one key handle associated with an account or identity, multiple key handles may need to be tried before getting a successful outcome. Furthermore, Clients that fail to send follow-up commands in a timely fashion may cause the Authenticator to drain its battery by staying powered up anticipating more commands.

A further consideration is to ensure that a user is not confused about which command she is confirming by completing the test of user presence.
That is, if a user performs the test of user presence, that action should perform exactly one operation.

We combine these considerations into the following series of recommendations:

- Upon initial connection to an Authenticator, and upon receipt of a response from an Authenticator, if a Client has more commands to issue, the Client must transmit the next command or fragment within \( k_{\text{MaxCommandTransmitDelayMillis}} \) milliseconds.
- Upon final response from an Authenticator, if the Client decides it has no more commands to send it should disable notifications on the \( \text{fido2Status} \) characteristic. When the notifications are disabled the Authenticator may enter a low power state or disconnect and shut down.
- Any time the Client wishes to send a FIDO 2.0 message, it must have first enabled notifications on the \( \text{fido2Status} \) characteristic and wait for the ATT acknowledgement to be sure the Authenticator is ready to process messages.
- Upon successful completion of a command which required a test of user presence, e.g. upon a successful authentication or registration command, the Authenticator can assume the Client is satisfied, and may reset its state or power down.
- Upon sending a command response that did not consume a test of user presence, the Authenticator must assume that the Client may wish to initiate another command, and leave the connection open until the Client closes it or until a timeout of at least \( k_{\text{ErrorWaitMillis}} \) milliseconds has elapsed without further commands from a Client, an Authenticator may reset its state or power down.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k_{\text{MaxCommandTransmitDelayMillis}} )</td>
<td>1500 milliseconds</td>
</tr>
<tr>
<td>( k_{\text{ErrorWaitMillis}} )</td>
<td>2000 milliseconds</td>
</tr>
<tr>
<td>( k_{\text{KeepAliveMillis}} )</td>
<td>500 milliseconds</td>
</tr>
</tbody>
</table>

### 6.3.11.4 Data throughput

BLE does not have particularly high throughput, this can cause noticeable latency to the user if request/responses are large. Some ways that implementers can reduce latency are:

- Support the maximum MTU size allowable by hardware (up to the 512-byte max from the BLE specifications).
- Make the attestation certificate as small as possible; do not include unnecessary extensions.

### 6.3.11.5 Advertising

Though the standard doesn’t appear to mandate it (in any way that we’ve found thus far), advertising and device discovery seems to work better when the Authenticators advertise on all 3 advertising channels and not just one.

### 6.3.11.6 Authenticator Address Type

In order to enhance the user's privacy and specifically to guard against tracking, it is recommended that Authenticators use Resolvable Private Addresses (RPAs) instead of static addresses.

### 7. Bibliography

[ISOIEC-7816-4-2013] ISO 7816-4: Identification cards - Integrated circuit cards; Part 4: Organization, security and commands for interchange

### A. References

#### A.1 Normative references

[FIDOKeyAttestation] FIDO 2.0: Key attestation format [FIDOPlatformApiReqs] FIDO 2.0: Requirements for Native Platforms [FIDOSignatureFormat] FIDO 2.0: Signature format [FIDOWebApi] FIDO 2.0: Web API for accessing FIDO 2.0 credentials

