FlexRadio

SmartLink API Reference

Revision

Rev. DRAFT

Date

April 2026

Contents

FlexRadio Systems

SmartLink API Reference

Developer reference for integrating with FlexRadio SmartLink — the cloudservice that enables remote access to FlexRadio transceivers over the internet.

March 2026

CONFIDENTIAL

Chapter 1

SmartLink API Reference

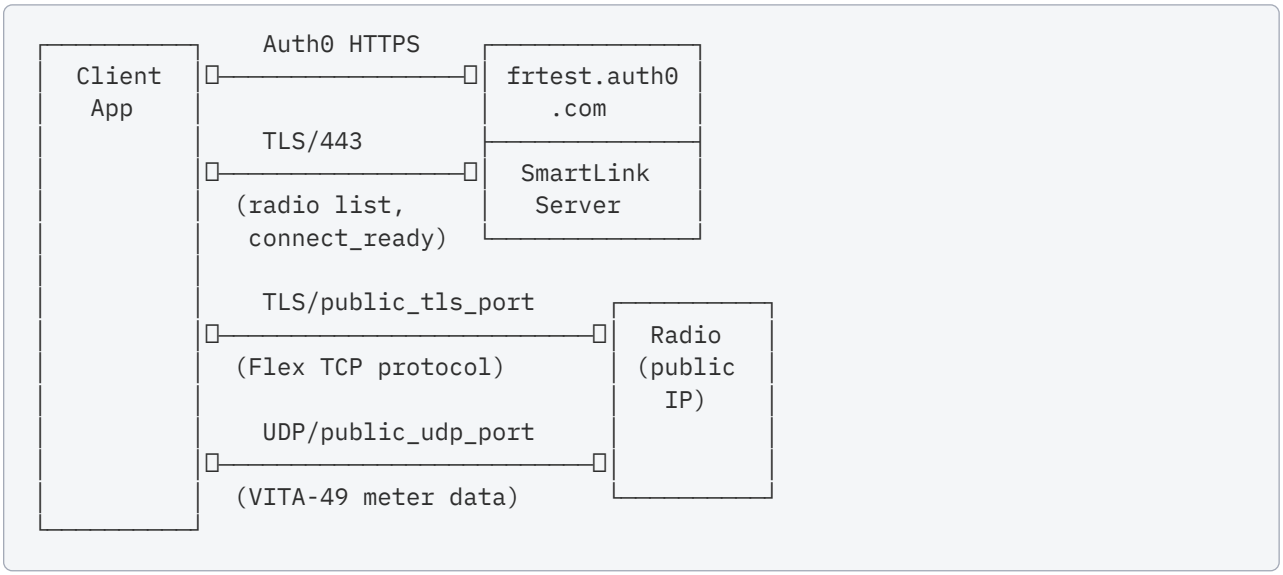
Developer reference for integrating with FlexRadio SmartLink — the cloud service that enables remote access to FlexRadio transceivers over the internet. This document covers the full connection lifecycle: authentication, server communication, radio discovery, command channel establishment, and VITA-49 UDP meter streaming.

Reference implementation: `src/ApiExplorer/SmartLink/SmartLinkService.cs`, `src/ApiExplorer/SmartLi` `src/ApiExplorer/DeviceManagement/DeviceManager.cs`, `src/Plugins/FlexRadio/Streams/VitaUDP.cs`. FlexLib reference: `FlexLib Source/FlexLib/WanServer.cs`, `FlexLib Source/FlexLib/Radio.cs`.

1.1 Overview

A SmartLink session involves three separate network connections:

1. **Auth0 HTTPS** — One-time browser-based login to obtain OAuth tokens
2. **SmartLink Server (TLS/443)** — Persistent SSL connection to `smartlink.flexradio.com` for radio discovery and connection brokering
3. **Radio Direct (TLS + UDP)** — TLS command channel and UDP VITA-49 data stream to the radio's public IP



1.2 Part 1: Authentication (Auth0)

SmartLink uses Auth0 for authentication. The login flow produces two tokens: a short-lived JWT for server authentication, and a long-lived refresh token for obtaining new JWTs without re-login.

