# Command Dictionary Tutorial & Reference

**Document Version:** 2.0
**Last Updated:** February 19, 2026
**Application Version:** UDP Server Manager v2.0+

---

## Table of Contents



---

## Purpose

Command dictionaries define the **complete set of commands** that the UDP Server Manager (Supervisor) uses to communicate with edge devices. They serve as the single source of truth for:

- Available commands and their organization
- Parameter requirements and types
- Input validation rules
- UI generation (hierarchical menus, parameter widgets)
- Message formatting and protocol

**Key Benefits:**

- **Consistency:** Both client and device use same command definitions
- **Maintainability:** Single location for all command specifications
- **Flexibility:** Add new commands without modifying application code
- **Documentation:** Self-documenting command structure

---

## What is a Command Dictionary?

A **command dictionary** is a JSON file that contains:

1. **Categories Array:** List of command categories for menu organization
2. **Commands Object:** Definitions of all available commands with parameters

**Location:** `core/workers/[device_type]/[device]_commandDictionary.json`

**Example:** `core/workers/capstanDrive/capstanDrive_commandDictionary.json`

---

# JSON Structure

## Top-Level Structure

```
{
  "categories": [
    "Device Control",
    "Device Status",
    "Error Handling",
    "System Administration"
  ],
  "commands": {
    "COMMAND_NAME": {
      "name": "COMMAND_NAME",
      "category": "Device Control",
      "description": "Command description",
      "parameters": [...]
    },
    ...
  }
}
```

## Required Fields

### Root Level

- `categories` (array): List of category names for hierarchical menu
- `commands` (object): Dictionary of command definitions

### Command Level

- `name` (string): Command name (must match key)
- `category` (string): Category name (must exist in categories array)
- `description` (string): Human-readable description
- `parameters` (array): Array of parameter objects (can be empty)

---

# Categories System

**NEW in v2.0:** Commands organized in hierarchical menu by category.

## Purpose

Categories provide:

- **Logical Grouping:** Related commands grouped together
- **Improved Navigation:** Flyout menus instead of long flat list
- **Scalability:** Easy to manage dozens of commands
- **Organization:** Clear separation of command types

### Standard Categories

**Recommended Categories:**

1. **Device Control** - Commands that control device hardware
2. **Device Status** - Commands that query device state
3. **Error Handling** - Commands for diagnostics and error management
4. **System Administration** - Commands for system configuration

### Custom Categories

**You can define any categories you need:**

```
{
  "categories": [
    "Motor Control",
    "Sensor Readings",
    "Calibration",
    "Network Settings",
    "Debugging"
  ]
}
```

**Category Rules:**

- Category names should be clear and descriptive
- Use Title Case (e.g., "Motor Control" not "motor_control")
- Limit to 2-6 categories for optimal UI
- Each command must belong to exactly one category

---

## Command Definitions

### Basic Command (No Parameters)

```
{
  "GET_ALL": {
    "name": "GET_ALL",
    "category": "Device Status",
    "description": "Query all device parameters at once",
    "parameters": []
  }
}
```

### Command with Parameters

```
{
  "LED": {
    "name": "LED",
    "category": "Device Control",
    "description": "Control LED state and behavior",
    "parameters": [
      {
        "name": "LED_MODE",
        "type": "enum",
        "options": ["OFF", "ON", "BLINK"],
        "default": "OFF",
        "description": "LED operating mode"
      },
      {
        "name": "BLINK_RATE",
        "type": "integer",
        "min": 1,
        "max": 1000,
        "default": 500,
        "description": "Blink rate in milliseconds",
        "condition": {
          "parameter": "LED_MODE",
          "value": "BLINK"
        }
      }
    ]
  }
}
```

### Command Fields

| Field | Type | Required | Description |
|---|---|---|---|
| name | string | Yes | Command identifier (uppercase with underscores) |
| category | string | Yes | Category name (must exist in categories array) |
| description | string | Yes | Human-readable description for UI/docs |
| parameters | array | Yes | Array of parameter objects (empty if no params) |

## Parameter Types

### Supported Parameter Types

**v2.0 supports up to 10 parameters per command.**

**1. Enum (Dropdown Selection)**

**Use Case:** Fixed set of mutually exclusive options

**JSON:**

```
{
  "name": "MOTOR_DIRECTION",
  "type": "enum",
  "options": ["FORWARD", "REVERSE", "BRAKE"],
  "default": "FORWARD",
  "description": "Motor rotation direction"
}
```

