HDC Protocol Specification

Specification of HDC 1.0.0-alpha.11

About this document

WARNING: This document is currently still being drafted!

The "*HDC Protocol Specification*", or “HDC-spec”, for shorthand, is the formal definition of the Host Device Communication protocol, and its purpose is to ensure the interoperability of its implementations over a wide variety of platforms. The HDC protocol aims to become an *open standard* for which this HDC-spec is intended to be the *single source of truth* of its specification.

Intended audience

This document is a rigorous specification of HDC concepts and its internals. It will therefore be of little use for anybody who only needs to use the API of a library that readily implements all of that. Please refer to the specific documentation of ready-to-use libraries for your language of choice: hdcproto for Python, or the STM32 HAL driver for C. (More to come, once a stable HDC-spec is released).
The chapters of this document, however, are sorted from high-level to low-level concepts, so its first chapter might also serve as an introduction to the terminology and basic architecture of HDC.

Contributing to this document

This document is being authored collectively on Google Docs, where everybody is invited to contribute their feedback. Alternatively it can be reported via the issue tracker.
The most recent snapshot of HDC-spec is published as a PDF document in the official repository.

License of the HDC Protocol Specification

About the license

- The license shown above **only** applies to this document.
  The source-code published in the repository is licensed under the “MIT License”, instead.

- Standards don't work well if everybody cooks its own variant.
  I’ve nevertheless chosen a permissive license, because “*the crowd”* is sometimes better at consolidating and evolving a standard, than a single individual may be.

- It would be greatly appreciated if any derivative work which modifies any of the conventions defined herein would also change the name of their protocol, as to avoid any potential confusion. Thanks! 🙂

Table of Contents

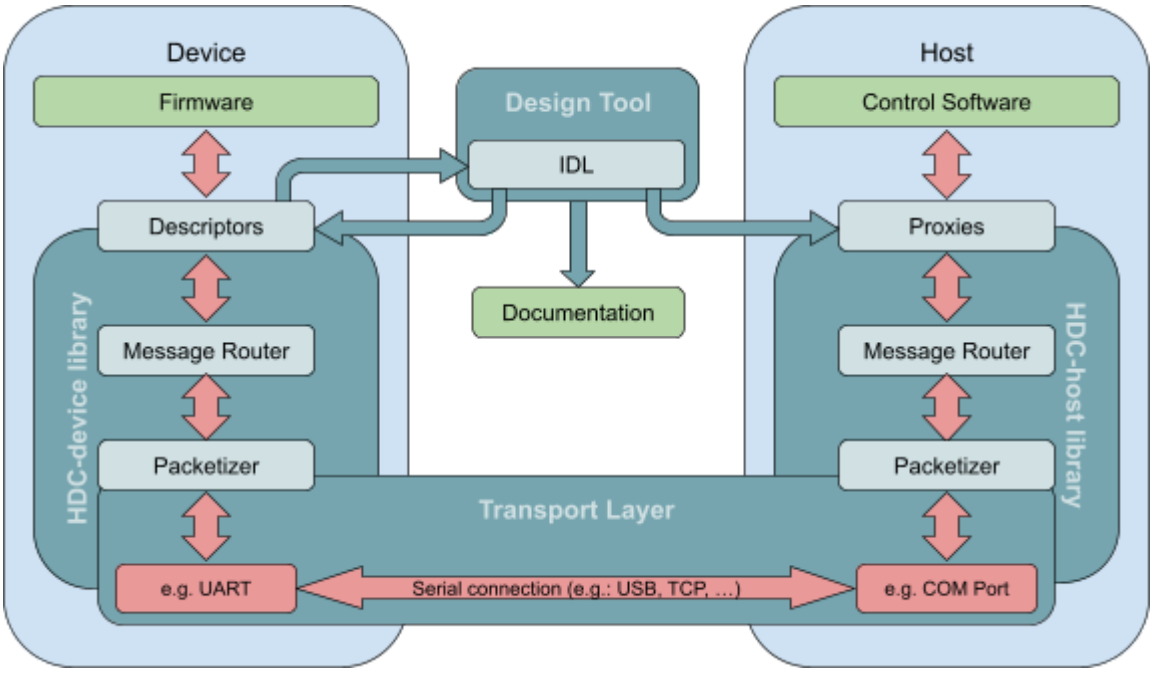
Summary (and terminology)

(Highlighted terms have specific meaning in the context of HDC.)

The HDC protocol exposes the capabilities of a ***device*** as an object-oriented API that the software on a ***host*** can interact with, thus unburdening developers from having to care about the specifics of the communication protocol.

At the highest level of abstraction, HDC uses ***descriptors*** to declare one or more ***features*** of a device, each exposing a set of ***properties***, ***commands*** and ***events***.
Descriptors can either be authored directly in the programming language of the device's firmware, or an Interface Description Language (IDL) generated with the assistance of an interactive design tool. The IDL descriptors can also be auto-generated by the firmware of the device.
The same design tool can translate the IDL descriptors into auto-generated source-code for matching descriptors and proxies. It also can generate human-readable documentation about a device's API.

