Briefing Document: DMIX128 WebSocket Protocol

Executive Summary

This document provides a comprehensive overview of the DMIX128 WebSocket Protocol, the real-time communication backbone for the DMIX128 digital mixer. The protocol facilitates bidirectional communication between a C++ backend server (mixer-web) and a JavaScript-based graphical user interface (GUI). Built upon the Socket.IO v4+ library, it ensures robust, state-synchronized control across multiple clients.

Key takeaways from the protocol architecture include:

- **Core Technology:** The protocol leverages WebSockets via Socket.IO for its primary communication on port 80. A secondary, direct TCP connection on port 7712 is available for high-frequency internal data streams like VU meters.
- **State-Based Synchronization:** The system operates on a full-state model. A client initiates a session by requesting the complete mixer state via an INIT command, after which it receives incremental updates. State changes from one client are broadcast to all others, ensuring all user interfaces remain synchronized.
- **Hierarchical Parameter Naming:** All mixer parameters are addressed using a consistent, dot-notation schema (e.g., i.0.mix). This structure is composed of a channel type prefix (i for input), a channel index (0), and the parameter name (mix), allowing for precise and unambiguous control over every function.
- **Comprehensive Command Set:** The protocol defines a rich set of JSON-based commands for managing shows, snapshots, and presets for various processing blocks (EQ, compressor, effects, etc.). It also includes commands for synchronizing GUI-specific actions like channel selection and page navigation across all connected clients.
- **Performance Optimization:** To handle the high volume of real-time data, the protocol implements several optimizations. High-frequency VU and RTA data are broadcast as Base64-encoded binary payloads to reduce overhead. The server also employs a "fast path" detection to process simple, single-value parameter changes without the need for full JSON parsing.

1. Protocol Architecture and Core Technology

The DMIX128 protocol is designed for real-time, bidirectional communication between the server and client components.

- **Server:** A C++ application named mixer-web.
- **Client:** A JavaScript-based graphical user interface (GUI).

- **Transport Layer:** The protocol is built on top of **Socket.IO v4+**, which manages the underlying WebSocket connection.

Connection Ports

Port	Protocol	Purpose
80	HTTP / WebSocket	Primary communication channel for the GUI, using Socket.IO.
7712	TCP	Secondary direct connection, typically used for local processes like high-frequency VU meter data transmission from the DSP.

Connection Flow

The connection is established through a standard Socket.IO handshake process:

1. **HTTP Polling:** The client may initially connect via HTTP long-polling.
2. **WebSocket Upgrade:** The connection is upgraded to a persistent WebSocket connection.
3. **Session Initialization:** Upon successful WebSocket connection, the server sends an initialization packet.
4. **Initial State Request:** The client sends an INIT command (["INIT"]) to request the full mixer state.
5. **State Transmission:** The server responds with a comprehensive JSON object containing the entire current state of the mixer.

2. Communication and Message Structure

All communication is encapsulated within the Socket.IO packet format. The core payload is a JSON object or array.

Message Types

The protocol defines three primary JSON payload structures:

1. **Command Messages:** Sent from the client to the server, typically expecting a response.
2. **Command Responses:** Sent from the server to the client in reply to a command.
3. **State Updates:** Broadcast from the server to all clients, representing a change in a mixer parameter. These do not require a response.

Payload Examples

Direction	Type	Example
Client → Server	Set Parameter	["S", {"i.0.mix": 0.7647}]
Server → Clients	Broadcast Update	["U", {"i.0.mix": 0.7647}]
Client → Server	Get Show List	["SHOW_LIST"]
Server → Client	Show List Response	["SHOW_LIST", {"shows": [...]}]

Heartbeat Mechanism

Socket.IO maintains the connection with a simple heartbeat:

- Client sends: "2"
- Server responds: "3"

3. Parameter Naming Convention

A hierarchical dot notation is used to address every controllable parameter in the mixer, providing a structured and predictable schema.

Structure: prefix.index.parameter.subparameter

- **Prefix:** A one or two-letter code identifying the channel type.
- **Index:** The zero-based numerical index of the channel.
- **Parameter:** The name of the function block or setting.

Channel Prefixes

Prefix	Type	Count	Description
i	Input	128	Main input channels
a	Aux	24	Aux bus masters
f	FX	8	Effects bus masters
s	Sub	8	Subgroup bus masters
r	Return	8	FX return channels
v	VCA	16	VCA/DCA groups
x	Matrix	8	Matrix outputs
X	Matrix Send	12	Matrix send channels

m	Master	1	Master LR bus
p	Player	1	Audio player
c	Click	1	Click/Metronome
h	Hardware	N/A	Hardware input gain settings
osc	Oscillator	1	Test oscillator

Key Parameter Categories

The protocol provides comprehensive control over all aspects of the signal chain:

