Message Creator Panel – Technical Reference

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

Table of Contents

Overview

The **Message Creator Panel** is a dynamic, adaptive UI component that enables users to construct and send device-specific UDP command messages. It provides an intuitive interface that automatically adjusts to the selected device's command dictionary.

Key Features

NEW in v2.0:

- **Hierarchical Command Menu:** Commands organized in flyout category menus (replaces flat dropdown)
- **10-Parameter Support:** Expanded from 8 to 10 parameters for complex commands
- **Conditional Display Logic:** Parameters shown/hidden based on dependencies
- **Real-Time Validation:** Instant feedback on parameter values
- **Automatic Adaptation:** UI updates dynamically based on command dictionary

Component Location

- **Position:** Center panel in top tier (Interactive Area)
- **Height:** 350px (reduced from 700px in v1.0)
- **Width:** Flexible, between Device Panel (200px) and Send Panel
- **File:** `app/ui/message_creator_panel.py`

Architecture

Component Lifecycle

1. **Initialization:**
    - Load command dictionary from device worker directory
    - Parse categories and commands from JSON
    - Build hierarchical menu structure
    - Create parameter input widgets (hidden initially)
2. **Command Selection:**
    - User clicks "SELECT COMMAND" button
    - Hierarchical menu displayed
    - User navigates category → command
    - Command selected, menu closes
3. **Parameter Configuration:**
    - Display relevant parameter widgets for selected command
    - Apply conditional visibility rules
    - Validate input in real-time
    - Assemble message string dynamically
4. **Message Transmission:**
    - User clicks "SEND MESSAGE" button (in Send Panel)
    - Validate all required parameters filled
    - Format message according to protocol
    - Emit signal to worker for transmission

Data Flow

```
Command Dictionary (JSON)
    ↓
MessageCreatorPanel.__init__()
    ↓
_load_command_dictionary()
    ↓
build_command_menu()  ← Creates hierarchical QMenu
    ↓
User Interaction (menu selection)
    ↓
on_command_selected(command)
    ↓
_update_parameter_widgets()  ← Show/hide widgets
    ↓
User fills parameters
    ↓
_validate_parameters()
    ↓
format_message()  ← Handler function
```

```
    ↓
Signal emitted to Worker
    ↓
UDP transmission
```

Hierarchical Command Menu

Menu Structure

Replaced: QComboBox dropdown (v1.0)
New: QPushButton + QMenu with categories and flyout submenus (v2.0)

Visual Hierarchy:

```
┌─ SELECT COMMAND (Button) ──────┐
│  When clicked, shows:          │
│                                │
│  ┌─ Device Control ────────────┐
│  │  → LED                     │
│  │  → HPL                     │
│  │  → STEPPER                 │
│  │  → ENCODER                 │
│  ├─ Device Status ─────────────┤
│  │  → GET_LED                 │
│  │  → GET_HPL                 │
│  │  → GET_ALL                 │
│  ├─ Error Handling ────────────┤
│  │  → CLR_WARN                │
│  │  → GET_ERROR_LOG           │
│  │  → CLEAR_DEBUG_LOG         │
│  ├─ System Administration ─────┤
│  │  → SET_TIME                │
│  │  → GET_TIME_STATUS         │
│  └────────────────────────────┘
└──────────────────────────────────┘
```

Implementation Details

Building the Menu:

```python
def build_command_menu(self):
    """
    Constructs hierarchical menu from command dictionary.

    Key steps:
    1. Extract unique categories from dictionary
    2. Create QAction for each category (acts as submenu)
    3. Populate each submenu with command items
```

```
    4. Connect each command to on_command_selected()
    """
    menu = QMenu(self)
    categories = self.command_dict.get("categories", [])

    for category in categories:
        # Create category submenu
        category_menu = menu.addMenu(category)

        # Add commands belonging to this category
        for cmd_name, cmd_data in self.command_dict.get("commands", {}).items():
            if cmd_data.get("category") == category:
                action = category_menu.addAction(cmd_name)
                action.triggered.connect(
                    lambda checked, c=cmd_name: self.on_command_selected(c)
                )

    self.command_button.setMenu(menu)
```

