can you find me a document that describes exactly how hart protocol commands are defined

Searched the web

The authoritative source for this is **FieldComm Group** (the organization that owns and maintains the HART protocol standard). They publish the exact specifications that define how commands are structured:

Core documents you want:

1. **Universal Command Specification (HCF_SPEC-127)** – contains definitions of all HART Protocol Universal Commands, and HART compatible devices must implement all universal commands exactly as described within this specification. FieldComm Group
2. **Command Summary Specification** – this is the prerequisite/foundation doc; conformance to the Universal Command Specification requires Command Summary Specification conformance as a prerequisite. It defines the general rules for how *any* command (universal, common practice, or device-specific) must be structured. FieldComm Group
3. **Common Practice Command Specification (HCF_SPEC-151)** – an Application Layer specification that builds on the Application Layer Requirements found in the Command Summary Specification, covering the less universal but still standardized commands. FieldComm Group
4. **Common Tables Specification (HCF_SPEC-183)** – many Universal Commands refer to tables from this spec, and when Common Tables are referenced, data from the tables must be used exactly as specified. FieldComm Group

5. **Data Link Layer Specification** – defines the HART frame format, device addressing structure, and error detection coding — this is what wraps the command bytes for transmission. FieldComm Group
6. **Response Code Specification** – specifies all requirements pertaining to the assignment, classification, definition, and application of Response Codes for every command, whether protocol-defined or manufacturer device-specific. FieldComm Group

You can find the full list here: **https://www.fieldcommgroup.org/hart-specifications**

One catch: these specifications can be viewed online at the FieldComm website but cannot be downloaded by non-member companies — they can only be purchased as a bound book, so full access typically requires membership or purchase. Texas Instruments

If you want something free and readable right now that explains the *structure* of how commands are defined (frame format, command classes, byte layout) without needing FieldComm membership:

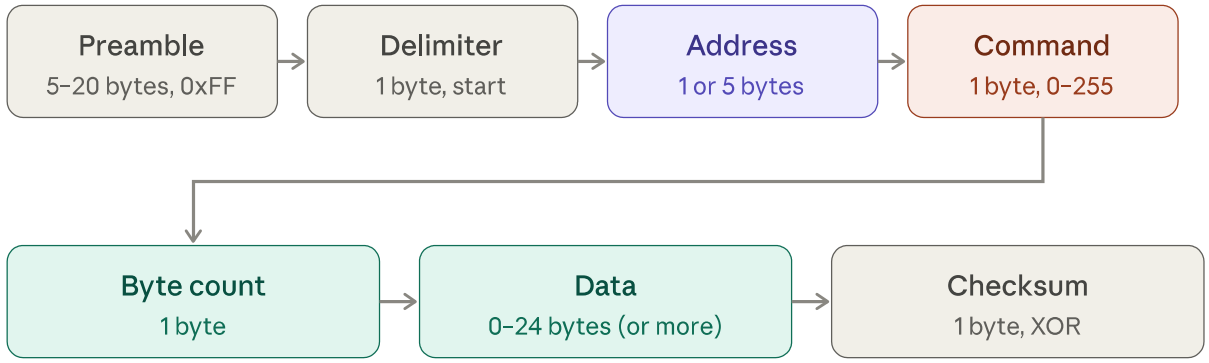
- TI's application note "A Basic Guide to the HART Protocol" (https://www.ti.com/lit/pdf/slaaeh0) — walks through the HART frame structure, the 9 sections of the HART data frame, and how universal commands (0-30, 38, 48) vs common-practice commands (32-121) are numbered and categorized. Texas Instruments
- FieldComm Group's own open GitHub example spec for a reference device: https://github.com/FieldCommGroup/HART-IP-Developer-Kit/blob/master/doc/hart-ip-flowdevice-spec.md — shows a real worked example of how a device documents its command support.

Want me to pull up more detail on any one of these — e.g., the actual byte-level layout of a command frame?

please show me the actual byte level layout of a command frame

Architected sequential byte-frame diagram with color-coded field categories

Good, now I have all the byte-level details. Let me lay this out visually since it's a structural byte layout.
Here's the byte-level layout of a HART request/response frame:



Gray = framing · Purple = address · Coral = command number · Teal = command's own data

A few notes on how the fields work, moving left to right:

Preamble — consists of repetitive bytes containing 0xFF, and allows the receiver(s) to synchronize. Bostontech

Delimiter — may have one of several values, indicating the type of message: master to slave, slave to master, or burst message from slave; also the address format: short frame or long frame. In HART 6+, it also encodes the length of an optional expansion field. Romilly

Address — includes both the master address (a single bit: 1 for a primary master, 0 for a secondary master) and the slave address. In the short frame format, the slave address is 4 bits containing the polling address (0 to 15). In the long frame format, it is 38 bits containing

a unique identifier for that particular device, with one bit also used to indicate if a slave is in burst mode. Romilly

Command — universal commands are in the range 0 to 30; common practice commands are in the range 32 to 126; device-specific commands are in the range 128 to 253. HART rev. 6 introduced 16-bit extended commands, which put 31 (hex 1f) in this byte with the 2-byte command number as the first 2 bytes in the data field. Romilly

Byte count — a single-byte field indicating how many more bytes are left for the message to be completed, excluding the checksum. The receiver uses it to know when the message is done. Instrumentation Tools

Data — this is where the command-specific request or response payload lives — content and length here depend entirely on which command is being sent, and this is where the actual "command definition" from the specs you asked about earlier matters most.

