

CHLORINE SYSTEMS

The Chloride Protocol

Specification and Implementation Notes

Version 1.0 — Experimental

Chlorine Systems

October 24, 2026

chloride:// | Bytecode Streaming Architecture | Draft Specification

Abstract

This document specifies Chloride, an experimental bytecode streaming architecture for network-delivered interactive applications. Chloride transports WebAssembly (Wasm) modules and associated assets to a host runtime that exposes a constrained rendering and input interface. Its design minimizes document-processing overhead by making executable modules, rather than HTML documents, the primary application payload. This document is an experimental draft; interoperability, security, and host-interface details remain subject to revision.

Contents

1 Introduction

1.1 Purpose and Scope

Chloride defines a lightweight framing protocol and an execution model for network-delivered interactive applications. It treats sandboxed Wasm modules as the primary execution unit, allowing compatible clients to render application output without requiring a conventional DOM-based interface.

Chloride does not replace the Wasm execution specification or define a universal graphics API. It specifies the transport framing and outlines a common host interface for compatible implementations.

1.2 Design Principles

Chloride is guided by four principles:

1. **Scripting-First:** Applications are delivered as compiled modules rather than document trees with scripting layered on top.
2. **Zero-DOM Overhead:** Native runtimes may render directly to host-provided pixel buffers or canvases without constructing an HTML document or CSS layout tree.
3. **Capability-Based Isolation:** Modules receive only the host functions and resources explicitly exposed by the runtime. Sandboxing is mandatory; additional resource limits are required for untrusted modules.
4. **Minimal Framing:** An eight-byte binary header provides message identification, flags, and payload length with low framing overhead.

1.3 Addressing and Port Allocation

Chloride resources use the following URI form:

`chloride://<host>[:<port>]/<path-to-module>`

During experimentation, the default TCP port is 49130, within the experimental range 49128-49150 described by RFC 7605, Section 8. This convention does not constitute an exclusive assignment to Chloride. Implementations **MUST** allow the port to be overridden by configuration.

Port 6767 is a proposed future service port, subject to availability and any applicable registration procedures. It is not an assigned Chloride port.

The `chloride://` scheme is a protocol URI convention in this draft; using it in a browser requires a compatible client or registered URI handler. Browser-based clients also require a supported networking transport: ordinary browser Wasm cannot open arbitrary raw TCP sockets.

2 Transport and Framing

2.1 Connection Model

This draft uses a TCP byte stream. A connection begins with a **HANDSHAKE** frame, followed by capability negotiation before application code is transmitted. TCP does not preserve application message boundaries; receivers **MUST** parse the stream using the frame header and payload length.

The precise handshake payload encoding and negotiation state machine are implementation details still requiring a normative definition in a future revision.

2.2 Frame Structure

Every Chloride frame consists of an eight-byte header followed by exactly the number of payload bytes declared in that header. Header integers use network byte order (big-endian).

Offset	Field	Definition
0x00	Magic	0x43 (ASCII C)
0x01	Frame Type	Message identifier; see Table 2
0x02-03	Flags	16-bit reserved field; MUST be zero
0x04-07	Payload Length	32-bit unsigned payload length

Table 1: Chloride frame header (8 bytes)

Receivers **MUST** reject frames with an invalid magic byte, an unknown frame type, nonzero reserved flags, or a payload length exceeding the receiver’s configured limit. Implementations **SHOULD** use a conservative payload limit and **MUST NOT** allocate memory solely on the basis of an unvalidated length field.

2.3 Frame Types

Value	Identifier	Purpose
0x01	HANDSHAKE	Capability and version negotiation
0x02	CODE_STREAM	WebAssembly module bytes
0x03	ASSET_STREAM	Asset data, including audio and textures
0x04	EVENT_INPUT	Input events sent by the client
0x05	TERMINATE	Connection teardown request

Table 2: Draft frame type assignments

A code stream may span multiple frames. Module completion, chunk ordering, asset identifiers, and input-event encoding **MUST** be defined by the application profile or a future revision of this

specification. A **TERMINATE** frame requests orderly shutdown; the connection **MUST** also be closed if the transport fails or the peer disconnects.