Command Selection Handler:

```
def on_command_selected(self, command):
    """
    Called when user selects a command from menu.

    Actions:
    1. Store selected command in _current_command
    2. Update button label to show command name
    3. Show/configure parameter widgets for this command
    4. Reset parameter values to defaults
    """
    self._current_command = command
    self.command_button.setText(command)
    self._update_parameter_widgets()
```

Customization

Add New Category:

1. Edit command dictionary JSON:

```
{
  "categories": [
    "Device Control",
    "Device Status",
    "Error Handling",
    "System Administration",
    "MY NEW CATEGORY"  ← Add here
```

```
    ]
  }
```

2. Assign commands to new category:

```
{
  "commands": {
    "MY_COMMAND": {
      "name": "MY_COMMAND",
      "category": "MY NEW CATEGORY",  ← Set category
      "parameters": [...]
    }
  }
}
```

3. Restart application → menu rebuilt automatically

See Also: command_menu_system.md for comprehensive menu customization guide.

Parameter System

Overview

Supports up to 10 parameters per command (expanded from 8 in v1.0).

Parameter Types

Enum (Dropdown)

Use Case: Fixed set of options (e.g., LED mode: OFF, ON, BLINK)

JSON Definition:

```
{
  "name": "LED_MODE",
  "type": "enum",
  "options": ["OFF", "ON", "BLINK"],
  "default": "OFF"
}
```

UI Widget: QComboBox with options ["OFF", "ON", "BLINK"]

Message Format: Index of selected option (0, 1, or 2)

Boolean (Dropdown)

Use Case: True/false choices (e.g., enable/disable feature)

JSON Definition:

```
{
  "name": "ENABLE_SAFETY",
  "type": "boolean",
  "default": true
}
```

UI Widget: QComboBox with options ["FALSE", "TRUE"]

Message Format: "0" for false, "1" for true

Integer (Text Input)

Use Case: Numeric values without decimals (e.g., step count, LED intensity)

JSON Definition:

```
{
  "name": "STEP_COUNT",
  "type": "integer",
  "min": 0,
  "max": 10000,
  "default": 1000
}
```

UI Widget: QLineEdit with integer validator

Validation: Min/max enforced, non-integers rejected

Message Format: String representation of integer (e.g., "1000")

Float (Text Input)

Use Case: Decimal values (e.g., temperature, voltage)

JSON Definition:

```
{
  "name": "VOLTAGE",
  "type": "float",
  "min": 0.0,
  "max": 5.0,
  "precision": 2,
  "default": 3.3
}
```

UI Widget: QLineEdit with double validator

Validation: Min/max enforced, precision checked

Message Format: String representation of float (e.g., "3.30")

String (Text Input)

Use Case: Alphanumeric data (e.g., device name, message text)

JSON Definition:

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

UI Widget: QLineEdit with character limit

Validation: Maximum length enforced

Message Format: Raw string (e.g., "Device001")

Conditional Parameters

Purpose: Show/hide parameters based on other parameter values.

Example: BLINK_RATE only shown when LED_MODE is "BLINK"

JSON Definition:

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

Implementation:

```
def _update_parameter_widgets(self):
    """
```

```
        Show/hide parameter widgets based on conditions.
        """
        for i, param in enumerate(self._current_command_params):
            widget = self.param_widgets[i]

            # Check if parameter has condition
            condition = param.get("condition")
            if condition:
                dep_param = condition["parameter"]
                required_value = condition["value"]

                # Get current value of dependency parameter
                dep_widget = self._find_widget_by_name(dep_param)
                current_value = self._get_widget_value(dep_widget)

                # Show widget only if condition met
                if current_value == required_value:
                    widget.show()
                else:
                    widget.hide()
            else:
                # No condition, always show
                widget.show()
```