Checksum — an XOR of all the bytes starting from the start byte and ending with the last byte of the data field, including those bytes. Wikipedia

One thing worth flagging: response frames insert two extra status bytes (response code + device status) right after the byte count, before the data field — that's how a device reports errors alongside the data you asked for. Want me to break down what a specific command's request/response data field looks like, e.g. Command 0 or Command 1?

what are the various values for the delimiter

Reconciled conflicting sources on HART delimiter byte structure

The delimiter byte packs several pieces of information into one byte, and its exact numeric value depends on the frame type. Here's the breakdown, drawing from several public technical references (the official FieldComm bit-level spec is paywalled, so treat the exact

bit positions below as the widely-corroborated community convention rather than a verbatim quote from the standard):

What the delimiter byte encodes

The start character may have one of several values, indicating the type of message: master to slave, slave to master, or burst message from slave; also the address format: short frame or long frame. Since HART rev. 6, it also indicates the number of bytes in the expansion field. Romilly

Message type (lower bits)

1 = BACK, Burst Acknowledge sent by a burst-mode device. 2 = STX, sent from master to field devices. 6 = ACK, the slave's acknowledge response to an STX frame. Wikipedia

Address format (top bit)

Bit 7, if high, means use the unique (5 byte / long-frame) address; if low, use the polling (1 byte / short-frame) address. Wikipedia

Putting those together, the common hex values you'll actually see on the wire are:

Hex value	Binary	Meaning
0x02	0000 0010	STX, short-frame address (master request)
0x06	0000 0110	ACK, short-frame address (slave response)
0x01	0000 0001	BACK, short-frame address (burst message)
0x82	1000 0010	STX, long-frame address (master request)
0x86	1000 0110	ACK, long-frame address (slave response)
0x81	1000 0001	BACK, long-frame address (burst message)

One more wrinkle (HART rev. 6+): the number of expansion bytes present is indicated by bits 6 and 5 in the start delimiter — a feature introduced in rev. 6 whose use was, as of that writing, still largely undefined, and the expansion field itself allows up to 3 additional bytes to be inserted between the address and command fields. Romilly Romilly

Worth flagging: at least one independent write-up (a hobbyist's protocol notes) describes the address-format bit the opposite way round from the sources above, which tells you the exact bit polarity isn't 100% consistently documented outside the official (paywalled) spec — if you're implementing against a real device, it's worth confirming against that device's manual or a HART analyzer capture rather than trusting any single secondary source.

how do i parse the hart protocol response

Architected parsing guide with worked example and reusable code

Parsing a HART response is mostly about walking the frame fields in order and knowing where the two status bytes fit in. Here's the step-by-step logic, then a worked example, then a small parser you can actually run.

Parsing steps

1. **Skip the preamble** — read bytes until you hit something that isn't `0xFF`. A preamble count of between 5 and 20 is typical. Kovleventer
2. **Read the delimiter** (1 byte) — its content denotes if the frame is a request from a master, a response from a slave, or a request from a slave in burst mode, and also indicates whether the address used is a polling address or a unique ID. Bit 7 tells you the address length that follows. Instrumentation Tools
3. **Read the address** — 1 byte if short-frame, 5 bytes if long-frame (per bit 7 of the delimiter).

4. **Read the command byte** (1 byte).
5. **Read the byte count** — indicates how many more bytes are left, excluding the checksum. Instrumentation Tools
6. **If this is a response frame, the first two bytes of what byte-count covers are the status bytes, not data:**
   - the first byte is the response code — if it's 0x00, communication is good and the device processed the command without error. Control Engineering
   - the second byte is the device status, indicating the current operating health of the field device as a whole. Romilly
   - if the response's top bit (bit 7) is set, it signals a communication error and the device-status byte has no practical meaning in that case. Scribd
7. **The rest of byte-count is the actual command data** (structure depends entirely on which command number this is — that's the part you need the per-command spec for).
8. **Read and verify the checksum** — an exclusive-or (longitudinal parity) of all previous bytes from the start character onward. Recompute it yourself and compare. Romilly

Worked example

Here's a real captured exchange for a long-frame query and response — the field device's response begins with the byte 86, then the same 5-byte address, then command 3, followed by 1A (byte count), then 00 00 as the two status bytes, then the actual data: a IEEE-754 float for loop current, a unit code, and further values. kartik's stuff

That `00 00` right after the byte count is exactly the response-code/device-status pair described above — everything after it is Command 3's real payload.

Now let me build you a small parser that does this generically:

Ran 2 commands, created a file

Runs correctly and parses the example cleanly — response code and device status both come back `0x00` (healthy), and the data field is left as raw bytes for you to decode using the specific command's data-field spec (e.g. for Command 3, the first 4 bytes there are an IEEE-754 float for loop current).

Presented file

A few things worth knowing when you use this on real captures:

- **The data field is command-specific.** This script only handles the data-link framing (preamble/delimiter/address/command/status/checksum). To decode the *contents* of the data field for a given command, you need that command's data-byte table from the Universal/Common Practice Command Specification, or the device's own manual.
- **Multi-byte numeric fields in HART are big-endian**, and floats are IEEE-754 single precision — so `40 80 11 92` unpacks with `struct.unpack('>f', bytes.fromhex("40801192"))`.
- **Communication errors**: if the response code's top bit is set (`response_code & 0x80`), the device status code has no practical meaning for that response — that top bit alone tells you the transmission itself had a problem, distinct from a normal command-processing error. Scribd
- If you're reading directly off a serial port rather than a pre-captured frame, you'll also need to strip the parity/start/stop bits at the UART level before this logic applies — that's a separate concern from the byte parsing shown here.

Want me to extend this to decode a specific command's data field (e.g. Command 0's device identification response, or Command 3's dynamic variables)?

Hart parse
PY
Download