At the lowest level of abstraction, HDC takes care of how ***request*** and ***reply messages*** are exchanged, and how those are serialized and deserialized in form of ***packets*** for transmission over a ***transport*** layer.



teal: HDC functionality / red: data flow / green: application specific

Devices

A device is essentially a server whose only client is the host.
All replies sent by a device must have been explicitly requested by the host, with exception of events, which a device can raise autonomously.

The HDC protocol is optimized for devices with very limited computing resources; typically microcontrollers with only a few kilobytes of RAM and FLASH memory, executing firmware written in C or C++.

Hosts

The host runs the software which operates/controls/monitors the device.
A host can be connected to multiple devices, each via a dedicated serial connection.
Typically this is a laptop, desktop or embedded PC, capable of executing software written in high level languages like Python, Java, C#, …

Features

The term ***feature*** provides a more natural way to refer to the capabilities of a device, while avoiding any collisions with related software development concepts like object, service, interface or endpoint.
HDC-spec requires that every device implements at least one feature, which will be sufficient for devices with very little functionality. Usually there's good reasons to organize functionality into multiple features. The following examples may help to understand how and why capabilities should be organized into features, and that it's no different than what conventional software development guidelines and common sense advise:

- Separation of concerns:
  - `core.serial_number` and `thermostat.setpoint`,
    instead of …
  - `core.serial_number` and `core.thermostat_setpoint`.
- Modularity:
  - e.g.: addressing multiple feature instances of the same class of functionality:
    - `thermostat_a.setpoint` and `thermostat_b.setpoint`,
      instead of …
    - `thermostat.setpoint_a` and `thermostat.setpoint_b`
  - e.g.: deploying the same `Thermostat` driver on different types of device.
- etcetera

The definition of features will typically mirror the internal structure of the firmware's implementation, but HDC does not enforce this in any way. It's perfectly feasible to expose an HDC-API which differs substantially from the internal APIs of the firmware.

Feature Descriptor			
Attribute	**M/O**	**Type**	**Description**
`id`	M	UINT8	Unique among all features implemented on a device. Used for routing of HDC messages.
`name`	M	UTF8	Unique name. Used for source-code generation.
`doc`	O	UTF8	Human readable docstring.
`class`	O	UTF8	Name of the implementation.
`version`	O	UTF8	Semantic Versioning of the implementation.
`states`	O	[State]	List of State-Descriptors.
`commands`	M	[Cmd]	List of Command-Descriptors.
`events`	M	[Evt]	List of Event-Descriptors.
`properties`	M	[Prop]	List of Property-Descriptors.

	State Descriptor		
Attribute	**M/O**	**Type**	**Description**
`id`	M	UINT8	Unique among all states of a feature.
`name`	M	UTF8	Unique name. Used for source-code generation.
`doc`	O	UTF8	Human readable docstring.

HDC-spec requirements related to HDC-features:

-
- Features of a device are identified by means of a unique UINT8 value named `FeatureID`.
    - A device can therefore implement a maximum of 256 features.
    - Features cannot be nested into sub-features.
    - `FeatureID` values may not be sequential.
        - e.g.: a device may implement `FeatureIDs` 0x00, 0x42, 0xD7
    - A `FeatureID` is only used for addressing purposes and only `FeatureID=0x00` is hard-coded for a particular purpose, while all other `FeatureIDs` may be assigned differently on different device implementations, e.g:
        - A Thermostat-feature may be `FeatureID=0x01` on one revision of a device and `FeatureID=0x44` on another.
        - It's up to the host's implementation to either hard-code `FeatureID` values for a specific device revision or to use introspection to infer the `FeatureID` of a certain feature.

The mandatory Core-Feature

Obsolete: No need for a Core-Feature anymore, because introspection was the only reason to originally require it.

A device must implement a Core-Feature with `FeatureID=0x00`, which besides implementing all mandatory properties of any feature, also must implement the following additional ones:

PropertyID	PropertyName	DataType	Description
0xFA	AvailableFeatures	BLOB	List of IDs of features available on the device.
0xFB	MaxReqMsgSize	UINT16	Maximum number of bytes of a request message that the device can cope with.

The Core-Feature may implement whatever further commands, properties and events are appropriate for the ***core*** of a particular device implementation.
Some examples of what a device's core-feature may typically implement:

- A read-only property to expose the serial-number of a device.
- A read-writable string property to store information about the most recent maintenance

A command to switch into the bootloader mode.
ToDo:

- The properties Name, Class, Version, States on the Core feature actually refer to the device implementation. (Those seem less useful for the mandatory Core feature and it would be kind of cumbersome to abuse the message-layer for stuff which already exists on the Feature-Property-layer.)

Feature state

Each feature can implement its own state-machine via the mandatory `FeatureState` property and notifying any state-transition by means of the mandatory `FeatureStateTransition` event.