**UI:** QComboBox dropdown with options
**Message Format:** Index of selected option (0, 1, 2)

**2. Boolean (True/False)**

**Use Case:** Binary on/off, enable/disable choices

**JSON:**

```
{
  "name": "ENABLE_SAFETY",
  "type": "boolean",
  "default": true,
  "description": "Enable safety interlocks"
}
```

**UI:** QComboBox with "FALSE" / "TRUE"
**Message Format:** "0" for false, "1" for true

**3. Integer (Whole Numbers)**

**Use Case:** Counts, steps, IDs, indices

**JSON:**

```
{
  "name": "STEP_COUNT",
  "type": "integer",
  "min": 0,
  "max": 10000,
  "default": 1000,
  "description": "Number of steps to execute"
}
```

**UI:** QLineEdit with integer validator
**Message Format:** String representation (e.g., "1000")

#### 4. Float (Decimal Numbers)

**Use Case:** Measurements, voltages, temperatures, percentages

**JSON:**

```
{
  "name": "TARGET_VOLTAGE",
  "type": "float",
  "min": 0.0,
  "max": 5.0,
  "precision": 2,
  "default": 3.3,
  "description": "Target output voltage in volts"
}
```

**UI:** QLineEdit with double validator
**Message Format:** String with precision (e.g., "3.30")

#### 5. String (Text Input)

**Use Case:** Names, labels, messages

**JSON:**

```
{
  "name": "DEVICE_NAME",
  "type": "string",
  "max_length": 32,
  "default": "Device001",
  "description": "Device identifier string"
}
```

**UI:** QLineEdit with length limit
**Message Format:** Raw string (e.g., "Device001")

### Parameter Fields

| Field | Type | Required | Applies To | Description |
|---|---|---|---|---|
| name | string | Yes | All | Parameter identifier (uppercase) |
| type | string | Yes | All | Parameter type (enum/boolean/integer/float/string) |
| description | string | Yes | All | Human-readable description |

| Field | Type | Required | Applies To | Description |
|---|---|---|---|---|
| default | varies | Yes | All | Default value when command selected |
| options | array | Yes | enum | List of valid options |
| min | number | No | integer, float | Minimum allowed value |
| max | number | No | integer, float | Maximum allowed value |
| precision | integer | No | float | Decimal places for display/validation |
| max_length | integer | No | string | Maximum character count |
| condition | object | No | All | Conditional display rule |

### Conditional Parameters

**Purpose:** Show parameter only when another parameter has specific value.

**Example:** Show BLINK_RATE only when LED_MODE is "BLINK"

**JSON:**

```
{
  "name": "BLINK_RATE",
  "type": "integer",
  "min": 1,
  "max": 1000,
  "default": 500,
  "description": "Blink rate in milliseconds",
  "condition": {
    "parameter": "LED_MODE",
    "value": "BLINK"
  }
}
```

**Condition Object:**

- `parameter` (string): Name of dependency parameter
- `value` (string): Required value to show this parameter

**Rules:**

- Dependency parameter must be defined before dependent parameter
- Condition value must match exactly (case-sensitive)
- Multiple levels of dependencies not supported (A depends on B, B depends on C ❌)

