CAT4OM Radio Handbook Manual

Version: 2026-06-30

Target: CAT4OM Service (.NET 10)

Purpose: complete guide for creating, understanding, validating, and debugging XML radio handbooks for amateur radio CAT control.

CAT4OM handbooks are the single source of truth for radio-specific behavior. The service derives runtime command availability from the `<Commands>` section and publishes it to clients through `RadioState.AvailableCommands`. Do not add a `<Capabilities>` section: it is not part of the current handbook schema.

Table of Contents

1. Introduction and Core Concepts

1.1 What a Handbook Is

A **handbook** is an XML file that fully describes how CAT4OM communicates with a specific radio model. It includes:

- **Metadata**: radio identity and protocol-level rules.
- **Commands**: frames to send and response parsing rules.
- **Polling**: commands that keep the runtime state synchronized.
- **Mappings**: conversions between radio-specific values and CAT4OM standard values.

1.2 Fundamental Principle

> ***Everything that is radio-specific belongs in XML.***

CAT4OM does not contain model-specific radio code. Adding a radio means adding or updating an XML handbook, not changing service logic for that individual model.

1.3 Data Flow

```
UI/Client  <----  RadioHandler  <----  Radio
                      |
                      v
                Handbook XML
```

1. **RadioHandler** reads the handbook and discovers the available commands.
2. It builds request frames from the templates defined in XML.
3. It sends commands to the radio through the selected adapter (`Serial`, `TCI`, or `UDP`).
4. It receives responses and parses them with the XML response rules.
5. It updates the internal `RadioState` with the parsed values.
6. State changes are pushed to connected clients.

2. General XML File Structure

```
<?xml version="1.0" encoding="utf-8"?>
<RadioHandbook>
    <!-- Radio identity and protocol configuration -->
    <Metadata>...</Metadata>

    <!-- Default connection parameters -->
    <ConnectionDefaults>...</ConnectionDefaults>
```

```
    <!-- Global reusable mappings -->
    <ValueMaps>...</ValueMaps>

    <!-- Available VFOs and their radio-specific values -->
    <AvailableVfos>...</AvailableVfos>

    <!-- Supported operating modes, also used by the UI -->
    <SupportedModes>...</SupportedModes>

    <!-- Initialization commands -->
    <Initialization>...</Initialization>

    <!-- Connection verification command -->
    <ConnectionCheck>...</ConnectionCheck>

    <!-- Command definitions -->
    <Commands>...</Commands>

    <!-- Polling configuration -->
    <Polling>...</Polling>
</RadioHandbook>
```

3. Metadata Section

The `<Metadata>` section defines the radio identity and global protocol rules.

3.1 Complete Example

```
<Metadata>
    <Manufacturer>Yaesu</Manufacturer>
    <Model>FTDX101D</Model>
    <Version>2.0.0</Version>
    <InterfaceType>Serial</InterfaceType>
    <Protocol>CAT</Protocol>
    <Terminator>;</Terminator>
    <FrequencyResolution>1</FrequencyResolution>
    <PushBased>false</PushBased>
    <CommandEncoding>ASCII</CommandEncoding>
    <FrequencyGroups hf="true" vhf="false" uhf="false" shf="false"/>
</Metadata>
```

3.2 Available Fields

Field	Type	Default	Description
`Manufacturer`	string	*required*	Manufacturer name, for example `Yaesu`, `Icom`, or `Kenwood`.
`Model`	string	*required*	Radio model, for example `FTDX101D` or `IC-7300`.
`Version`	string	`1.0.0`	Handbook version, not the CAT4OM software version.
`InterfaceType`	string	*required*	Interface type: `Serial`,

			`TCI`, `UDP`, or `Flex`.
`Protocol`	string	*required*	Protocol name: `CAT`, `CI-V`, `TCI`, or `FLEX`.
`Terminator`	string	`;`	Message terminator. CI-V handbooks normally use `FD`.
`FrequencyResolution`	int	1	Frequency resolution in Hz: `1`, `10`, `100`, or `1000`.
`PushBased`	bool	false	When true, the radio sends spontaneous updates and does not need normal polling.
`SharedVfoMode`	bool	false	When true, the operating mode (and filter) is shared across all VFOs because the protocol exposes it per-receiver, not per-channel (e.g. TCI). A mode change then applies to every VFO. See §3.3.
`CommandEncoding`	string	`ASCII`	`ASCII` for text commands, `Hex` for binary frames.
`FrequencyGroups`	element	all true	Declares which frequency groups the radio hardware covers. When the element is absent all groups are enabled. See §3.4.

3.4 FrequencyGroups

`<FrequencyGroups>` declares which frequency groups the radio hardware physically covers. The UI uses these flags to show only the relevant Band Select buttons.

```
<FrequencyGroups hf="true" vhf="false" uhf="false" shf="false"/>
```

Attribute	Bands included	Example radios
`hf`	160m, 80m, 60m, 40m, 30m, 20m, 17m, 15m, 12m, 10m, **6m**	Most HF transceivers
`vhf`	2m	IC-705, IC-7100, FT-991, FT-817, TS-2000
`uhf`	70cm	IC-705, IC-7100, FT-991, FT-817, TS-2000
`shf`	23cm	IC-9700, TS-2000 (with optional 23cm module)

All attributes default to `true` when the `<FrequencyGroups>` element is absent, so existing handbooks without the element show all band groups. When the element is present, attributes that are not listed

also default to `true`, so you only need to explicitly set `false` for groups the radio does not cover. In practice it is clearer to list all four attributes explicitly.

Note: 6m is included in the HF group because most HF transceivers that support 6m use the same CAT connection and band plan as their HF coverage.

3.3 Field Notes

InterfaceType selects the adapter used for communication:

- `Serial`: physical or virtual serial port.
- `TCI`: WebSocket transport for the TCI protocol, used by Expert Electronics and compatible software.
- `UDP`: packet-based UDP communication.
- `Flex`: TCP connection to a FlexRadio SmartSDR API endpoint (default port 4992). Push-based; all inbound state is driven by S-line messages processed by `FlexStatusHandler`, not by the handbook's `<PushMessages>` section. The `<PushMessages>` element should be left empty with a comment. Set `<PushBased>true</PushBased>` and `<Protocol>FLEX</Protocol>` in Metadata.

CommandEncoding controls how the `<Frame>` strings in the Commands section are converted to bytes on the wire. It is **independent of `Protocol`**: two radios can share the same protocol dialect but differ in encoding, and a future binary protocol from a non-Icom manufacturer would need `Hex` even if it is not CI-V. Always declare this field explicitly.

- `ASCII`: frames are text strings sent as ASCII bytes — `FA;`, `MD0;`, etc. Used by CAT (Yaesu, Kenwood, Elecraft) and TCI.
- `Hex`: frames are hexadecimal byte sequences decoded to binary before transmission — `FE FE A4 E0 25 00 FD`. Currently used by Icom CI-V; may apply to other future binary protocols.

Terminator is used by the service to:

7. Detect when a response is complete.
8. Validate received responses.
9. Build matching patterns.

SharedVfoMode declares that the radio exposes a single operating mode (and filter) per receiver, shared by all of its VFOs, rather than an independent mode per VFO. TCI is the canonical case: the `MODULATION` and `RX_FILTER_BAND` messages carry only a receiver index, no channel — the radio shows one MODE indicator for both VFOs.

When `SharedVfoMode` is true:

- a `SetMode` command applies the mode (and any filter) to **every** VFO in `RadioState`, not just the active one, so the VFOs never diverge even if the radio does not echo a mode push back;
- inbound mode/filter updates from the radio are likewise applied to all VFOs (for TCI this is done by the protocol's push handler — so a TCI handbook must **not** declare a per-active-VFO `MODULATION` push message; the value is shared in code).

Leave it `false` (the default) for radios such as Yaesu/Icom/Kenwood, where each VFO can hold an independent mode.

4. ConnectionDefaults Section

This section defines default serial connection parameters for the radio. These values are presented to users and are used when the radio configuration does not override them.

4.1 Example

```
<ConnectionDefaults>
    <BaudRate>38400</BaudRate>
    <DataBits>8</DataBits>
    <StopBits>1</StopBits>
    <Parity>None</Parity>
    <RtsEnable>true</RtsEnable>
    <DtrEnable>true</DtrEnable>
    <CIVAddress>A4</CIVAddress>
</ConnectionDefaults>
```

4.2 Available Fields

Field	Type	Default	Description
`BaudRate`	int	9600	Serial speed in baud.
`DataBits`	int	8	Data bits, usually `7` or `8`.
`StopBits`	int	1	Stop bits, usually `1` or `2`.
`Parity`	string	`None`	`None`, `Even`, or `Odd`.
`RtsEnable`	bool	false	Enables the RTS signal.
`DtrEnable`	bool	false	Enables the DTR signal.
`CIVAddress`	hex	-	CI-V address for Icom radios. **Always in hexadecimal**, exactly as written in the Icom manual (e.g. `88`, `A4`, `94`).

4.3 Notes

- These are **defaults**. The user can override them in the radio configuration.
- `CIVAddress` is used only by CI-V handbooks. Its value replaces the `{CIV}` placeholder in frames.
- RTS/DTR may be needed by some CAT interfaces, USB adapters, opto-isolators, or PTT-related hardware.

4.4 CI-V Address Notation

Icom manuals always express CI-V addresses in **hexadecimal**, writing for example `88h` or `0x88`. In the handbook XML, write the value as a plain hex string without prefix or suffix:

```
<CIVAddress>88</CIVAddress>   <!-- IC-7100: hex 88 = decimal 136 -->
<CIVAddress>A4</CIVAddress>   <!-- IC-7300: hex A4 = decimal 164 -->
<CIVAddress>94</CIVAddress>   <!-- IC-705:  hex 94 = decimal 148 -->
```

Common mistake: writing `88` and intending decimal 88 (= `0x58`). The value `88` in the XML is always parsed as hex 0x88 = 136. If the CI-V frames sent by the service show the wrong address byte in the wire log, verify that the handbook `CIVAddress` matches the address reported in the Icom menu (set via `MENU > CI-V Address` on the radio).

5. Capabilities Section

The `<Capabilities>` section is not part of the current handbook schema.

Radio capabilities are derived directly from `<Commands>`, which is the single source of truth:

- The service automatically populates `AvailableCommands`, for example `["SetFrequencyA", "GetModeA", "VfoAtoB"]`, from the `<Command name="...">` entries defined in the handbook.
- The list is sent to clients through `RadioState.AvailableCommands` in `stateUpdate` messages.
- The UI enables or disables controls by checking whether the corresponding command exists. For example, the `A to B` control is enabled only when `VfoAtoB` is available.

Handbook authoring rule: do not write `<Capabilities>...</Capabilities>`. Define accurate commands and let CAT4OM derive runtime availability from them.

See **Section 9, Commands Overview** and **Section 15, Standard API Commands** for command names recognized by the system.

6. ValueMaps Section

ValueMaps are dictionaries that map radio-specific values to standard CAT4OM values. They are defined once at handbook level and can be reused by multiple response fields.

6.1 Structure

```
<ValueMaps>
    <ValueMap name="MapName">
        <Map key="radioValue" value="standardValue"/>
        <Map key="radioValue" value="standardValue"/>
        ...
    </ValueMap>
</ValueMaps>
```

6.2 Complete Example

```
<ValueMaps>
    <!-- Operating mode mapping -->
    <ValueMap name="ModeMap">
        <Map key="1" value="LSB"/>
        <Map key="2" value="USB"/>
        <Map key="3" value="CW"/>
        <Map key="4" value="FM"/>
        <Map key="5" value="AM"/>
        <Map key="6" value="RTTY-LSB"/>
        <Map key="7" value="CW-R"/>
        <Map key="8" value="DATA-LSB"/>
        <Map key="9" value="RTTY-USB"/>
        <Map key="A" value="DATA-FM"/>
        <Map key="B" value="FM-N"/>
        <Map key="C" value="DATA-USB"/>
    </ValueMap>

    <!-- VFO selection mapping -->
    <ValueMap name="VfoSelectMap">
        <Map key="0" value="A"/>
        <Map key="1" value="B"/>
    </ValueMap>

    <!-- TX VFO mapping, including split states -->
```

```
    <ValueMap name="TxVfoMap">
        <Map key="0" value="A"/>
        <Map key="1" value="B"/>
        <Map key="2" value="A"/>  <!-- Split mode -->
        <Map key="3" value="B"/>  <!-- Split mode -->
    </ValueMap>

    <!-- Boolean mapping for PTT, split, and similar flags -->
    <ValueMap name="PttMap">
        <Map key="0" value="false"/>
        <Map key="1" value="true"/>   <!-- CAT TX -->
        <Map key="2" value="true"/>   <!-- Mic TX -->
    </ValueMap>

    <!-- RX VFO mapping for Dual Watch -->
    <ValueMap name="RxVfosMap">
        <Map key="00" value="A,B"/>   <!-- Dual watch, both VFOs receiving -->
        <Map key="01" value="A"/>     <!-- VFO-A only -->
        <Map key="10" value="B"/>     <!-- VFO-B only -->
    </ValueMap>
</ValueMaps>
```

6.3 Using ValueMaps

Maps are referenced by response fields through the `map` attribute:

```
<Field name="mode.A" type="string" startIndex="3" length="1" format="MAP"
map="ModeMap"/>
```

6.4 Naming Conventions

Map Name	Typical Use
`ModeMap`	Converts radio mode codes to standard mode names.
`VfoSelectMap`	Converts the active VFO code to a canonical token such as `A` or `B`.
`TxVfoMap`	Converts the TX VFO code to a canonical token.
`RxVfosMap`	Converts RX state to the list of receiving VFOs.
`PttMap`	Converts PTT state to a boolean.
`SplitMap`	Converts split state to a boolean.

6.5 Special ModeMap Behavior

When a `ValueMap` named `ModeMap` exists, CAT4OM uses it automatically for:

10. **Parsing**: converting radio mode codes into standard mode names in responses.
11. **Encoding**: converting standard mode names back to radio codes for SET commands.

The conversion is **bidirectional**: the map is inverted during initialization for normal `SetMode` operations.

First-entry wins for SET commands. When two radio values map to the same standard name (e.g. `AM` and `SAM` both map to `AM`), the **first** entry declared in the XML is used for outgoing `SetMode` commands. Subsequent entries with the same standard name are still recognized for parsing (so incoming `SAM` is correctly mapped to `AM`) but are never used to encode a command. Declare the preferred wire value first.

6.6 ModeSetMap for SET Payloads That Differ from Parsing

Normally `ModeMap` is enough for both reading and setting the mode. If the radio replies with `MD2;` and `2` means `USB`, CAT4OM can invert the map and use `2` when the client requests `SetMode("USB")`.

Some radios do not use the same payload in both directions. A typical case is the Elecraft K4: state can report a mode through a simple code, while setting some digital modes requires a full sequence such as mode selection, data sub-mode selection, and application to the second target. Use the optional `ModeSetMap` section for these cases.

```
<ValueMaps>
    <ValueMap name="ModeMap">
        <Map key="6" value="DIG-U"/>
    </ValueMap>
</ValueMaps>

<ModeSetMap>
    <Mode name="DIG-U" value="MD6;DT0;MD$6;DT$0;"/>
</ModeSetMap>
```

Rules:

12. `ModeMap` remains the source for **parsing** radio responses.
13. `ModeSetMap`, when it contains the requested mode, takes precedence only for `SetMode`.
14. When a mode is not present in `ModeSetMap`, CAT4OM uses the normal inverted `ModeMap` behavior.
15. The `ModeSetMap` value can be a single token such as `2`, or a complete frame/sequence when `SetMode` is defined with a passthrough frame such as `<Frame>{MODE}</Frame>`.

Use `ModeSetMap` only when it is truly required. Its presence documents that the SET payload is not a simple inversion of the response mapping.