Parameter Slots

Total Slots: 10 (indices 0-9)

Dynamic Allocation:

- Only slots needed for current command are shown
- Unused slots hidden to reduce visual clutter
- Slots reused when switching commands

Memory Efficiency:

- Widgets created once at initialization
- Show/hide instead of create/destroy
- Reduces memory allocation overhead

Message Assembly

Format Protocol

Standard Format:

```
COMMAND:param0:param1:param2:...:param9
```

Example:

```
LED:2:500      ← LED command, mode=2 (BLINK), rate=500ms
STEPPER:1000:1 ← STEPPER command, steps=1000, direction=1 (forward)
GET_ALL:       ← GET_ALL command, no parameters
```

Assembly Process

1. **Collect Parameters:**

```
params = []
for i in range(10):
    if self.param_widgets[i].isVisible():
        value = self._get_widget_value(self.param_widgets[i])
        params.append(value)
```

2. **Format Message:**

```
command = self._current_command
message = f"{command}:{':'.join(params)}"
```

3. **Validate:**

```
if not self._validate_parameters():
    # Show error, prevent sending
    return None
```

4. **Return for Transmission:**

```
return message
```

Handler Integration

Handler Function:

```
def format_message(command, params):
    """
    Device-specific message formatting.

    Override this function in device handler to customize
    message format, add checksums, encode binary data, etc.
    """
    # Default implementation
```

```
    param_str = ":".join(str(p) for p in params)
    return f"{command}:{param_str}"
```

Custom Handler Example:

```
def format_message(command, params):
    """
    Custom format with checksum.
    """
    param_str = ":".join(str(p) for p in params)
    message = f"{command}:{param_str}"

    # Add checksum
    checksum = calculate_checksum(message)
    return f"{message}:{checksum}"
```

UI Components

Command Button

Component: QPushButton with attached QMenu

Properties:

- **Text:** "SELECT COMMAND" or current command name
- **Menu:** Hierarchical command menu
- **Behavior:** Click opens menu at button position

Styling:

```
self.command_button.setStyleSheet("""
    QPushButton {
        text-align: left;
        padding: 5px;
    }
    QPushButton::menu-indicator {
        subcontrol-position: right center;
        subcontrol-origin: padding;
        left: 5px;
    }
""")
```

Parameter Widgets

Widget Types:

- **QComboBox:** Enum and boolean parameters

- **QLineEdit:** Integer, float, and string parameters

Layout:

- **Orientation:** Vertical (top to bottom)
- **Labels:** Parameter name above each widget
- **Spacing:** 5px between parameter groups

Example Layout:

```
┌─ LED_MODE ──────────┐
│ [BLINK ▼]           │ ← QComboBox
├─ BLINK_RATE ────────┤
│ [500    ]           │ ← QLineEdit (integer)
├─ ENABLE_SAFETY ─────┤
│ [TRUE ▼]            │ ← QComboBox (boolean)
└─────────────────────┘
```

Validation Feedback

Valid Input:

- Normal border color
- No error message
- Send button enabled

Invalid Input:

- Red border on invalid widget
- Tooltip shows error message (e.g., "Value must be between 0 and 1000")
- Send button disabled

Implementation:

```
def _validate_parameter(self, widget, param):
    """
    Validate single parameter widget.
    """
    value = self._get_widget_value(widget)

    # Type-specific validation
    if param["type"] == "integer":
        try:
            int_value = int(value)
            min_val = param.get("min", -2147483648)
            max_val = param.get("max", 2147483647)

            if not (min_val <= int_value <= max_val):
                self._show_error(widget, f"Must be between {min_val} and
{max_val}")
```

```
                return False
            except ValueError:
                self._show_error(widget, "Must be an integer")
                return False

        # If validation passed
        self._clear_error(widget)
        return True
```

