Independent Submission
Internet-Draft
Intended status: Informational
Expires: 2 November 2026

Y. Sun
Independent Contributor

1 May 2026

HACP: A Capability-Contract Protocol for AI Agents and Edge Hardware
draft-sunyi-hacp-protocol-00

Abstract

HACP (Hardware Agent Capability Protocol) is a JSON-RPC 2.0 transport- agnostic protocol that lets a Large Language Model (LLM) agent — or any program acting on its behalf — discover, plan, execute, observe, and audit operations on physical edge hardware (GPIO, I2C, UART, sensors, system telemetry, files) through a single, stable, security-checked surface.

HACP sits below higher-level agent protocols such as the Model Context Protocol and above the operating system. It is designed to make heterogeneous edge devices interoperable with diverse AI agent implementations without requiring either side to know the details of the other.

This document specifies the HACP wire format, core methods, lifecycle, error model, security requirements, and audit semantics. Streaming, attestation, and MCP-bridge profiles are sketched as optional or preview.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 2 November 2026.

Table of Contents

1. Introduction

LLM-based agents are increasingly asked to operate on physical hardware: read sensors, set GPIO lines, query system telemetry, manage files. Today, each agent framework reinvents this surface, often by shelling out to operating-system primitives. The result is fragile, unauditable, non-portable, and hard to secure.

HACP defines a small, stable JSON-RPC 2.0 protocol that an agent or an agent-side framework speaks to a host-side daemon. The daemon owns the hardware, enforces security policy, and emits an audit trail. The agent owns the intent, the planning, and the natural-language surface.

Goals:

1. Capability contract, not a driver API. HACP describes what an agent may ask for (read this GPIO, list I2C buses, write this file under a safe path). It does not describe how the daemon talks to silicon.

2. Safe by construction. Every operation passes through an allowlist, a path guard, a per-tool risk level, and an audit log. Agents cannot escape into a shell.

3. Plannable, not just callable. A request expresses an intent and an ordered list of steps; the server executes them as a single task with one task identifier, one status, and one audit trail.

4. Transport-neutral. The wire encoding is JSON-RPC 2.0 framed by line-delimited JSON; the carrier may be a Unix domain socket, a TCP socket over mutual TLS, an SSH stream, or a WebSocket.

5. Portable across edge boards. A capability name (e.g. "gpio.set", "hw.i2c.list", "sys.thermal") means the same thing on different physical platforms.

6. Forward-compatible. New methods, new tools, and new fields can be added without breaking existing clients.

Non-goals: HACP does not prescribe an agent SDK shape, a hardware description schema, a deployment manifest format, or an internal bus between daemons. Those are the subject of separate companion specifications outside this document's scope.

1.1. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The following terms are used:

Agent: The component that initiates HACP requests. Typically an LLM-based program or a framework on its behalf.

Server: The host-side component that owns hardware access and serves HACP. Also called the "daemon" in this document.

Tool: A single named capability registered at server startup. A tool has a name, a risk level, a JSON Schema for its arguments, and a validate/execute pair on the server side.

Task: An intent plus an ordered list of steps. Each step invokes a single tool. A task has exactly one identifier and one terminal status.

Session: An authenticated context between an agent and a server, identified by a server-generated session identifier.

A conformant server is one that implements the methods marked "Required" in Section 3 and obeys the framing, error, and lifecycle rules in Section 2 and Section 5.

A conformant client is one that issues only well-formed requests per Section 2, honors the lifecycle in Section 5, and treats unknown optional fields as defined in Section 2.5.

2. Wire format

2.1. JSON-RPC 2.0

HACP MUST use JSON-RPC 2.0 [JSONRPC] over UTF-8 encoded JSON [RFC8259]. Every request is a JSON object with at least:

```
{ "jsonrpc": "2.0", "id": 7, "method": "tool.list",
  "params": { "session_id": "..." } }
```

Every response is either a result:

```
{ "jsonrpc": "2.0", "id": 7, "result": { ... } }
```

or an error (see Section 6):

```
{ "jsonrpc": "2.0", "id": 7,
  "error": { "code": -32602, "message": "Invalid params" } }
```

2.2. Framing

The default framing is line-delimited JSON: each request and response is a single JSON document followed by exactly one LF (0x0A). Clients MUST NOT split a single document across multiple LF boundaries. Servers MUST accept arbitrary whitespace between documents.

Alternative framings (Content-Length headers, WebSocket frames) MAY be negotiated by transport binding documents (see Section 10).

2.3. Identifiers

* id — JSON-RPC request identifier; MUST be unique per connection while outstanding. Strings, integers, or null are allowed; null indicates a notification (no response expected).

* session_id — opaque server-generated string returned by session.open. MUST be at most 64 bytes drawn from [0-9a-zA-Z_-].