6.7 List Values: the Special `RxVfosMap` Case

`ValueMap` entries are structurally **string scalar to string scalar** mappings: both `key` and `value` are single strings. This covers almost every use case: mode codes, VFO selection, booleans, split flags, and similar values.

The exception is the `rxVfos` state field. In the runtime model (`CAT4OM.Contracts.Radio.RadioState.RxVfos`) it is typed as `List<string>` to support Dual Watch on dual-receiver radios, where both VFOs can receive at the same time. Because the XML schema does not define a dedicated list-mapping syntax, CAT4OM uses this convention:

- **Handbook XML side**: the mapped value is a comma-separated string of VFO tokens. For example, `value="A,B"` means both VFOs are receiving, while `value="A"` means only VFO-A is receiving.
- **Runtime side**: `RadioHandler.NormalizeRxVfos` recognizes the CSV string, splits it by comma, trims and normalizes each token, and produces the `List<string>` assigned to `RadioState.RxVfos`.

Complete example for an FTDX101D-style `GetFunctionRx` command that returns `FR<mainMute><subMute>;`, where `0` means unmuted and `1` means muted:

```
<ValueMaps>
    <ValueMap name="RxVfosMap">
        <Map key="00" value="A,B"/>   <!-- main=on, sub=on, dual watch -->
        <Map key="01" value="A"/>     <!-- main=on, sub=off, VFO-A only -->
        <Map key="10" value="B"/>     <!-- main=off, sub=on, VFO-B only -->
    </ValueMap>
</ValueMaps>
```

```
<Commands>
    <Command name="GetFunctionRx">
        <Request><Frame>FR;</Frame></Request>
        <Response>
            <Pattern>FR{P1}{P2};</Pattern>
            <Fields>
                <Field name="rxVfos" type="string" startIndex="2" length="2"
format="MAP" map="RxVfosMap"/>
            </Fields>
        </Response>
    </Command>
</Commands>
```

Practical rules for handbook authors:

16. **Use comma as the separator**, without spaces. `"A,B"` is preferred; `"A, B"` also works because the runtime trims each token. `"A;B"` and `"A|B"` do not work.
17. **Tokens must be canonical VFO names** present in `<AvailableVfos>`. Unknown values are ignored after normalization.
18. **An empty list** produced by `value=""` or by an unmatched key triggers the defensive `[ActiveVfo]` fallback, so the UI always has at least one RX badge to display.
19. **`RxVfosMap` cannot be inverted automatically** like `ModeMap`. There is no standard `set rxVfos` command, and even if a radio-specific one existed, inversion would be ambiguous because multiple keys can map to the same scalar value. Treat the map as **read-only response parsing**.

This pattern is the only current case where a `ValueMap` semantically produces a list. All other `RadioState` fields are scalar and the single `value="..."` syntax is sufficient.

7. AvailableVfos Section

This section defines the VFOs available on the radio and their radio-specific values. **This section is required**.

7.1 Structure

```
<AvailableVfos>
    <Vfo name="canonicalToken" value="radioValue"/>
    ...
</AvailableVfos>
```

7.2 Example

```
<AvailableVfos>
    <Vfo name="A" value="0"/>
    <Vfo name="B" value="1"/>
</AvailableVfos>
```

7.3 Attributes

Attribute	Description
`name`	Stable VFO name exposed on the wire, such as `A`, `B`, `C`, `MAIN`, `SUB`, or a protocol-specific name. Treat it as an opaque identifier and keep casing consistent across the handbook.
`value`	Radio-specific value used in CAT commands.

7.4 How It Is Used

When the API receives a request such as `SetFrequency(frequency, vfo="A")`:

20. The system resolves the requested VFO. When omitted, it uses `ActiveVfo`.
21. When the VFO exists in the VFO map, CAT4OM calculates:
    - `{VFO_NAME}` = VFO name, for example `"A"`.
    - `{VFO_VALUE}` = radio-specific mapped value, for example `"0"`. It can be identical to `{VFO_NAME}` for identity mappings.
22. It builds the frame by replacing the placeholders required by the command template.
23. Typical Yaesu FA/FB example: template `F{VFO_NAME}{FREQ};` sends `FA{FREQ};` for `vfo="A"`, for example `FA014025000;`.

7.5 Automatic Parameters

When a command is executed, CAT4OM automatically adds these parameters:

Parameter	Value	Description
`VFO`	`"A"`, `"B"`, ...	VFO name.
`VFO_VALUE`	`"0"`, `"1"`, ...	Radio-specific value from the VFO map.
`VFO_NAME`	`"A"`, `"B"`, ...	Alias of `VFO`.
`CHANNEL`	`"A"`, `"B"`, ...	Alias of `VFO`, mainly for TCI compatibility.
`TRX`	`0`, `1`, ...	Transceiver index for multi-TRX protocols.

8. SupportedModes Section

This section lists the operating modes supported by the radio. It is primarily used by the UI to populate selectors and validate user input.

> ***TCI radios**: the `<SupportedModes>` list is a **fallback only**. At connection time the TCI adapter receives a `MODULATIONS_LIST` message from the radio with the complete authoritative mode list. `RadioHandler` overrides both the handbook's `SupportedModes` and the live `RadioState.SupportedModes` with the radio-reported values. The XML list is only used before the first connection is established. Keep it approximately correct as a reasonable default, but it does not need to be exhaustively maintained.*

8.1 Compact Format

```
<SupportedModes>LSB,USB,CW,CW-R,FM,FM-N,AM,AM-N,RTTY-LSB,RTTY-USB,DATA-FM,DATA-USB,DATA-LSB,DATA-FM-N,PSK</SupportedModes>
```

8.2 Extended Format

```
<SupportedModes>
    <Mode>LSB</Mode>
    <Mode>USB</Mode>
    <Mode>CW</Mode>
</SupportedModes>
```

8.3 Standard Modes

Mode	Description
`LSB`	Lower Side Band
`USB`	Upper Side Band
`CW`	Continuous Wave
`CW-R`	CW Reverse
`AM`	Amplitude Modulation
`AM-N`	AM Narrow
`FM`	Frequency Modulation
`FM-N`	FM Narrow
`RTTY-LSB`	RTTY on LSB
`RTTY-USB`	RTTY on USB
`DATA-LSB`	Digital data on LSB
`DATA-USB`	Digital data on USB
`DATA-FM`	Digital data on FM
`PSK`	Phase Shift Keying

9. Commands Section - Overview

The `<Commands>` section contains all CAT command definitions supported by the radio.

9.1 Basic Command Structure

```
<Command name="CommandName">
    <Description>Command description</Description>
    <Request>
        <Frame>TEMPLATE_FRAME</Frame>
        <Parameters>
            <Param name="NAME" type="type" format="format"/>
        </Parameters>
    </Request>
    <Response>
        <Pattern>RESPONSE_PATTERN</Pattern>
        <ExpectEcho>false</ExpectEcho>
        <ExpectedLength>12</ExpectedLength>
        <Fields>
            <Field name="stateKey" type="type" startIndex="N" length="M"
format="format" map="MapName"/>
        </Fields>
    </Response>
</Command>
```

9.2 Command Types

Type	Naming Convention	Example	Description
GET	`Get*`	`GetFrequencyA`	Reads a value from the radio.
SET	`Set*`	`SetFrequency`	Sets a value on the radio.

ACTION	Descriptive name	`VfoSwap`, `VfoAtoB`	Executes an action.

9.3 Internal Commands and API Commands

Internal commands are used by polling and are not directly exposed as client actions.

- `GetFrequencyA`, `GetFrequencyB` read specific VFO frequencies.
- `GetModeA`, `GetModeB` read specific VFO modes.
- `GetStatus` reads general status such as RIT/XIT.
- `GetSMeter` reads the S-meter.

API commands are exposed through the external API and can be requested by clients.

- `SetFrequency` sets frequency.
- `SetMode` sets mode.
- `SetVfo` selects the active VFO.
- `SetPtt` controls PTT.
- `SetSplit` enables or disables split.
- `CwSend` sends a CW text string via the radio's built-in keyer.
- `CwStop` aborts the current CW transmission.
- `CwSpeed` sets the CW keyer speed in WPM.
- `CwKeyDown` keys or unkeys the transmitter manually (tune).

10. Commands: Read Commands (GET)

GET commands read values from the radio and update internal state.

10.1 Example: GetFrequencyA

```
<Command name="GetFrequencyA">
    <Description>Read VFO-A frequency</Description>
    <Request>
        <Frame>FA;</Frame>
    </Request>
    <Response>
        <Pattern>FA{FREQ};</Pattern>
        <Fields>
            <Field name="frequency.A" type="long" startIndex="2" length="9"
format="ASCII"/>
        </Fields>
    </Response>
</Command>
```

Flow:

24. The system sends `FA;` to the radio.
25. The radio replies, for example, with `FA014025000;`.
26. The system extracts 9 characters from position 2: `014025000`.
27. It converts the value to `long`: `14025000`.
28. It updates state: `frequency.A = 14025000`.

10.2 Example: GetModeA with ValueMap

```
<Command name="GetModeA">
    <Description>Get VFO-A operating mode</Description>
```

```
    <Request>
        <Frame>MD0;</Frame>
    </Request>
    <Response>
        <Pattern>MD0{MODE};</Pattern>
        <Fields>
            <Field name="mode.A" type="string" startIndex="3" length="1"
format="MAP" map="ModeMap"/>
        </Fields>
    </Response>
</Command>
```

Flow:

29. It sends `MD0;`.
30. The radio replies, for example, with `MD02;`.
31. It extracts the character at position 3: `2`.
32. It looks up `2` in `ModeMap` and obtains `USB`.
33. It updates state: `mode.A = USB`.

> ***Why `<Pattern>` matters — declare it for sibling commands.*** *Field extraction is purely positional, so a frame that arrives in place of the expected answer but happens to have the right layout would be parsed into the wrong field. `GetModeA` (`MD0{MODE};`) and `GetModeB` (`MD1{MODE};`) are exactly such siblings: the mode digit sits at the same offset, so a stray `MD05;` (MAIN/AM) read where a `MD1…;` (SUB) answer was expected would silently store `AM` as the SUB mode. When a command declares a `<Pattern>`, the runtime validates the frame against it before extracting any field; a non-matching frame is discarded and a `WARN ResponsePatternMismatch` is logged instead of producing a wrong value (see Cat4OM_Server.md §8.1).* ***Always declare a `<Pattern>` for any command whose frame shares its field layout with a sibling*** *(per-VFO reads `MD0`/`MD1`, `FA`/`FB`, metered reads like `SM0`, etc.). Commands with no `<Pattern>` — including all CI-V (hex) reads, which are validated by echo/`SuccessIndicator` instead — keep the legacy positional-only behaviour.*

10.3 Example: GetStatus with Multiple Fields

```
<Command name="GetStatus">
    <Description>Read RIT/XIT status via IF command</Description>
    <Request>
        <Frame>IF;</Frame>
    </Request>
    <Response>
        <Pattern>IF{DATA};</Pattern>
        <Fields>
            <Field name="ritOffset" type="int" startIndex="14" length="5"
format="ASCII_SIGNED"/>
            <Field name="ritEnabled" type="bool" startIndex="19" length="1"
format="BOOL"/>
            <Field name="xitEnabled" type="bool" startIndex="20" length="1"
format="BOOL"/>
        </Fields>
    </Response>
</Command>
```

A single command can extract multiple fields from the same response.

11. Commands: Write Commands (SET)

SET commands send values to the radio. Parameters are substituted into the frame template.

11.1 Example: SetFrequency

```
<Command name="SetFrequency">
    <Description>Set frequency on target VFO</Description>
    <Request>
        <Frame>F{VFO_NAME}{FREQ};</Frame>
        <Parameters>
            <Param name="VFO_NAME" type="string" format=""/>
            <Param name="FREQ" type="frequency" format="ASCII9"/>
        </Parameters>
    </Request>
    <Response>
        <ExpectEcho>false</ExpectEcho>
    </Response>
</Command>
```

Flow for `SetFrequency(14025000, vfo="A")`:

34. The system resolves VFO `A` to `VFO_NAME = "A"` and `VFO_VALUE = "0"`.
35. It substitutes the frame placeholders:
    - `{VFO_NAME}` becomes `A`.
    - `{FREQ}` with `ASCII9` format becomes `014025000`.
36. The resulting frame is `FA014025000;`.
37. The frame is sent to the radio.

11.1.1 SetFrequency on the Active VFO Only

Some radios do not have a command that directly sets the frequency of a non-active VFO. For example, several CI-V radios expose command `05 {FREQ}`, which always modifies only the VFO currently selected on the physical radio.

This detail matters because the CAT4OM API is more expressive than the radio command. A client can request `SetFrequency(freq, vfo="B")`, but an active-only radio frame contains no byte that identifies VFO B. If the service sent the frame anyway, the physical radio would change the currently active VFO while software state could attribute the frequency to the client-requested VFO. The tag prevents this ambiguity by declaring that the command is valid only when the requested VFO matches `RadioState.ActiveVfo`.

In these cases the command must declare the optional `vfoScope="activeOnly"` attribute:

```
<Command name="SetFrequency" vfoScope="activeOnly">
    <Description>Set operating frequency on the selected VFO only</Description>
    <Request>
        <Frame>FE FE {CIV} E0 05 {FREQ} FD</Frame>
        <Parameters>
            <Param name="FREQ" type="frequency" format="BCD5LE"/>
        </Parameters>
    </Request>
    <Response>
        <ExpectEcho>true</ExpectEcho>
        <SuccessIndicator>FB</SuccessIndicator>
    </Response>
```

```
</Command>
```

Runtime behavior:

- If the API requests `SetFrequency(freq, vfo=null)`, or the requested VFO matches `ActiveVfo`, the command is sent normally.
- If the API requests a VFO different from `ActiveVfo`, the service does not send any frame to the radio, writes a warning to the log, and returns the non-destructive error `INVALID_TARGET_VFO`.
- If the attribute is not present, CAT4OM uses the standard behavior: the command is considered able to apply the change to the requested VFO.

The service also publishes this information to clients through radio state:

```
"commandVfoScopes": {
  "SetFrequency": "activeOnly"
}
```

The map is intentionally sparse: it contains only commands with non-default behavior. A command absent from the map is interpreted as normally addressable to the requested VFO. The UI uses this information to disable the command and show a message before sending it; the service remains authoritative for external clients or clients that ignore the tag.

11.2 Example: SetMode

```
<Command name="SetMode">
    <Description>Set operating mode</Description>
    <Request>
        <Frame>MD{VFO_VALUE}{MODE};</Frame>
        <Parameters>
            <Param name="VFO_VALUE" type="string" format=""/>
            <Param name="MODE" type="mode" format="CHAR1"/>
        </Parameters>
    </Request>
    <Response>
        <ExpectEcho>false</ExpectEcho>
    </Response>
</Command>
```

Note: CAT4OM automatically converts the standard mode name, for example `USB`, to the radio code, for example `2`, by using the inverted `ModeMap`.

`SetMode` accepts the same optional `vfoScope="activeOnly"` attribute as `SetFrequency` (§11.1.1), with identical runtime behavior: an explicit `vfo` request that does not match `ActiveVfo` is rejected with `INVALID_TARGET_VFO` instead of being sent to the radio. Absent the attribute, `SetModeAsync` can target any VFO in `AvailableVfos`, which is required on radios with no `SetVfo` command (§15.4).

11.3 Example: SetSplit with Boolean Encoding

```
<Command name="SetSplit">
    <Description>Enable/disable SPLIT</Description>
    <Request>
        <Frame>ST{ENABLED};</Frame>
        <Parameters>
            <Param name="ENABLED" type="boolean" format="BOOL"/>
        </Parameters>
    </Request>
    <Response>
        <ExpectEcho>false</ExpectEcho>
```

```
        </Response>
    </Command>
```

With `ENABLED=true` and `BOOL` format, the resulting frame is `ST1;`.