1.2.1 Auth0 Parameters

Parameter	Value
Domain	`frtest.auth0.com`
Client ID	FlexRadio's Auth0 client ID (tenant-specific, not shown here)
Response Type	`token` (implicit flow)
Redirect URI	`https://frtest.auth0.com/mobile`
Scope	`openid offline_access email given_name family_name picture`
Device	Application name (e.g., `ApiExplorer`)

1.2.2 Login URL Construction

Build the authorize URL with a random `state` parameter per request:

```
var state = Guid.NewGuid().ToString("N");
var url = $"https://frtest.auth0.com/authorize"
    + $"?client_id=<client-id>"
    + "&response_type=token"
    + $"&redirect_uri={Uri.EscapeDataString("https://frtest.auth0.com/mobile")}"
    + $"&scope={Uri.EscapeDataString("openid offline_access email given_name
      ↪ family_name picture")}"
    + $"&state={state}"
    + "&device=MyApp";

// Optional: pre-fill email for returning users
url += $"&login_hint={Uri.EscapeDataString("user@example.com")}";
```

Open this URL in a browser or embedded WebView. The user enters their FlexRadio SmartLink credentials.

1.2.3 Token Extraction

On successful login, Auth0 redirects to the redirect URI with tokens in the **URL fragment** (after #):

```
]
  ↪ https://frtest.auth0.com/mobile#refresh_token=abc123&id_token=eyJhbG...&token_type=Bearer&state
```

Parse the fragment as query string parameters:

```
var uri = new Uri(redirectUrl);
var fragment = uri.Fragment.TrimStart('#');
var parsed = HttpUtility.ParseQueryString(fragment);

string refreshToken = parsed["refresh_token"];  // Long-lived, store securely
string idToken = parsed["id_token"];            // JWT, expires in 60 seconds
```

Important: If using WebView2, intercept the redirect in `NavigationStarting` and cancel the navigation after extracting tokens. Clear all WebView2 cookies on logout (`CookieManager.DeleteAllCookies()`)

to prevent Auth0 session cookies from auto-completing future logins.

1.2.4 Token Lifecycle

Token	Lifetime	Purpose	Storage
`id_token` (JWT)	**60 seconds**	Authenticate with SmartLink server	Memory only
`refresh_token`	Long-lived	Obtain new JWTs without re-login	Encrypted persistent storage

The JWT payload contains standard OIDC claims. Extract the user's email without a JWT library:

```
// JWT structure: header.payload.signature (base64url-encoded)
var payload = jwt.Split('.')[1];
payload = payload.Replace('-', '+').Replace('_', '/');
switch (payload.Length % 4)
{
    case 2: payload += "=="; break;
    case 3: payload += "="; break;
}
var json = Encoding.UTF8.GetString(Convert.FromBase64String(payload));
using var doc = JsonDocument.Parse(json);
var email = doc.RootElement.GetProperty("email").GetString();
```

1.2.5 Refreshing the JWT

The JWT expires after 60 seconds. Before connecting (or reconnecting) to the SmartLink server, refresh it using the Auth0 delegation endpoint:

```
var content = new FormUrlEncodedContent(new Dictionary<string, string>
{
    ["client_id"] = "<client-id>",
    ["target"]    = "<client-id>",
    ["grant_type"] = "urn:ietf:params:oauth:grant-type:jwt-bearer",
    ["refresh_token"] = storedRefreshToken,
    ["scope"] = "openid email given_name family_name picture"
});

var response = await httpClient.PostAsync("https://frtest.auth0.com/delegation",
  ↪ content);
using var doc = JsonDocument.Parse(await response.Content.ReadAsStringAsync());
string freshJwt = doc.RootElement.GetProperty("id_token").GetString()!;
```

The response JSON contains a new `id_token`. The refresh token itself does not change.

