Host-Device Communication

Specification of a generic host-device communication protocol

Table of Contents

Summary and glossary

HDC is a set of conventions for the implementation of protocols for the communication between a ***device*** and its ***host***.

At the highest level of abstraction, HDC groups capabilities of a device into ***features***, each exposing a set of ***properties***, ***commands*** and ***events***. A host can use ***introspection*** to discover all of these things and obtain further details about each of them.

At the lowest level of abstraction, HDC specifies how the above is implemented by means of ***messages*** and how those are ***packetized*** for transmission over a serial communication layer.

Devices and their host

The HDC specification is not meant for a particular kind of device.
ToDo!

Features

The term ***feature*** is intentionally ambiguous, because HDC handles all features in exactly the same manner, regardless of their nature and purpose.
It's therefore only for the sake of comprehension, that features can be categorized as follows:

- Hardware-Features
    - Represent a device's hardware module.
    - e.g.: a thermostat module, a linear axis, an optics-module, …
- Activity-Features
    - Represent a device's ability to perform a certain operation.
    - e.g.: a measurement operation that acquires data.
- State-Features
    - Provide introspection into (and configuration of) a device's state machine.
    - Note how a state-feature is essentially also an activity-feature, but it's usually a pretty dumb one, that might be more intuitive to comprehend as a state. ;-)
    - e.g.: the error-state, exposing information about the error via its properties.
- … and it's ultimately up to the implementation of each particular device to define what the nature of a feature may be.

Facts about HDC-features:

- The feature abstraction in HDC is meant to match the object-oriented abstraction of the device's implementation, which as a consequence can easily be projected as an equivalent proxy object on the host.
    - If the firmware of the device organizes for example all temperature control related capabilities in a Thermostat object/module, it will likely also make sense for its HDC interface to expose an equivalent Thermostat-feature, which in turn can be handled with a Thermostat-proxy on the host.
    - Note how the HDC commands, properties and events map quite naturally into the methods, properties and events/callbacks of object-oriented code.
    - Feature implementations are consequently easy to reuse across multiple device implementations. (aka "Code reusability" and modularity)
- All commands, properties and events must be implemented in the context of a feature.
    - Except for the `EchoCommand` described further below.
- 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.

Feature States

Each feature may implement its own state-machine.
If it does so, it must expose it via the `FeatureState` property and notify its transitions by means of the `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'}"`
- The FeatureID of activity-features can be used as value of the `Core.FeatureState` property whenever they are active. This does not apply to any activities that may be able to execute concurrently with others.
- Naming recommendations for typical states that may occur in activity-features:
  - Off
  - Initializing
  - NotReady
  - Ready
  - Preparing
  - Acquiring
  - Error

The mandatory Core-Feature

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.
A device core-feature typically may implement:

- A state-machine for the device as a whole(e.g.: Initializing, Ready, Busy, Error, … and the like). (Note how other features may implement a state-machine of their own, too)
- A property for the serial-number of a device.

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

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

- `FeatureTypeRevision`
    - A UINT8 number, to be incremented whenever a feature's implementation changes, without significantly affecting the HDC-interface of the feature. More disrupting changes should preferably be documented by using a different `FeatureTypeName`.

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

- `FeatureTags`
    - A UTF8 string containing a semi-colon delimited list of tags.
    - Tags allow for the flexible categorization of features and the implementation of future hacks and workarounds.
      Some ideas for tags:
        - Types of features: Hardware-feature, Activity-feature, State-feature, …
        - Declaration of more granular interfaces implemented by a feature.
            - E.g.: `ImplementsStateMachine`
        - Injection of further meta-data by means of a key=value syntax
            - E.g.: `ReleaseDate=2028-Nov-24`
    - TBD: Rough guidelines for the tagging system.

Commands

Note how the `EchoCommand` is so exotic (and useless), that when the generic term ***command*** is used, it typically refers to a `FeatureCommand`.

FeatureCommands

`FeatureCommands` are essentially "remote procedure calls" executed on a specific feature of a device. The host sends a ***request*** message to the device, which will acknowledge completion by sending a ***reply*** message to the host:

The structure of a `FeatureCommand` ***request*** message is:

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

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

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

Whenever the `ReplyErrorCode` 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 with more specific details about the failure.

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

Message[0]	**Message[1]**	**Message[2]**	**Message[3]**	***remainder***
`MessageTypeID =0xCF`	`FeatureID`	`CommandID`	`ReplyErrorCode >0x00`	Typically none, but may be a UTF8 string of an error message.

Facts about `FeatureCommands`:

- 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.
  - This requirement applies to both: FeatureCommands and EchoCommands.
- 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 256 commands (including the mandatory ones).

Mandatory FeatureCommands

Every feature must implement the following commands:

CommandID	CommandName	Arguments	Reply
0xF1	GetPropertyName	UINT8: PropertyID	UTF8
0xF2	GetPropertyType	UINT8: PropertyID	UTF8
0xF3	GetPropertyReadonly	UINT8: PropertyID	BOOL
0xF4	GetPropertyValue	UINT8: PropertyID	according to PropertyType
0xF5	SetPropertyValue	UINT8: PropertyID	according to PropertyType
0xF6	GetPropertyDescription	UINT8: PropertyID	UTF8
0xF7	GetCommandName	UINT8: CommandID	UTF8
0xF8	GetCommandDescription	UINT8: CommandID	UTF8
0xF9	GetEventName	UINT8: EventID	UTF8
0xFA	GetEventDescription	UINT8: EventID	UTF8

Developers usually don't need to bother about any of these, as explained further below.