11.4 Parameter Attributes

```
    <Param name="NAME" type="type" format="format" divisor="N" offset="M"/>
```

Attribute	Description
`name`	Parameter name. It must match the `{NAME}` placeholder in the frame.
`type`	Semantic type: `string`, `frequency`, `mode`, `bool`, or `int`.
`format`	Encoding format for the value. See section 12.
`divisor`	Divisor applied before encoding, for example Hz to 10 Hz with `divisor=10`.
`offset`	Offset added after division.

Conversion formula: `encoded_value = (original_value / divisor) + offset`

11.5 Placeholder runtime speciali (runtime-injected)

Alcuni placeholder nei frame XML non corrispondono a un `<Param>` dichiarato nel comando — vengono iniettati direttamente dal runtime prima della costruzione del frame.

Placeholder	Sorgente	Handbook che lo usano
`{CIV}`	`ConnectionDefaults.CivAddress` (letto dalla configurazione radio al momento della connessione)	Tutti gli handbook Icom CI-V e Xiegu G90

Regola per gli autori handbook: `{CIV}` **non va dichiarato come `<Param>`** nel comando. Il validatore `HandbookValidator.CheckFrameParams` ha una whitelist dei placeholder runtime e ignora `{CIV}` durante la verifica della corrispondenza `{placeholder} ↔ <Param>`. Se lo si dichiara come `<Param>` si ottiene un WARNING `UNUSED_PARAM`.

Attualmente `{CIV}` è l'unico placeholder runtime speciale. I placeholder automatici del sistema (`{VFO}`, `{VFO_VALUE}`, `{VFO_NAME}`, `{CHANNEL}`, `{TRX}`) sono documentati in §7.5 e §15.2 e seguono meccanismi separati.

12. Encoding Formats (Request)

These formats define how a value is converted into the request payload for SET commands.

12.1 ASCII Numeric Formats

Format	Description	Example Input	Output
`ASCII1`	1 digit, zero-padded when applicable.	`7`	`"7"`
`ASCII2`	2 digits, zero-padded.	`7`	`"07"`
`ASCII3`	3 digits, zero-padded.	`7`	`"007"`
`ASCII4`	4 digits, zero-padded.	`42`	`"0042"`
`ASCII5`	5 digits, zero-padded.	`100`	`"00100"`

`ASCII8`	8 digits, zero-padded.	`14025000`	`"14025000"`
`ASCII9`	9 digits, zero-padded.	`14025000`	`"014025000"`
`ASCII11`	11 digits, zero-padded.	`14025000`	`"00014025000"`
`ASCII_SIGNED`	5 digits with sign.	`50`	`"+00050"`
`ASCII_SIGNED`	5 digits with sign.	`-25`	`"-00025"`
`MHZ`	Frequency in MHz, 6 decimal places.	`14074000`	`"14.074000"`

> ***`MHZ` format*** *converts an internal Hz Long value to a MHz string with exactly 6 decimal places (e.g. 14074000 Hz → "14.074000"). Used by FlexRadio SmartSDR commands such as slice tune which require frequency in MHz. Only valid for <Param type="frequency">.*

12.2 Boolean Formats

Format	true	false	Notes
`BOOL`	`"1"`	`"0"`	Single character.
`BOOL01`	`"01"`	`"00"`	Two characters.

12.3 String Formats

Format	Description	Example
`STRING`	Passthrough without modification.	`"ABC"` to `"ABC"`
`CHAR1`	Single character, truncating or padding as needed.	`"AB"` to `"A"`
`CHAR2`	Two characters, truncating or padding as needed.	`"A"` to `"A "`
empty string	Passthrough.	Same as `STRING`.

12.4 Binary Formats (Icom CI-V)

Format	Description	Byte Count	Endianness
`BCD4LE`	4-byte BCD frequency.	4	Little Endian
`BCD4BE`	4-byte BCD frequency.	4	Big Endian
`BCD5LE`	5-byte BCD frequency.	5	Little Endian
`BCD5BE`	5-byte BCD frequency.	5	Big Endian
`BYTE`	Single hexadecimal byte.	1	-
`HEX`	Alias of `BYTE`.	1	-
`HEX2`	Byte encoded as two hexadecimal characters.	1	-

13. Parsing Formats (Response)

These formats define how CAT4OM interprets values in radio responses.

13.1 Field Structure

```
<Field name="stateKey" type="type" startIndex="N" length="M" format="format"
map="MapName" multiplier="X" offset="Y"/>
```

Attribute	Description
`name`	State key. See section 14.
`type`	Value type: `long`, `int`, `string`, or `bool`.
`startIndex`	Initial zero-based position in the response buffer.
`length`	Number of bytes or characters to read.
`format`	How bytes are interpreted. See below.
`map`	ValueMap name used for conversion.
`multiplier`	Post-parsing multiplier.
`offset`	Post-parsing offset.
`mask`	Bit mask for `BITMASK` format (hex, e.g. `"20"` for bit 5).
`invert`	When `true`, inverts the `BITMASK` result.

Post-parsing formula: `final_value = (parsed_value * multiplier) + offset`

startIndex for CI-V commands

`startIndex` is always **response-relative** (zero-based from the first byte of the response frame, i.e. from the leading `FE`). The echo is stripped by the adapter before field parsing; handbook authors must NOT add the echo length to `startIndex`.

```
startIndex = response_header_length
           = 5  (simple commands:      FE FE E0 {CIV} cmd → data at index 5)
           = 6  (commands with subcmd: FE FE E0 {CIV} cmd subcmd → data at index
6)
```

See §23 for the complete CI-V echo-handling architecture.

13.2 Available Parsing Formats

Format	Use	Description
`ASCII`	Text numbers.	Parses the string as `long`.
`ASCII_SIGNED`	Signed numbers.	Parses values such as `+0050` or `-0025`.
`MAP`	Mapped values.	Uses a `ValueMap` for conversion.
`BOOL`	Booleans.	`1` or `ON` maps to true.
`BCD4LE/BE`	Icom frequencies.	Decodes BCD.
`BCD5LE/BE`	Icom frequencies.	Decodes BCD.
`BYTE`	Single byte.	Returns a numeric value.
`HEX2`	Byte as hex.	Returns a string such as `A4`.
`BYTE_MAP`	Byte plus map.	Converts the byte through a `ValueMap`.
`BITMASK`	Bit flag.	Tests a single bit via `mask`; optionally `invert`s the result.

13.3 MAP vs BYTE_MAP

MAP is used by ASCII protocols. It extracts a string and looks it up in the map.

```
<Field name="mode.A" format="MAP" map="ModeMap"/>
<!-- Response: "MD02;" -> extracts "2" -> looks up ModeMap -> "USB" -->
```

BYTE_MAP is used by binary protocols. It reads a byte, converts it to a hexadecimal string, then looks it up in the map.

```
<Field name="mode" format="BYTE_MAP" map="ModeMap"/>
<!-- Binary response: byte 0x02 -> "02" -> looks up ModeMap -> "USB" -->
```

13.4 BITMASK

`BITMASK` extracts a single boolean flag from a byte using a bit mask. It is useful when multiple states are encoded as bit flags in the same byte (legacy Yaesu binary protocols, etc.).

Logic: `result = (byte & mask) != 0`. With `invert="true"`: `result = (byte & mask) == 0`.

Required attributes:

Attribute	Description
`mask`	Hexadecimal mask of the bit to test (for example `"20"` = bit 5, `"80"` = bit 7)
`invert`	`true` when the flag uses inverted logic (bit=0 means active)

Example: Yaesu FT-817 opcode F7

The radio returns 1 byte where bit 5 (0x20) indicates split and bit 7 (0x80) indicates PTT, both using inverted logic (0=active, 1=inactive):

```
<Command name="GetTxStatus">
    <Request><Frame>00 00 00 00 F7</Frame></Request>
    <Response>
        <ExpectedLength>1</ExpectedLength>
        <Fields>
            <Field name="split" type="bool" startIndex="0" length="1"
                   format="BITMASK" mask="20" invert="true"/>
            <Field name="ptt" type="bool" startIndex="0" length="1"
                   format="BITMASK" mask="80" invert="true"/>
        </Fields>
    </Response>
</Command>
```

With response byte `0x00`: split=true (bit 5=0, inverted), ptt=true (bit 7=0, inverted).

With response byte `0xA0`: split=false (bit 5=1, inverted), ptt=false (bit 7=1, inverted).

Example without inversion (direct logic: bit=1 means active):

```
<Field name="agcOn" type="bool" startIndex="0" length="1"
       format="BITMASK" mask="04"/>
```

14. State Keys Recognized by the System

When a polling command returns values, those values are applied to radio state. The system recognizes these field names.

14.1 VFO-Qualified Keys with `.A`, `.B`, and Similar Suffixes

Key	Type	Description
`frequency.A`	long	VFO-A frequency in Hz.
`frequency.B`	long	VFO-B frequency in Hz.
`frequency.unselected`	long	Frequency of the VFO that is **not** `ActiveVfo`. Reserved engine suffix — see §25.7 for usage.
`mode.A`	string	VFO-A operating mode.
`mode.B`	string	VFO-B operating mode.
`filterwidth.A`	int	VFO-A filter width in Hz.
`filterwidth.B`	int	VFO-B filter width in Hz.

`frequency.unselected` — reserved suffix: The engine resolves this suffix to the single VFO in `AvailableVfos` that is not `ActiveVfo`. Valid only for radios with exactly two VFOs. If zero or two or more candidates exist (single-VFO radio, ambiguous state), the update is skipped and a WARNING is logged — no silent fallback is applied. This suffix is **read-only**: it must never appear in a write (`SetFrequency`) command. See §25.7 for the complete usage model and affected radio list.

14.2 Global Keys

Key	Type	Description
`activeVfo`	string	Currently selected VFO, for example `A` or `B`.
`txVfo`	string	VFO used for TX.
`rxVfos`	string (CSV)	Receiving VFOs as a comma-separated list, for example `A,B`. Runtime type: `List<string>`.
`split`	bool	Active split state.
`ptt`	bool	Active PTT state.
`ritEnabled`	bool	RIT enabled state.
`ritOffset`	int	RIT offset in Hz.
`xitEnabled`	bool	XIT enabled state.
`xitOffset`	int	XIT offset in Hz.
`smeter`	int	S-meter level, normally 0-255.
`swr`	float	TX SWR.
`power`	int	TX power.
`alc`	int	ALC level.

Note about the `rxVfos` type:

The `Type` column describes the **mapped XML-side value type**, meaning what the handbook author writes in `<ValueMap>` or `<Field>`, not always the final runtime state type. All fields in the table have XML type equal to runtime type except `rxVfos`:

- **XML type**: `string` in CSV format, for example `"A,B"` or `"A"`.

- **Runtime type** in `CAT4OM.Contracts.Radio.RadioState.RxVfos`: `List<string>`, for example `["A", "B"]`.

The CSV to `List<string>` conversion is performed automatically by `RadioHandler.NormalizeRxVfos` before state is updated. The handbook author only needs to provide the CSV value. For accepted syntax, VFO token normalization, and the defensive fallback used when the map returns an empty list, see [Section 6.7, List Values: the Special `RxVfosMap` Case](#67-list-values-the-special-rxvfosmap-case).

14.3 Unsupported Legacy Keys

The following keys are not supported by the current schema and produce warnings:

- `frequency0`, `frequency1`, `frequency2`
- `mode0`, `mode1`

Always use VFO-qualified keys such as `frequency.A` and `mode.B`.

14.4 Unqualified Keys

Keys without a VFO suffix, such as `frequency`, `mode`, and `filterwidth`, are applied to the current active VFO.

15. Standard API Commands

These are the commands exposed by the `RadioHandler` API. Each operation looks for the corresponding command definition in the handbook.

15.1 Command Table

API Method	Handbook Command	Parameters	Description
`SetFrequencyAsync()`	`SetFrequency`	FREQ, VFO, VFO_VALUE, VFO_NAME	Sets frequency.
`SetModeAsync()`	`SetMode`	MODE, FILTER, VFO, VFO_VALUE	Sets mode.
`SetVfoAsync()`	`SetVfo`	VFO, VFO_VALUE	Selects the active VFO.
`SetPttAsync()`	`SetPtt`	ENABLED, VFO, VFO_VALUE	Controls PTT.
`SetSplitAsync()`	`SetSplit`	ENABLED, TXVFO, TXVFO_VALUE	Enables split.
`SetRitAsync()`	`SetRit`	ENABLED, OFFSET	Controls RIT.
`SetXitAsync()`	`SetXit`	ENABLED, OFFSET	Controls XIT.

15.2 Automatic Parameters

CAT4OM automatically adds these parameters to every command:

Parameter	Description
`VFO`	Canonical target VFO token.
`VFO_VALUE`	Radio-specific VFO value.
`VFO_NAME`	Alias of `VFO`.
`CHANNEL`	Alias of `VFO`, mainly for TCI compatibility.
`TRX`	Transceiver index, default `0`.

15.3 Automatic Mode Conversion

When `SetModeAsync("USB")` is called:

38. The system looks up `USB` in the inverted `ModeMap`.
39. It finds the radio code, for example `2`.
40. It passes `MODE = "2"` to the command.

15.4 Radios Without `SetVfo`: Independently Addressable VFOs

Not every radio has a CAT concept of "the active VFO" that a client can switch. Some protocols address each VFO directly instead — for example FlexRadio SmartSDR, where VFO A and B are separate slices (`slice tune {idx} {freq}`, `slice set {idx} mode=...`), or TCI, which is push-based with no VFO-select command at all.

Convention: omit `SetVfo` from `Commands` entirely for these radios. Do not declare a fake `SetVfo` that does nothing, and do not add a separate handbook flag to signal this — `AvailableCommands` already communicates command existence to every layer (server validation, UI button gating, external client capability checks), and adding a second signal for the same fact would only risk drifting out of sync with it. `RadioHandler.SetVfoAsync` fails with `NOT_SUPPORTED` if a client calls `setVfo` on a radio whose handbook has no `SetVfo` command; clients are expected to check `AvailableCommands` instead of relying on that failure.

Consequence for `SetFrequency` and `SetMode`: both commands accept an explicit `vfo` parameter/token independent of whatever VFO the radio last reported as active (`RadioHandler.SetFrequencyAsync` / `SetModeAsync`, both resolve their target VFO the same way — see §15.1). For a radio with no `SetVfo`, this `vfo` parameter is the *only* way a client can address VFO B: there is no "switch to VFO B, then set frequency/mode" two-step flow, because there is nothing to switch. Do **not** mark `SetFrequency`/`SetMode` as `vfoScope="activeOnly"` on a handbook with no `SetVfo` — that combination would make VFO B permanently unreachable (see §11.1.1 for `vfoScope`). `activeOnly` is only for radios where the protocol *does* have an active-VFO switch, but the frequency/mode command specifically cannot target the inactive one.

The reference UI (`Cat4OM.UI`) reflects this: for a radio with no `SetVfo`, its VFO A/B selector never sends a wire command — it only changes which VFO the next `setFrequency`/`setMode` call from the UI targets. See `Cat4OM_UI.md` §9.1.

16. Custom Handbook Commands

In addition to standard commands mapped to specific API methods, CAT4OM supports **custom commands** defined in the handbook. This allows radio-specific or protocol-specific features without changing the engine code.

16.1 Standard Commands vs Custom Commands

Standard commands have a dedicated `RadioHandler` API method with a typed signature and specific validation logic.

Category	Characteristics
****Standard****	Dedicated API method, for example `SetFrequencyAsync`.
	Built-in parameter validation.
	Automatic state update.
	Supported by all radios that expose the corresponding command.
****Custom****	Executed through `ExecuteCustomCommandAsync`.
	Defined only in the XML handbook.
	No automatic state update.
	Radio-specific or protocol-specific.