Recommendations:

- Document all possible values of the FeatureState property via its property-description string expressed in python dictionary curly brace syntax, like for example:
  `"{0:'Initializing', 1:'NotReady', 2:'Ready', 3:'Acquiring', 0xFF:'Error'}"`
- Naming recommendations for typical states:
  - Off
  - Initializing
  - NotReady
  - Ready
  - Preparing
  - Acquiring
  - Error

Feature logger

Each feature can implement its own logger by means of the mandatory `Log` event and also allow tuning its verbosity by means of the mandatory `LogEventThreshold` property whose UINT8 values mimic the logging levels of the python logging module:

Overview of LogLevel values		
UINT8 value of `LogEventThreshold`	**Name**	**Description**
`10`	DEBUG	Most detailed information, which might be useful when proactively debugging a certain issue.
`20`	INFO	Useful information to confirm that everything is working as it should.
`30`	WARNING	Something unexpected has happened or some problem is about to happen in the near future.
`40`	ERROR	An error has occurred. The software was unable to perform some function.
`50`	CRITICAL	A serious error has occurred. Device itself may not be able to continue running properly, i.e. entered a safe-state or even shut itself down.

Other logging levels are not supported.

It's up to each device's implementation to decide to which level the `LogEventThreshold` gets initialized, but a device must not change this value afterwards, because the host caches this value and does not expect a device to change it dynamically.

Feature introspection

A host can obtain a list of `FeatureID` values implemented on a device, via the mandatory property:

- `CoreFeature.AvailableFeatures`

It can then obtain further details about each feature, via the mandatory properties implemented on every feature:

- `FeatureName`
  - A UTF8 string, unique among all features implemented on a device.
  - Typically a human-readable name in camel-case without any whitespace.
  - Example values:
    - Core
    - Thermostat
    - AxisX

- `FeatureClassName`
  - A UTF8 string, ideally unique among all existing feature implementations.
  - TBD: Namespace dot syntax?

- `FeatureClassVersion`
  - A UTF8 string with the SemVer 2.0.0 compliant version of the implementation (aka class) of a feature.

- `FeatureDescription`
  - A UTF8 string, that may be empty.
  - May contain multiple lines of text, which must be delimited with newlines (`\n`) only.

Commands

`Commands` are essentially “remote procedure calls” of procedures implemented on a specific feature of a device.

	Command Descriptor		
Attribute	**M/O**	**Type**	**Description**
`id`	M	UINT8	Unique among all commands implemented on a feature. Used for routing of HDC messages.
`name`	M	UTF8	Unique name. Used for source-code generation.
`doc`	O	UTF8	Human readable docstring.
`args`	O	[Arg]	List of Argument-Descriptors
`returns`	O	[Ret]	List of Return-Descriptors
`raises`	O	[Exc]	List of Exception-Descriptors

Facts about `Commands`:

- A command must be implemented in the context of a feature.
- The request may contain ***arguments*** and the reply may contain ***return values***, depending on the specific command implementation.
- A host must wait for the reply to the previous request, before sending the next request.
    - Commands that take longer than a few milliseconds to execute should preferably be refactored into a non-blocking command and use events or a proper state-machine to notify completion.
- Request messages must not exceed the maximum size specified by the `CoreFeature.MaxReqMsgSize` property.
- Every feature must implement all mandatory commands.
- Each feature may implement up to 240 custom commands
- Command name must at least be 1 character long.
- Command description might be an empty string.

Mandatory Commands

Every feature must implement the `GetPropertyValue` and `SetPropertyValue` commands, which is how Properties are implemented in HDC.

Overview of Command.id values			
id	**name**	**args**	**returns**
0x00→0xEF	Available for custom use		
0xF0	GetPropertyValue	UINT8: PropertyID	var
0xF1	SetPropertyValue	UINT8: PropertyID var NewValue	var
0xF2→0xFF	Reserved for future use		

ToDo: Should we also get rid of Setter and Getter commands and use Messages instead?

Command introspection (Obsolete!)

A host can query a feature for details about the commands it implements, via:

- the mandatory property
  - `AvailableCommands`
    - A BLOB listing all the `CommandID` values implemented on the feature.
- … and the mandatory commands:
  - `GetCommandName(UINT8 CommandID)` → UTF8
    - Takes the CommandID as an argument.
    - Returns a UTF8 string: the name of the corresponding command, which will be unique among all commands implemented on a feature.
    - Typically a human-readable name in camel-case without any whitespace, like this: `MyCustomCommand`
  - `GetCommandDescription(UINT8 CommandID)` → UTF8
    - Takes the CommandID as an argument
    - Returns UTF8 string, that may be empty.
    - May contain multiple lines of text, which must be delimited with newlines (`\n`) only.
    - First line of text may describe the signature of arguments and return values like this:

      ```
      (UINT8 FirstArg, INT32 SecondArg) -> UINT16 FirstRetVal, UINT32 SecondRetVal
      ```

Arguments

	Argument Descriptor		
Attribute	**M/O**	**Type**	**Description**
`type`	M	TYPE	Data type. ToDo: Current HDC data types might be too limited. Should we bloat it, or allow devs to do their custom thing.
`name`	O	UTF8	Name of the argument. Used for source-code generation. Note that HDC can only cope with positional arguments!
`doc`	O	UTF8	Human readable docstring.