* task_id — opaque server-generated string returned by task.submit. Same constraints as session_id.

2.4. Encoding

UTF-8 only. Numbers fit in IEEE 754 double precision unless explicitly declared otherwise in a method’s schema. Binary payloads (for example file.read contents) MUST be encoded using base64 as defined in [RFC4648].

2.5. Forward compatibility

Servers MUST ignore unknown fields in params. Clients MUST ignore unknown fields in result. New error codes MUST be additive (no re-purposing of existing numbers). Tool authors SHOULD version breaking-change tools by appending .v2 (for example gpio.set.v2) rather than mutating the original.

3. Core methods

The following methods make up the HACP Core and MUST be implemented by every conformant server.

Method	Required	Purpose
session.open	Yes	Establish an authenticated session
session.close	Yes	Tear down a session
tool.list	Yes	Enumerate capabilities available
task.submit	Yes	Submit an intent and ordered steps
task.get	Yes	Poll a task’s status and per-step results
task.cancel	Yes	Request cooperative cancellation

Table 1

Optional method namespaces:

Namespace	Optional	Purpose
task.events	SHOULD	Server-sent task progress (see Section 8)
attest.*	MAY	Device or agent attestation (see Section 11)
mcp.*	MAY	MCP-bridge introspection (see Section 10.1)

Table 2

3.1. session.open

Establish a session. The server MUST allocate a fresh session_id and MAY advertise capability flags.

Request params:

Field	Type	Required	Notes
client_name	string	SHOULD	Free-form e.g. "ai-shell"
client_version	string	SHOULD	Semantic version
protocol_version	string	MAY	Highest HACP version supported

Table 3

Result:

```
{
  "session_id": "01J9...XYZ",
  "capabilities": ["CAP_GPIO_RW", "CAP_FILE_READ"],
  "protocol_version": "0.1.0"
}
```

The server MAY reject session.open with RPC_PERMISSION_DENIED if authentication fails. Authentication is transport-bound (see Section 10).

3.2. session.close

Request params: { "session_id": "..." }

Result: { "ok": true }

Closing a session MUST cancel any tasks still owned by it.

3.3. tool.list

Request params: { "session_id": "..." }

Result:

```
{
  "tools": [
    {
      "name": "gpio.set",
      "version": 1,
      "risk_level": 2,
      "timeout_ms": 1000,
      "supports_rollback": true,
      "description": "Set a GPIO line to 0 or 1.",
      "params_schema": { "type": "object" }
    }
  ]
}
```

risk_level is one of:

Level	Name	Meaning
0	safe	Pure read; no side effects
1	low	Local mutation, easily reversible
2	medium	Physical actuation or cross-process effect
3	high	Irreversible, destructive, or safety-critical

Table 4

Clients MUST treat unknown additional tool fields as opaque and preserve them when forwarding.

3.4. task.submit

A task is an intent plus an ordered list of steps. Each step is a single tool invocation. Steps run sequentially within a task. Two tasks in the same session MAY run concurrently if the server supports it.

Request params:

```
{
  "session_id": "...",
  "task": {
    "intent": "Read sensor and toggle status LED",
    "steps": [
      { "tool": "i2c.read",
        "args": { "bus": 1, "addr": "0x48", "reg": "0x00", "len": 2 } },
      { "tool": "gpio.set",
        "args": { "line": 17, "value": 1 } }
    ],
    "constraints": {
      "max_duration_ms": 5000,
      "abort_on_step_failure": true,
      "max_risk_level": 2
    }
  }
}
```

Result:

```
{ "task_id": "01J9...ABC", "status": "QUEUED" }
```

The server MUST validate every step’s tool name and argument schema before returning. If any step fails validation, the entire submission is rejected with RPC_INVALID_PARAMS and no step is executed.

The server MUST reject any step whose tool’s risk_level exceeds the session’s permitted maximum (default 2; configurable via constraints.max_risk_level only if the session is permitted to relax it).

3.5. task.get

Request params: `{ "session_id": "...", "task_id": "..." }`

Result:

```
{
  "task_id": "01J9...ABC",
  "status": "SUCCESS",
  "intent": "...",
  "steps": [
    { "tool": "i2c.read", "status": "SUCCESS",
      "result": { "data": "ABcd" }, "latency_ms": 4 },
    { "tool": "gpio.set", "status": "SUCCESS",
      "result": { "line": 17, "value": 1 }, "latency_ms": 1 }
  ]
}
```

Task status is one of: QUEUED, RUNNING, SUCCESS, FAILED, CANCELLED. Step status uses the same enum (without QUEUED).

A failed step MUST include an "error" string. The whole task fails fast unless abort_on_step_failure: false was set.