Standard command list:

GET Command	SET Command	Description
`GetFrequency`, `GetFrequencyA`, `GetFrequencyB`	`SetFrequency`	VFO frequency.
`GetMode`, `GetModeA`, `GetModeB`	`SetMode`	Operating mode.
`GetActiveVfo`	`SetVfo`	VFO selection.
`GetPtt`	`SetPtt`	Push-to-Talk
`GetSplit`	`SetSplit`	Split function.
`GetStatus`	`SetRit`, `SetXit`	RIT/XIT
`GetSMeter`, `GetSWR`, `GetPower`	-	Metering
-	`VfoAtoB`, `VfoBtoA`, `VfoSwap`	VFO operations.
-	`PowerOn`	Powers on the radio. Used by one-shot Management Port adapter; never invoked through a live `RadioHandler`.
-	`PowerOff`	Powers off the radio. Invoked through `RadioHandler.PowerOffAsync`. After a successful command the server disconnects the adapter immediately.

All other commands defined in the handbook are considered **custom**.

16.2 Executing Custom Commands

The `ExecuteCustomCommandAsync` method executes any command defined in the handbook:

```
// Method signature
```

```
public async Task<CommandResult> ExecuteCustomCommandAsync(
    string commandName,
    CancellationToken ct,
    params (string Name, object? Value)[] parameters)
```

Automatic parameters: CAT4OM adds these parameters automatically when they are not provided explicitly:

- `VFO` - canonical active VFO token.
- `VFO_VALUE` - radio-specific VFO value.
- `VFO_NAME` - alias of `VFO`.
- `CHANNEL` - alias for TCI compatibility.
- `TRX` - transceiver index, default `0`.

16.3 Usage Examples

Example 1: TCI - Set Panorama Center Frequency (DDS)

Handbook definition (`sunsdr-tci.xml`):

```
<Command name="SetDds">
    <Description>Set panorama center frequency</Description>
    <Request>
        <Frame>dds:{TRX},{FREQ};</Frame>
        <Parameters>
            <Param name="TRX" type="int" format="ASCII1"/>
            <Param name="FREQ" type="frequency" format="ASCII"/>
        </Parameters>
    </Request>
</Command>
```

Client call:

```
// Set DDS to 14.100 MHz on TRX 0.
await handler.ExecuteCustomCommandAsync("SetDds", ct,
    ("FREQ", 14100000));

// With explicit TRX.
await handler.ExecuteCustomCommandAsync("SetDds", ct,
    ("FREQ", 7050000),
    ("TRX", 1));
```

Example 2: TCI - Audio Volume Control

Handbook definition:

```
<Command name="SetVolume">
    <Description>Set audio volume (0-100)</Description>
    <Request>
        <Frame>volume:{LEVEL};</Frame>
        <Parameters>
            <Param name="LEVEL" type="int" format="ASCII"/>
        </Parameters>
    </Request>
</Command>
```

Client call:

```
await handler.ExecuteCustomCommandAsync("SetVolume", ct, ("LEVEL", 75));
```

Example 3: TCI - TX Drive

Handbook definition:

```
<Command name="SetDrive">
    <Description>Set TX drive level (0-100)</Description>
    <Request>
        <Frame>drive:{TRX},{LEVEL};</Frame>
        <Parameters>
            <Param name="TRX" type="int" format="ASCII1"/>
            <Param name="LEVEL" type="int" format="ASCII"/>
        </Parameters>
    </Request>
</Command>
```

Client call:

```
await handler.ExecuteCustomCommandAsync("SetDrive", ct, ("LEVEL", 50));
```

Example 4: Icom CI-V - Specific Command

Handbook definition (`icom-ic705.xml`):

```
<Command name="SetNoiseBlanker">
    <Description>Enable/disable noise blanker</Description>
    <Request>
        <Frame>FE FE {CIV} E0 16 22 {ENABLED} FD</Frame>
        <Parameters>
            <Param name="ENABLED" type="bool" format="BYTE"/>
        </Parameters>
    </Request>
</Command>
```

Client call:

```
await handler.ExecuteCustomCommandAsync("SetNoiseBlanker", ct, ("ENABLED", true));
```

16.4 Discovering Available Custom Commands

`GetAvailableCustomCommands` returns the list of custom commands defined by the current handbook:

```
// Method signature
public IReadOnlyList<(string Name, string? Description)>
GetAvailableCustomCommands()
```

Usage example:

```
var customCommands = handler.GetAvailableCustomCommands();

foreach (var (name, description) in customCommands)
{
    Console.WriteLine($"  {name}: {description ?? "(no description)"}");
}

// Example output for SunSDR TCI:
//   SetDds: Set panorama center frequency
//   SetVolume: Set audio volume (0-100)
//   SetMute: Mute/unmute audio
//   SetDrive: Set TX drive level (0-100)
//   SetAgcMode: Set AGC mode
//   SetLock: Lock/unlock frequency
```

This method is useful for:

41. **Dynamic UIs**: automatically generating controls for radio-specific features.
42. **Documentation**: listing the features available for a radio.
43. **Validation**: checking whether a command exists before executing it.

16.5 State Management with Custom Commands

Custom commands **do not automatically update radio state** because CAT4OM does not know which properties they modify. To update state, use the overload with a state updater:

```
// Overload with state updater.
public async Task<CommandResult> ExecuteCustomCommandAsync(
    string commandName,
    Action<RadioState> stateUpdater,
    CancellationToken ct,
    params (string Name, object? Value)[] parameters)
```

Example:

```
// Set volume and update state, assuming RadioState exposes a Volume-like
property.
await handler.ExecuteCustomCommandAsync(
    "SetVolume",
    s => s.ExtendedProperties["Volume"] = 75,
    ct,
    ("LEVEL", 75));
```

16.6 Guidelines for Defining Custom Commands

When adding custom commands to a handbook:

44. **Naming**: use descriptive CamelCase names, for example `SetNoiseBlanker` or `GetBandScope`.
45. **Description**: always include a clear description:

```
<Command name="SetAgcMode">
    <Description>Set AGC mode: 0=OFF, 1=SLOW, 2=MID, 3=FAST</Description>
    ...
</Command>
```

46. **Parameters**: define all required parameters with clear names:

```
<Parameters>
    <Param name="MODE" type="int" format="ASCII1"/>  <!-- 0-3 -->
</Parameters>
```

47. **Default values**: rely on automatic parameters such as `VFO` and `TRX` where appropriate.
48. **Documentation**: for complex commands, add XML comments:

```
<!-- SetBandScope: configures the band scope.
     WIDTH: span width in Hz, for example 48000, 96000, 192000.
     CENTER: when true, centers on the VFO frequency. -->
<Command name="SetBandScope">
    ...
</Command>
```

16.7 Complete Example: Handbook with Custom Commands

```
<Commands>
    <!-- Standard commands (required) -->
    <Command name="SetFrequency">...</Command>
    <Command name="SetMode">...</Command>
    <Command name="SetVfo">...</Command>
```

```
<Command name="SetPtt">...</Command>
<Command name="GetFrequencyA">...</Command>
<Command name="GetModeA">...</Command>

<!-- Custom commands (optional, radio-specific) -->

<!-- Audio -->
<Command name="SetVolume">
    <Description>Set audio volume (0-100)</Description>
    <Request>
        <Frame>volume:{LEVEL};</Frame>
        <Parameters>
            <Param name="LEVEL" type="int" format="ASCII"/>
        </Parameters>
    </Request>
</Command>

<Command name="SetMute">
    <Description>Mute/unmute audio output</Description>
    <Request>
        <Frame>mute:{ENABLED};</Frame>
        <Parameters>
            <Param name="ENABLED" type="bool" format="BOOL_TCI"/>
        </Parameters>
    </Request>
</Command>

<!-- TX Control -->
<Command name="SetDrive">
    <Description>Set TX drive level (0-100)</Description>
    <Request>
        <Frame>drive:{TRX},{LEVEL};</Frame>
        <Parameters>
            <Param name="TRX" type="int" format="ASCII1"/>
            <Param name="LEVEL" type="int" format="ASCII"/>
        </Parameters>
    </Request>
</Command>

<!-- SDR Specific -->
<Command name="SetDds">
    <Description>Set panorama center frequency</Description>
    <Request>
        <Frame>dds:{TRX},{FREQ};</Frame>
        <Parameters>
            <Param name="TRX" type="int" format="ASCII1"/>
            <Param name="FREQ" type="frequency" format="ASCII"/>
        </Parameters>
    </Request>
</Command>

<!-- DSP -->
```

```
        <Command name="SetAgcMode">
            <Description>Set AGC mode: OFF, SLOW, NORMAL, FAST</Description>
            <Request>
                <Frame>rx_agc_mode:{TRX},{CHANNEL},{MODE};</Frame>
                <Parameters>
                    <Param name="TRX" type="int" format="ASCII1"/>
                    <Param name="CHANNEL" type="string" format=""/>
                    <Param name="MODE" type="string" format=""/>
                </Parameters>
            </Request>
        </Command>
    </Commands>
```

17. Polling Section

This section defines which commands CAT4OM executes periodically to keep radio state current.

17.1 Structure

```
    <Polling>
        <PollGroup name="Essential" interval="100">
            <Poll command="GetFrequencyA" updates="frequency.A"/>
            <Poll command="GetFrequencyB" updates="frequency.B"/>
        </PollGroup>
        <PollGroup name="Extended" interval="500">
            <Poll command="GetModeA" updates="mode.A"/>
            <Poll command="GetActiveVfo" updates="activeVfo"/>
            <Poll command="GetPtt" updates="ptt"/>
        </PollGroup>
        <PollGroup name="Metering" interval="200">
            <Poll command="GetSMeter" updates="smeter" condition="PTT=OFF"/>
            <Poll command="GetSWR" updates="swr" condition="PTT=ON"/>
        </PollGroup>
    </Polling>
```

17.2 PollGroup

Attribute	Description
`name`	Group name: `Essential`, `Extended`, or `Metering`.
`interval`	Polling interval in milliseconds.

Recognized groups:

- `Essential`: critical polling such as frequency, normally with a short interval.
- `Extended`: secondary polling such as mode and split, normally with a medium interval.
- `Metering`: meter readings such as S-meter and SWR, often with conditions.

17.3 Poll

Attribute	Description
`command`	Command name to execute.
`updates`	State keys updated by the command. This is documentary.

`condition`	Optional condition required to execute the poll.

17.4 Conditions

```
<Poll command="GetSWR" condition="PTT=ON"/>
<Poll command="GetSMeter" condition="PTT=OFF"/>
```

Supported conditions:

- `PTT=ON` / `PTT=OFF`
- `PTT` (equivale a PTT=ON)
- `SPLIT=ON` / `SPLIT=OFF`

17.5 `updates` Attribute

The `updates` attribute is **documentary**: it indicates which state keys are updated by the command. The runtime update is determined by the command `Field` definitions, not by `updates`. The attribute is useful for:

49. Documentation.
50. Future optimization.
51. Debugging.

18. Initialization Section

Commands sent when the radio connection is initialized.

18.1 Structure

```
<Initialization>
    <Command name="DisableAutoInfo">
        <Description>Disable automatic information mode</Description>
        <Frame>AI0;</Frame>
        <ExpectResponse>false</ExpectResponse>
        <RequiresTelemetry>false</RequiresTelemetry>
        <DelayAfterMs>50</DelayAfterMs>
    </Command>
</Initialization>
```

18.2 Attributes

Tag	Description
`name`	Command identifier.
`Description`	Human-readable description.
`Frame`	Command frame to send.
`ExpectResponse`	Whether CAT4OM waits for a response.
`RequiresTelemetry`	Optional. When `true`, the command is sent only if the radio connection has `enableTelemetry: true`. Default `false`.
`DelayAfterMs`	Pause after sending, in milliseconds.

18.3 Common Use

```
<Initialization>
```

```
        <!-- Disable auto-info for controlled polling -->
        <Command name="DisableAutoInfo">
            <Frame>AI0;</Frame>
            <ExpectResponse>false</ExpectResponse>
        </Command>

        <!-- Force split off -->
        <Command name="SplitOff">
            <Frame>ST0;</Frame>
            <ExpectResponse>false</ExpectResponse>
        </Command>
    </Initialization>
```

18b. PushMessages Section

Defines how CAT4OM processes spontaneous messages sent by push-based protocols such as TCI. Each `<Message>` maps an incoming push command to one or more state fields via `<Field>` elements. This section is only relevant for handbooks with `<PushBased>true</PushBased>`.

18b.1 Structure

```
    <PushMessages>
        <!-- VFO frequency: TCI format VFO:trx,channel,freq; -->
        <Message command="VFO" trxArg="0" vfoArg="1">
            <Field stateKey="frequency" argIndex="2" format="ASCII" type="long"/>
        </Message>
        <!-- PTT state: TCI format TRX:trx,enabled; -->
        <Message command="TRX" trxArg="0">
            <Field stateKey="ptt" argIndex="1" format="BOOL_TCI" type="bool"/>
        </Message>
        <!-- Split: TCI format SPLIT_ENABLE:trx,enabled; -->
        <Message command="SPLIT_ENABLE" trxArg="0">
            <Field stateKey="split" argIndex="1" format="BOOL_TCI" type="bool"/>
        </Message>
        <!-- RIT enable/offset: TCI format RIT_ENABLE:trx,enabled; and
    RIT_OFFSET:trx,offset; -->
        <Message command="RIT_ENABLE" trxArg="0">
            <Field stateKey="ritEnabled" argIndex="1" format="BOOL_TCI" type="bool"/>
        </Message>
        <Message command="RIT_OFFSET" trxArg="0">
            <Field stateKey="ritOffset" argIndex="1" format="ASCII" type="int"/>
        </Message>
    </PushMessages>
```

18b.2 `<Message>` Attributes

Attribute	Description
`command`	Incoming TCI command name (matched case-insensitively, stored uppercase).
`trxArg`	Index of the argument that carries the TRX index. When present, messages are TRX-filtered: only those whose `trxArg` argument matches the configured TRX index are processed. Absent means no TRX filter.
`vfoArg`	Index of the argument that carries the VFO token (e.g.

	`"0"`, `"1"`). When present, fields with a key such as `frequency` are applied to that specific VFO (e.g. `frequency.A`). Absent means the field is applied as a global (active-VFO) key.

18b.3 `<Field>` Attributes

Attribute	Description
`stateKey`	Target state key. Same keys as `ApplyParsedValuesToState` (see §20.4). When `vfoArg` is set, `frequency` becomes `frequency.A` / `frequency.B`, etc.
`argIndex`	Zero-based index of the message argument to extract.
`format`	Parsing format: `ASCII`, `MAP`, `BOOL`, `BOOL_TCI`, etc. (same as §13.2).
`type`	Target type: `long`, `int`, `string`, `bool`.
`map`	Reference to a global `<ValueMap>` for MAP-format conversion.
`multiplier`	Multiplier applied after parsing (default `1`).
`offset`	Offset applied after parsing × multiplier (default `0`).

18b.4 TCI Implementation Notes

Declarative vs. computed messages. The `<PushMessages>` mechanism covers direct cases: argument-to-state mapping with format conversion and value maps. Some TCI messages require computation beyond declarative mapping and are handled in code by `TciPushHandler`, not by handbook XML:

TCI message	Why not declarative
`MODULATION`	Mode is shared across all VFOs (`SharedVfoMode`); must be applied to every VFO of the TRX.
`RX_FILTER_BAND`	Filter width = |high − low| (two arguments), applied to all VFOs.
`RX_CHANNEL_SENSORS`	S-meter from dBm via formula (dBm → 0–255); populates both `SMeter` and `SignalDbm`.
`RX_SENSORS` / `RX_SMETER`	Periodic message enabled by `RX_SENSORS_ENABLE`; same treatment as `RX_CHANNEL_SENSORS`.
`TX_SENSORS`	Power and SWR from a multi-value message (fixed argument positions).
`TX_SWR` / `TX_FORWARD_POWER`	Alternative telemetry messages sent by some TCI servers.
`RX_CHANNEL_ENABLE`	Manages the `RxVfos` list (dual watch) with add/remove logic.