Return values

	Return(-Value) Descriptor		
Attribute	**M/O**	**Type**	**Description**
`type`	M	TYPE	Data type. ToDo: Current HDC data types might be too limited. Should we bloat it, or allow devs to do their custom thing.
`name`	O	UTF8	Name of the argument. Used for source-code generation. Note that HDC can only cope with positional return values!
`doc`	O	UTF8	Human readable docstring.

Exceptions

Devices can raise exceptions while executing a command and the HDC protocol will forward those to the host and raise them via the corresponding proxy.

	Exception Descriptor		
Attribute	**M/O**	**Type**	**Description**
`id`	M	UINT8	Unique among all exceptions potentially raised by a command. ID range 0xF0 to 0xFF is reserved for exceptions predefined by HDC-spec. The ID 0x00 is reserved to mean NO_ERROR. The remaining range 0x01 to 0xEF is available for application specific exceptions.
`name`	M	UTF8	Unique name. Used for source-code generation.
`doc`	O	UTF8	Human readable docstring.

The id of different types of exceptions must be unique among all those potentially raised by each command. Whether this requirement for uniqueness should be extended for all exceptions raised in the context of one feature or even one device is up to the designers' of the API to decide.

Overview of Exception.id values		
`id`	`name`	**Raised by**
0x00	NO_ERROR	No exception is raised. Used in Command reply messages to signify success.
0x01→0xEF	Available for application specific use	
0xF0	CommandFailed	All commands. Typically used for unexpected exceptions, or by very lazy developers, who do not care about application specific exceptions.
0xF1	UnknownFeature	Any command addressed to a non-existent feature
0xF2	UnknownCommand	Any command not implemented on the given feature.
0xF3	InvalidArgs	All commands. Command arguments are incorrect.
0xF4	NotNow	All commands. Command can not be executed at this point in time. Typically because the current feature-state disallows it.
0xF5	UnknownProperty	GetPropertyValue SetPropertyValue
0xF6	ReadOnly	SetPropertyValue Attempted to modify a read-only property
0xFA→0xFF	Reserved for future use	

	in HDC	

Exceptions may also carry a human-readable text, but only if it provides additional context beyond what the name of the exception already provides. In some scenarios it might be also convenient to emit additional Log-events to convey more detailed information about the situation. Those Log events are typically sent before the failed command raises its exception.

Keep in mind that **some types of errors can not be raised as Exceptions** and will be notified as `CoreFeature.Log` events, instead:

- Reading-frame errors detected while de-packetizing messages
- Receiving a request featuring an unknown `MessageTypeID` value
- Receiving a request that is larger than allowed by the `CoreFeature.MaxReqMsgSize` property

ToDo:

-

Properties

Properties are implemented by means of the following, mandatory `Commands`:

- `GetPropertyValue(UINT8 PropertyID)` → *`PropType`*
  - Takes the `PropertyID` as an argument
  - Returns a value of the data-type of the property.
- `SetPropertyValue(UINT8 PropertyID,` *`PropType`* `NewValue)` → *`PropType`*
  - Takes the `PropertyID` and the `NewValue` as arguments.
  - Returns a value of the data-type of the property.
  - The returned value may differ from the given `NewValue` argument and reports the actual value of the property, according to any trimming or discretization that the property-setter might consider appropriate.
    - e.g.: Attempting to set a value of 3.567% might be discretized into 3.6%

Properties may serve a wide range of purposes:

- Expose immutable meta-data/capabilities of a feature, as for example:
  - `CoreFeature.SerialNumber`
  - `AxisX.MaxPos`
  - `Thermostat.MaxTargetTemp`
- Expose mutable configuration parameters, as for example:
  - `Axis.MaxAccel`
  - Warning: Avoid abusing properties as a replacement for `Command` arguments!
- Expose the internal state of a feature as read-only properties, as for example:
  - `*.FeatureState`
  - `Thermostat.Setpoint`
- Expose sensor data that might be acquired on request, as for example:
  - `Thermostat.ObjectTemperature`
    - Beware of sensor latency issues!
    -

Facts:

- All properties must be implemented in the context of a feature.

Property Descriptor			
Attribute	**M/O**	**Type**	**Description**
`id`	M	UINT8	Unique among all properties implemented on a feature. Used for routing of HDC messages.
`name`	M	UTF8	Unique name. Used for source-code generation.
`doc`	O	UTF8	Human readable docstring.
`type`	O	TYPE	Data type
`ro`	O	BOOL	Whether it's "read-only" via the HDC interface. Note it still may be *mutable* !

Mandatory properties

Every feature must implement the following mandatory properties:

PropertyID	PropertyName	DataType	Readonly	Immutable
0x01→0xEF	Available for custom use			
0xF0	LogEventThreshold	UINT8	No	No
0xF1	FeatureState	UINT8	Yes	No
0xFA→0xFF	Reserved for future use			

Property introspection (Obsolete!)

