ChipCap Python - Developer Documentation

Version: 1.0

Date: January 24, 2026

Application: Chip Inspection System

Table of Contents

1. Project Overview

ChipCap Python is a PySide6-based desktop application for automated chip/device inspection. The system supports three inspection stations (FEED, TOP, BOTTOM) with configurable inspection parameters, teach mode for setting up reference data, and automated test mode for quality control.

Key Features

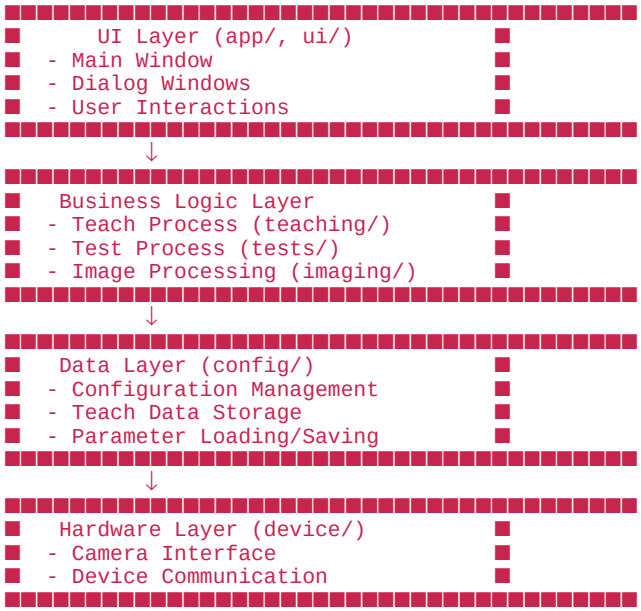
- **Multi-Station Support**: FEED, TOP, and BOTTOM stations with independent teach data
- **Teach Mode**: Interactive teaching system for setting up inspection parameters
- **Test Mode**: Automated inspection with pass/fail results
- **Step-by-Step Debug**: Optional step-by-step inspection for troubleshooting
- **Image Processing**: Real-time image capture, rotation, and binary threshold adjustment
- **License Management**: Built-in license verification system
- **Configurable Inspections**: Enable/disable individual inspection types per station

2. System Architecture

Technology Stack

- **GUI Framework**: PySide6 (Qt for Python)
- **Image Processing**: OpenCV (cv2)
- **Language**: Python 3.8+
- **Data Storage**: JSON files for configuration and teach data

Application Layers



3. Folder Structure

Root Directory

```
chip-chap-python/
■■■ main.py                      # Application entry point
■■■ requirements.txt             # Python dependencies
■■■ README.md                   # Basic project information
■■■ [Configuration JSON files]  # Runtime configuration
```

`/app` - Main Application

Purpose: Contains the main application window and core UI logic

Files:

- `main_window.py` - **MainWindow class**
- Central application controller (2100+ lines)
- Manages all UI components
- Handles teach/test workflows
- Coordinates between different modules
- Key sections:
- Menu bar creation
- Toolbar setup (GRAB, LIVE, TEACH, TEST, NEXT)
- Track/Station management
- Image display and zoom
- Binary mode controls
- Teach mode orchestration
- Test mode execution

Key Responsibilities:

- Application state management
- UI event handling
- Workflow coordination
- Station switching
- Mode transitions (ONLINE/OFFLINE)

`/ui` - Dialog Windows

Purpose: Reusable dialog windows for various settings and configurations

Files:

- `inspection_parameters_range_dialog.py` - Inspection parameters editor (1100+ lines)
- Tabbed interface for all inspection types
- Numeric field editors
- Checkbox controls for enabling/disabling inspections
- Station-specific parameter management

- `license_dialog.py` - License key entry and validation
- `lot_information_dialog.py` - Production lot information
- `device_location_dialog.py` - Device location settings
- `device_inspection_dialog.py` - Device inspection configuration
- `ignore_fail_count_dialog.py` - Fail count threshold settings
- `alert_messages_dialog.py` - Alert message configuration
- `autorun_setting_dialog.py` - Auto-run configuration
- `autorun_withdraw_setting_dialog.py` - Auto-withdraw settings
- `body_color_dialog.py` - Body color detection settings
- `mark_color_dialog.py` - Mark color settings
- `terminal_color_dialog.py` - Terminal color settings
- `para_mark_config_dialog.py` - Parameter mark configuration
- `pocket_location_dialog.py` - Pocket location settings
- `image_rotation_dialog.py` - Image rotation angle adjustment
- `inspection_debug_dialog.py` - Inspection debug settings
- `step_debug_dialog.py` - Step-by-step debug dialog
- `simulator_prompt.py` - Simulator mode prompt

Design Pattern: Each dialog is a standalone QDialog subclass with its own layout and logic