These messages **must not be declared** in the `<PushMessages>` section of a TCI handbook — they are fully handled by the code.

> ***`signalDbm` is not a handbook state key.*** *The dBm → `RadioState.Metering.SignalDbm` conversion is performed by `TciPushHandler` in code. There is no `signalDbm` key in*

ApplyParsedValuesToState; do not use it in <Field stateKey="...">. For normalized S-meter (0–255) via polling on serial radios, use the smeter key instead.

18b.5 Sensor relay for multi-TRX endpoints (ExpertSDR3)

ExpertSDR3 sends enabled sensor messages (RX_SENSORS, TX_SENSORS, etc.) to only **one** of the WebSocket connections open to the same endpoint. When more than one radio/TRX is configured against the same TCI host/port (e.g. TRX 0 and TRX 1), the adapter that receives a sensor message automatically re-publishes it to all sibling adapters on the same endpoint. Each sibling applies its own TRX filter, so only the correct RadioHandler processes the message. The relay mechanism is adapter-level and transparent to the handbook.

19. ConnectionCheck Section

This section defines how CAT4OM verifies that the radio responds correctly.

19.1 Structure

```
<ConnectionCheck>
    <Command>FA;</Command>
    <ExpectedPattern>FA*;</ExpectedPattern>
    <Timeout>1000</Timeout>
    <Retries>3</Retries>
    <ExpectEcho>false</ExpectEcho>
</ConnectionCheck>
```

19.2 Fields

Field	Description
`Command`	Command sent for the test. Use a **read** command that makes the radio send a data response (for CI-V, `03` read-frequency or `25 00` read-selected-VFO). Do not use a write-only command: it produces no answer to prove the radio is alive.
`ExpectedPattern`	Documentation/informational pattern for the expected response. `*` is a wildcard.
`Timeout`	Timeout in milliseconds.
`Retries`	Number of attempts.
`ExpectEcho`	Whether the transport echoes the transmitted frame, matching the per-command `Response/ExpectEcho`. When `true`, the check is run through the echo-stripping read so the radio's own echo is never mistaken for a real response. **Optional — defaults to `true` for CI-V handbooks** (`Protocol` = `CI-V` or `Terminator` = `FD`) and `false` otherwise. Set it explicitly only to override that default.

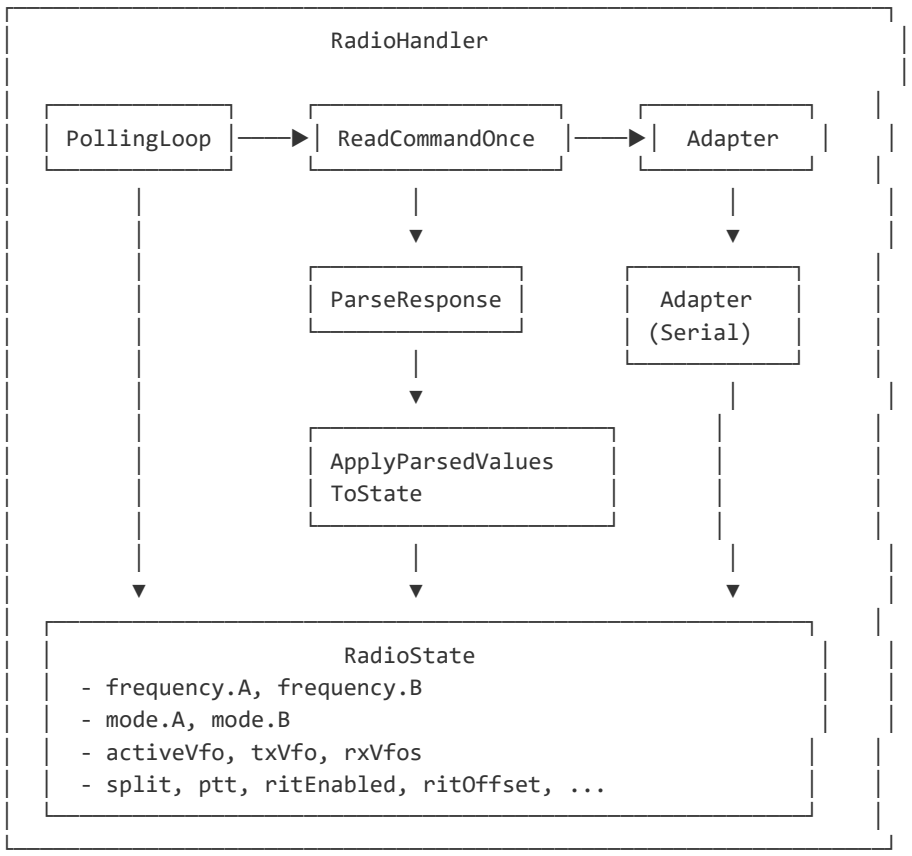
Why `ExpectEcho` matters for CI-V. *A CI-V bus echoes every frame you transmit before the radio answers. If the check accepted that echo as the response, a radio that is powered but not actually answering (or a serial port that is only half-owned by another program) would be reported as "connected" and then fail every poll. With echo stripping the check requires a genuine FE FE E0 xx … answer from the radio, so it fails cleanly instead of producing a false positive. This is why every Icom/Xiegu/Ten-Tec CI-V handbook uses a read command here.*

19.3 Behavior

52. On connection, the system sends the command. For CI-V (`ExpectEcho = true`) it uses the two-phase echo-stripping read.
53. It waits for a genuine response within the timeout (the radio's own echo does not count).
54. Success requires a response frame; the check does not enforce `ExpectedPattern` byte-for-byte (that field is informational).
55. If the check fails, it retries up to `Retries` times.
56. If all attempts fail, the connection check is considered failed and the radio is reported as not responding.

20. Internal Architecture: How the System Processes Commands

20.1 Polling Flow



20.2 Detail: ExecutePollItemAsync

```
// File: RadioHandler.cs, ExecutePollItemAsync method.

private async Task<PollOutcome> ExecutePollItemAsync(PollItem pollItem,
CancellationToken ct)
{
    // 1. Check conditions.
    if (!string.IsNullOrEmpty(pollItem.Condition) &&
!EvaluateCondition(pollItem.Condition))
        return new PollOutcome(PollOutcomeKind.Skipped);
```

```
    // 2. Find the command in the handbook.
    if (!_handbook.Commands.TryGetValue(pollItem.Command, out var commandDef))
        return new PollOutcome(PollOutcomeKind.MissingCommand);

    // 3. Build command bytes.
    var commandBytes = commandDef.BuildRequest(_config.Connection.CivAddress,
_handbook.IsHexEncoding);

    // 4. Send and receive the response.
    var tx = await _adapter.TransactAsync(commandBytes, timeout, expectedLen, ct);

    // 5. Parse the response.
    var parsedValues = _handbook.ParseResponse(pollItem.Command, tx.Response);

    // 6. Apply values to state.
    ApplyParsedValuesToState(parsedValues, pollItem.Command);

    return new PollOutcome(PollOutcomeKind.Success, ...);
}
```

20.3 Detail: ApplyParsedValuesToState

```
// File: RadioHandler.cs, ApplyParsedValuesToState method.

private void ApplyParsedValuesToState(Dictionary<string, object?> values, string
source)
{
    UpdateState(s =>
    {
        foreach (var (key, value) in values)
        {
            var normalizedKey = key.ToLowerInvariant();

            // VFO-qualified keys: frequency.A, mode.B.
            if (TryGetQualifiedToken(normalizedKey, "frequency.", out var
freqVfo))
            {
                if (s.Vfos.TryGetValue(freqVfo, out var vfoState))
                    vfoState.Frequency = Convert.ToInt64(value);
                continue;
            }

            if (TryGetQualifiedToken(normalizedKey, "mode.", out var modeVfo))
            {
                if (s.Vfos.TryGetValue(modeVfo, out var vfoState))
                    vfoState.Mode = value.ToString();
                continue;
            }

            // Global keys.
            switch (normalizedKey)
            {
```

```
                    case "activevfo":
                        s.ActiveVfo = NormalizeVfoToken(value, s);
                        break;
                    case "txvfo":
                        s.TxVfo = NormalizeVfoToken(value, s);
                        break;
                    case "split":
                        s.Split = Convert.ToBoolean(value);
                        break;
                    case "ptt":
                        s.Ptt = Convert.ToBoolean(value);
                        break;
                    case "ritenabled":
                        s.Rit.Enabled = Convert.ToBoolean(value);
                        break;
                    case "ritoffset":
                        s.Rit.Offset = Convert.ToInt32(value);
                        break;
                    // ... other cases.
                }
            }
        });
    }
```

20.4 Keys Recognized by ApplyParsedValuesToState

This is the **exact** list of keys recognized by the system:

First pass (VFO selection):

- `activevfo` -> `RadioState.ActiveVfo`
- `txvfo` -> `RadioState.TxVfo`

Second pass (VFO-qualified):

- `frequency.{VFO}` -> `RadioState.Vfos[VFO].Frequency`
- `mode.{VFO}` -> `RadioState.Vfos[VFO].Mode`
- `filterwidth.{VFO}` -> `RadioState.Vfos[VFO].FilterWidth`

Second pass (global):

- `frequency` -> frequency of the active VFO.
- `mode` -> mode of the active VFO.
- `filterwidth` -> filter width of the active VFO.
- `rxvfos` -> `RadioState.RxVfos` list.
- `smeter` -> `RadioState.Metering.SMeter` (normalized value 0–255).
- `swr` -> `RadioState.Metering.Swr`.
- `power` -> `RadioState.Metering.Power`.
- `alc` -> `RadioState.Metering.Alc`.

> ***`signalDbm` is not a handbook key*** *and cannot be set via `<Field stateKey="...">` in polling or PushMessages. `RadioState.Metering.SignalDbm` is populated exclusively by `TciPushHandler` from TCI messages RX_CHANNEL_SENSORS, RX_SENSORS, RX_SMETER (formula: `(dbm + 127) × 2`, clamped 0–255 for `SMeter`; raw float stored in `SignalDbm`). For serial radios that expose a normalized S-meter level, use the `smeter` key.*

- `ptt` -> `RadioState.Ptt`.
- `split` -> `RadioState.Split`.
- `ritenabled` -> `RadioState.Rit.Enabled`.
- `ritoffset` -> `RadioState.Rit.Offset`.
- `xitenabled` -> `RadioState.Xit.Enabled`.
- `xitoffset` -> `RadioState.Xit.Offset`.

Frequency offset / transverter handling is intentionally **not** a handbook state key. Handbooks, parsers, polling, diagnostics, and `RadioHandler` all work in real physical radio frequencies. Optional `VfoState.FrequencyOffset` metadata is attached later by the server boundary when it projects state for clients, using the exact VFO names from `<AvailableVfos>` as dictionary keys. Do not add XML fields such as `frequencyOffset`, `publishedFrequency`, or `transverterFrequency` to a handbook response definition; the correct handbook output remains `frequency` or `frequency.{VFO}` with the value reported by the radio.

21. Troubleshooting and Debugging

21.1 Wire Logs

Wire logs are **always active** — no configuration required. For every radio that connects, the service creates a dedicated log file in the `/log/RADIO/` folder under the data root:

```
<DataRoot>/log/RADIO/<RadioId>_yyyyMMdd.log
```

Files are cleaned up automatically at startup, keeping only the current day's log (same policy as the main service log).

21.2 Log Level Controls Verbosity

The amount of detail captured depends on the server log level configured in `cat4om-server.json` under `hub.logLevel`:

Level	What is logged
`warning` (default)	Connection events, protocol errors (malformed responses), application exceptions
`information`	Everything above + connect/disconnect lifecycle + **UI command execution** (`CMD>HL` / `CMD<HL`: command name, parameters, result)
`debug`	Everything above + TX frames for all operations (poll and command) + poll command names (`POLL>HL`) + CI-V echo/response phases
`trace`	Everything above + RX frames during poll cycles (very verbose)

To diagnose a radio problem, set `hub.logLevel` to `"debug"` or `"trace"`, reproduce the issue, then revert:

```
{
  "hub": {
    "logLevel": "debug"
  }
}
```

The change takes effect immediately without restarting the service (applied via the UI or by editing the file directly).

21.3 Wire Log Format

```
2026-05-30 14:23:01.123 [ftdx101d]                  CONN  Connected to COM3 @
38400 baud
2026-05-30 14:23:01.200 [ftdx101d] op=000042 CMD    CMD>HL Execute GetFrequencyA
Params:
2026-05-30 14:23:01.200 [ftdx101d] op=000042 CMD    TX[a1b2c3d4] [3 bytes] HEX: 46
41 3B | ASCII: "FA;"
2026-05-30 14:23:01.212 [ftdx101d] op=000042 CMD    RX[a1b2c3d4] (complete in
12.4ms) [12 bytes] HEX: 46 41 30 31 34 30 32 35 30 30 30 3B | ASCII:
"FA014025000;"
2026-05-30 14:23:01.212 [ftdx101d] op=000042 CMD    CMD<HL Ok GetFrequencyA
ParsedValues=1: frequency.a=14025000
2026-05-30 14:23:05.000 [ftdx101d] op=000099 POLL   TX[e5f6a7b8] [3 bytes] HEX: 46
41 3B | ASCII: "FA;"
2026-05-30 14:23:05.015 [ftdx101d] op=000099 POLL   RX[e5f6a7b8] (complete in
15.2ms) [12 bytes] HEX: ...
2026-05-30 14:23:10.001 [ftdx101d]                  WARN  ParseFieldFailed
GetModeA.mode.A: startIndex out of range
2026-05-30 14:23:15.003 [ftdx101d]                  ERR   Exception GetFrequencyA:
TimeoutException: read timeout after 2000ms
```

Legend:

- `[ftdx101d]` — radio ID.
- `op=000042` — operation sequence number, correlating TX and RX lines for the same command.
- `CMD` / `POLL` — operation type (command issued by a client vs automatic polling).
- `CMD>HL` / `CMD<HL` — high-level entry/exit of UI command execution (Information level).
- `POLL>HL` / `POLL<HL` — high-level entry/exit of poll command execution (Debug level).
- `TX[...]` / `RX[...]` — raw wire direction with correlation ID.
- `CONN` — connection lifecycle event (always logged).
- `WARN` — protocol warning, always logged regardless of level.
- `ERR` — application exception, always logged regardless of level.

21.4 Common Problems

The command is not sent

Symptom: the log shows `CMD>HL Execute...` but no `TX`.

Cause: `BuildRequest` fails because a placeholder is not replaced or a format is not recognized.

Solution: verify that:

57. Every `{PARAM}` in the frame has a corresponding `<Param>`.
58. The format is supported. See section 12.
59. The value type is compatible with the format.

Parsing returns no values

Symptom: `Parsed ... Values=0`.

Cause: incorrect `startIndex`/`length`, or wrong format.

Solution: verify against the real response:

60. Count characters using zero-based positions.
61. Check that the format matches the data type.

The ValueMap does not work

Symptom: the returned value is the radio code, not the mapped value.

Cause: missing map reference or incorrect map name.

Solution:

62. Verify that the map exists in `<ValueMaps>`.
63. Verify that the `map="..."` attribute is correct.
64. Verify that the format is `MAP` or `BYTE_MAP`.

Mode is not set

Symptom: `SetMode` sends nothing.

Cause: mode conversion failed.

Solution:

65. Verify that the requested mode is present in `<SupportedModes>`.
66. Verify that `ModeMap` contains the inverse mapping.
67. Verify the format of the `MODE` parameter.