A host can query a feature for details about the properties it implements, via:

- the mandatory property
  - `AvailableProperties`
    - A BLOB listing all the `PropertyID` values implemented on the feature.
- … and the mandatory commands:
  - `GetPropertyName(UINT8 PropertyID) → UTF8`
    - Takes the `PropertyID` as an argument.
    - Returns a UTF8 string: the name of the corresponding property, which will be unique among all properties implemented on a feature.
    - Typically a human-readable name in camel-case without any whitespace.
      - E.g.: `MyCustomProperty`
  - `GetPropertyType(UINT8 PropertyID) → UINT8`
    - Takes the `PropertyID` as an argument.
    - Returns the UINT8 code for the data-type.
  - `GetPropertyReadonly(UINT8 PropertyID) → BOOL`
    - Takes the `PropertyID` as an argument.
    - Returns a BOOL:
      - A value of TRUE means the property is read-only and any attempt to change its value will produce a CommandErrorCode=0x07.
      - A value of FALSE means the host is allowed to modify the value of the property.
  - `GetPropertyDescription(UINT8 PropertyID) → UTF8`
    - Takes the `PropertyID` as an argument
    - Returns a UTF8 string that may be empty.
    - May contain multiple lines of text, which must be delimited with newlines (`\n`) only.
    - First line of text may describe the units and meaning like this:
      - `[°C] Current heat-sink temperature.`

Events

	Event Descriptor		
Attribute	**M/O**	**Type**	**Description**
`id`	M	UINT8	Unique among all events implemented on a feature. Used for routing of HDC messages.
`name`	M	UTF8	Unique name. Used for source-code generation.
`doc`	O	UTF8	Human readable docstring.
`args`	O	[Arg]	List of Argument-Descriptors

Facts:

- All events must be implemented in the context of a feature.

Mandatory events

(Strictly speaking an event cannot be mandatory, but that's too philosophical and beyond scope.)

Any feature requiring any of the following functionality, must implement it as follows:

EventID	EventName	Payload	Description
0x01→0xEF	Available for custom use		
0xF0	Log	[0]: UINT8 LogLevel [remainder]: UTF8 LogMessage	See Feature logger
0xF1	FeatureStateTransition	[0]: UINT8 PreviousStateID [1]: UINT8 NewStateID	See Feature state
0xFA→0xFF	Reserved for future use		

Data types

The data-type of properties, command-arguments and return values are limited to the following list introspection uses the following `DataTypeCode`s:

DataType ID	DataType name
`0x01`	UINT8
`0x02`	UINT16
`0x04`	UINT32
`0x11`	INT8
`0x12`	INT16
`0x14`	INT32
`0x24`	FLOAT
`0x28`	DOUBLE
`0xB1`	BOOL
`0xBF`	BLOB
`0xFF`	UTF8

The ID values of each DataType can be interpreted as follows:

- Upper Nibble: Kind of DataType
  - 0x0_ --> Unsigned integer
  - 0x1_ --> Signed integer
  - 0x2_ --> Floating point number
  - 0xB_ --> Binary data (Either variable size 0xBF, or boolean 0xB1)
  - 0xF_ --> UTF-8 string (Always variable size 0xFF, without zero-termination)
- Lower Nibble: Size of DataType, given in number of bytes.
  - i.e. 0x14 --> INT32, whose size is 4 bytes
    - Exception to the rule: 0x_F denotes a variable size DataType.
    - Special case 0xB1 --> BOOL, size is 1 byte, although only using 1 bit.

Endianness

Numeric values are serialized in little-endian order when being transmitted in messages.

For example the UINT32 value `0xAABBCCDD` will be serialized in a message like this:

Message[n-1]	Message[n+0]	Message[n+1]	Message[n+2]	Message[n+3]	Message[n+4]
…	`0xDD`	`0xCC`	`0xBB`	`0xAA`	…

The reason being that the firmware will most likely run on little-endian processors (e.g. STM32 microcontrollers) and the pointer arithmetics can be kept less cryptic and more performant when sending numeric values in the same endianness.
Note how also most hosts will run on little-endian processors (e.g. Intel Core)
We disregard the tradition of transmitting data in big-endian order.

Messages

ToDo: Explain Message-layer, e.g. in form of a summary with glossary.

Overview of MessageTypeID values

The first byte of any message is its `MessageTypeID`, which roughly specifies its purpose:

MessageTypeID	Name
`0x00 .. 0xEF`	Available for custom use. e.g. Tunneling of other protocols
`0xF0`	Meta-message
`0xF1`	Echo-message
`0xF2`	Command-message
`0xF3`	Event-message
`0xF4 .. 0xFF`	Reserved by HDC for future use

Any other MessageTypeID is reserved for future use and must be treated as a reading-frame-error when deserializing messages.

ToDo:

- Explain MaxReqMsgSize.
- Remove related remarks that are scattered in the wrong chapters, i.e. Commands, because it is easier to specify all this at the message-layer.

Request-Reply message exchange pattern

Typically messages are exchanged according to a strict request-reply pattern, which is initiated by the host sending a request message to the device and the device responding to it with a specific reply message.
Note how the `MessageTypeID` of a reply message always matches that of the request message.
In the case of CommandMessages, also its second and third byte will match those of the request, regardless of the FeatureID or CommandID being known to the device.