3.6. task.cancel

Request params: { "session_id": "...", "task_id": "..." }

Result: { "task_id": "...", "status": "CANCELLING" }

Cancellation is cooperative. Steps in progress are signaled; the task moves to CANCELLED only after the current step yields. Servers MUST NOT force-kill a step that has begun a hardware transaction (for example, an in-flight I2C write) just because cancel arrived.

4. Capability namespaces (informative)

Tool names are dot-separated. The first segment is a namespace; the rest is implementation-defined. The following namespaces are reserved by this specification:

Namespace	Purpose	Examples
hw.*	Read-only hardware discovery	hw.gpio.list, hw.i2c.list
gpio.*	GPIO read/write	gpio.get, gpio.set
i2c.*	I2C transactions	i2c.read, i2c.write
uart.*	Serial / UART	uart.write, uart.read
file.*	Path-guarded filesystem	file.read, file.write
sys.*	Linux system telemetry	sys.cpuinfo, sys.thermal
proc.*	Process control	proc.exec, proc.signal

Table 5

Vendors SHOULD prefix custom namespaces with their reverse-DNS, for example com.acme.modbus.read, to avoid collisions.

5. Lifecycle

A session begins with session.open and ends with session.close or an idle timeout. While a session is active, an agent MAY submit any number of tasks. Each task progresses through:

QUEUED -> RUNNING -> { SUCCESS | FAILED | CANCELLED }

Idle sessions SHOULD be reaped after a server-defined time-to-live (default 300 seconds); reaping MUST run session.close semantics, including cancelling any in-flight tasks.

A server MUST reject any post-close request that uses the closed session identifier with RPC_SESSION_INVALID.

6. Error model

HACP reuses the JSON-RPC 2.0 standard codes:

Code	Name	Meaning
-32700	Parse error	Malformed JSON
-32600	Invalid Request	Not a valid request object
-32601	Method not found	Unknown method
-32602	Invalid params	Missing/wrong params
-32603	Internal error	Server bug

Table 6

Plus HACP-specific codes (range -32099 to -32000):

Code	Name	Meaning
-32000	RPC_SESSION_INVALID	session_id unknown or expired
-32001	RPC_TASK_NOT_FOUND	task_id unknown for session
-32002	RPC_TOOL_NOT_FOUND	Step references unregistered tool
-32003	RPC_PERMISSION_DENIED	Allowlist or risk guard rejected
-32004	RPC_RESOURCE_BUSY	Hardware contention

Table 7

Implementations MAY add codes in -32099 to -32005. They MUST NOT re-use the codes above with different meaning.

The optional error.data field MAY carry a structured detail object:

```
{ "code": -32003, "message": "Permission denied",
  "data": { "tool": "gpio.set",
            "reason": "max_risk_level=1 < tool=2" } }
```

7. Audit semantics

Every accepted task.submit, every step start, every step finish, and every session.open and session.close MUST generate an audit event.

The recommended event shape is illustrative; a full audit log format is the subject of a separate companion document:

```
{
  "ts":         "2026-04-19T22:48:01.234Z",
  "event":      "task.step.finish",
  "session_id": "...",
  "task_id":    "...",
  "step_index": 0,
  "tool":       "gpio.set",
  "status":     "SUCCESS",
  "latency_ms": 1,
  "args_hash":  "sha256:..."
}
```

Audit logs SHOULD be append-only and rotation-friendly (newline-delimited JSON). Implementations MAY chain consecutive records cryptographically (for example, by including the SHA-256 hash of the previous record in the next record) to provide tamper evidence.

8. Streaming extension (optional)

The polling pattern (task.submit followed by repeated task.get) is sufficient but verbose. A server MAY advertise the CAP_STREAMING_TASK_EVENTS capability in session.open. If present, clients MAY call:

```
{ "method": "task.events.subscribe",
  "params": { "session_id": "...", "task_id": "..." } }
```

The server then sends one or more notifications (JSON-RPC objects with no id):

```
{ "jsonrpc": "2.0", "method": "task.event",
  "params": { "task_id": "...", "step_index": 0,
              "phase": "start",
              "ts": "2026-04-19T22:48:01.234Z" } }
```

Phases: start, progress, finish, task_finish. The subscription ends implicitly when the task reaches a terminal status.

9. Versioning

The protocol uses Semantic Versioning [SEMVER] as MAJOR.MINOR.PATCH:

* PATCH: clarifications and editorial fixes only.

* MINOR: additive — new methods, fields, or error codes. Older clients still work.

* MAJOR: breaking. Servers MAY support multiple majors side by side via protocol_version in session.open.

Servers MUST advertise their highest supported version in the session.open result.

10. Transport bindings

10.1. Unix Domain Socket (default)

* Default path: implementation-defined; the reference implementation uses /run/hearth/hearth.sock when started as a system service or /tmp/hearth/hearth.sock for unprivileged use.