`/config` - Configuration Management

Purpose: Handles loading/saving of configuration data and inspection parameters

Files:

Parameter Management

- `inspection_parameters.py` - **InspectionParameters dataclass**
- Defines all inspection parameter fields
- Includes ranges (min/max values)
- Includes flags (enable/disable switches)
- Includes teach data (package/pocket ROI, rotation angle)

- `inspection_parameters_io.py` - Load/save inspection parameters
- `load_parameters()` - Load from inspection_parameters.json
- `save_parameters()` - Save to inspection_parameters.json

Teach Data Management

- `teach_store.py` - **Station-specific teach data storage**
- `save_teach_data(params_by_station)` - Saves teach data for all stations
- `load_teach_data()` - Loads teach data for all stations
- Storage format: `teach_data.json` with station keys (Feed, Top, Bottom)

Configuration Modules

- `store.py` - General application settings
- App data directory management
- Settings persistence
- Station-specific configurations

- `debug_flags.py` - Debug flag definitions
- `debug_flags_io.py` - Debug flags I/O operations

- `auto_run_setting.py` - Auto-run settings dataclass
- `auto_run_setting_io.py` - Auto-run settings I/O

- `device_location_setting.py` - Device location settings
- `device_location_setting_io.py` - Device location I/O

- `lot_information.py` - Lot information dataclass
- `lot_information_io.py` - Lot information I/O

- `alert_messages.py` - Alert messages dataclass
- `alert_messages_io.py` - Alert messages I/O

- `ignore_fail_count.py` - Ignore fail count settings
- `ignore_fail_count_io.py` - Ignore fail count I/O

Design Pattern:

- Dataclass definition file (`.py`)

- Corresponding I/O operations file (`_io.py`)
- JSON persistence for all configurations

`/imaging` - Image Processing

Purpose: Image capture, manipulation, and ROI management

Files:

- `grab_service.py` - **Camera/image capture service**
- `grab()` - Single image capture
- `toggle_live()` - Start/stop live video feed
- Camera device management
- Simulator mode support

- `image_loader.py` - **Image loading from disk**
- File browser integration
- Image format support (BMP, JPG, PNG)

- `pocket_teach_overlay.py` - **Interactive ROI overlay widget**
- Drawable red/green rectangles
- Draggable and resizable
- Handle-based resizing (4 corner handles)
- Used for:
- Pocket position teaching
- Package position teaching
- Rotation ROI selection
- Key methods:
- `paintEvent()` - Draw ROI with handles
- `mousePressEvent()` - Handle drag start
- `mouseMoveEvent()` - Handle dragging/resizing
- `mouseReleaseEvent()` - Handle drop
- `confirm()` - Finalize ROI selection
- `set_confirmed()` - Change color to green

- `roi.py` - **ROI data structure**
- Simple rectangle representation (x, y, w, h)

Key Concepts:

- ROI (Region of Interest): Rectangular area selected by user
- Overlay: Semi-transparent widget drawn over image
- Teach Overlay: Turns green when confirmed

`/device` - Hardware Interface

Purpose: Camera and device communication

Files:

- `camera_device.py` - Camera device abstraction
- Camera initialization
- Frame capture
- Device settings (exposure, gain, etc.)

Note: In simulator mode, no actual camera is used - images are loaded from disk instead

`/tests` - Testing System

Purpose: Automated inspection test execution

Files:

- `test_runner.py` - **Test execution framework**
- `TestResult` dataclass - Test result container
- `TestStatus` enum - PASS, FAIL, SKIP statuses

- `test_top_bottom.py` - **TOP/BOTTOM station tests**
- `test_top_bottom()` - Main test function for TOP/BOTTOM stations
- `test_feed()` - Main test function for FEED station
- Checks all enabled inspections in order:

1. Package Location
2. Dimension Measurements (Body Length, Width, etc.)
3. Terminal Inspections
4. Body Inspections
5. Body Smear
6. Body Edge
7. TQS Inspections (FEED only)

- Supports step-by-step mode with callbacks

- `measurements.py` - **Measurement algorithms**
- `measure_body_width()` - Measure chip body width
- `measure_body_length()` - Measure chip body length
- Uses OpenCV edge detection and contour analysis

- `test_draw.py` - **Result visualization**
- `draw_test_result()` - Draw PASS/FAIL overlay on image
- Color coding: Green (PASS), Red (FAIL), Yellow (SKIP)

- `measurement_draw.py` - **Measurement visualization**
- `draw_measurement_result()` - Draw measurement lines and values
- `add_status_text()` - Add status text to image