Extending the System

Adding a New Device Class

Step 1: Create device directory structure:

```
core/workers/my_device/
    __init__.py
    my_device_config.py
    my_device_commandDictionary.json
    my_device_handler.py
    my_device_worker.py
```

Step 2: Define command dictionary (`my_device_commandDictionary.json`):

```
{
  "categories": [
    "Device Control",
    "Device Status"
  ],
  "commands": {
    "TURN_ON": {
      "name": "TURN_ON",
      "category": "Device Control",
      "description": "Turn device on",
      "parameters": []
    },
    "SET_MODE": {
      "name": "SET_MODE",
      "category": "Device Control",
      "description": "Set operating mode",
      "parameters": [
        {
          "name": "MODE",
          "type": "enum",
          "options": ["STANDBY", "ACTIVE", "SLEEP"],
          "default": "STANDBY"
        }
      ]
```

```
    }
  }
}
```

Step 3: Create handler (`my_device_handler.py`):

```python
def format_message(command, params):
    """Format message for my_device protocol."""
    return f"{command}:{':'.join(params)}"

def parse_response(response):
    """Parse device response."""
    parts = response.split(":")
    return {
        "command": parts[0],
        "status": parts[1] if len(parts) > 1 else "OK"
    }
```

Step 4: Register device in `data/servers.json`:

```
{
  "servers": [
    {
      "name": "My Device",
      "ip": "192.168.1.100",
      "port": 5000,
      "worker_type": "my_device"
    }
  ]
}
```

Step 5: Restart application → device appears in Device Panel with custom commands.

Adding a New Parameter Type

Example: Add "hex" parameter type for hexadecimal input.

Step 1: Define in command dictionary:

```
{
  "name": "ADDRESS",
  "type": "hex",
  "min": "0x0000",
  "max": "0xFFFF",
  "default": "0x1234"
}
```

Step 2: Add widget creation logic (`message_creator_panel.py`):

```
def _create_parameter_widget(self, param):
    """Create widget for parameter."""
    if param["type"] == "hex":
        widget = QLineEdit()
        widget.setPlaceholderText("0x0000")
        # Set validator for hex input
        hex_regex = QRegularExpression("0x[0-9A-Fa-f]{1,4}")
        validator = QRegularExpressionValidator(hex_regex, widget)
        widget.setValidator(validator)
        return widget
    # ... existing type handling ...
```

Step 3: Add validation logic:

```
def _validate_parameter(self, widget, param):
    """Validate parameter value."""
    if param["type"] == "hex":
        value = widget.text()
        try:
            int_value = int(value, 16)  # Convert hex to int
            min_val = int(param.get("min", "0x0"), 16)
            max_val = int(param.get("max", "0xFFFF"), 16)

            if not (min_val <= int_value <= max_val):
                self._show_error(widget, f"Must be between {param['min']} and
{param['max']}")
                return False
        except ValueError:
            self._show_error(widget, "Invalid hexadecimal value")
            return False
    # ... existing type handling ...
```

Best Practices

Command Dictionary Design

DO:

- Use clear, descriptive command names (e.g., "SET_LED_MODE" not "SLM")
- Group related commands in same category
- Provide comprehensive descriptions
- Define reasonable min/max ranges for numeric parameters
- Use consistent naming conventions across device classes

DON'T:

- Create deeply nested categories (max 2 levels: category → command)
- Use special characters in command names (stick to A-Z, 0-9, underscore)
- Define parameters without type information
- Leave default values blank (always provide sensible defaults)

Parameter Organization

Logical Ordering:

1. Primary parameters (required, most important)
2. Secondary parameters (optional, less common)
3. Conditional parameters (dependent on primary parameters)

Example: LED Command

```
{
  "parameters": [
    {"name": "LED_MODE", "type": "enum", ...},        ← Primary
    {"name": "BRIGHTNESS", "type": "integer", ...},   ← Secondary
    {"name": "BLINK_RATE", "condition": {...}, ...}   ← Conditional
  ]
}
```