FeatureCommand introspection

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`

ReplyErrorCodes

Each `FeatureCommand` implementation may define its own ReplyErrorCodes as long as they do not collide with the following reserved codes and meanings:

Reply Error Code	Meaning	Context
0x00	No error	All commands
0x01	Unknown feature	Any command addressed to a non-existent feature
0x02	Unknown command	Any command not implemented on the given feature, or when attempting to introspect about thereof.
0x03	Incorrect command arguments	All commands
0x04	Command not allowed now	All commands
0x05	Command failed	All commands
0xF0	Unknown property	GetPropertyName GetPropertyType GetPropertyReadOnly GetPropertyValue SetPropertyValue
0xF1	Invalid property value	SetPropertyValue
0xF2	Property is read-only	SetPropertyValue
0xF3	Unknown event	GetEventName GetEventDescription

Reply error codes are kept intentionally simple, because more sophisticated error scenarios can resort to Log-events to convey more detailed information about the situation. Note that it's allowed to send events while the host is awaiting for the reply to a command.

Some kinds of errors can't be reported via reply messages and will instead be notified via Log-events. This is especially the case for reading-frame errors and errors that can't be differentiated thereof, like "`Unknown MessageTypeID`", or requests that are larger than allowed by the `CoreFeature.MaxReqMsgSize` property.

EchoCommand

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

The structure of a `EchoCommand` ***request*** message is as follows:

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

→ The corresponding reply will be identical to the request and will be sent as quickly as possible.

Facts about `EchoCommands`:

- Since this resembles the request/reply pattern of the more usual commands, this is named `EchoCommand`, 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 `EchoCommand.`
- Hosts may use the `EchoCommand` 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: `FeatureCommand`s and `EchoCommand`s.
- There's no introspection on the `EchoCommand`.
    - Hosts can simply count on it being implemented on any device.

Properties

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

- `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 `FeatureCommand` 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!

Mandatory properties

Every feature must implement the following mandatory properties:

PropertyID	PropertyName	DataType	Readonly	Immutable
0xF0	FeatureName	UTF8	Yes	Yes
0xF1	FeatureTypeName	UTF8	Yes	Yes
0xF2	FeatureTypeRevision	UINT8	Yes	Yes
0xF3	FeatureDescription	UTF8	Yes	Yes
0xF4	FeatureTags	UTF8	Yes	Yes
0xF5	AvailableCommands	BLOB	Yes	Yes
0xF6	AvailableEvents	BLOB	Yes	Yes
0xF7	AvailableProperties	BLOB	Yes	Yes
0xF8	FeatureState	UINT8	Yes	No
0xF9	LogEventThreshold	UINT8	No	No

Developers usually don’t need to bother about any of these, as explained further below.

Property introspection

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 ReplyErrorCode=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

A `FeatureEvent`, just named ***event*** for short, 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 `FeatureEvent` message is as follows:

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

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**
0xF0	Log	[0]: UINT8 LogLevel [remainder]: UTF8 LogMessage
0xF1	FeatureStateTransition	[0]: UINT8 PreviousStateID [1]: UINT8 NewStateID

Developers usually don't need to bother about any of these, as explained further below.

Data types

Property introspection uses the following `DataTypeCode`s:

DataTypeCode	DataType
`0x01`	UINT8
`0x02`	UINT16
`0x04`	UINT32
`0x11`	INT8
`0x12`	INT16
`0x14`	INT32
`0x24`	FLOAT
`0x28`	DOUBLE
`0xB0`	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 0xB0)
  - 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.
    - Exception to the rule: 0xB0 --> BOOL, whose size is 1 byte.

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)

Messages

Types of messages

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

MessageTypeID	Name	Description
0xCE	EchoCommand	A request sent by the host which will be echoed by the device. (Only useful for debugging and testing purposes.)
0xCF	FeatureCommand	Either a request (sent by the host) or a reply (sent by the device) for a command implemented on a specific feature.
0xEF	FeatureEvent	Event raised by a specific feature.

Any other MessageTypeID is reserved for future use and must be treated as a reading-frame-error when deserializing messages.
The actual values of the MessageTypeID are meant to be mnemonics to aid the interpretation of raw message buffers.
The EchoCommand is so exotic and useless, that when the term ***command*** is used, it typically refers to a FeatureCommand.

Overview of message syntax

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

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

(request & reply)	D→H	`= 0xCE`	

Packetizing

Summary and glossary

Packetizing is the way the 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 package 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 package containing less than 255 bytes. Should the message size be an exact multiple of 255 bytes, then an empty package will be signaling that the message transfer is complete.

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

Building and decoding of Packets

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

Byte Index	Description
0	L=Payload size in number of 8-bit bytes Thus a single package can contain payloads of up to 255 bytes. Messages larger than 254 bytes are sent as multiple, consecutive packages, 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 package (L=0). Empty packages are also allowed on their own, but they are pointless and will be ignored.
1 to L (or none, when L=0)	Bytes of the payload. Usually a single message. Can be empty (L=0). May be only a fragment of the message when this package or its immediately preceding one(s) have a payload size of L=255.
L+1	Checksum of Payload Two’s complement of the sum of all payload bytes. If there’s no payload (L=0) then this byte is 0x00 .
L+2	Record Separator (0x1E)

The receiver algorithm assumes that the byte at buffer[0] is the size of the payload (L) and uses its value to check whether the byte at buffer[L+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 is valid by ensuring the the sum of all payload bytes and the checksum byte yield a value of 0x00. The checksum is computed literally 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 [L+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 L 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.

Implementation details

Mandatory stuff looks like boilerplate, but it isn't!

Developers usually don’t need to bother about the mandatory stuff required by the HDC specification, because it’s taken care of in the `hdc.c` module of the firmware and the proxy base-classes on the host.

Hence, mandatory and custom implementations are kept separate in the source-code, thus improving the readability of custom implementations, because those won’t become polluted with any mandatory boilerplate.