HDC only tolerates the following *exceptions* to the pattern described above:

- Devices might send EventMessages at any time.
  Hosts are never allowed to send any EventMessages to a device!
- Hosts and devices might send custom message types at any time.
  I.e.: Tunneling of other protocols through through HDC
  It's up to the specific application to ensure that custom messages won't saturate the connection.
- Hosts may desist from waiting for a reply whenever they observe XXXXXX What? Log-Events emitted by the Core-feature? Should we dedicate a special MessageType for this purpose?
  - ToDo: Specify how to deal with time-outs while waiting for a reply.

Limiting the size of messages sent to a device

Devices are typically limited in the amount of RAM available for HDC message processing.
A device must report the maximum message size it can cope with via the mandatory property `Core.MaxReqMsgSize`, whose UINT32 value represents the number of bytes of a message (not to be confused with the size of a packet!!)
Hosts are therefore required to ensure that all messages they send to a given device comply with this size limit.

Meta-Messages

The purpose of this type of message is to allow for hosts to infer different kinds of meta-information about the HDC-API:

- the HDC-spec version implemented by a device, so that they may decide very early on how to communicate with a given device.
- The MaxReqMsgSize: Largest request-message size, that a device can cope with.
- The JSON representation of the HDC-API implemented by a device.

ToDo: Reword this chapter! There are multiple kinds of “meta” requests and we should arguments in the message-payload to allow for future extensions.

The structure of a `VersionMessage` ***request*** is as follows:

Message[0]	**Message[1...]**
MessageTypeID = 0xF0	Any message payload will be ignored silently.

The structure of a `VersionMessage` ***reply*** is as follows:

Message[0]	**Message[1...7]**
MessageTypeID = 0xF0	“`HDC 1.0.0`” (UTF-8 encoded string. Without the quotes)

Facts about `VersionMessages`:

- Must be implemented by all HDC-devices, including those implementing future HDC-spec versions.
- Syntax and meaning of the version number are those defined by Semantic Versioning 2.0.0.
    - A device compliant with HDC-spec 1.0.0 will return: `HDC 1.0.0`
    - An alpha version firmware might reply: `HDC 1.0.0-alpha.42`
    - Forks of HDC may reply with a different prefix: `HDC++ 1.0.0`
- Note how this resembles a `Property`, but it isn’t, because it is not associated with any Feature. It is intentionally implemented at the lower “message layer”, instead, to ensure forward-compatibility with future HDC-spec versions that might need to redefine conventions about how `Features` and `Properties` are handled.

Echo-Messages

Messages starting with the `MessageTypeID=0xF1` will immediately be echoed back to the host, regardless of their content.

The structure of a `EchoMessage` ***request*** is as follows:

Message[0]	***remainder***
MessageTypeID = 0xF1	Whatever, as long as the size of the message does not exceed `CoreFeature.MaxReqMsgSize`

→ Device must ***reply*** with an identical message.

Facts about `EchoMessages`:

- Since this resembles the request/reply pattern of the more usual commands, this is named `EchoMessage`, but it's otherwise a quite atypical command, because it's unrelated to any feature and its content is discretionary, except for the first `MessageTypeID` byte and that the message size may not exceed `CoreFeature.MaxReqMsgSize`.
- All devices must handle the `EchoMessage`.
- Hosts may use the `EchoMessage` to:
    - Test that the serial communication connection is working.
    - Test that the firmware of a device is responsive.
    - Benchmark the bandwidth and latency of the serial communication.
    - Stress-test the bandwidth while other messages are being sent.
- A host must wait for the reply to the previous request, before sending the next request.
    - This requirement applies to both: `Command`s and `EchoMessage`s.
- There's no introspection on the `EchoMessage`.
    - Hosts can simply count on it being implemented on any device.

Command-Messages

See Commands

ToDo: Move CommandMessage syntax into this chapter, to make the high-level explanation of Commands less confusing to readers who actually do not care about the message-layer.

The host sends a ***request*** message to the device, which will acknowledge completion to the host by sending a ***reply*** message.

The structure of a `CommandMessage` ***request*** is:

Message[0]	**Message[1]**	**Message[2]**	***remainder***
`MessageTypeID =0xF2`	`FeatureID`	`CommandID`	Command arguments, if any.

The structure of a `Command` ***reply*** message of a ***successful*** command execution is:

Message[0]	**Message[1]**	**Message[2]**	**Message[3]**	***remainder***
`MessageTypeID =0xF2`	`FeatureID`	`CommandID`	`CmdErrorCode =0x00`	Return values, if any.

Whenever the `CommandErrorCode` is not 0x00, it indicates a failure, and the reply message must omit the usual return value(s) and may instead send a UTF8 string of a human-readable error message explaining the failure.

The structure of a `Command` ***reply*** message of a ***failed*** command execution is:

Message[0]	**Message[1]**	**Message[2]**	**Message[3]**	***remainder***
`MessageTypeID =0xF2`	`FeatureID`	`CommandID`	`CmdErrorCode >0x00`	Typically none, but may be a human-readable, UTF8 encoded error message string

Event-Messages

See Events

ToDo: Move EventMessage syntax into this chapter, to make the high-level explanation of Events less confusing to readers who actually do not care about the message-layer.

