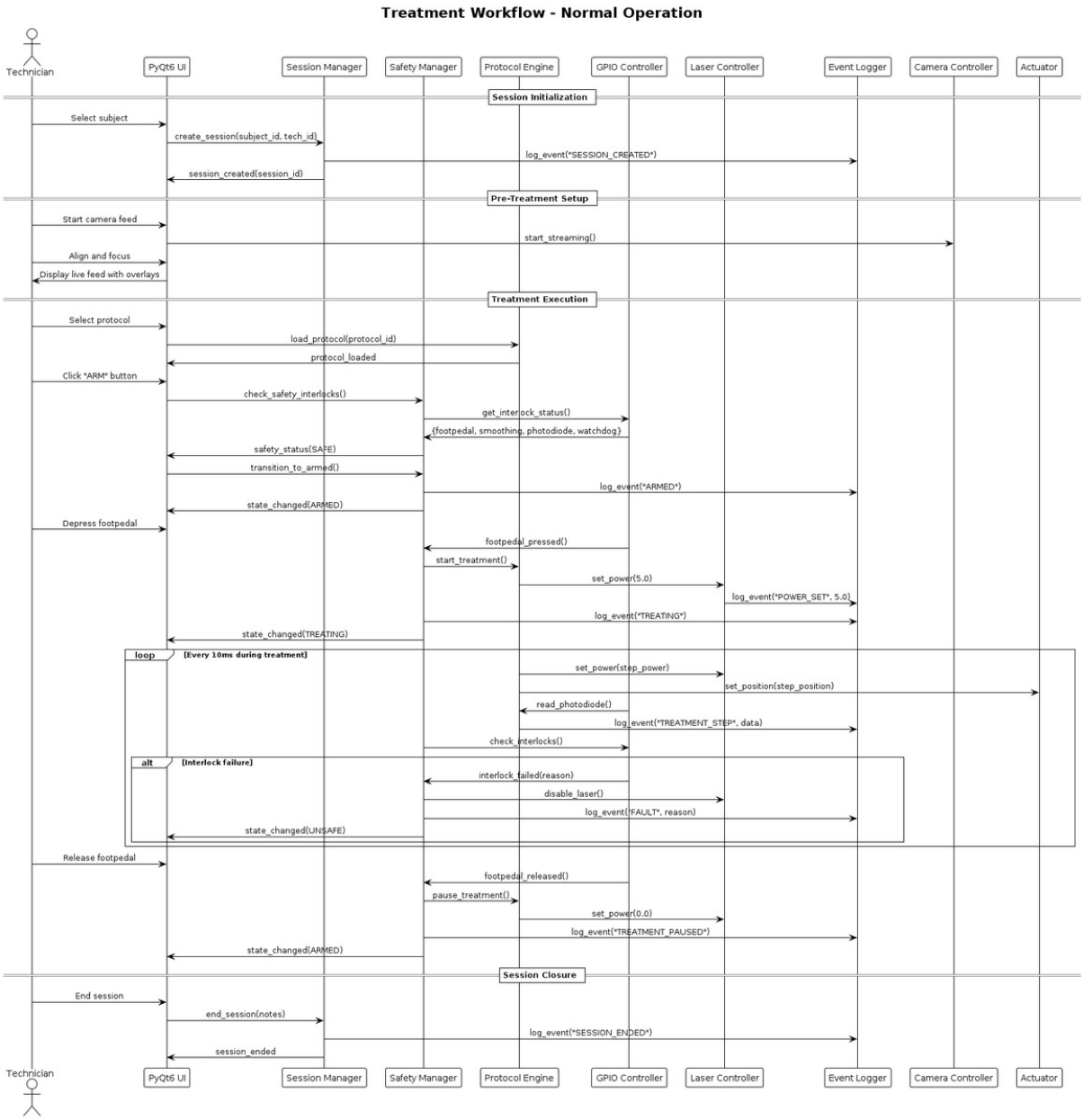
TOSCA Laser Control System - Treatment Protocol Engine