3 Runtime and Security Architecture

3.1 Sandboxing and Resource Limits

Host runtimes **MUST** validate incoming Wasm modules and execute them inside a sandboxed runtime. Modules **MUST NOT** receive implicit access to the host filesystem, operating system, devices, or network.

Hosts **MUST** enforce appropriate memory and execution limits and **SHOULD** provide a way to terminate modules that exceed their resource budgets. Only explicitly permitted host imports may expose capabilities. Wasm sandboxing alone does not guarantee application safety or protect against resource exhaustion.

For deployments across untrusted networks, implementations **SHOULD** authenticate servers and use an appropriately secured transport. The authentication mechanism and transport security profile are not standardized by this draft.

3.2 Host Interface

The following proposed imports illustrate the intended host ABI. They are Chloride conventions, not standardized Wasm functions.

- `env.draw_frame(ptr, size)` — submits a pixel buffer for rendering to the host surface.
- `env.poll_input(ptr)` — retrieves available input events into module memory.
- `env.audio_write(ptr, size)` — submits audio samples to the host audio queue.

A conforming runtime **MUST** define the signatures, memory bounds checks, return values, and error behavior of every exposed import. Implementations **MUST** validate pointer and length arguments before accessing module memory. The ABI above remains provisional until these details are specified normatively.

4 Implementation Guidelines

4.1 Compiling a Wasm Module

A minimal module can be compiled with a suitable Clang and Wasm toolchain. The exact linker options depend on whether the target runtime uses WASI or a custom, freestanding host ABI.

```
1 clang --target=wasm32-unknown-unknown -O2 -nostdlib \  
2 -Wl,--no-entry -Wl,--export-all \  
3 -o app.wasm app.c
```

Listing 1: Illustrative freestanding build

This command is illustrative, not a complete build recipe. The module must export the entry points expected by its target runtime and must not depend on unavailable imports.

4.2 Header Representation

The following C structure illustrates the eight-byte wire layout on toolchains supporting packed structures. Integer fields must be serialized in network byte order; implementations should avoid assuming that native structure layout is itself a portable wire format.

```

1 #include <stdint.h>
2
3 struct __attribute__((__packed__)) chloride_header {
4     uint8_t  magic;
5     uint8_t  frame_type;
6     uint16_t flags; /* Big-endian */
7     uint32_t payload_len; /* Big-endian */
8 };

```

Listing 2: Illustrative C header

4.3 Reference Send Routine

TCP writes may be partial. A sender must loop until the requested bytes have been written or an error occurs. The following POSIX example illustrates complete transmission of one frame.

```

1 #include <arpa/inet.h>
2 #include <errno.h>
3 #include <stdint.h>
4 #include <sys/socket.h>
5
6 struct __attribute__((__packed__)) chloride_header {
7     uint8_t magic;
8     uint8_t frame_type;
9     uint16_t flags;
10    uint32_t payload_len;
11 };
12
13 static int send_all(int fd, const void *buf, uint32_t len) {
14     const uint8_t *p = buf;
15     uint32_t sent = 0;
16
17     while (sent < len) {
18         ssize_t n = send(fd, p + sent, len - sent, 0);
19         if (n < 0 && errno == EINTR)
20             continue;
21         if (n <= 0)
22             return -1;
23         sent += (uint32_t)n;
24     }
25     return 0;
26 }
27
28 int chloride_send_frame(int fd, uint8_t type,
29                        const uint8_t *data, uint32_t len) {
30     struct chloride_header h = {
31         .magic = 0x43,
32         .frame_type = type,
33         .flags = htons(0),
34         .payload_len = htonl(len)
35     };
36
37     if (send_all(fd, &h, sizeof(h)) < 0)
38         return -1;
39
40     if (len != 0 && data == NULL)
41         return -1;
42
43     return len ? send_all(fd, data, len) : 0;
44 }

```

Listing 3: POSIX frame transmission

Production implementations must also define a maximum permitted frame size, handle peer disconnects, and account for platform-specific socket behavior such as broken-pipe signals. A corresponding receive routine must read the complete header and payload before processing a frame.

Status: Experimental specification. This document defines a draft framing format and proposed runtime interface; it does not claim final interoperability or security conformance.

Reference: [RFC 7605 — A Survey of Transport Protocol Port Use](#).