1.3 Part 2: SmartLink Server Communication

The SmartLink server at `smartlink.flexradio.com:443` brokers connections between clients and radios. Communication is line-based text over TLS — each message is a single `\n`-terminated line.

1.3.1 Connection and Registration

```
// 1. TCP + TLS connect
var tcp = new TcpClient();
await tcp.ConnectAsync("smartlink.flexradio.com", 443);

var ssl = new SslStream(tcp.GetStream(), leaveOpen: false);
await ssl.AuthenticateAsClientAsync("smartlink.flexradio.com");

var writer = new StreamWriter(ssl, Encoding.UTF8) { AutoFlush = true };
var reader = new StreamReader(ssl, Encoding.UTF8);

// 2. Register your application (must be the first message sent)
await writer.WriteLineAsync(
    $"application register name=MyApp platform=Windows token={freshJwt}");
```

The `name` identifies your application. The `platform` is your OS. The `token` is a fresh JWT (must be less than 60 seconds old).

1.3.2 Server Messages

After registration, the server sends messages asynchronously. Read them in a loop:

```
while (!cancellationToken.IsCancellationRequested)
{
    var line = await reader.ReadLineAsync(cancellationToken);
    if (line is null) break;  // Server closed connection

    if (line.StartsWith("radio list "))
        HandleRadioList(line["radio list ".Length..]);
    else if (line.StartsWith("application info "))
        HandleApplicationInfo(line["application info ".Length..]);
    else if (line.StartsWith("radio connect_ready "))
        HandleConnectReady(line["radio connect_ready ".Length..]);
    else if (line.StartsWith("radio test_connection "))
        HandleTestConnection(line["radio test_connection ".Length..]);
    else if (line.StartsWith("application user_settings "))
        HandleUserSettings(line["application user_settings ".Length..]);
    else if (line.StartsWith("application registration_invalid"))
        HandleRegistrationInvalid();  // JWT expired or invalid
}
```

1.3.2.1 `radio list` — Radio Discovery

Sent periodically by the server. Contains all radios registered to the authenticated user's account. Multiple radios are separated by `|`; fields within each radio are space-separated `key=value` pairs.

```
radio list serial=1234-5678-9012-3456 model=FLEX-6700 nickname=My_Radio
  ↪ callsign=W5ABC
  version=3.6.8.40296 status=Available public_ip=203.0.113.45 public_tls_port=4992
```

```
public_udp_port=4993 public_upnp_tls_port=-1 public_upnp_udp_port=-1
↪ upnp_supported=0
licensed_clients=2 max_licensed_version=v3 last_seen=2026-02-28T12:00:00Z
gui_client_ips=192.168.1.100 gui_client_programs=SmartSDR
↪ gui_client_stations=MyStation
gui_client_handles=0x12345678|serial=...
```

Key fields for connection:

Field	Description
`serial`	Radio serial number (used for `application connect`)
`public_ip`	Radio's public IP address (for direct TLS/UDP)
`public_tls_port`	Port for TLS command channel (`-1` if no port forwarding)
`public_udp_port`	Port for UDP VITA-49 data (`-1` if no port forwarding)
`public_upnp_tls_port`	UPnP-mapped TLS port (fallback if `public_tls_port` is `-1`)
`public_upnp_udp_port`	UPnP-mapped UDP port (fallback if `public_udp_port` is `-1`)
`status`	`Available`, `In Use`, etc.
`nickname`	User-assigned name (underscores replace spaces: `My_Radio`)

Port selection logic: Prefer manually forwarded ports over UPnP. If both are `-1`, NAT hole-punching is required (not covered here).

```
int tlsPort = publicTlsPort > 0 ? publicTlsPort
            : publicUpnpTlsPort > 0 ? publicUpnpTlsPort
            : -1;  // Requires hole-punch

int udpPort = publicUdpPort > 0 ? publicUdpPort
            : publicUpnpUdpPort > 0 ? publicUpnpUdpPort
            : -1;
```

1.3.2.2 `application info` — Client Info

Sent once after registration. Contains your public IP as seen by the server:

```
application info public_ip=198.51.100.42
```

1.3.2.3 `application user_settings` — User Profile from Server

Sent once after registration. Contains the user's SmartLink account profile data (separate from JWT claims — these come from the SmartLink account registration, not Auth0):

```
application user_settings callsign=N7HQ first_name=Dan last_name=Quigley
```

Field	Description
`callsign`	User's registered callsign
`first_name`	First name from SmartLink account
`last_name`	Last name from SmartLink account

These values supplement the JWT claims (`given_name`, `family_name`, `email`, `picture`). The JWT values take priority for name fields when both are available; the callsign is only available from this message.