- `utils.py` - Test utility functions

Test Flow:

```
1. Validate teach data exists
2. Check which inspections are enabled (params.flags)
3. For each enabled inspection:
   a. Run measurement/detection algorithm
   b. Compare against thresholds
   c. If step mode: show dialog on fail
   d. Return FAIL if out of range
4. Return PASS if all enabled tests pass
```

`/license` - License Management

Purpose: License generation and validation

Files:

- `licensce_genrator.py` - License key generator
- Generate license keys
- Encryption/validation logic
- `manager.py` - License validation manager
- Check license validity
- License expiration handling

Usage: Run `python license/licensce_genrator.py` to generate license.json

`/resources` - UI Resources

Purpose: Stylesheets and UI assets

Files:

- `styles.qss` - Qt stylesheet (CSS-like styling for UI)
- Button styles
- Dialog styles
- Color schemes

`/teaching` - Teaching Module

Purpose: Teach mode logic (currently minimal, most logic is in main_window.py)

Note: Most teach logic is currently in `app/main_window.py`. This folder may be expanded in future versions.

`/inspection` - Inspection Module

Purpose: Reserved for future inspection algorithms

Note: Currently empty. Future home for advanced inspection algorithms.

4. Key Concepts

4.1 Stations

The system supports three inspection stations:

FEED Station:

- Purpose: Inspect pocket location in tape/reel
- Teaches: Package location, Pocket location
- Tests: Pocket-related inspections, TQS inspections, dimension measurements

TOP Station:

- Purpose: Inspect top side of chip
- Teaches: Package location, rotation angle
- Tests: Dimension measurements, terminal inspections, body inspections

BOTTOM Station:

- Purpose: Inspect bottom side of chip
- Teaches: Uses same teach data as TOP station
- Tests: Same inspections as TOP station

Implementation:

- `Station` enum in `app/main_window.py`
- Station selection via menu bar or track buttons
- Station-specific parameters stored in `inspection_parameters_by_station` dictionary

4.2 Run States

ONLINE Mode:

- Production mode
- Real camera capture
- Auto-run capabilities
- TEST button disabled

OFFLINE Mode:

- Setup/debug mode
- Manual image loading
- Teach mode available
- Test mode available

Implementation: `RunState` enum in `app/main_window.py`

4.3 Teach Mode

Interactive process to set up reference data for inspections.

Teach Phases (`TeachPhase` enum):

1. `NONE` - No teach active
2. `POCKET_DONE` - Pocket position learned (FEED only)
3. `ROTATION_ASK` - Ask if image is rotated
4. `ROTATION_ROI` - Select rotation reference ROI
5. `ROTATION_DONE` - Rotation angle set
6. `PACKAGE_ASK` - Ask to teach package
7. `PACKAGE_ROI` - Select package ROI
8. `PACKAGE_CONFIRM` - Confirm package location
9. `DONE` - Teach complete

Teach Workflow:

FEED Station:

```
1. Ask: Teach pocket position?
2. Draw red ROI on image (PocketTeachOverlay)
3. User adjusts ROI with mouse
4. Press Enter to confirm
5. ROI turns green
6. Pocket data saved to teach_data.json
7. Teach complete (overlay removed)
```

TOP/BOTTOM Stations:

```
1. Ask: Is image rotated?
   - If NO: Skip to step 5
   - If YES: Continue to step 2
2. Draw ROI for rotation reference
3. Open rotation dialog to set angle
4. Preview rotation live
5. Draw ROI for package location
6. Confirm package location
7. Package data saved to teach_data.json
8. Teach complete
```

Data Saved:

- FEED: `package_x, package_y, package_w, package_h, pocket_x, pocket_y, pocket_w, pocket_h`
- TOP: `package_x, package_y, package_w, package_h, rotation_angle`
- BOTTOM: Uses TOP station data

4.4 Test Mode

Automated inspection process that validates devices against taught parameters.

Test Workflow:

```
1. Load image
2. Get station-specific teach data
3. Check which inspections are enabled (params.flags)
4. For each enabled inspection:
   - Run measurement algorithm
   - Compare against thresholds
   - Record result
5. Return PASS/FAIL with overlay image
```

Threshold Sources (in priority order):

1. `device_inspection.json` - Device-specific thresholds
2. `inspection_parameters.json` - User-configured defaults
3. Hardcoded fallbacks

Step Mode:

• Optional step-by-step inspection
• Shows dialog after each inspection
• User can:
• Continue to next step
• Abort test
• Adjust parameters

Enable in: Production Menu → Inspection Debug → Enable Step Mode

4.5 Inspection Flags

All inspection types are controlled by boolean flags in `params.flags` dictionary.