An `Event`, is a kind of message that is sent autonomously by a device's feature, without having been explicitly requested by the host. Besides the `FeatureID` of the emitting feature, it also includes an `EventID`, which enables each feature to emit multiple types of events.

Events may serve a diverse range of purposes, like:

- Notify the host about state-machine transitions of the Core or any other HDC-feature.
- Transfer a stream of data acquired by an ongoing measurement-activity-feature, by sending each tuple of data as the payload of an event message. Note how a single feature may transmit multiple data-streams concurrently, by using different EventIDs for each stream.
- Enable the firmware of a device to send log messages to the host (infos, warnings, errors, …)

The structure of a `Event` message is as follows:

Message[0]	**Message[1]**	**Message[2]**	***remainder***
`MessageTypeID` = `0xF3`	`FeatureID`	`EventID`	Payload of an event

Cheat-sheet of message syntax

The following table describes the syntax of messages in terms of bytes:

		Message[0]	Message[1]	Message[2]	Message[3]	Message[...]
Command (request)	H→D	MessageTypeID = 0xF2	FeatureID	CommandID	*Command arguments*	
Property-getter (request)	H→D	MessageTypeID = 0xF2	FeatureID	CommandID = 0xF4	PropertyID	
Property-getter (reply)	D→H	MessageTypeID = 0xF2	FeatureID	CommandID = 0xF4	CmdErrorCode	*Property value*
Property-setter (request)	H→D	MessageTypeID = 0xF2	FeatureID	CommandID = 0xF5	PropertyID	*New value*
Property-setter (reply)	D→H	MessageTypeID = 0xF2	FeatureID	CommandID = 0xF5	CmdErrorCode	*Actual new value (May differ!)*
Event (unrequested reply)	D→H	MessageTypeID = 0xF3	FeatureID	EventID	*Event payload*	
LogEvent	D→H	MessageTypeID = 0xF3	FeatureID	EventID = 0xF0	LogLevel	*Log message*
FeatureState TransitionEvent	D→H	MessageTypeID = 0xF3	FeatureID	EventID = 0xF1	StateID *(before)*	StateID *(after)*
EchoMessage (request & reply)	H→D D→H	MessageTypeID = 0xF1	*Echo payload*			

Custom MessageTypeID values

Note how only `MessageTypeID` values `0xF0` to `0xFF` are reserved for HDC protocol purposes.
The remaining range of values is freely available for any custom purpose, like for example:

- Encapsulation for the tunneling of whatever other communication protocol or raw data stream needs to be transmitted through the HDC connection.
- Note how tunneling of one HDC-connection through another HDC-connection can be implemented more elegantly by the sender mapping the MessageTypeID into the range of custom IDs and the receiver restoring the original ID before forwarding said message to the intended sub-system. Similar to how NAT works.
  This technique is preferable to encapsulation, because it does not incur any additional transmission overhead, but it is limited to tunnel up to 15 HDC-connections through an existing one.

Packets

Summary and glossary

Packetizing is the procedure of how a sender wraps a ***message*** as ***payload*** into one or more ***packets*** in a way that allows the receiver to decode ***chunks*** of received bytes back into the original message. When sending many messages or very large messages, they usually are received as a ***burst*** of multiple chunks of bytes.

Multi-packet messages

Most messages fit in a single packet.
Messages larger than 254 bytes must be sent in multiple, consecutive packets.
The first packet contains the first 255 bytes of the message, the second packet the next 255 bytes and so forth until the remainder of the message is sent in a packet containing less than 255 bytes. Should the message size be an exact multiple of 255 bytes, then an empty packet will be signaling that the message transfer is complete.

Multi-packet messages must be sent as strictly consecutive packets. No other packets should be sent in between. (Be especially aware of event-messages not interrupting the transmission of any ongoing multi-packet message!)

Building and decoding of Packets

Messages are encoded into and decoded from a stream of bytes as payload of packets that have the following structure:

Byte Index	Description
`0`	Payload size given as number of 8-bit bytes and subsequently referred to as `PS`. Thus a single packet can contain payloads of up to 255 bytes. Messages larger than 254 bytes are sent as multiple, consecutive packets, the last of which is characterized by containing less than 255 bytes. Messages containing an exact multiple of 255 bytes must be terminated with an additional empty packet (`PS`=0). Empty packets are also allowed on their own, but they are pointless and will be silently ignored.
`1..PS` (or none, whenever `PS`=0)	Bytes of the payload. Usually a single message. Can be empty (`PS=0`). May be only a fragment of the message when this packet or its immediately preceding one(s) have a payload size of `PS`=255.
`PS+1`	Checksum of Payload Two's complement of the sum of all payload bytes. If there's no payload (`PS`=0) then this byte is 0x00 .
`PS+2`	Record Separator (0x1E)

The receiver algorithm assumes that the byte at `buffer[0]` is the size of the payload (`PS`) and uses its value to check whether the byte at `buffer[PS+2]` contains the value `0x1E` (A homage to