1.3.2.4 `radio test_connection` — WAN Connectivity Test Results

Sent in response to a `radio test_connection` request. Contains the results of firewall/NAT probing for a specific radio:

```
radio test_connection serial=1234-5678-9012-3456 success=true
  upnp_tcp_port_working=1 upnp_udp_port_working=1
  forward_tcp_port_working=0 forward_udp_port_working=0
  nat_supports_hole_punch=0 wan_method=UPnP
  public_tls_port=4992 public_udp_port=4993
```

Used by the Remote Configuration dialog to display connectivity test results.

1.3.2.5 `application registration_invalid` — Auth Failure

Sent when the JWT is expired or invalid. Disconnect and re-authenticate.

1.3.3 Keep-Alive

Send a ping to the server every 10 seconds to maintain the connection:

```
// Every 10 seconds:
await writer.WriteLineAsync("ping from client");
```

If the server doesn't receive a ping within its timeout window, it drops the connection.

1.3.4 Requesting a Radio Connection

To connect to a specific radio, send the `application connect` command and wait for `radio connect_ready`:

```
// Send connection request
await writer.WriteLineAsync(
    $"application connect serial={radioSerial} hole_punch_port=0");

// Wait for response (with timeout)
// The server responds with:
//   radio connect_ready serial=<serial> handle=<handle>

string wanHandle = await WaitForConnectReady(radioSerial, timeout:
  ↪ TimeSpan.FromSeconds(15));
```

The `hole_punch_port` is `0` when port forwarding or UPnP is available. For NAT hole-punching, use a random port between 25000-65000.

The `handle` from `connect_ready` is the **WAN connection handle** — you will need it to validate the direct radio connection.

1.4 Part 3: Direct Radio Connection

After receiving `connect_ready`, connect directly to the radio's public IP. This replaces the LAN discovery + TCP connect flow.

1.4.1 Step 1: TLS Command Channel

Connect via TLS to `public_ip:public_tls_port`. The radio uses a self-signed certificate — you must accept it.

```
var radioTcp = new TcpClient();
await radioTcp.ConnectAsync(publicIp, tlsPort);

var radioSsl = new SslStream(radioTcp.GetStream(), leaveOpen: false,
    (sender, cert, chain, errors) => true);  // Accept self-signed certs
await radioSsl.AuthenticateAsClientAsync(publicIp.ToString());
```

Do not send an ICMP ping before connecting — WAN addresses behind NAT are not ICMP-reachable. The ping will fail and waste time.

1.4.2 Step 2: `wan validate` (MUST Be First Command)

The absolute first command sent over the TLS connection must be `wan validate` with the handle from `connect_ready`. **Any other command sent first will cause the radio to disconnect.**

```
wan validate handle=<wanConnectionHandle>

await radioWriter.WriteLineAsync($"wan validate handle={wanHandle}");
```

1.4.3 Step 3: Receive Version and Client Handle

After `wan validate`, the radio sends its standard handshake messages:

```
V3.6.8.40296          ← Version message (first character 'V')
H0000ABCD             ← Client handle message (first character 'H')
```

Critical implementation detail: You must detect these messages by their **first character** (V or H), not by ordinal position. The `wan validate` command you sent in step 2 is also processed by the message parser — if you use a counter-based scheme (message #1 = Version, #2 = Handle), the sent `wan validate` command consumes counter slot #1, shifting the actual V/H messages to incorrect positions where they are parsed as regular text messages. The client handle is never assigned, and all subsequent `handle=0x0` commands fail silently.