22. Complete Examples

22.1 Radio ASCII (Yaesu/Kenwood Style)

```
<?xml version="1.0" encoding="utf-8"?>
<RadioHandbook>
    <Metadata>
        <Manufacturer>Yaesu</Manufacturer>
        <Model>FTDX101D</Model>
        <Version>2.0.0</Version>
        <InterfaceType>Serial</InterfaceType>
        <Protocol>CAT</Protocol>
        <Terminator>;</Terminator>
        <CommandEncoding>ASCII</CommandEncoding>
        <FrequencyResolution>1</FrequencyResolution>
    </Metadata>

    <ConnectionDefaults>
        <BaudRate>38400</BaudRate>
        <DataBits>8</DataBits>
        <StopBits>1</StopBits>
        <Parity>None</Parity>
        <RtsEnable>true</RtsEnable>
        <DtrEnable>true</DtrEnable>
    </ConnectionDefaults>

    <ValueMaps>
        <ValueMap name="ModeMap">
            <Map key="1" value="LSB"/>
            <Map key="2" value="USB"/>
            <Map key="3" value="CW"/>
            <Map key="4" value="FM"/>
            <Map key="5" value="AM"/>
```

```
            <Map key="7" value="CW-R"/>
            <Map key="8" value="DATA-LSB"/>
            <Map key="C" value="DATA-USB"/>
        </ValueMap>
        <ValueMap name="VfoSelectMap">
            <Map key="0" value="A"/>
            <Map key="1" value="B"/>
        </ValueMap>
        <ValueMap name="BoolMap">
            <Map key="0" value="false"/>
            <Map key="1" value="true"/>
            <Map key="2" value="true"/>
        </ValueMap>
    </ValueMaps>

    <AvailableVfos>
        <Vfo name="A" value="0"/>
        <Vfo name="B" value="1"/>
    </AvailableVfos>

    <SupportedModes>LSB,USB,CW,CW-R,FM,AM,DATA-LSB,DATA-USB</SupportedModes>

    <Initialization>
        <Command name="DisableAutoInfo">
            <Frame>AI0;</Frame>
            <ExpectResponse>false</ExpectResponse>
        </Command>
    </Initialization>

    <ConnectionCheck>
        <Command>FA;</Command>
        <ExpectedPattern>FA*;</ExpectedPattern>
        <Timeout>1000</Timeout>
        <Retries>3</Retries>
    </ConnectionCheck>

    <Commands>
        <!-- GET Commands -->
        <Command name="GetFrequencyA">
            <Request><Frame>FA;</Frame></Request>
            <Response>
                <Fields>
                    <Field name="frequency.A" type="long" startIndex="2"
length="9" format="ASCII"/>
                </Fields>
            </Response>
        </Command>

        <Command name="GetFrequencyB">
            <Request><Frame>FB;</Frame></Request>
            <Response>
                <Fields>
```

```
                    <Field name="frequency.B" type="long" startIndex="2"
length="9" format="ASCII"/>
                </Fields>
            </Response>
        </Command>

        <Command name="GetModeA">
            <Request><Frame>MD0;</Frame></Request>
            <Response>
                <Fields>
                    <Field name="mode.A" type="string" startIndex="3" length="1"
format="MAP" map="ModeMap"/>
                </Fields>
            </Response>
        </Command>

        <Command name="GetActiveVfo">
            <Request><Frame>VS;</Frame></Request>
            <Response>
                <Fields>
                    <Field name="activeVfo" type="string" startIndex="2"
length="1" format="MAP" map="VfoSelectMap"/>
                </Fields>
            </Response>
        </Command>

        <Command name="GetPtt">
            <Request><Frame>TX;</Frame></Request>
            <Response>
                <Fields>
                    <Field name="ptt" type="bool" startIndex="2" length="1"
format="MAP" map="BoolMap"/>
                </Fields>
            </Response>
        </Command>

        <!-- SET Commands -->
        <Command name="SetFrequency">
            <Request>
                <Frame>F{VFO_NAME}{FREQ};</Frame>
                <Parameters>
                    <Param name="VFO_NAME" type="string" format=""/>
                    <Param name="FREQ" type="frequency" format="ASCII9"/>
                </Parameters>
            </Request>
            <Response><ExpectEcho>false</ExpectEcho></Response>
        </Command>

        <Command name="SetMode">
            <Request>
                <Frame>MD{VFO_VALUE}{MODE};</Frame>
                <Parameters>
```

```
                    <Param name="VFO_VALUE" type="string" format=""/>
                    <Param name="MODE" type="mode" format="CHAR1"/>
                </Parameters>
            </Request>
            <Response><ExpectEcho>false</ExpectEcho></Response>
        </Command>

        <Command name="SetVfo">
            <Request>
                <Frame>VS{VFO_VALUE};</Frame>
                <Parameters>
                    <Param name="VFO_VALUE" type="string" format=""/>
                </Parameters>
            </Request>
            <Response><ExpectEcho>false</ExpectEcho></Response>
        </Command>

        <Command name="SetPtt">
            <Request>
                <Frame>TX{ENABLED};</Frame>
                <Parameters>
                    <Param name="ENABLED" type="bool" format="BOOL"/>
                </Parameters>
            </Request>
            <Response><ExpectEcho>false</ExpectEcho></Response>
        </Command>
    </Commands>

    <Polling>
        <PollGroup name="Essential" interval="100">
            <Poll command="GetFrequencyA" updates="frequency.A"/>
            <Poll command="GetFrequencyB" updates="frequency.B"/>
        </PollGroup>
        <PollGroup name="Extended" interval="500">
            <Poll command="GetModeA" updates="mode.A"/>
            <Poll command="GetActiveVfo" updates="activeVfo"/>
            <Poll command="GetPtt" updates="ptt"/>
        </PollGroup>
    </Polling>
</RadioHandbook>
```

22.2 Radio CI-V (Icom)

```
<?xml version="1.0" encoding="utf-8"?>
<RadioHandbook>
    <Metadata>
        <Manufacturer>Icom</Manufacturer>
        <Model>IC-7100</Model>
        <Version>1.0.0</Version>
        <InterfaceType>Serial</InterfaceType>
        <Protocol>CI-V</Protocol>
        <Terminator>FD</Terminator>
        <CommandEncoding>Hex</CommandEncoding>
        <FrequencyResolution>1</FrequencyResolution>
```

```
</Metadata>

<ConnectionDefaults>
    <BaudRate>19200</BaudRate>
    <DataBits>8</DataBits>
    <StopBits>1</StopBits>
    <Parity>None</Parity>
    <CIVAddress>88</CIVAddress>
</ConnectionDefaults>

<ValueMaps>
    <ValueMap name="ModeMap">
        <Map key="00" value="LSB"/>
        <Map key="01" value="USB"/>
        <Map key="02" value="AM"/>
        <Map key="03" value="CW"/>
        <Map key="04" value="RTTY"/>
        <Map key="05" value="FM"/>
        <Map key="07" value="CW-R"/>
        <Map key="08" value="RTTY-R"/>
    </ValueMap>
    <ValueMap name="BoolMap">
        <Map key="00" value="false"/>
        <Map key="01" value="true"/>
    </ValueMap>
</ValueMaps>

<AvailableVfos>
    <Vfo name="A" value="00"/>
    <Vfo name="B" value="01"/>
</AvailableVfos>

<SupportedModes>LSB,USB,CW,CW-R,AM,FM,RTTY,RTTY-R</SupportedModes>

<Initialization>
    <Command name="DisableCivTransceive">
        <Description>Disable CI-V transceive mode</Description>
        <Frame>FE FE {CIV} E0 1A 05 00 95 00 FD</Frame>
        <ExpectResponse>true</ExpectResponse>
        <DelayAfterMs>50</DelayAfterMs>
    </Command>
</Initialization>

<ConnectionCheck>
    <Command>FE FE 88 E0 03 FD</Command>
    <ExpectedPattern>FE FE E0 88</ExpectedPattern>
    <Timeout>1000</Timeout>
    <Retries>3</Retries>
</ConnectionCheck>

<Commands>
    <!-- GET: Response FE FE E0 88 03 {5 BCD freq} FD (11 bytes)
```

```
             startIndex = header(5) = 5  (response-relative, echo stripped by
adapter) -->
        <Command name="GetFrequency">
            <Request>
                <Frame>FE FE {CIV} E0 03 FD</Frame>
            </Request>
            <Response>
                <ExpectedLength>11</ExpectedLength>
                <ExpectEcho>true</ExpectEcho>
                <Fields>
                    <Field name="frequency" type="long" startIndex="5" length="5"
format="BCD5LE"/>
                </Fields>
            </Response>
        </Command>

        <!-- SET: Response FB/FA is always 6 bytes: FE FE E0 88 FB FD -->
        <Command name="SetFrequency" vfoScope="activeOnly">
            <Request>
                <Frame>FE FE {CIV} E0 05 {FREQ} FD</Frame>
                <Parameters>
                    <Param name="FREQ" type="frequency" format="BCD5LE"/>
                </Parameters>
            </Request>
            <Response>
                <ExpectedLength>6</ExpectedLength>
                <ExpectEcho>true</ExpectEcho>
                <SuccessIndicator>FB</SuccessIndicator>
            </Response>
        </Command>
    </Commands>

    <Polling>
        <PollGroup name="Essential" interval="100">
            <Poll command="GetFrequency" updates="frequency"/>
        </PollGroup>
    </Polling>
</RadioHandbook>
```

23. CI-V Protocol: Echo Handling

23.0 CI-V Address

Every CI-V command frame contains the radio address as the third byte (`FE FE {CIV} E0 ...`). If this byte is wrong, the radio echoes the command (it sees the bus traffic) but never responds — because the destination address does not match its own.

The address must match exactly what the radio reports in its own menu (typically `MENU > CI-V > CI-V Address`). Default values per model are documented in section 4.4.

Diagnosing an address mismatch: in the wire log at Warning level, a timeout whose TX and RX bytes are identical confirms the radio is only echoing and not responding:

```
WARN[a1b2] RX TIMEOUT 2000ms partial [6 bytes] TX: FE FE 58 E0 03 FD | RX: FE FE
58 E0 03 FD
```

TX == RX → only echo, no response → wrong CI-V address.

23.1 How CI-V Echo Works

The Icom CI-V protocol operates on a half-duplex bus. When a command is sent via USB or CI-V jack, the radio **always returns the command echo** before the actual response.

Example for `GetFrequency` (command 03) on IC-7100 (CIV address `88`):

```
Sent:        FE FE 88 E0 03 FD                              (6 bytes)
                                                             ↓
On the bus:  FE FE 88 E0 03 FD  FE FE E0 88 03 {freq} FD
             ├── echo (6 bytes) ─┤├── response (11 bytes) ──┤
```

The adapter (`SerialAdapter.TransactCivAsync`) performs a **two-phase read**:

68. **Phase 1 — echo**: reads until the first `0xFD` and discards those bytes.
69. **Phase 2 — response**: scans for the next `FE FE` preamble, then reads until the second `0xFD`.

The method returns **only the response frame**. The echo is never passed to the handbook parser.

This design is robust against:

- Bus noise and reserved bytes (e.g. `0xFC`) between echo and response.
- Spontaneous CI-V transceive messages arriving between echo and response (skipped by the `FE FE` preamble search).

23.2 Handbook Rules

Element	Value	Description
`ExpectEcho`	`true`	Activates the two-phase CI-V read in the adapter.
`ExpectedLength`	Response length only, **without echo**	Used for sanity checks only; the adapter uses the `FD` terminator.
`startIndex`	**Response-relative** (zero-based from the first `FE` of the response)	Do NOT add echo length.

startIndex calculation:

```
startIndex = response_header_length
           = 5  (simple commands: FE FE E0 {CIV} cmd → data starts at index 5)
           = 6  (cmd + subcmd:    FE FE E0 {CIV} cmd sub → data starts at index 6)
```

ExpectedLength calculation:

```
ExpectedLength = response_header_length + data_length + 1 (FD terminator)
```

Examples (IC-7100, CIV `88`):

Command	Response frame	ExpectedLength	startIndex
`03` GetFrequency	`FE FE E0 88 03 {5 BCD} FD` = 11 B	**11**	**5**
`04` GetMode	`FE FE E0 88 04 {mode} {filter} FD` = 8 B	**8**	**5**
`0F` GetSplit	`FE FE E0 88 0F {status} FD` = 7 B	**7**	**5**

`1C 00` GetPtt	`FE FE E0 88 1C 00 {status} FD` = 8 B	**8**	**6**
`15 02` GetSMeter	`FE FE E0 88 15 02 {2 BCD} FD` = 9 B	**9**	**6**
`25 00` GetFreqA	`FE FE E0 A4 25 00 {5 BCD} FD` = 12 B	**12**	**6**
Any SET (FB/FA)	`FE FE E0 {addr} FB FD` = 6 B	**6**	n/a

23.3 Two-Phase Read Architecture

`SerialAdapter.TransactCivAsync` implements the two-phase CI-V protocol read:

```
Phase 1 — echo drain
  Read bytes until 0xFD found (inclusive) → echo consumed and discarded
  Wire log: (civ echo) [N bytes] HEX: ...

Phase 2 — spurious byte skip
  Scan bytes until 0xFE 0xFE preamble found
  Wire log: (civ skip) [0xXX] for each discarded byte

Phase 3 — response capture
  Read from 0xFE 0xFE until next 0xFD → actual response
  Wire log: (civ response Xms) [N bytes] HEX: ...
```

The CI-V path is selected automatically when `ExpectEcho > 0` (i.e. `echoLength` parameter to `TransactAsync` is non-zero) AND the adapter is configured with the `FD` terminator. For all other protocols (Yaesu CAT, Kenwood CAT, TCI) the standard read path is used.

23.4 Older vs Newer Radios

Generation	Examples	USB Echo	EnableCivEchoBack	Notes
Older	IC-756, IC-7100, IC-7600, IC-7700	Always active	Not available	Implicit bus echo
Newer	IC-705, IC-7300, IC-7610, IC-9700	Configurable	Yes, via `1A 05 xxxx 01`	Explicitly enabled in Initialization

For newer radios, Initialization must include `EnableCivEchoBack`:

```
<Command name="EnableCivEchoBack">
    <Description>Enable CI-V echo back</Description>
    <Frame>FE FE {CIV} E0 1A 05 {REGISTER} 01 FD</Frame>
    <ExpectResponse>true</ExpectResponse>
    <DelayAfterMs>50</DelayAfterMs>
</Command>
```

The register varies by model — always verify it in the official radio manual or CI-V reference.

24. Validation Checklist

Before considering a handbook complete, verify the following items.

24.1 Metadata

- [ ] `Manufacturer` and `Model` are specified.

- [ ] `InterfaceType` is correct (`Serial`, `TCI`, `UDP`, or `Flex`).
- [ ] `Protocol` is specified.
- [ ] `CommandEncoding` is coherent: `ASCII` for CAT, `Hex` for CI-V.
- [ ] `Terminator` is correct: `;` for ASCII, `FD` for CI-V.

24.2 Connection

- [ ] `ConnectionDefaults` contains reasonable parameters.
- [ ] For CI-V, `CIVAddress` is specified.
- [ ] `ConnectionCheck` is defined.

24.3 ValueMaps

- [ ] `ModeMap` defines every supported mode.
- [ ] `ModeSetMap` is present only when the `SetMode` payload is not the inverted `ModeMap`.
- [ ] VFO mappings are defined when required.
- [ ] No duplicated `ValueMap` names exist.

24.4 VFOs and Modes

- [ ] `AvailableVfos` defines at least one VFO.
- [ ] Every VFO has a non-empty `value`.
- [ ] `SupportedModes` lists every mode supported by the radio.

24.5 GET Commands

- [ ] `GetFrequency*` exists for each VFO when the radio can read them independently.
- [ ] `GetMode*` exists for each VFO, or at least for the active VFO on active-only radios.
- [ ] `GetActiveVfo` exists when the radio has more than one VFO.
- [ ] Field names use state keys recognized by the system.