Shared Across All Stations:

- `enable_package_location`
- `enable_pocket_location`
- `check_body_length`
- `check_body_width`
- `check_terminal_width`
- (40+ more inspection types)

Station-Specific Teach Data:

- Each station has its own `InspectionParameters` object
- Teach data (ROI coordinates) stored per station
- Flags are shared across all stations

Storage:

- Flags: `inspection_parameters.json`
- Teach Data: `teach_data.json` (per station)

5. Core Components

5.1 MainWindow Class

Location: `app/main_window.py`

Key Attributes:

```
# State
self.state                          # AppState (station, run_state, track)
self.teach_phase                    # Current teach phase
self.is_teach_mode                  # Teach mode active flag
self.step_mode_enabled              # Step-by-step debug flag

# Images
self.current_image                  # Current image being displayed
self.teach_overlay                  # PocketTeachOverlay widget

# Parameters
self.inspection_parameters_by_station  # Dict[Station, InspectionParameters]
self.shared_flags                   # Shared inspection flags (dict)

# UI Components
self.image_label                    # QLabel for image display
self.track_combo                    # Track selector (1-8)
self.binary_slider                  # Binary threshold slider
```

Key Methods:

Initialization:

- `__init__()` - Setup UI, load configs, initialize state
- `_build_menu_bar()` - Create menu bar
- `_build_main_toolbar()` - Create main toolbar (GRAB, LIVE, TEACH, TEST, NEXT)
- `_build_track_bar()` - Create track selector bar
- `_build_center_layout()` - Create image display area

Station Management:

- `_select_station(station)` - Switch to a station
- `_set_station(station)` - Apply station change
- `_sync_station_actions()` - Update UI to reflect current station
- `current_params()` - Get current station's parameters

Teach Mode:

- `_on_teach()` - Start teach process
- `_teach_feed_station()` - FEED teach workflow
- `_teach_top_bottom_station()` - TOP/BOTTOM teach workflow
- `_start_pocket_teach()` - Start pocket teaching
- `_confirm_pocket_teach(roi)` - Save pocket teach data
- `_start_package_teach()` - Start package teaching
- `_confirm_package_teach(roi)` - Save package teach data
- `_exit_teach_mode()` - Clean up and exit teach mode

Test Mode:

- `_on_test()` - Execute test for current station

- `_handle_test_step(step_data)` - Handle step-by-step dialog

Image Management:

- `_show_image(image)` - Display image in label
- `_apply_binary()` - Apply binary threshold to image
- `_map_label_roi_to_image(roi)` - Convert label coordinates to image coordinates

Event Handlers:

- `_on_image_clicked(event)` - Handle image click (binary mode UI)
- `_on_track_changed(index)` - Handle track selection change
- `_on_next()` - Handle NEXT button (advance teach workflow)

5.2 InspectionParameters Class

Location: `config/inspection_parameters.py`

Purpose: Central data structure for all inspection parameters

Structure:

```
@dataclass
class InspectionParameters:
    # Ranges (numeric thresholds)
    ranges: dict[str, int]          # Dynamic ranges dict

    # Flags (enable/disable inspections)
    flags: dict[str, bool]          # Shared across stations

    # Teach Data (station-specific)
    rotation_angle: float = 0.0
    package_x: int = 0              # Package ROI
    package_y: int = 0
    package_w: int = 0
    package_h: int = 0
    pocket_x: int = 0               # Pocket ROI (FEED only)
    pocket_y: int = 0
    pocket_w: int = 0
    pocket_h: int = 0

    # Status
    is_defined: bool = False        # True after teach
```

Usage:

```
# Get station parameters
params = self.inspection_parameters_by_station[Station.TOP]

# Check if inspection is enabled
if params.flags.get("check_body_length", False):
    # Run body length inspection

# Get threshold range
min_val = params.ranges.get("body_length_min", 0)
max_val = params.ranges.get("body_length_max", 255)
```

5.3 PocketTeachOverlay Class

Location: `imaging/pocket_teach_overlay.py`

Purpose: Interactive ROI selection widget

Key Methods:

- `paintEvent()` - Draw ROI rectangle and handles
- `mousePressEvent()` - Start drag or resize
- `mouseMoveEvent()` - Update position during drag
- `mouseReleaseEvent()` - Finish drag
- `confirm()` - Finalize ROI and call callback
- `set_confirmed(True)` - Change color to green

Visual States:

- Red rectangle: Being adjusted
- Green rectangle: Confirmed
- Handles: 4 corner squares for resizing

Usage:

```
# Create overlay
self.teach_overlay = PocketTeachOverlay(self.image_label, self)
self.teach_overlay.setGeometry(self.image_label.rect())
self.teach_overlay.show()
self.teach_overlay.setFocus()

# When user presses Enter, confirm() is called
# which triggers main_window._confirm_overlay(roi)
```

5.4 Test Functions

Location: `tests/test_top_bottom.py`

Functions:

- `test_top_bottom(image, params, step_mode, step_callback)` - TOP/BOTTOM test
- `test_feed(image, params, step_mode, step_callback)` - FEED test

Return: `TestResult` object with:

- `status`: TestStatus.PASS, FAIL, or SKIP
- `message`: Detailed result message
- `result_image`: Image with overlays drawn

Algorithm:

```
1. Validate teach data exists
2. Load device thresholds (device_inspection.json)
3. Build list of enabled inspections from params.flags
4. For each enabled inspection:
   a. Run measurement algorithm if available
   b. Get thresholds (device file > user params > defaults)
   c. Compare measured value to range
   d. If step_mode and fail: show dialog
   e. Return FAIL if out of range
5. Return PASS if all enabled tests pass
```

6. Configuration Files

All configuration files are stored as JSON in the root directory.

teach_data.json

Purpose: Stores station-specific teach data (ROI coordinates, rotation angle)

Format:

```
{
  "Feed": {
    "package_x": 73,
    "package_y": 70,
    "package_w": 185,
    "package_h": 107,
    "pocket_x": 169,
    "pocket_y": 104,
    "pocket_w": 523,
    "pocket_h": 239,
    "rotation_angle": 0.0,
    "is_defined": true,
    "ranges": { ... },
    "flags": { ... }
  },
  "Top": { ... },
  "Bottom": { ... }
}
```

Access:

```
from config.teach_store import load_teach_data, save_teach_data

# Load
data = load_teach_data()  # Returns dict[str, InspectionParameters]

# Save
save_teach_data(self.inspection_parameters_by_station)
```

inspection_parameters.json

Purpose: Stores shared inspection flags and default ranges

Format:

```
{
  "flags": {
    "enable_package_location": true,
    "enable_pocket_location": false,
    "check_body_length": true,
    "check_body_width": true,
    ...
  },
  "ranges": {
    "body_length_min": 0,
    "body_length_max": 255,
    ...
  }
}
```

Access:

```
from config.inspection_parameters_io import load_parameters, save_parameters

params = load_parameters()
params.flags["check_body_length"] = True
save_parameters(params)
```

device_inspection.json

Purpose: Device-specific inspection thresholds (highest priority)

Format:

```
{
  "UnitParameters": {
    "body_length_min": "100",
    "body_length_min2": "150",  # Actually max (legacy naming)
    "body_width_min": "80",
    "body_width_min2": "120",
    ...
  }
}
```

Usage: Test functions check this file first for thresholds

Other Configuration Files

- `camera_settings.json` - Camera exposure, gain, etc.
- `device_location_setting.json` - Device location parameters
- `lot_information.json` - Current lot information
- `alert_messages.json` - Alert message templates
- `auto_run_setting.json` - Auto-run configuration
- `ignore_fail_count.json` - Fail count thresholds
- `debug_flags.json` - Debug flags
- `license.json` - License key data

7. Data Flow

7.1 Teach Data Flow

```
User clicks TEACH button
        ↓
main_window._on_teach()
        ↓
Check station type (FEED vs TOP/BOTTOM)
        ↓
Start appropriate teach workflow
        ↓
Show PocketTeachOverlay
        ↓
User adjusts ROI and presses Enter
        ↓
PocketTeachOverlay.confirm()
        ↓
main_window._confirm_overlay(roi)
        ↓
Save to current_params() object
        ↓
save_teach_data(inspection_parameters_by_station)
        ↓
Write to teach_data.json
```

7.2 Test Data Flow

```
User clicks TEST button
        ↓
main_window._on_test()
        ↓
Get current station's params
        ↓
Call test_feed() or test_top_bottom()
        ↓
Load device_inspection.json thresholds
        ↓
Check params.flags for enabled inspections
        ↓
For each enabled inspection:
  - Run measurement algorithm
  - Compare to thresholds
  - If step_mode: show dialog on fail
        ↓
Return TestResult
        ↓
Display result image with overlays
        ↓
Show message dialog (PASS/FAIL)
```

7.3 Parameter Update Flow

```
User opens Inspection Parameters dialog
        ↓
InspectionParametersRangeDialog.__init__()
        ↓
Load current station's params
        ↓
Load shared flags
        ↓
Populate UI fields
        ↓
User modifies values
        ↓
User clicks OK/Apply
        ↓
_apply_to_model()
        ↓
Update params.ranges (station-specific)
        ↓
Update shared_flags (shared across stations)
        ↓
save_parameters(shared_model)  # Save flags
        ↓
save_teach_data(params_by_station)  # Save ranges
```

8. Testing System

8.1 Test Types

Dimension Measurements (Implemented):