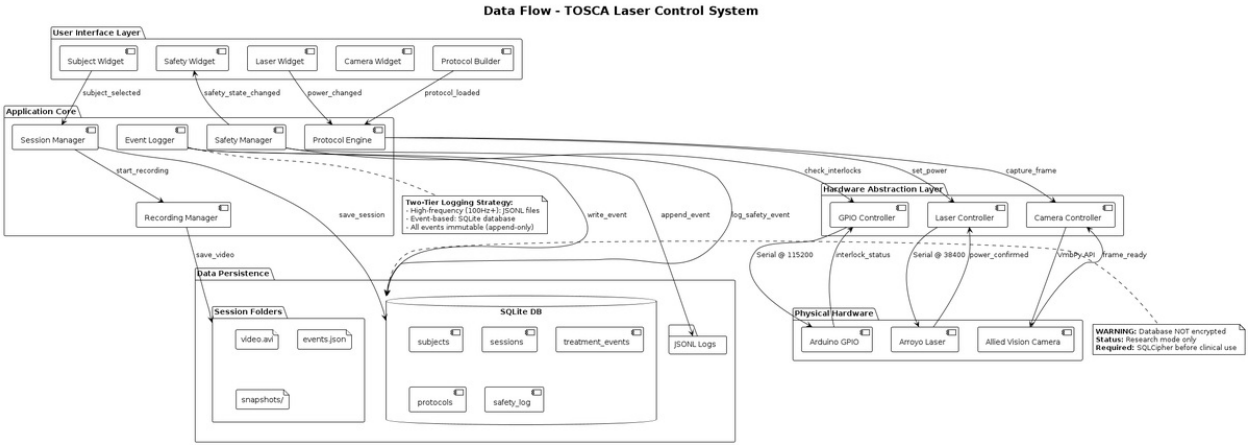
Architecture Diagrams

Figure 1: TOSCA Treatment Workflow Sequence



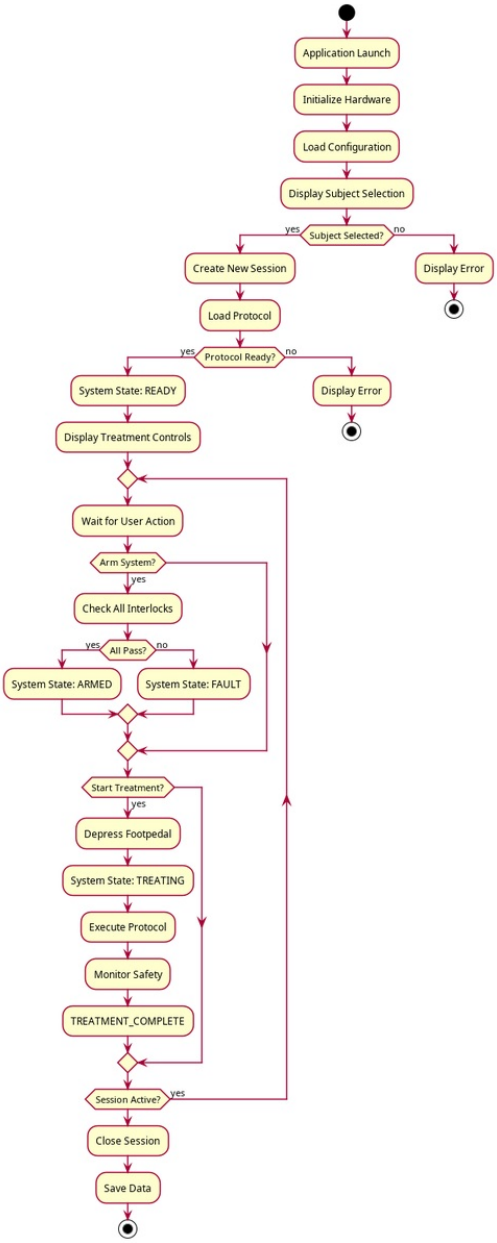
TOSCA Treatment Workflow Sequence

Figure 2: TOSCA Data Flow Diagram



TOSCA Data Flow Diagram

Figure 3: session-workflow



session-workflow

WARNING: **DEPRECATED DOCUMENT**

This document describes an older step-based protocol model that has been superseded.

Current Design: See `06_protocol_builder.md` for the action-based protocol engine (current implementation).

Status: Kept for historical reference only. Do not implement new features based on this document.

Last Updated: 2025-10-26 (marked as deprecated)

Document Version: 1.0 **Date:** 2025-10-15

Overview

The Treatment Protocol Engine executes pre-defined or custom treatment plans, controlling laser power and ring size over time. It provides:

- Pre-configured protocol templates
- Real-time protocol execution with precise timing
- Power ramping (constant, linear, custom curves)
- Ring size control via actuator positioning
- In-treatment protocol adjustments
- Complete execution logging