24.6 SET Commands

- [ ] `SetFrequency` uses `FREQ` plus `VFO_VALUE`/`VFO_NAME`, or declares `vfoScope="activeOnly"` when the radio can set only the active VFO.
- [ ] `SetMode` has a `MODE` parameter.
- [ ] `SetVfo` exists when the radio supports active VFO selection.
- [ ] Every `{PARAM}` in the frame has a corresponding `<Param>`.

24.7 Polling

- [ ] `Essential` group contains frequency commands.
- [ ] Intervals are reasonable, usually 100-500 ms for essential polling.
- [ ] Conditions are correct for conditional commands.

24.8 CI-V (Icom Radios Only)

- [ ] `ExpectEcho` is `true` on all commands (GET and SET).
- [ ] All GET commands have `ExpectedLength` = response frame length (without echo).
- [ ] All SET commands have `ExpectedLength=6` (FB/FA response).
- [ ] All SET commands have `SuccessIndicator=FB`.
- [ ] `startIndex` is calculated as `echo_length + response_header_length`.
- [ ] Newer radios (IC-705, IC-7300, IC-7610, IC-9700): `EnableCivEchoBack` is present in Initialization.

- [ ] Command bytes, response lengths, field offsets, and register values are verified against the official radio documentation.

24.9 Test

- [ ] The radio responds to `ConnectionCheck`.
- [ ] Frequency is read correctly.
- [ ] Mode is read correctly.
- [ ] Frequency can be set.
- [ ] Mode can be set.

25. Icom Radio-Specific Notes

25.1 CI-V Command `0F`: Split and FM Repeater Duplex

On Icom radios, CI-V command `0F` is **dual-purpose**: it controls both **split operation** (RX on VFO-A, TX on VFO-B) and **FM repeater duplex** (DUP−/DUP+) on radios that cover VHF/UHF bands. As a result, the value returned in response to a `GetSplit` query is not limited to `00` (Off) and `01` (Split ON): on radios with VHF/UHF capability, the radio may also return a duplex-mode code when it is in FM mode with a repeater offset selected.

This distinction matters for handbook authoring: the `SplitMap` must cover all values that the radio can actually return in read mode. The table below lists the valid read responses by model, as verified against the official Icom CI-V reference manuals.

25.2 CI-V Command `0F` Read Responses by Model

Sources: IC-7100 CI-V Reference Manual, IC-705 CI-V Reference Manual, IC-9700 CI-V Reference Manual.

Response byte	Meaning	IC-7100	IC-705	IC-9700	IC-7300 / IC-7300mk2	IC-7600 / IC-7610 / IC-7700 / IC-7800
`00`	Split OFF — normal simplex operation	✓	✓	✓	✓	✓
`01`	Split ON — TX on VFO-B, RX on VFO-A	✓	✓	✓	✓	✓
`11`	DUP− — FM repeater, minus frequency offset	✓	✓	✓	—	—
`12`	DUP+ — FM repeater, plus frequency offset	✓	✓	✓	—	—
`13`	DD Repeater Simplex (RPS) — D-STAR DD mode on 1.2	—	—	✓	—	—

	GHz					

Legend:

- ✓ = documented read response; entry present in the handbook `SplitMap`.
- — = not applicable; the radio does not have the band or mode that produces this value.

Why the columns differ:

- **IC-7100, IC-705** cover HF + 6 m + 2 m + 70 cm. FM repeater operation on VHF/UHF can produce DUP− (`11`) or DUP+ (`12`). Neither has D-STAR DD mode.
- **IC-9700** covers 2 m + 70 cm + 1.2 GHz only. It supports D-STAR DD mode on 1.2 GHz, which introduces the additional `13` (DD Repeater Simplex) code.
- **IC-7300, IC-7300mk2** cover HF + 6 m only. With no VHF/UHF bands, FM repeater duplex modes are mechanically impossible. The radio will never return `11` or `12`.
- **IC-7600, IC-7610, IC-7700, IC-7800** are HF (and optionally 6 m) only. Same reasoning as IC-7300.

25.3 Why `10` (Set Simplex) Does Not Appear in Any `SplitMap`

CI-V command `0F` uses sub-command value `10` as a **write-only** instruction meaning "set simplex operation". It instructs the radio to exit any split or duplex mode. It is **never returned** as a read response: when the radio is in simplex mode, it returns `00`, not `10`.

Including `10` in a `SplitMap` is therefore incorrect — it implies a documented read response that does not exist. The handbook XML comments reproduce this rule verbatim to prevent future confusion.

25.4 Mapping DUP−/DUP+ to the CAT4OM `split` State

CAT4OM maps all DUP−/DUP+ codes (`11`, `12`, `13`) to `false` in the `split` state field. This is deliberate.

The `split` flag in `RadioState` represents the state where **TX and RX VFOs are independently tuned** — the amateur radio meaning of "split". FM repeater duplex does not involve independent VFOs: both RX and TX use the same VFO, offset by a fixed repeater shift. The radio itself treats them as distinct modes (separate `SetSplit` and `SetDuplex` concepts), but from CAT4OM's perspective the meaningful question is "is the TX VFO different from the RX VFO?", and the answer for DUP−/DUP+ is always no.

The consequence is that when the radio is in FM repeater mode, CAT4OM correctly reports `split = false`, and the UI does not show the split indicator.

25.5 The IC-756 and Other Older Radios

The handbook for the **IC-756** defines a `SplitMap` but does **not** implement `GetSplit`. The IC-756 is a late 1990s radio; command `0F` behavior on that generation has not been verified against original Icom documentation and the risk of silent misinterpretation is higher than for newer models. Until the CI-V reference for this radio is reviewed, the split state is not polled and remains at its default value.

If you need to add `GetSplit` to the IC-756 handbook, verify the following against the IC-756 CI-V manual before writing any code:

70. Whether command `0F` exists and returns a single status byte.
71. Whether the response format is identical to the IC-7100 (`00`/`01`/`11`/`12`).
72. The total frame lengths (echo + response) to set `ExpectedLength` and `startIndex` correctly.

25.6 Adding `GetSplit` to a New Icom Handbook

When adding split polling to a new CI-V handbook, follow this checklist:

73. **Check the manual for command `0F` read responses.** Do not assume the values are the same as an existing model. The IC-9700 has `13` (DD RPS); older radios might have none of the duplex codes.
74. **Determine whether the radio has VHF/UHF bands.** If it is HF-only, the `SplitMap` needs only `00` and `01`.
75. **Verify `ExpectedLength` and `startIndex`.** For all current Icom handbooks, the pattern is:
    - Request: `FE FE {CIV} E0 0F FD` = 6 bytes (echo)
    - Response: `FE FE E0 {CIV} 0F {status} FD` = 7 bytes
    - `ExpectedLength = 7`, `startIndex = 11` (6 echo + 5 response header)
76. **Use `BYTE_MAP` format, not `BOOL`.** The field must be declared as `format="BYTE_MAP" map="SplitMap"`, not `format="BOOL"`. The BOOL format expects the strings `"1"` / `"0"` / `"true"` / `"false"`, not raw binary byte values.
77. **Annotate the `SplitMap` with a comment** listing each entry and its source, following the pattern used in existing IC-7100/IC-705/IC-9700 handbooks.

25.7 CI-V `0x25` is *selected/unselected*, not absolute A/B (critical)

This is the single most error-prone area of Icom handbooks. Command `0x25` does **not** address VFO A and VFO B by absolute index on many modern Icom radios. Confirmed by the **official IC-705 CI-V reference** (`IC-705_ENG_CI-V_1_20200721`): command `25` is *"Selected or unselected VFO frequency settings"* with sub-command `00 = Selected VFO`, `01 = Unselected VFO`. So `0x25 00` returns the **currently selected (operating) VFO** and `0x25 01` the **unselected** one — *relative to whichever VFO is active*, not absolute A/B. The same manual lists the operating frequency under commands `00 / 03 / 05`: `03` reads it, `05` sets it (and moves the live dial). There is **no** command to read which VFO is selected (`07` only *selects* A/`07 00`, B/`07 01`, equalizes, or exchanges — there is no `07 D2` read on the IC-705).

How it was confirmed: with CAT4OM polling `25 00`→`frequency.A` and `25 01`→`frequency.B`, the operator tuned the radio **from the front panel only**. The value that changed while tuning was **always `frequency.A` (`25 00`)**, regardless of whether VFO A or VFO B was selected; switching VFO on the radio swapped the two readings. Proof that `25 00` = *selected*, `25 01` = *unselected*.

Consequence of modelling it as absolute (the trap): mapping `25 00`→`frequency.A` and `25 01`→`frequency.B` is correct **only while the radio is on VFO A**. The moment the operator selects VFO B, `25 00` returns VFO B's frequency and CAT4OM files it under "A" — the displayed VFO label inverts and the frequency lands on the wrong VFO. This also breaks writes: setting "VFO B" via `25 01` while B is selected actually writes the *unselected* register (A), so the live dial never moves.

Two compounding facts:

78. `0x25` writes only the **background register** of a VFO; they do **not** move the radio's live operating dial when the target is the selected VFO. The live/operating frequency moves only with command `0x05` (or `0x00`). So `SetFrequency` via `0x25` leaves the dial visually stuck on the old value.
79. Some Icoms expose **no** command to read which VFO is selected. The IC-7600 has `07 D2` (`GetSelectedVfo`); the **IC-705 and IC-7700 do not**. Without that readback CAT4OM cannot know the active VFO when the operator works the front panel, so it cannot maintain a correct A/B split at all.

Recommended model — selected/unselected with `frequency.unselected` (for dual-VFO CI-V radios where `0x25` is selected/unselected **and** that lack a selected-VFO readback — currently **IC-705, IC-7100, IC-7700**):

- **Read the active VFO frequency** with `0x03` mapped to the **unqualified** key `frequency`. The engine applies an unqualified `frequency` to `ActiveVfo` (§14.1 two-pass logic). This is always correct.

- **Read the unselected VFO frequency** with `0x25 01` mapped to the **reserved** key `frequency.unselected`. The engine resolves this at runtime to the single VFO that is not `ActiveVfo` — so the mapping stays correct even when the operator switches VFO on the front panel. The A/B label may be temporarily wrong in that case (unavoidable without a VFO readback), but both frequencies are always present and always land on the right register.
- **Write** with `0x05`, `vfoScope="activeOnly"` — the operating frequency, which moves the live dial. **Never** use `0x25` for writes.
- **Do not** map `25 01` as `frequency.B` (absolute): that is the inversion trap described above.

`frequency.unselected` — engine mechanics:

- The suffix `unselected` is **reserved** by the engine (`RadioHandler.ReservedUnselectedVfoToken = "UNSELECTED"`, compared case-insensitively against the upper-cased qualifier produced by `TryGetQualifiedToken`).
- Resolution rule: `AvailableVfos.Where(v => v ≠ ActiveVfo)` → exactly one result → target VFO. If zero or ≥2 candidates (single-VFO radio, misconfigured handbook), the update is **skipped** and a WARNING is logged (project rule: WARNING on degraded result).
- **Read-only**: `frequency.unselected` must only appear in `<Field>` definitions of `<Response>` blocks. Never use it in a write command or `SetFrequency`.
- Scope: frequency only. `mode.unselected` and `filterwidth.unselected` are out of scope.

Frame geometry for `25 01` (same for all affected radios):

```
Echo:     FE FE {CIV} E0 25 01 FD        (7 bytes, stripped before indexing)
Response: FE FE E0 {CIV} 25 01 [5×BCD] FD (12 bytes)
ExpectedLength="12"  ExpectEcho="true"
Field startIndex="6" length="5" format="BCD5LE"
<Command name="GetFrequencyUnselected">
    <Description>Read the unselected VFO frequency. CI-V command 25 01.
        Maps to frequency.unselected (engine resolves to the VFO that is not
ActiveVfo).
    </Description>
    <Request>
        <Frame>FE FE {CIV} E0 25 01 FD</Frame>
    </Request>
    <Response>
        <ExpectedLength>12</ExpectedLength>
        <ExpectEcho>true</ExpectEcho>
        <Fields>
            <Field name="frequency.unselected" type="long" startIndex="6"
length="5" format="BCD5LE" />
        </Fields>
    </Response>
</Command>
```

And in the `Essential` poll group:

```
<Poll command="GetFrequency" updates="frequency" />
<Poll command="GetFrequencyUnselected" updates="frequency.unselected" />
```

Affected radios (dual-VFO, `0x25` selected/unselected, no `07 D2` readback):

Radio	CIV Address	Handbook file
IC-705	`A4`	`icom/icom-ic705.xml`

IC-7100	`88`	`icom/icom-ic7100.xml`
IC-7700	`74`	`icom/icom-ic7700.xml`

Radios that already use absolute `frequency.A` + `frequency.B` + `GetSelectedVfo` (`07 D2`) — such as IC-7600, IC-7610, IC-9700, IC-7800, IC-7850/7851, IC-7760 — are **correct as-is and must not be changed**.

If the radio *does* support `07 D2` (e.g. IC-7600): add `GetSelectedVfo` → `updates="activeVfo"` to the polling group. Combined with the unqualified `frequency` read above, the active-VFO label then stays correct even when the operator uses the front panel.

> ⚠ *Do **not** map `25 01` as absolute `frequency.B` on a radio without `07 D2` — that is the inversion trap. Do **not** use `25 {VFO}` for writes — it writes the background register and does not move the dial. When in doubt, use `frequency.unselected` for the read and `05/vfoScope="activeOnly"` for the write.*

26. CW Keyer Commands

CAT4OM supports hardware CW keying through radios that expose a built-in CW keyer over their native protocol (TCI or SmartSDR/Flex). Four standard command names are reserved for this purpose and are routed by the server as structured Control Port actions — not as raw pass-through bytes. This guarantees that the TX safety gate is applied and that the TX watchdog is armed for the duration of the transmission.

26.1 Standard CW Command Names

Command name	Purpose	TX action?
`CwSend`	Send a CW text string	Yes — arms TX watchdog
`CwStop`	Abort current CW transmission	No — disarms TX watchdog
`CwSpeed`	Set keyer speed (WPM)	No
`CwKeyDown`	Manual key-down/up (tune)	Yes when `enabled=true`

These names are **reserved** across all protocol types. A handbook that supports CW must use exactly these names so the server dispatcher can route the `sendCw`, `stopCw`, `setCwSpeed`, and `cwKeyDown` Control Port actions correctly.

26.2 TX Safety and Watchdog

`CwSend` and `CwKeyDown(enabled=true)` key the transmitter. Unlike `SetPtt`, the radio does not always send an unsolicited push that would update `RadioState.Ptt`. The server handler therefore **explicitly sets `RadioState.Ptt = true`** before issuing the handbook command frame and arms the TX watchdog. `CwStop` and `CwKeyDown(enabled=false)` explicitly clear `Ptt` and disarm the watchdog.

This means all existing TX safety rules apply unchanged:

- Only the master client may issue `CwSend` or `CwKeyDown(true)`.
- While the CW watchdog is armed, ownership cannot change and no other mutating command is accepted on the transmitting radio.