- Body Length - Uses `measure_body_length()`
- Body Width - Uses `measure_body_width()`
- Terminal Width - Not implemented (stub)
- Terminal Length - Not implemented (stub)
- Term-Term Length - Not implemented (stub)

Terminal Inspections (Not implemented):

- Terminal Pogo
- Incomplete Termination 1 & 2
- Terminal to Body Gap
- Terminal Color
- Terminal Oxidation
- Inner/Outer Terminal Chipoff

Body Inspections (Not implemented):

- Body Stain 1 & 2
- Body Color
- Body to Term Width
- Body Width Diff
- Body Crack
- Low/High Contrast
- Black/White Defect

Body Smear (Not implemented):

- Body Smear 1, 2, 3
- Reverse Chip
- Smear White

Body Edge (Not implemented):

- Body Edge Black
- Body Edge White

TQS Inspections - FEED Only (Not implemented):

- Sealing Stain, Sealing Stain 2
- Sealing Shift
- Black to White Scar
- Hole Reference
- White to Black Scan
- Emboss Tape Pickup

8.2 Measurement Algorithms

Body Length (`measurements.py`):

```
    def measure_body_length(image, package_roi, body_contrast=75, debug=False):
        """
        Measures chip body length using edge detection

        Algorithm:
        1. Extract package ROI from image
        2. Convert to grayscale
        3. Apply binary threshold (body_contrast)
        4. Find contours
        5. Get largest contour (assume it's the body)
        6. Calculate bounding rectangle width
        7. Return width in pixels
        """
```

Body Width (`measurements.py`):

```
    def measure_body_width(image, package_roi, body_contrast=75, debug=False):
        """
        Measures chip body width using edge detection

        Algorithm:
        1. Extract package ROI from image
        2. Convert to grayscale
        3. Apply binary threshold (body_contrast)
        4. Find contours
        5. Get largest contour (assume it's the body)
        6. Calculate bounding rectangle height
        7. Return height in pixels
        """
```

8.3 Adding New Tests

To add a new inspection type:

1. Add flag to InspectionParameters (`config/inspection_parameters.py`):

```
@dataclass
class InspectionParameters:
    # ...
    check_new_inspection: bool = False  # Add new flag
```

2. Add to inspection_checks list (`tests/test_top_bottom.py`):

```
inspection_checks = [
    # ...
    ("New Inspection", params.flags.get("check_new_inspection", False),
     "check_new_inspection", "measure_new_feature"),  # Add entry
]
```

3. Implement measurement function (`tests/measurements.py`):

```
def measure_new_feature(image, package_roi, **kwargs):
    """
    Your measurement algorithm here

    Returns:
        int or None: Measured value in pixels, or None if detection fails
    """
    # Your implementation
    return measured_value
```

4. Add measurement handling (`tests/test_top_bottom.py`):

```
for test_name, attr_name, measurement_func in enabled_tests:
    if measurement_func == "measure_new_feature":
        value = measure_new_feature(
            image,
            (params.package_x, params.package_y, params.package_w, params.package_h)
        )
        metric_key_min = "new_feature_min"
        metric_key_max_from_dev = "new_feature_min2"
        fallback_min = params.ranges.get("new_feature_min", 0)
        fallback_max = params.ranges.get("new_feature_max", 255)
```

5. Add UI controls (`ui/inspection_parameters_range_dialog.py`):

```
# Add checkbox
checkbox = QCheckBox("New Inspection")
self._inspection_checkboxes["check_new_inspection"] = checkbox

# Add range fields if needed
min_field = QLineEdit()
max_field = QLineEdit()
self._numeric_fields["new_feature_min"] = min_field
self._numeric_fields["new_feature_max"] = max_field
```

6. Update device_inspection.json (optional):

```
{
  "UnitParameters": {
    "new_feature_min": "100",
    "new_feature_min2": "200"
  }
}
```

9. Adding New Features

9.1 Adding a New Station

1. Add to Station enum (`app/main_window.py`):

```
class Station(str, Enum):
    FEED = "Feed"
    TOP = "Top"
    BOTTOM = "Bottom"
    NEW = "New"  # Add new station
```

2. Initialize station parameters (`app/main_window.py`):

```
self.inspection_parameters_by_station = {
    Station.FEED: InspectionParameters(),
    Station.TOP: InspectionParameters(),
    Station.BOTTOM: InspectionParameters(),
    Station.NEW: InspectionParameters(),  # Add entry
}
```

3. Add menu item (`app/main_window.py` in `_build_menu_bar()`):

```
self.act_station_new = QAction("     New", self, checkable=True)
self.act_station_new.triggered.connect(lambda: self._select_station(Station.NEW))
station_group.addAction(self.act_station_new)
m_station.addAction(self.act_station_new)
```

