WPX Prefix Lookup Specification

Version: 0.31.60-Beta Date: 2025-08-24

--- Revision History ---

[0.31.60-Beta] - 2025-08-24

Changed

- Updated the "Data Flow" section to reflect the module's refactoring

from a `calculation_module` to a `custom_multiplier_resolver`.

- Updated function names to match the refactored source code.

[0.31.59-Beta] - 2025-08-15

Changed

- Overhauled the algorithm description to accurately reflect the

sophisticated, multi-stage logic from the code, including the

critical DXCCPfx override.

[1.2.0-Beta] - 2025-08-11

Changed

- Updated the description of the `calculate_wpx_prefixes` function to

reflect the correct "once per contest" logic.

[1.1.0-Beta] - 2025-08-11

Changed

- Updated the algorithm and implementation sections to match the

corrected, hierarchical logic in the `cq_wpx_prefix.py` module.

[1.0.0-Beta] - 2025-08-10

Added

- Initial release of the WPX Prefix Lookup technical specification.

---

1. Introduction

This document provides the definitive technical specification for how WPX (Worked All Prefixes) multipliers are calculated and processed by the Contest Log Analyzer. It covers both the abstract algorithm and its concrete implementation in the Python source code.

2. The Prefix Lookup Algorithm

This section defines the formal, step-by-step logic for determining a valid WPX prefix from a single amateur radio

callsign. The process is hierarchical, checking for special cases before proceeding to the standard calculation.

2.1. Pre-processing

The algorithm begins by cleaning the raw callsign string.

1. **Strip Suffixes**: Common non-prefix suffixes like `/P`, `/M`, `/QRP`, etc., are removed.
2. **Maritime Mobile**: The call is checked for `/MM`. If present, the call is immediately classified as having an "Unknown" prefix, and the process terminates.

2.2. Portable Call Logic

If the cleaned callsign contains a `/`, it is processed by the portable call heuristics in `get_cty.py` to determine a `portableid`.

- **call/letters** (e.g., `LX/KD4D`): The `portableid` is `LX`. The prefix is formed by appending a `0` (e.g., `LX0`).
- **call/digit** (e.g., `WN5N/7`): The `portableid` is `7`. The prefix is formed by replacing the digit in the base call's prefix with the portable digit (e.g., `WN5` becomes `WN7`).
- **call/prefix** (e.g., `VP2V/KD4D`): The `portableid` is `VP2V`. The prefix is the `portableid` itself.

2.3. Standard Prefix Calculation

If the call is not a portable, the following logic is used:

1. **Default Prefix**: The default prefix is calculated as "everything up to and including the last digit" (e.g., `K3LR` -> `K3`, `WX3B` -> `WX3`). A fallback rule creates a prefix for calls with no numbers (e.g., `PA` -> `PA0`).
2. **DXCCPfx Override**: The algorithm then checks the `DXCCPfx` value for the callsign (provided by `get_cty.py`). If the `DXCCPfx` is a longer, more specific prefix than the default (e.g., for call `R1FJ`, the default is `R1` but the `DXCCPfx` is `R1F`), the **`DXCCPfx` is used as the final prefix**. This is the highest-priority rule for standard calls.
3. **Final Validation**: If the final calculated prefix is only a single digit, it is invalidated and classified as "Unknown".

---

3. The Python Implementation

This section describes how the algorithm and the higher-level "first-worked" logic are implemented in the project's source code.

3.1. Overview

The full process involves two stages, handled by two separate functions within the `cq_wpx_prefix.py` module:

- A low-level helper function (`_get_prefix`) that implements the hierarchical algorithm from Section 2.
- A high-level orchestrator function (`resolve_multipliers`) that uses this helper to implement the stateful "first-worked per contest" logic.

`_get_prefix(row)` function

This helper function is the direct, line-by-line implementation of the hierarchical algorithm described in Section 2.

It accepts a full DataFrame row (containing the `Call`, `DXCCPfx`, and `portableid`) and returns the final, calculated prefix string, correctly handling all special cases.

`resolve_multipliers(df)` function

This is the main function called by the log processing engine. It implements the "first-worked per contest" logic. Its process is as follows:

1. It takes the full, unprocessed QSO DataFrame as input.
2. It applies the `_get_prefix` helper to every row to generate a `WPXPfx` column containing the prefix for every QSO.
3. It then sorts the DataFrame chronologically and iterates through it to find the first time each unique prefix was worked.
4. The function returns the DataFrame with two new columns: a dense `WPXPfx` column and a sparse `Mult1` column (for scoring) that contains the prefix only on the row where it was first worked.

3.3. Data Flow and Orchestration

`contest_log.py`

This script is the central orchestrator for all log processing. Its `apply_contest_specific_annotations` method reads the `custom_multiplier_resolver` key from the relevant `.json` file. It then uses this key to dynamically import and execute the `resolve_multipliers` function from the `cq_wpx_prefix.py` module. This happens early in the annotation sequence, before standard multiplier rules are processed.

`get_cty.py`

This utility is fundamental to the overall log processing pipeline. The prefix is derived from a combination of the callsign string itself and its associated `DXCCPfx` value, which is provided by the `get_cty.py` utility. The `DXCCPfx` serves as a critical override in the logic.