UI Responsiveness

Keep UI Snappy:

- Avoid expensive operations in parameter change handlers
- Use debouncing for real-time validation (wait 300ms after last keystroke)
- Cache command dictionary in memory (don't re-parse JSON on every selection)
- Use signals/slots for asynchronous operations

Example: Debounced Validation

```
def _on_parameter_changed(self):
    """Called when parameter value changes."""
    # Cancel previous timer
    if hasattr(self, '_validation_timer'):
        self._validation_timer.stop()

    # Start new timer (300ms delay)
    self._validation_timer = QTimer()
    self._validation_timer.setSingleShot(True)
    self._validation_timer.timeout.connect(self._validate_all_parameters)
    self._validation_timer.start(300)
```

Testing New Commands

Test Checklist:

- [ ] Command appears in correct category
- [ ] All parameters displayed for command
- [ ] Parameter widgets match expected types
- [ ] Conditional parameters show/hide correctly
- [ ] Validation catches invalid inputs
- [ ] Message format matches device protocol
- [ ] Device responds correctly to command
- [ ] Response parsed and displayed properly

Troubleshooting

Issue: Command Not Appearing in Menu

Possible Causes:

1. Command not added to "commands" object in JSON
2. Category field missing or misspelled
3. Category not in "categories" array

Solutions:

1. Verify command exists in `commandDictionary.json`
2. Check `"category"` field matches exactly (case-sensitive)
3. Add category to `"categories"` array if new category

Issue: Parameter Widgets Not Showing

Possible Causes:

1. Parameters array empty or missing
2. Parameter type not recognized
3. All parameters hidden by conditions

Solutions:

1. Add parameters to command definition in JSON
2. Use supported types: enum, boolean, integer, float, string
3. Check conditional logic - are dependencies set correctly?

Issue: Validation Always Fails

Possible Causes:

1. Min/max values inverted (min > max)
2. Default value outside valid range
3. Widget value not being read correctly

Solutions:

1. Verify min <= default <= max in JSON
2. Check validator configuration matches parameter definition
3. Add debug logging: `print(f"Widget value: {widget.text()}")`

Issue: Message Not Sent

Possible Causes:

1. Required parameters not filled
2. Validation failing silently
3. Send button signal not connected

Solutions:

1. Check all visible parameter widgets have values
2. Add debug logging to validation functions
3. Verify signal connection: `self.send_button.clicked.connect(...)`

API Reference

Public Methods

`__init__(self, device_type, parent=None)`

Initialize message creator panel for specific device type.

Parameters:

- `device_type` (str): Device type name (e.g., "capstan_drive")
- `parent` (QWidget): Parent widget

`get_current_command(self) -> str`

Get currently selected command name.

Returns: Command name string or None if no command selected

`get_parameters(self) -> List[str]`

Get current parameter values for selected command.

Returns: List of parameter values as strings

`get_message(self) -> str`

Assemble and return complete message string.

Returns: Formatted message string ready for transmission

`reset(self)`

Reset panel to initial state (no command selected, all parameters cleared).

Signals

`command_selected(str)`

Emitted when user selects a command from menu.

Parameter: Command name

`parameters_changed()`

Emitted when any parameter value changes.

`message_ready(str)`

Emitted when valid message assembled and ready to send.

Parameter: Message string

Properties

`current_command` **(str, read-only)**

Currently selected command name.

`parameter_count` **(int, read-only)**

Number of visible parameters for current command.

Related Documentation

- **Usage Guide:** See usage.md for user instructions
- **Command Menu System:** See command_menu_system.md for menu customization
- **Architecture:** See architecture.md for system design
- **Command Dictionary Tutorial:** See command_dictionary_tutorial.md for JSON format details

---

Note: This document reflects v2.0 features. For v1.0 documentation (flat dropdown, 8 parameters), see project history.