ASCII, because that's the code for the `RecordSeparator`) and then it further checks if the checksum byte at `buffer[PS+1]` is valid by checking whether the sum of all payload bytes, plus the checksum byte, yield a value of 0x00. The checksum is computed as the byte-wise sum with numeric overflow at 0xFF.
If the RecordSeparator is not where expected or the checksum is invalid, it assumes a reading-frame-error and skips the first byte and tries again as often as necessary to restore the correct reading frame.
If the buffer does not yet contain any byte at the index `[PS+2]`, then the receiver allows for a certain timeout, based on the assumption that any pending bytes of that packet must arrive as a quick burst. But if that timeout elapses, it also must assume a reading-frame-error (i.e. the tentative `PS` value actually isn't) and therefore skips the first byte and tries again as a way to restore the correct reading frame.
And finally, the payload is then passed to the message parser, which will additionally check whether the given payload is *well-formed*, i.e. whether the first byte of the message is one of the known `MessageTypeID` values and the second byte of the message is a valid `FeatureID` value, and so on.

Beware of the following:

- The byte value 0x1E can also occur in the payload.
- The receiver buffer might still be missing some pending bytes of upcoming chunks.

Serial connections

A point-to-point connection between one device and one host, that allows bi-directional transmission of a stream of bytes.
Typically this can be:

- a UART module on a microcontroller bridged into USB-CDC and exposed as a Virtual-COM-Port on a PC
- a USB-CDC module on a microcontroller, exposed as a Virtual-COM-Port on a PC
- A TCP/IP socket

IDL Descriptors

ToDo: Coin a different term for the IDL stuff and reserve the term ***descriptor*** for the classes used to interface firmware and HDC-device library.

Workflows made possible by the IDL Descriptors

- Design first, implement later:
    - Define an HDC-API in a language neutral manner with a graphical Design-Tool. The tool can auto-generate source-code for Descriptors and Proxies of whatever the targeted implementation languages of Device and Host are.
- Implement Device first:
    - Device developer takes the lead and defines HDC-API by means of descriptors of their preferred HDC library, e.g. C or C++ for microcontrollers or Python for a resource-rich device.
    - Design-Tool connects via HDC with the device and requests the JSON descriptors by means of a Meta-request message. The Device's HDC library is able to generate the JSON descriptors based on its language specific descriptors..
    - Design-Tool can auto-generate documentation and source-code of Proxies tailored to that device in whatever programming language the Host is going to be implemented.
- Implement Host first:
    - Application developer takes the lead and implements a mock-up device that simulates the HDC-API and behavior of a device that does not exist yet.
    - The Host software can be implemented in parallel with said mock-up device, because it is implemented in the same programming language and IDE environment that

JSON as IDL

HDC currently uses JSON as IDL, because:

- It's well-known and widespread and although many developers complain about its limitations, most have extensive experience working with it.Has good support in most languages and frameworks.
- Its syntax is simple enough for it to be generated by the firmware.
- It's human- and machine-readable.
- It's relatively concise. Not as bloated as other alternatives.
- Its syntax is strictly specified in RFC 8259 , unburdening HDC-spec from having to do so.
- JSON Schema allows strict definition of how to validate data structures.
- Syntax and schema can be validated reliably by plenty of already existing tools.

Other languages as IDL

There's no reason to disallow the use of other languages as IDL for HDC APIs.
And there's also no reason to use other languages as IDL for HDC APIs.
Anyways, some options worth keeping an eye on may be:

- CBOR: Concise Binary Object Representation (RFC 8949)
  - Pros:
    - Smaller size..
    - Easier to generate by firmware.
      No string literals and no need for sprintf()
  - Cons:
    - Not human-readable.
    - Not widespread, yet.
- YAML
  - Pros:
    - ?
  - Cons:
    - Benefits over JSON do not create any value for the needs as an IDL.
    - Multiple syntax variants to encode for the same data structures.

Overview of descriptors and their attributes

Some attributes are used across different descriptors, but beware of their slightly different meaning and requirements.
M: Mandatory attribute // O: Optional attribute // Dash: Attribute does not apply

	Type of Descriptor							
Attribute	**Feat**	**Cmd**	**Evt**	**Prop**	**Arg**	**Ret**	**Err**	**State**
`id`	M	M	M	M	-	-	M	M
`name`	M	M	M	M	O	O	O	O
`doc`	O	O	O	O	O	O	O	O
`type`	O	-	-	M	M	M	-	-
`version`	O	-	-	-	-	-	-	-
`states`	O	-	-	-	-	-	-	-
`args`	-	O	O	-	-	-	-	-
`returns`	-	O	-	-	-	-	-	-
`raises`	-	O	-	-	-	-	-	-

History of changes

HDC-spec version	Change description
1.0.0-alpha-11	Major change: Replaced “introspection” with “IDL” architecture. No HDC-internals polluting the Features anymore! No requirement for a Core-Feature anymore! Refactored Core.MaxReqMsgSize property into a Meta-message.
1.0.0-alpha-10	