FlexLib's `Radio.ParseRead()` uses a content-based `switch (s[0])` check and ApiExplorer follows the same approach:

```
// Correct: content-based detection (order-independent)
if (received && message[0] == 'V' && !versionReceived)
{
    versionReceived = true;
    ParseVersion(message);    // e.g., "V3.6.8.40296"
}
else if (received && message[0] == 'H' && !handleReceived)
{
    handleReceived = true;
    ParseClientHandle(message);  // e.g., "H0000ABCD" → 0xABCD
}
```

Another critical detail for TLS connections: Create your message parser **before** starting the TCP listener. For WAN (TLS) connections, the radio sends V and H immediately after the TLS handshake completes — they are already buffered in the TLS stream by the time your `OnConnected` callback fires. If the listener reads them before the parser exists, both messages are silently dropped and the client handle stays zero.

1.4.4 Step 4: Normal Flex TCP Protocol

After the V/H handshake, the connection operates identically to a LAN connection. Send commands, receive responses and status messages using the standard Flex TCP protocol (`C<seq>|command`, `R<seq>|code|message`, `S<handle>|status`).

Typical initial commands:

```
C1|client program MyApp        ← Identify your application
C2|unsub radio all              ← Unsubscribe from all status updates
C3|sub meter all                ← Subscribe to meter descriptors
```

SmartLink radios should always use Flex TCP ping for heartbeat (not ICMP, which won't reach the radio through NAT):

```
C<seq>|ping
```

The radio responds with `R<seq>|0|` — use the round-trip time for disconnect detection.

1.5 Part 4: VITA-49 UDP Meter Streaming

WAN meter streaming replaces the LAN `client udpport` TCP command with UDP-based registration. The UDP channel carries VITA-49 packets containing meter data.

1.5.1 Socket Setup

Create a raw UDP socket (not `UdpClient` — it has issues with Windows ICMP resets). Bind to a local port, and configure it to survive NAT port-unreachable responses:

```
var socket = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
  ↪ ProtocolType.Udp);
socket.ExclusiveAddressUse = true;
socket.ReceiveBufferSize = 150_000 * 5;

// CRITICAL: Disable WSAECONNRESET on Windows.
// Without this, the first udp_register (sent before NAT is established)
// triggers an ICMP port-unreachable response, which Windows converts into
// a SocketException (WSAECONNRESET/10054) on the next ReceiveFromAsync,
// killing the receive loop. FlexLib avoids this via BeginReceiveFrom
// (which doesn't throw); with ReceiveFromAsync you must disable it.
const int SIO_UDP_CONNRESET = unchecked((int)0x9800000C);
socket.IOControl(SIO_UDP_CONNRESET, new byte[] { 0, 0, 0, 0 }, null);

// Bind to a local port (scan for available port)
int port = 4991;
while (true)
{
    try
    {
        socket.Bind(new IPEndPoint(IPAddress.Any, port));
        break;
    }
    catch (SocketException)
    {
        socket.Dispose();
        socket = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
  ↪ ProtocolType.Udp);
        // ... (reapply settings) ...
        if (++port > 6010)
            throw new InvalidOperationException("No available UDP port");
    }
}

// Radio's public endpoint for sending registration/ping
var radioEndpoint = new IPEndPoint(publicIp, publicUdpPort);
```

Do NOT call `socket.Connect()` — the socket must remain unconnected so `ReceiveFromAsync` accepts packets from any source. This is critical for WAN because NAT may remap the radio's source port on outbound VITA packets, and a connected socket would reject them.