---

## Complete Example

**File:** `capstanDrive_commandDictionary.json`

```
{
  "categories": [
    "Device Control",
    "Device Status",
    "Error Handling",
    "System Administration"
  ],
  "commands": {
    "LED": {
      "name": "LED",
      "category": "Device Control",
      "description": "Control LED state and blinking",
      "parameters": [
        {
          "name": "LED_MODE",
          "type": "enum",
          "options": ["OFF", "ON", "BLINK"],
          "default": "OFF",
          "description": "LED operating mode"
        },
        {
          "name": "BLINK_RATE",
          "type": "integer",
          "min": 1,
          "max": 1000,
          "default": 500,
          "description": "Blink rate in milliseconds",
          "condition": {
            "parameter": "LED_MODE",
            "value": "BLINK"
          }
        }
      ]
    },
    "STEPPER": {
      "name": "STEPPER",
      "category": "Device Control",
      "description": "Execute stepper motor movement",
      "parameters": [
        {
          "name": "STEP_COUNT",
          "type": "integer",
          "min": -10000,
          "max": 10000,
          "default": 1000,
          "description": "Number of steps (negative for reverse)"
        },
        {
          "name": "SPEED",
          "type": "integer",
          "min": 1,
          "max": 1000,
          "default": 100,
```

```
            "description": "Steps per second"
          },
          {
            "name": "ACCELERATION",
            "type": "integer",
            "min": 1,
            "max": 500,
            "default": 50,
            "description": "Acceleration in steps/sec²"
          }
        ]
      },
      "GET_STEPPER": {
        "name": "GET_STEPPER",
        "category": "Device Status",
        "description": "Query current stepper motor position and status",
        "parameters": []
      },
      "GET_ALL": {
        "name": "GET_ALL",
        "category": "Device Status",
        "description": "Query all device parameters",
        "parameters": []
      },
      "GET_ERROR_LOG": {
        "name": "GET_ERROR_LOG",
        "category": "Error Handling",
        "description": "Retrieve error log entries",
        "parameters": []
      },
      "CLEAR_DEBUG_LOG": {
        "name": "CLEAR_DEBUG_LOG",
        "category": "Error Handling",
        "description": "Erase debug log memory",
        "parameters": []
      },
      "SET_TIME": {
        "name": "SET_TIME",
        "category": "System Administration",
        "description": "Set device real-time clock",
        "parameters": [
          {
            "name": "TIMESTAMP",
            "type": "integer",
            "min": 0,
            "max": 2147483647,
            "default": 1640995200,
            "description": "Unix timestamp (seconds since 1970-01-01)"
          }
        ]
      }
    }
  }
```

---

# Best Practices

## Command Naming

**DO:**

- Use UPPERCASE with underscores: `GET_MOTOR_STATUS`
- Be descriptive: `SET_LED_BRIGHTNESS` not `LED_BR`
- Use verb prefix for actions: GET_, SET_, START_, STOP_, CLEAR_
- Be consistent across device types

**DON'T:**

- Use lowercase or camelCase: `getMotorStatus` ❌
- Use special characters: `GET-STATUS` ❌
- Use ambiguous abbreviations: `SLB` ❌
- Mix naming styles: `getLED`, `SET_HPL` ❌

## Category Organization

**DO:**

- Group related commands logically
- Use 3-6 categories (optimal for UI)
- Name categories clearly: "Motor Control" not just "Motors"
- Keep category names concise (2-3 words max)

**DON'T:**

- Create too many categories (>10 becomes unwieldy)
- Use vague names: "Misc", "Other" ❌
- Mix abstraction levels: "Hardware" and "LED" in same list ❌

## Parameter Design

**DO:**

- Provide sensible defaults for all parameters
- Define min/max for numeric types
- Use enums instead of integers for modes
- Add descriptions for all parameters
- Order parameters logically (most important first)

**DON'T:**

- Leave default values blank
- Use magic numbers without validation: `{"type": "integer"}` ❌
- Create overly complex conditional chains
- Exceed 10 parameters per command

### Documentation

**DO:**

- Write clear descriptions for commands and parameters
- Document units: "milliseconds", "degrees", "steps/second"
- Explain parameter interactions in descriptions
- Keep command dictionary in version control

**DON'T:**

- Use terse descriptions: "LED" ❌ (use "Control LED state and blinking" ✓)
- Assume users know units: "Speed: 100" ❌ (use "Steps per second: 100" ✓)
- Leave parameters undocumented

---

## For Embedded C Developers

### Conceptual Mapping

**Command Dictionary ↔ C Code Equivalents:**

**Categories Array:**

```
// Similar to grouping in header comments or separate files
// deviceControl.h, deviceStatus.h, errorHandling.h, sysAdmin.h
```

**Command Definition:**

```
// JSON: "LED": {"name": "LED", "category": "Device Control", ...}

// C equivalent:
#define CMD_LED 0x01

typedef struct {
    uint8_t cmd_id;      // CMD_LED
    uint8_t mode;        // 0=OFF, 1=ON, 2=BLINK
    uint16_t blink_rate; // milliseconds
} led_command_t;
```

**Parameters:**

```
// JSON: "parameters": [{"name": "LED_MODE", "type": "enum", ...}]

// C equivalent:
typedef enum {
    LED_OFF = 0,
    LED_ON = 1,
```

```
    LED_BLINK = 2
} led_mode_t;
```