4. Update teach logic (`app/main_window.py` in `_on_teach()`):

```
if self.state.station == Station.NEW:
    self._teach_new_station()
```

5. Implement teach workflow:

```
def _teach_new_station(self):
    # Your teach workflow for new station
    pass
```

6. Create test function (`tests/test_top_bottom.py`):

```
def test_new(image, params, step_mode=False, step_callback=None):
    # Your test logic
    pass
```

7. Update test routing (`app/main_window.py` in `_on_test()`):

```
if station == Station.NEW:
    result = test_new(
        image=self.current_image.copy(),
        params=params
    )
```

9.2 Adding a New Dialog

1. Create dialog file (`ui/my_new_dialog.py`):

```
from PySide6.QtWidgets import QDialog, QVBoxLayout, QPushButton

class MyNewDialog(QDialog):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.setWindowTitle("My New Dialog")
        self.resize(400, 300)

        layout = QVBoxLayout()

        # Add your widgets here
        btn = QPushButton("OK")
        btn.clicked.connect(self.accept)
        layout.addWidget(btn)

        self.setLayout(layout)
```

2. Import in main_window.py:

```
from ui.my_new_dialog import MyNewDialog
```

3. Add menu item:

```
act_new = QAction("My New Dialog", self)
act_new.triggered.connect(self._open_my_new_dialog)
m_settings.addAction(act_new)
```

4. Implement handler:

```
def _open_my_new_dialog(self):
    dlg = MyNewDialog(self)
    if dlg.exec() == QDialog.Accepted:
        # Handle dialog result
        pass
```

9.3 Adding a New Configuration File

1. Create dataclass (`config/my_config.py`):

```
from dataclasses import dataclass

@dataclass
class MyConfig:
    setting1: str = "default"
    setting2: int = 0
    setting3: bool = False
```

2. Create I/O module (`config/my_config_io.py`):

```
import json
from pathlib import Path
from config.my_config import MyConfig

CONFIG_FILE = Path("my_config.json")

def load_my_config() -> MyConfig:
    if not CONFIG_FILE.exists():
        return MyConfig()

    with open(CONFIG_FILE, "r") as f:
        data = json.load(f)

    return MyConfig(**data)

def save_my_config(config: MyConfig):
    with open(CONFIG_FILE, "w") as f:
        json.dump(config.__dict__, f, indent=2)
```

3. Use in application:

```
from config.my_config_io import load_my_config, save_my_config

# Load
config = load_my_config()

# Modify
config.setting1 = "new value"

# Save
save_my_config(config)
```

10. Troubleshooting

Common Issues

Issue: Teach data not saving

Solution:

- Check that `save_teach_data()` is called after ROI confirmation
- Verify `inspection_parameters_by_station` dictionary is updated
- Check write permissions on teach_data.json

Issue: Test fails with "Package not taught"

Solution:

- Run Teach mode for the station first
- Check that package_w and package_h are > 0
- Verify teach_data.json contains station data

Issue: Inspection not running even though enabled

Solution:

- Check params.flags dictionary contains the flag
- Verify flag name matches between UI and test code
- Ensure measurement function is implemented

Issue: Binary mode enables automatically

Solution:

- Fixed in latest version
- Binary mode now only enables from menu bar
- Check `_on_image_clicked()` doesn't set `self.binary_mode = True`

Issue: BOTTOM station has different teach data than TOP

Solution:

- BOTTOM should use TOP's teach data
- Check `_on_test()` maps BOTTOM → TOP station
- Verify `test_station = Station.TOP if station == Station.BOTTOM else station`

Debug Tips

Enable Debug Output:

```
# In test functions, set debug=True
value = measure_body_width(image, package_roi, debug=True)
```

Step-by-Step Mode:

- Production Menu → Inspection Debug → Enable Step Mode
- Shows dialog after each inspection
- Can inspect intermediate results

Check Console Output:

- All test functions print detailed logs
- Look for `[TEST]`, `[INFO]`, `[FAIL]`, `[PASS]` prefixes
- Logs show which inspections are enabled/skipped

Inspect Configuration Files:

- Open teach_data.json to see taught ROIs
- Check inspection_parameters.json for flags
- Verify device_inspection.json for thresholds

Appendix A: File Reference

Critical Files (Must understand)

1. `app/main_window.py` - Application core (2100+ lines)
2. `config/inspection_parameters.py` - Parameter definitions
3. `config/teach_store.py` - Teach data persistence
4. `tests/test_top_bottom.py` - Test execution
5. `imaging/pocket_teach_overlay.py` - ROI selection widget

Configuration Files (JSON)

1. `teach_data.json` - Station teach data (ROIs, angles)
2. `inspection_parameters.json` - Shared flags and defaults
3. `device_inspection.json` - Device-specific thresholds
4. `camera_settings.json` - Camera configuration
5. `license.json` - License key data