1.5.2 UDP Registration Loop

Registration uses a three-phase loop:

1.5.2.1 Phase 0: Wait for Client Handle

The client handle is assigned asynchronously via the TCP H message. The UDP registration must not start until it arrives — `handle=0x0` is silently ignored by the radio.

```
// Wait up to 10 seconds for the client handle to be assigned
var waitStart = Environment.TickCount64;
```

```
while (client.ClientHandle == 0)
{
    if (Environment.TickCount64 - waitStart > 10_000)
    {
        // Abort — the radio never sent the 'H' message
        Log.Warning("Client handle still 0x0 after 10s");
        return;
    }
    await Task.Delay(100, cancellationToken);
}
uint clientHandle = client.ClientHandle;
```

1.5.2.2 Phase 1: Registration (send every 50ms until confirmed)

Send `client udp_register` over UDP (not TCP) to the radio's public endpoint. Repeat every 50ms until the radio responds with VITA-49 data:

```
byte[] registerCmd = Encoding.ASCII.GetBytes(
    $"client udp_register handle=0x{clientHandle:X}");

var registrationStart = Environment.TickCount64;
int sendCount = 0;

while (!udpRegistered && !cancellationToken.IsCancellationRequested)
{
    // Timeout after 30 seconds (FlexLib has no timeout, but this catches
    // misconfigured networks and prevents infinite retries)
    if (Environment.TickCount64 - registrationStart > 30_000)
    {
        Log.Warning($"Registration timed out — sent {sendCount} packets, no
 ↪ response");
        return;
    }

    socket.SendTo(registerCmd, radioEndpoint);
    sendCount++;
    await Task.Delay(50, cancellationToken);
}
```

Registration is confirmed when the first VITA-49 packet arrives with the Flex OUI (0x1C2D in the VITA class ID). Set a flag in the packet dispatch loop:

```
// In the UDP receive/dispatch loop:
var preamble = new VitaPacketPreamble(packetData);
if (preamble.class_id.OUI == 0x00001C2D)  // Flex OUI
{
    if (!udpRegistered)
        udpRegistered = true;  // Confirms the radio received our registration

    DispatchVitaPacket(preamble, packetData);
```

```
}
```

1.5.2.3 Phase 2: Keep-Alive Ping (every 5 seconds)

After registration is confirmed, switch to a 5-second keep-alive ping to maintain the NAT mapping:

```
byte[] pingCmd = Encoding.ASCII.GetBytes(
    $"client ping handle=0x{clientHandle:X}");

while (!cancellationToken.IsCancellationRequested)
{
    await Task.Delay(5000, cancellationToken);
    socket.SendTo(pingCmd, radioEndpoint);
}
```

If the keep-alive stops, the NAT mapping expires and meter data stops flowing.

1.5.3 LAN vs. WAN Comparison

Aspect	LAN	WAN (SmartLink)
Meter port command	`client udpport <port>` (TCP)	`client udp_register handle=0x<handle>` (UDP)
Target endpoint	Radio's LAN IP + port 4993	Radio's public IP + `public_udp_port`
Socket type	Same raw socket	Same raw socket
Keep-alive	Not needed (LAN doesn't NAT)	`client ping` every 5s over UDP
Registration confirmation	Immediate (TCP response)	Wait for first VITA packet with Flex OUI

LAN + SmartLink radios: When a radio is available on both the local network and SmartLink (discovery source contains "LAN"), always use the LAN meter path (`client udpport` over TCP). The `DiscoveryManager` overlays SmartLink data onto LAN-discovered radios, populating `WanUdpPort` even though the radio is locally accessible. The decision must check the discovery source, not the presence of `WanUdpPort`:

```
// Correct: check discovery source, not WAN port presence
bool useWanMeters = !discoverySource.Contains("LAN",
  ↪ StringComparison.OrdinalIgnoreCase)
                    && int.TryParse(wanUdpPort, out _);
```