## Message Format

**JSON Command:**

```
{
  "command": "LED",
  "parameters": [2, 500]
}
```

**UDP Message String:**

```
LED:2:500
```

**C Parsing Example:**

```
void parse_command(char* message) {
    char* token;
    char* cmd_name;
    int params[10];
    int param_count = 0;

    // Extract command name
    cmd_name = strtok(message, ":");

    // Extract parameters
    while ((token = strtok(NULL, ":")) != NULL && param_count < 10) {
        params[param_count++] = atoi(token);
    }

    // Process command
    if (strcmp(cmd_name, "LED") == 0 && param_count >= 2) {
        led_mode_t mode = (led_mode_t)params[0];
        uint16_t blink_rate = (uint16_t)params[1];
        execute_led_command(mode, blink_rate);
    }
}
```

## Best Practices for Embedded Integration

1. **Keep IDs Consistent:** Command names match on both sides
2. **Document Protocol:** Maintain shared protocol specification document
3. **Validate Parameters:** Check min/max ranges on device side too

4. **Error Responses:** Return clear error codes for invalid commands
5. **Version Control:** Track command dictionary changes with firmware versions

---

# Adding New Commands

## Step-by-Step Guide

### 1. Plan Your Command

- Determine command name and category
- List required parameters and types
- Define validation rules (min/max, options)
- Set sensible defaults

### 2. Edit Command Dictionary JSON

**Location:** `core/workers/[device]/[device]_commandDictionary.json`

**Add to commands object:**

```
{
  "commands": {
    ...existing commands...,
    "MY_COMMAND": {
      "name": "MY_COMMAND",
      "category": "Device Control",
      "description": "Description of what command does",
      "parameters": [
        {
          "name": "PARAM1",
          "type": "integer",
          "min": 0,
          "max": 100,
          "default": 50,
          "description": "First parameter description"
        }
      ]
    }
  }
}
```

### 3. Add Category (if new)

If using new category, add to categories array:

```
{
  "categories": [
    "Device Control",
    "Device Status",
```

```
    "Error Handling",
    "System Administration",
    "My New Category"  ← Add here
  ]
}
```

**4. Implement Handler**

**File:** `core/workers/[device]/[device]_handler.py`

Add handling for new command:

```
def handle_command(command, params):
    """Handle device commands."""
    if command == "MY_COMMAND":
        param1 = int(params[0])
        # Implement command logic
        result = execute_my_command(param1)
        return format_response(result)
    # ...existing command handlers...
```

**5. Test**

1. Restart application
2. Select device
3. Click "SELECT COMMAND" → navigate to category → select command
4. Verify parameters appear correctly
5. Fill parameters and send command
6. Verify device responds correctly

---

# Troubleshooting

## Issue: Command Not Appearing in Menu

**Check:**

1. ✓ Command added to `commands` object?
2. ✓ `category` field present and correct?
3. ✓ Category exists in `categories` array?
4. ✓ JSON syntax valid? (no trailing commas, quotes matched)
5. ✓ Application restarted after changes?

**Validate JSON:**

```
python -m json.tool capstanDrive_commandDictionary.json
```

### Issue: Parameters Not Displaying

**Check:**

1. ✓ `parameters` array present (even if empty)?
2. ✓ All required parameter fields present (name, type, description, default)?
3. ✓ Parameter type is supported (enum/boolean/integer/float/string)?
4. ✓ Conditional parameters: dependency exists and spelled correctly?

**Debug:** Add logging to `message_creator_panel.py`:

```
def _update_parameter_widgets(self):
    print(f"Command: {self._current_command}")
    print(f"Parameters: {self._current_command_params}")
```

### Issue: Validation Failing

**Check:**

1. ✓ `min` ≤ `default` ≤ `max` for numeric types?
2. ✓ `default` in `options` array for enum types?
3. ✓ `max_length` reasonable for string types?
4. ✓ No typos in field names?

**Test Validation:**

```
# In Python console
import json
with open('capstanDrive_commandDictionary.json') as f:
    data = json.load(f)

cmd = data['commands']['MY_COMMAND']
param = cmd['parameters'][0]

# Check default in range
assert param['min'] <= param['default'] <= param['max']
```

### Issue: Category Menu Structure Wrong

**Check:**

1. ✓ All commands have `category` field?
2. ✓ Category names match exactly (case-sensitive)?
3. ✓ No duplicate category names?
4. ✓ Categories array is proper JSON array `[...]` not object `{...}`?

---

## Related Documentation

- **Message Creator Panel:** See message_creator_panel.md for UI details
- **Command Menu System:** See command_menu_system.md for menu customization
- **Architecture:** See architecture.md for system design
- **Usage Guide:** See usage.md for user instructions

---

**Note:** This reflects v2.0 with hierarchical categories. For v1.0 flat dropdown format, see project history.