UI Dialogs (Optional to understand)

- `ui/inspection_parameters_range_dialog.py` - Parameter editor
- `ui/image_rotation_dialog.py` - Rotation adjustment
- `ui/step_debug_dialog.py` - Step-by-step debug
- (Many other dialogs for various settings)

Appendix B: Development Workflow

Typical Development Session

1. **Setup**:

```
python license/licensce_genrator.py  # Generate license if needed
python main.py  # Launch application
```

2. **Testing Changes**:

• Make code changes
• Restart application
• Switch to OFFLINE mode
• Load test image
• Run TEACH to set reference
• Run TEST to verify changes

3. **Adding Inspection**:

• Add flag to InspectionParameters
• Implement measurement function
• Add to test function's inspection_checks
• Add UI controls in dialog
• Test with sample image

4. **Debugging**:

• Enable step mode for detailed inspection
• Check console output for [TEST] logs
• Inspect JSON files for configuration issues
• Use debug=True in measurement functions

Appendix C: Code Style Guidelines

Naming Conventions

- **Classes**: PascalCase (`MainWindow`, `TestResult`)
- **Functions**: snake_case (`measure_body_width`, `save_teach_data`)
- **Private Methods**: `_prefix` (`_on_teach`, `_apply_binary`)
- **Constants**: UPPER_SNAKE_CASE (`TEACH_FILE`, `CONFIG_FILE`)

UI Method Prefixes

- `_build_*` - Construct UI components
- `_on_*` - Event handlers
- `_open_*` - Open dialogs
- `_start_*` - Begin workflows
- `_confirm_*` - Confirm actions
- `_apply_*` - Apply settings

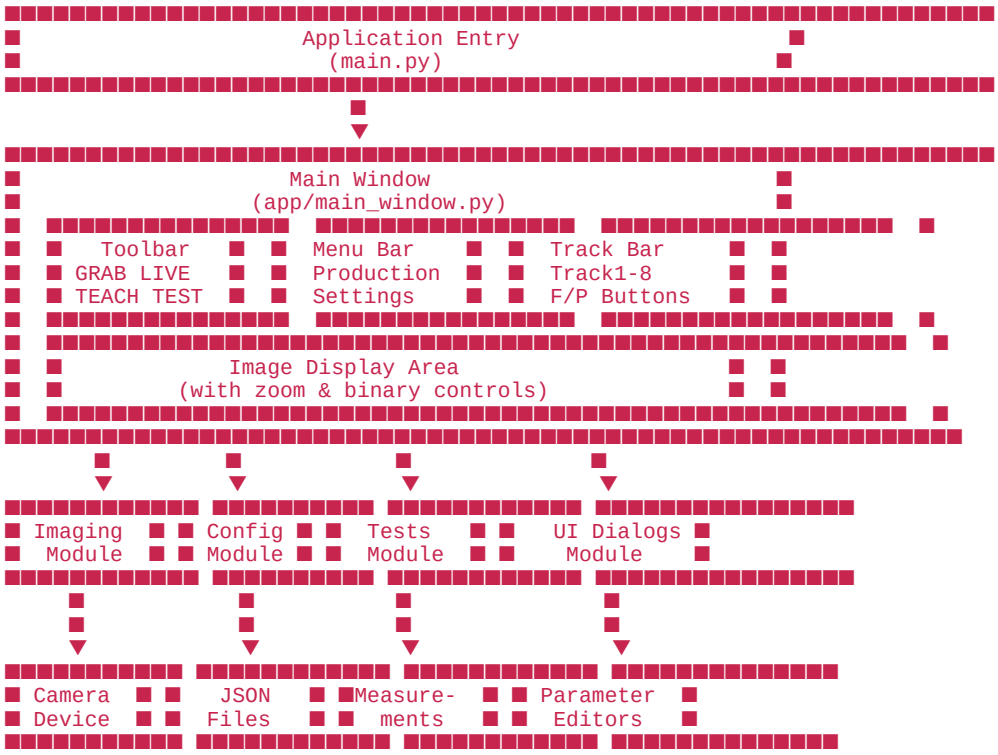
Comment Style

```
# Section headers
# ================================================
# SECTION NAME
# ================================================

# Subsections
# ------------------------------
# Subsection Name
# ------------------------------

# Important notes
# ■ Loop indicator
# ■ Error condition
# ■ Success condition
# ■ Prevention/blocking
```

Appendix D: Architecture Diagram



Version History

v1.0 (January 24, 2026)

- Initial documentation release
- Covers all major components
- Complete API reference
- Development guidelines

Document maintained by: Development Team

Last updated: January 24, 2026

For questions: Contact project maintainer