1.6 Part 5: Complete Connection Sequence

Here is the full sequence from login to receiving meter data:

```
Step  Action                                      Transport    Direction
────  ──────────────────────────────────────────  ──────────  ─────────
 1    Build Auth0 URL, open in browser/WebView    HTTPS        → Auth0
```

```
 2    User authenticates                          HTTPS        ⇔ Auth0
 3    Extract refresh_token, id_token from URL    (local)      —
 4    POST /delegation to refresh JWT             HTTPS        → Auth0
 5    TLS connect to smartlink.flexradio.com:443  TLS          → Server
 6    application register name=... token=<jwt>   TLS          → Server
 7    radio list serial=... public_ip=...         TLS          ← Server
 8    application connect serial=<serial>         TLS          → Server
 9    radio connect_ready serial=... handle=...   TLS          ← Server
10    TLS connect to radio public_ip:tls_port     TLS          → Radio
11    wan validate handle=<handle>                TLS          → Radio
12    V3.6.8.40296                                TLS          ← Radio
13    H0000ABCD                                   TLS          ← Radio
14    C1|client program MyApp                     TLS          → Radio
15    C2|unsub radio all                          TLS          → Radio
16    C3|sub meter all                            TLS          → Radio
17    Bind UDP socket, start receive loop         UDP          (local)
18    client udp_register handle=0xABCD           UDP          → Radio
      (repeat every 50ms until VITA arrives)
19    <VITA-49 meter packet>                      UDP          ← Radio
      (registration confirmed)
20    client ping handle=0xABCD                   UDP          → Radio
      (repeat every 5s)
```

Steps 1-3 can be skipped on subsequent connections if the refresh token is stored — just refresh the JWT (step 4) and reconnect from step 5. Steps 7-9 happen through the persistent SmartLink server connection (maintained with `ping from client` every 10s).

1.7 Timeouts and Constants

Parameter	Value	Notes
Auth0 domain	`frtest.auth0.com`	
Auth0 client ID	`<client-id>` (tenant-specific)	
SmartLink server	`smartlink.flexradio.com:443`	TLS
JWT lifetime	60 seconds	Refresh before each server connect
Server keep-alive	10 seconds	`ping from client`
`connect_ready` timeout	15 seconds	Time to wait after `application connect`
TLS connect timeout	10 seconds	Radio direct connection
Client handle wait	10 seconds	Wait for H message before UDP register
UDP register interval	50 ms	Send rate during registration phase
UDP register timeout	30 seconds	Abort if no VITA response
UDP keep-alive ping	5 seconds	`client ping` to maintain NAT
UDP port scan range	4991-6010	Local bind port search
Flex VITA UDP port (LAN)	4993	Not used for WAN

1.8 Common Pitfalls

1. `wan validate` **not sent first** — The radio disconnects immediately if any other command precedes `wan validate` on the TLS connection.
2. **Counter-based V/H detection** — Sent commands (like `wan validate`) increment the message counter, shifting the radio's V and H messages to unexpected positions. Use content-based detection (`message[0] == 'V'` / `'H'`).
3. **Message parser not ready for TLS** — For TLS connections, the radio sends V and H immediately after the TLS handshake. If your TCP listener starts before the message parser is initialized, both messages are read and discarded.
4. `handle=0x0` **in UDP register** — The client handle arrives asynchronously via TCP. If you start the UDP registration loop before the handle is assigned, `client udp_register handle=0x0` is silently ignored by the radio and meter data never arrives.
5. `UdpClient` **instead of raw** `Socket` — `UdpClient` on Windows does not handle ICMP port-unreachable responses gracefully. The first `udp_register` (before NAT is established) triggers WSAECONNRESET on the next receive, killing the loop. Use raw `Socket` with `SIO_UDP_CONNRESET`.
6. **Connected UDP socket** — Calling `socket.Connect()` locks the socket to a specific remote endpoint. WAN NAT may remap the radio's source port, causing the connected socket to reject legitimate VITA packets.
7. **ICMP ping to WAN address** — Public IPs behind NAT are not ICMP-reachable. Skip reachability checks for WAN connections.
8. **Auth0 session cookies persist** — WebView2 caches Auth0 session cookies. After logout, clear all cookies or the next login auto-completes without prompting for credentials.
9. **Server** `registration_invalid` — The server sends this if the JWT has expired. Refresh the JWT and reconnect.
10. **No server keep-alive** — The SmartLink server drops connections that don't send periodic pings. Send `ping from client` every 10 seconds.