* Permissions: 0660 owned by the server’s user, with a dedicated group. Adding a user to that group is the authentication act.

* Framing: line-delimited JSON.

10.2. MCP bridge (informative)

A small adapter MAY present a HACP server as an MCP server [MCP]. Mapping:

MCP	HACP
tools/list	session.open then tool.list
tools/call	task.submit (single step)
MCP error	HACP error per Section 6

Table 8

10.3. TCP plus mutual TLS (informative)

For multi-host deployments, HACP MAY be served over TCP wrapped in mutual TLS. Authentication is by client certificate; the certificate's subject is mapped to a session policy. The same line-delimited framing applies.

11. Future Considerations

Future revisions of this specification may explore:

* Remote operation across network boundaries with stronger authentication and confidentiality requirements.

* Mutual authentication and authorization between agents and capability providers based on hardware identities.

* Hardware-backed attestation of capability providers (for example, via TPM 2.0 quotes binding the running daemon binary, the kernel, and the device identity to a client-supplied nonce).

* Catalog versioning and capability negotiation evolution as the agent ecosystem matures.

The detailed mechanisms are out of scope for this document.

12. Security Considerations

HACP exposes physical hardware to programs that act on behalf of LLMs. This is inherently security-sensitive.

A conformant server MUST:

1. Allowlist tools at startup. The set of registered tools is fixed per process; agents cannot install new tools at runtime over HACP.

2. Path-guard every filesystem operation. file.* and any tool that accepts a path MUST consult a configured read/write allowlist; reject with RPC_PERMISSION_DENIED on miss.

3. Enforce per-tool risk caps. A session MUST NOT execute a tool whose risk_level exceeds the session's configured maximum.

4. Audit every accepted task (Section 7).

5. Bound resource use. Server MUST reject task.submit when internal queues are full.

A conformant server SHOULD:

* Run as an unprivileged user, joining only the groups required for the buses it exposes.

* Apply rate limits per (session_id, tool) pair.

* Apply per-tool timeouts.

* Drop privileges and apply syscall filtering before serving.

Servers MUST NOT expose a proc.exec-style escape hatch in the default tool set. Operators that need shell access MUST opt in explicitly and MUST flag such tools as risk_level: 3.

The default Unix Domain Socket transport relies on filesystem permissions for authentication. Operators deploying HACP over network transports MUST apply mutual TLS or an equivalent authentication mechanism, as the JSON-RPC envelope itself carries no confidentiality or integrity protection.

The audit log is the primary mechanism for detecting and investigating abuse. Operators SHOULD store audit logs on append-only media or use cryptographic chaining (Section 7) to detect tampering after the fact.

A malicious or compromised agent that gains a HACP session is limited by the configured tool allowlist, risk cap, and path guard. HACP does not protect against an attacker who can reconfigure the server itself (for example, by editing its configuration file as root); operators are expected to apply standard host hardening to the server’s runtime environment.

13. Privacy Considerations

The audit log records the full sequence of operations performed on behalf of an agent, including a hash of arguments. Operators SHOULD treat audit logs as sensitive: they may incidentally correlate hardware events with user activity (for example, motion sensor reads taken at specific times). Standard log retention, access control, and minimization practices apply.

HACP itself does not transmit personally identifiable information as part of the protocol envelope. Implementations that expose tools returning personal data (for example, a microphone capture tool) MUST classify those tools at an appropriate risk_level and SHOULD require explicit operator opt-in.

14. IANA Considerations

This document has no IANA actions.

A future revision may register a media type for line-delimited HACP framing and a default port number for the optional TCP transport. No registrations are requested at this time.

15. Implementation Status

A reference implementation, the HEARTH project [HEARTH], includes a server (hearthd, written in C) and an agent client (ai-shell). The reference implementation covers all six core methods, the audit log format, the Unix Domain Socket transport, and the MCP bridge. Streaming task events and attestation are partial.

This section is intended to be removed by the RFC Editor before publication.

16. References

16.1. Normative References

[JSONRPC] "JSON-RPC 2.0 Specification", 2013, <https://www.jsonrpc.org/specification>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/rfc/rfc8259>.

16.2. Informative References

[HEARTH] "HEARTH: Open Capability Layer Between AI Agents and Edge Hardware", 2026, <https://github.com/hearthworks/hearthd>.

[MCP] "Model Context Protocol", 2024, <https://modelcontextprotocol.io>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/rfc/rfc4648>.

[SEMVER] "Semantic Versioning 2.0.0", 2013, <https://semver.org>.

Appendix A. Acknowledgments

The author thanks the early reviewers and contributors of the HEARTH project for their feedback on the protocol surface, and the broader LLM agent community whose function-calling conventions informed the design of the task.submit shape.

Author’s Address

Yi Sun
Independent Contributor
Email: syqust@gmail.com
URI: https://github.com/hearthworks