26.3 TCI Protocol — Command Definitions

```
<!-- Send CW text. TCI CW escaping (':'^, ','~, ';'*) applied by server handler. -
->
<Command name="CwSend">
```

```
    <Description>Send CW text via TCI cw_macros.</Description>
    <Request>
        <Frame>cw_macros:{TRX},{TEXT};</Frame>
        <Parameters>
            <Param name="TRX"  type="int"    format="ASCII"/>
            <Param name="TEXT" type="string" format=""/>
        </Parameters>
    </Request>
    <Response><ExpectEcho>false</ExpectEcho></Response>
</Command>

<!-- Stop CW. No TRX argument: TCI stops CW globally. -->
<Command name="CwStop">
    <Description>Stop current CW transmission (cw_macros_stop).</Description>
    <Request>
        <Frame>cw_macros_stop;</Frame>
    </Request>
    <Response><ExpectEcho>false</ExpectEcho></Response>
</Command>

<!-- Set CW speed in WPM. -->
<Command name="CwSpeed">
    <Description>Set CW macro speed in WPM.</Description>
    <Request>
        <Frame>CW_MACROS_SPEED:{WPM};</Frame>
        <Parameters>
            <Param name="WPM" type="int" format="ASCII"/>
        </Parameters>
    </Request>
    <Response><ExpectEcho>false</ExpectEcho></Response>
</Command>

<!-- Manual key-down/up (tune). -->
<Command name="CwKeyDown">
    <Description>Key-down (manual tune) via TCI TUNE command.</Description>
    <Request>
        <Frame>TUNE:{TRX},{ENABLED};</Frame>
        <Parameters>
            <Param name="TRX"     type="int"  format="ASCII"/>
            <Param name="ENABLED" type="bool" format="BOOL_TCI"/>
        </Parameters>
    </Request>
    <Response><ExpectEcho>false</ExpectEcho></Response>
</Command>
```

TCI CW text escaping: the TCI `cw_macros` command uses `,` and `;` as field delimiters. Any CW text that contains these characters must be escaped before the frame is built:

Character	Escaped as
`:`	`^`
`,`	`~`
`;`	`*`

This escaping is **not** declared in the handbook XML. It is applied by the `sendCw` action handler in `RadioGroup` before the parameter is passed to the handbook command executor. Handbook authors do not need to handle it.

26.4 Flex/SmartSDR Protocol — Command Definitions

```
    <!-- Send CW text. Directed to the active TX slice. -->
    <Command name="CwSend">
        <Description>Send CW text via SmartSDR "cw send {text}" command.</Description>
        <Request>
            <Frame>cw send {TEXT}</Frame>
            <Parameters>
                <Param name="TEXT" type="string" format=""/>
            </Parameters>
        </Request>

    <Response><ExpectResponse>false</ExpectResponse><ExpectEcho>false</ExpectEcho></Re
    sponse>
    </Command>

    <!-- Stop CW (clear queue). -->
    <Command name="CwStop">
        <Description>Stop current CW transmission via SmartSDR "cw clear"
    command.</Description>
        <Request>
            <Frame>cw clear</Frame>
        </Request>

    <Response><ExpectResponse>false</ExpectResponse><ExpectEcho>false</ExpectEcho></Re
    sponse>
    </Command>

    <!-- Set CW speed in WPM. -->
    <Command name="CwSpeed">
        <Description>Set CW speed in WPM via SmartSDR "cw wpm {wpm}"
    command.</Description>
        <Request>
            <Frame>cw wpm {WPM}</Frame>
            <Parameters>
                <Param name="WPM" type="int" format="ASCII"/>
            </Parameters>
        </Request>

    <Response><ExpectResponse>false</ExpectResponse><ExpectEcho>false</ExpectEcho></Re
    sponse>
    </Command>

    <!-- Manual key-down/up (tune). Uses xmit; BOOL format: true→"1", false→"0". -->
    <Command name="CwKeyDown">
        <Description>Key-down (manual tune) via SmartSDR "xmit {0|1}" command. Radio-
    wide.</Description>
        <Request>
```

```
            <Frame>xmit {ENABLED}</Frame>
            <Parameters>
                <Param name="ENABLED" type="bool" format="BOOL"/>
            </Parameters>
        </Request>

    <Response><ExpectResponse>false</ExpectResponse><ExpectEcho>false</ExpectEcho></Re
    sponse>
    </Command>
```

Notes for Flex:

- `CwSend` is directed to the active TX slice; no VFO index is needed.
- `CwStop` clears the entire CW queue. There is no per-slice stop in SmartSDR API 3.x.
- `CwKeyDown` reuses the generic `xmit` command (same as `SetPtt`). This is correct for tune purposes. A future SmartSDR firmware version may expose a dedicated `cw key` command.
- CW speed set via `CwSpeed` persists in the SmartSDR session until changed again.

26.5 Adding CW Support to a New Handbook

A handbook declares CW support by including **all four** standard command names (`CwSend`, `CwStop`, `CwSpeed`, `CwKeyDown`). The server infers CW capability from the presence of `CwSend` in `availableCommands` on the published `RadioState`.

Protocol-specific notes:

- For TCI handbooks: use `BOOL_TCI` for boolean parameters; apply `{TRX}` where the protocol requires a receiver index.
- For Flex handbooks: use `BOOL` (1/0); omit `{TRX}` where SmartSDR commands are radio-wide.
- For CAT/serial handbooks: CW keying via CAT varies by model. If the radio supports it, map the four command names to the appropriate CAT frames. If it does not, omit the CW commands entirely — the server will not expose `CwSend` in `availableCommands` and clients will not offer the CW keyer option for that radio.

27. Power Commands

CAT4OM supports two optional power control commands: `PowerOn` and `PowerOff`. Both are purely data-driven and defined in the handbook like any other command. Neither requires engine changes — simply add the command to the handbook and the server exposes the capability automatically.

27.1 PowerOff

`PowerOff` is invoked through the live `RadioHandler` when the operator sends a `powerOff` action (Control Port) or a `powerOffRadio` action (Management Port). After the command executes successfully the server disconnects the adapter immediately rather than waiting for the serial link to go silent.

A typical single-frame definition:

```
<Command name="PowerOff">
  <Description>Power off the radio via CAT</Description>
  <Request><Frame>PS0;</Frame></Request>
  <Response><ExpectEcho>false</ExpectEcho></Response>
</Command>
```

Notes:

- Set `<ExpectEcho>false</ExpectEcho>` because the radio powers down and cannot echo the frame.

- CI-V radios do not need `<ExpectEcho>` — echo handling is part of the two-phase read and unaffected by power-off.
- Once defined, `PowerOff` appears in `availableCommands` in the published `RadioState`. Clients should gate the power-off UI element on its presence.

27.2 PowerOn

`PowerOn` is invoked via a **one-shot adapter** created by the Management Port. The radio does not need to be in a running group and no `RadioHandler` is created. The server opens a temporary serial connection, executes the command, and disposes the adapter.

This works because many radios (e.g., Yaesu FTDX-101D with USB connection) keep the USB-to-serial chip powered from the USB bus even when the radio is in standby, allowing CAT frames to be received before the radio finishes booting.

For radios that require multiple wake-up frames with delays, use a `<Sequence>`:

```
<Command name="PowerOn">
  <Description>Power on the radio via USB CAT (three-step PS1; wake
sequence)</Description>
  <Sequence>
    <Step name="WakeUSB" delayAfterMs="1000">
      <Frame>PS1;</Frame>
      <Response><ExpectsResponse>false</ExpectsResponse></Response>
    </Step>
    <Step name="PowerOn1" delayAfterMs="1000">
      <Frame>PS1;</Frame>
      <Response><ExpectsResponse>false</ExpectsResponse></Response>
    </Step>
    <Step name="PowerOn2">
      <Frame>PS1;</Frame>
      <Response><ExpectsResponse>false</ExpectsResponse></Response>
    </Step>
  </Sequence>
</Command>
```

Notes:

- Use `<ExpectsResponse>false</ExpectsResponse>` for each step — the radio is booting and cannot reply reliably.
- `delayAfterMs` on the step element is already supported by `CommandExecutor.ExecuteSequenceAsync`. There is no minimum or maximum — use a value appropriate for the radio's boot time.
- `PowerOn` does **not** appear in `availableCommands` because it is not callable through a live `RadioHandler`. It is silently omitted from the published state.

27.3 Supported Handbooks

Handbook	`PowerOn`	`PowerOff`	Notes
`yaesu/ftdx-101d.xml`	✓ (3-step `PS1;` sequence, 1 s delays)	✓ (`PS0;`)	USB serial chip stays powered in standby
`yaesu/ft-710.xml`	✓ (3-step `PS1;` sequence, 1 s delays)	✓ (`PS0;`)	USB serial chip stays powered in standby
`yaesu/ft-891.xml`	✓ (3-step `PS1;` sequence,	✓ (`PS0;`)	USB serial chip stays

	1 s delays)		powered in standby
`yaesu/ft-991.xml`	✓ (3-step `PS1;` sequence, 1 s delays)	✓ (`PS0;`)	USB serial chip stays powered in standby
`yaesu/ft-991a.xml`	✓ (3-step `PS1;` sequence, 1 s delays)	✓ (`PS0;`)	USB serial chip stays powered in standby
`yaesu/ftdx10.xml`	✓ (3-step `PS1;` sequence, 1 s delays)	✓ (`PS0;`)	USB serial chip stays powered in standby
`yaesu/ftx-1.xml`	✓ (3-step `PS1;` sequence, 1 s delays)	✓ (`PS0;`)	USB serial chip stays powered in standby

Icom CI-V handbooks — CI-V command `18 00` (off) / `18 01` (on). `PowerOn` requires 50 FE preamble bytes to wake the USB-serial chip (covers baud rates up to 38400 bps; see §29). `PowerOff` is a standard CI-V frame with no preamble.

Handbook	`PowerOn`	`PowerOff`	Notes
`icom/icom-ic705.xml`	✓ (50-FE preamble + `18 01`)	✓ (`18 00`)	Confirmed from official CI-V manual
`icom/icom-ic7000.xml`	✓ (50-FE preamble + `18 01`)	✓ (`18 00`)	USB, post-2004
`icom/icom-ic7100.xml`	✓ (50-FE preamble + `18 01`)	✓ (`18 00`)	USB
`icom/icom-ic7200.xml`	✓ (50-FE preamble + `18 01`)	✓ (`18 00`)	USB
`icom/icom-ic7300.xml`	✓ (50-FE preamble + `18 01`)	✓ (`18 00`)	Confirmed from official CI-V manual
`icom/icom-ic7300mk2.xml`	✓ (50-FE preamble + `18 01`)	✓ (`18 00`)	Same family as IC-7300
`icom/icom-ic7410.xml`	✓ (50-FE preamble + `18 01`)	✓ (`18 00`)	USB
`icom/icom-ic7600.xml`	✓ (50-FE preamble + `18 01`)	✓ (`18 00`)	USB
`icom/icom-ic7610.xml`	✓ (50-FE preamble + `18 01`)	✓ (`18 00`)	USB
`icom/icom-ic7700.xml`	✓ (50-FE preamble + `18 01`)	✓ (`18 00`)	USB
`icom/icom-ic7760.xml`	✓ (50-FE preamble + `18 01`)	✓ (`18 00`)	USB
`icom/icom-ic7800.xml`	✓ (50-FE preamble + `18 01`)	✓ (`18 00`)	USB
`icom/icom-ic7850.xml`	✓ (50-FE preamble + `18 01`)	✓ (`18 00`)	USB
`icom/icom-ic7851.xml`	✓ (50-FE preamble + `18 01`)	✓ (`18 00`)	USB
`icom/icom-ic9100.xml`	✓ (50-FE preamble + `18 01`)	✓ (`18 00`)	USB

`icom/icom-ic9700.xml`	✓ (50-FE preamble + `18 01`)	✓ (`18 00`)	Confirmed from official CI-V manual

Other handbooks (vintage radios without USB: IC-756, IC-746, IC-735, IC-706, IC-751, IC-725/6/8/9, IC-736/7/8, IC-761/5, IC-781, IC-820/1, IC-910, IC-970, IC-703, IC-707, IC-718, IC-77/8, IC-275) do not define power commands — these radios use RS-232C or DE-9 serial and have no USB standby power for remote wake-up.

28. Yaesu IF Response Layout Variants

Yaesu radios share the `IF;` polling command but the response layout differs between radio generations. The difference is in **P1** (channel/memory indicator): older Yaesu HF radios use a 3-byte P1 while newer radios (FTX-1 and later multi-band/dual-VFO flagships) use a 5-byte P1.

28.1 3-byte P1 layout (FTDX-101D, FTDX10, FT-710, FT-891, FT-991, FT-991A)

```
Byte:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 ...
       I  F [P1 P1 P1][P2 P2 P2 P2 P2 P2 P2 P2 P2][P3 P3 P3 P3 P3] P4 P5 ...
              3 bytes       9 bytes (freq Hz)       sign+4dig(offset)
```

Field	`startIndex`	`length`	Format
`frequency` (FTDX10 only)	5	9	`ASCII`
`ritOffset` (sign + offset)	14	5	`ASCII_SIGNED`
`ritEnabled` (RX CLAR)	19	1	`BOOL`
`xitEnabled` (TX CLAR)	20	1	`BOOL`

28.2 5-byte P1 layout (FTX-1)

```
Byte:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 ...
       I  F [P1 P1 P1 P1 P1][P2 P2 P2 P2 P2 P2 P2 P2 P2][P3 P3 P3 P3 P3] P4 P5 ...
                 5 bytes           9 bytes (freq Hz)        sign+4dig(offset)
```

Field	`startIndex`	`length`	Format
`frequency`	7	9	`ASCII`
`ritOffset` (sign + offset)	16	5	`ASCII_SIGNED`
`ritEnabled` (RX CLAR)	21	1	`BOOL`
`xitEnabled` (TX CLAR)	22	1	`BOOL`

When porting a new Yaesu radio, check the byte column numbers in the manufacturer's CAT manual to determine which layout applies. The P1 length is listed in the `IF` command's "Answer" row count.

29. Icom CI-V Power On: FE Preamble Wake-up Mechanism

When an Icom radio is in standby the USB-to-serial chip stays powered, but the CI-V parser is not yet running. To guarantee the power-on command (`18 01`) is received, the Icom CI-V specification requires sending a burst of `FE` bytes before the standard frame to wake the UART.

29.1 Required FE count by baud rate

Values from the IC-7300 CI-V manual (use the higher figure when manuals disagree):

Baud rate	FE bytes needed
4800	7
9600	13
19200	25
38400	50
57600	75
115200	150

29.2 How the preamble works

`FE` is the CI-V frame start marker. The radio's parser is looking for the pattern `FE FE {addr} E0 ...`. When it sees repeated `FE` bytes in the stream, it keeps reading until it matches a valid frame. The extra `FE` bytes before the address are discarded.

The full power-on sequence at 38400 bps for an IC-7300 (address `94`) looks like:

```
FE FE FE FE FE FE FE FE FE FE   ← 10 FEs
FE FE FE FE FE FE FE FE FE FE   ← 20
FE FE FE FE FE FE FE FE FE FE   ← 30
FE FE FE FE FE FE FE FE FE FE   ← 40
FE FE FE FE FE FE FE FE FE FE   ← 50
94 E0 18 01 FD                   ← CI-V command (no leading FE FE — the last two
FEs above serve as the frame header)
```

In CAT4OM handbooks the preamble and frame are written as a single `<Frame>` string in a Sequence step. The `{CIV}` placeholder is substituted at runtime.

29.3 Handbook implementation

All Icom handbooks listed in §27.3 use 50 FE bytes, which covers baud rates up to 38400 bps — the highest practical default for these radios. At baud rates above 38400 bps (57600 / 115200) the preamble may be insufficient and power-on via CAT may fail; in that case use the front-panel power button.

```
<Command name="PowerOn">
    <Description>Power on the radio via CI-V (50-FE preamble + command 18 01, USB
standby required)</Description>
    <Sequence>
        <Step name="WakeAndPowerOn">
            <Frame>FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE
                   FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE
                   FE FE FE FE FE FE FE FE FE FE {CIV} E0 18 01 FD</Frame>
            <Response>
                <ExpectsResponse>false</ExpectsResponse>
            </Response>
        </Step>
    </Sequence>
</Command>
```

`ExpectsResponse>false` is correct: the radio is booting and cannot respond during the power-on sequence.

End of Manual - 2026-06-13