- **Mixing:** .mix (fader), .pan, .solo, .mute, .mgmask (mute group), .vcamask (VCA assignment).
- **Input Processing:** .src (source assignment), .trim, .gain, .phantom, .invert, .delay.
- **Equalizer (.eq):** 4-band parametric EQ with types (bell, hpf, lpf, etc.), frequency, gain, and Q controls for each band (.eq.b1.freq). Includes dedicated high-pass and low-pass filters.
- **Dynamic EQ:** Each EQ band can be configured as a dynamic EQ, with parameters for .thresh, .ratio, .attack, .release, and sidechain controls.
- **Graphic EQ (.geq):** 31-band GEQ with individual band levels (.geq.peak.0 to .geq.peak.30).
- **Compressor (.comp):** Full-featured compressor with .thresh, .ratio, .gain, .attack, .release, .knee, and sidechain filtering. Multiple types (clean, vca, fet, opto) and models are available.
- **Gate (.gate):** Includes .thresh, .depth, .attack, .hold, and .release.
- **De-esser (.ds):** Controlled by .thresh, .freq, .q, and .range.
- **Multiband Compressor (.mcomp):** 4-band compressor with crossover frequencies and full dynamics controls for each band.
- **Sends:** Control over levels (.value), pan, and pre/post settings for Aux sends (.aux.N.*) and FX sends (.fx.N.*).
- **Effects (.f.X):** Control over internal FX processors, including type selection (Reverb, Delay, Chorus, DX480) and all associated parameters (v0, v1, etc.).
- **Processing Flow:** The order of processing blocks (Gate, EQ, Compressor, etc.) can be rearranged using the .flow string parameter (e.g., "gcqdmei").

- **Metadata:** .name (channel name), .color, .linked (stereo linking).

4. Core Commands and Data Management

The protocol includes a rich command set for managing the mixer's state.

State and Data Streams

Command	Direction	Description
INIT	Client → Server	Requests the entire current state of the mixer. The server responds with a complete JSON object.
VU	Server → Clients	Broadcasts VU meter levels for all channels as Base64-encoded binary data.
RTA	Server → Clients	Broadcasts Real-Time Analyzer data.
BMSG	Client → Server	Requests the server to broadcast a message to all other connected clients.

Show & Snapshot Management

A full suite of commands is provided for Shows (entire mixer states) and Snapshots (subsets of a Show).

- **List:** SHOW_LIST, SNAP_LIST
- **Save:** SHOW_SAVE, SNAP_SAVE
- **Load:** SHOW_LOAD, SNAP_LOAD
- **Delete:** SHOW_DELETE, SNAP_DELETE
- **Rename:** SHOW_RENAME, SNAP_RENAME

Preset Management

The protocol allows for saving and loading presets for individual processing blocks. The PRESET_LIST command requires a type argument.

- **Preset Types:** eq, comp, gate, geq, ds, fx, mcomp.
- **Commands:** PRESET_LIST, PRESET_READ, PRESET_WRITE, PRESET_DELETE, PRESET_RENAME.

GUI Synchronization Commands

To ensure a consistent user experience across multiple control surfaces, certain UI actions are broadcast to all clients.

- SELECT_CHANNEL: Syncs the currently selected channel.
- SCROLL_TO_CHANNEL: Syncs the visible channel strip area.
- SWITCH_PAGE: Syncs the current view or page in the GUI.
- SYNC_ID: Syncs a selected channel across clients.

5. Performance and Optimization

Several strategies are employed to ensure low-latency performance, especially for high-frequency data.

- **Targeted Broadcasting:** When a client changes a parameter, the server broadcasts the update to all *other* clients, avoiding redundant traffic to the originator.
- **Fast Path Optimization:** The server uses pattern matching to identify and rapidly process common, simple messages without the overhead of full JSON parsing. This applies to:
    - Single key-value parameter updates (e.g., ["S", {"i.0.mix": 0.5}]).
    - VU and RTA data packets.
- **Optimized VU/RTA Data:** VU meter and RTA data are sent as Base64-encoded binary arrays rather than JSON objects to minimize packet size. Clients can opt out of receiving this data by sending a novurta message.
- **Direct TCP Connection:** A secondary TCP connection on port **7712** is used for internal, high-frequency data loops, such as sending VU meter data from the DSP engine to the mixer-web server application, bypassing the main WebSocket channel.

6. Data Representation

Most continuous parameters are transmitted as normalized floating-point values between 0.0 and 1.0. The client GUI is responsible for converting these values to and from a human-readable format.

Normalized Value Examples

Parameter	Real-World Range	Normalized Range	Conversion
Mix/Fader	-∞ to +6 dB	0.0 – 1.0	Logarithmic
EQ Frequency	20 to 20000 Hz	0.0 – 1.0	Logarithmic
EQ Gain	-15 to +15 dB	0.0 – 1.0	Linear

Comp Ratio	1:1 to 20:1	0.0 – 1.0	Logarithmic
Comp Attack	0.5 to 500 ms	0.0 – 1.0	Logarithmic
Comp Release	10 to 2000 ms	0.0 – 1.0	Logarithmic

A key reference point is the fader's unity gain position ("0 dB"), which corresponds to the normalized value of 0.7647058823531552.

7. System and Security Considerations

- **Security:** The protocol has no authentication enabled by default. For secure environments, network isolation is recommended. The documentation notes that JWT support is available but optional.
- **Error Handling:** The client is responsible for implementing a reconnection strategy in case of connection loss, which involves randomized backoff delays and re-sending the INIT command upon successful reconnection to re-synchronize its state.
- **State Persistence:** The mixer state is automatically saved to disk by the server every 5 seconds as a failsafe.
- **Versioning:** The current protocol is **Version 1**, which is JSON-based. This supersedes a deprecated legacy version (Version 0) that used a string-based format.