Protocol Data Model

Protocol JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "required": ["protocol_name", "version", "steps"],
  "properties": {
    "protocol_name": {
      "type": "string",
      "description": "Unique protocol identifier"
    },
    "version": {
      "type": "string",
      "description": "Protocol version (semantic versioning)"
    },
    "description": {
      "type": "string"
    },
    "created_date": {
      "type": "string",
      "format": "date-time"
    },
    "safety_limits": {
      "type": "object",
      "properties": {
        "max_power_watts": {"type": "number"},
        "max_duration_seconds": {"type": "number"},
        "min_ring_size_mm": {"type": "number"},
        "max_ring_size_mm": {"type": "number"}
      }
    },
    "steps": {
      "type": "array",
      "minItems": 1,
      "items": {
        "type": "object",
        "required": ["step_number", "duration_seconds", "power_start_watts", "power_end_watts", "ring_size_mm"],
        "properties": {
          "step_number": {
            "type": "integer",
            "minimum": 1
          },
          "duration_seconds": {
            "type": "number",
            "minimum": 0
          },
          "power_start_watts": {
            "type": "number",
            "minimum": 0
          },
          "power_end_watts": {
            "type": "number",
            "minimum": 0
          },
          "ring_size_mm": {
            "type": "number",
            "minimum": 0
          },
          "ramp_type": {
            "type": "string",
            "enum": ["constant", "linear", "logarithmic", "exponential"]
          },
          "notes": {
            "type": "string"
          }
        }
      }
    }
  }
}
```

Example Protocols

1. Constant Power Treatment

```
{
  "protocol_name": "Standard_5W_60s",
  "version": "1.0.0",
  "description": "5 watts constant power for 60 seconds at 3mm ring",
  "created_date": "2025-10-15T10:00:00Z",
```

```
  "safety_limits": {
    "max_power_watts": 6.0,
    "max_duration_seconds": 90,
    "min_ring_size_mm": 2.5,
    "max_ring_size_mm": 4.0
  },
  "steps": [
    {
      "step_number": 1,
      "duration_seconds": 60,
      "power_start_watts": 5.0,
      "power_end_watts": 5.0,
      "ring_size_mm": 3.0,
      "ramp_type": "constant",
      "notes": "Maintain steady 5W throughout"
    }
  ]
}
```

2. Linear Power Ramp

```
{
  "protocol_name": "Ramp_2to8W_90s",
  "version": "1.0.0",
  "description": "Linear ramp from 2W to 8W over 90 seconds",
  "created_date": "2025-10-15T10:00:00Z",
  "safety_limits": {
    "max_power_watts": 10.0,
    "max_duration_seconds": 120
  },
  "steps": [
    {
      "step_number": 1,
      "duration_seconds": 90,
      "power_start_watts": 2.0,
      "power_end_watts": 8.0,
      "ring_size_mm": 3.5,
      "ramp_type": "linear",
      "notes": "Smooth linear increase"
    }
  ]
}
```

3. Multi-Step Treatment

```
{
  "protocol_name": "MultiStep_Therapeutic",
  "version": "1.0.0",
  "description": "Warm-up, treatment, and cool-down phases",
  "created_date": "2025-10-15T10:00:00Z",
  "safety_limits": {
    "max_power_watts": 10.0,
    "max_duration_seconds": 300
  },
  "steps": [
    {
      "step_number": 1,
      "duration_seconds": 30,
      "power_start_watts": 1.0,
      "power_end_watts": 3.0,
      "ring_size_mm": 4.0,
      "ramp_type": "linear",
      "notes": "Warm-up phase - gradual power increase"
    },
    {
      "step_number": 2,
      "duration_seconds": 90,
      "power_start_watts": 5.0,
      "power_end_watts": 5.0,
      "ring_size_mm": 3.0,
      "ramp_type": "constant",
      "notes": "Treatment phase - steady 5W"
    },
    {
      "step_number": 3,
      "duration_seconds": 60,
      "power_start_watts": 5.0,
      "power_end_watts": 8.0,
      "ring_size_mm": 3.0,
      "ramp_type": "linear",
      "notes": "Intensification phase"
    },
    {
      "step_number": 4,
      "duration_seconds": 30,
      "power_start_watts": 8.0,
      "power_end_watts": 2.0,
      "ring_size_mm": 3.5,
      "ramp_type": "linear",
      "notes": "Cool-down phase"
    }
  ]
}
```

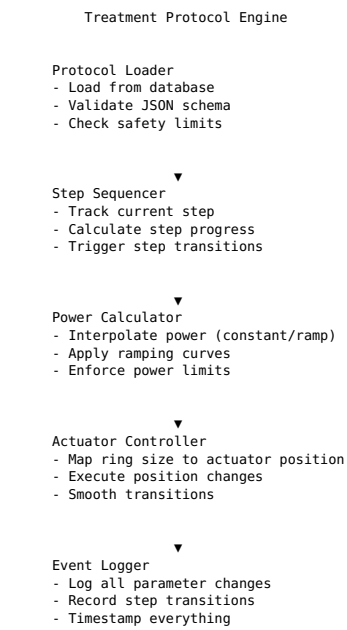
4. Variable Ring Size

```
{
  "protocol_name": "Variable_Ring_Treatment",
  "version": "1.0.0",
  "description": "Treatment with changing ring size",
  "created_date": "2025-10-15T10:00:00Z",
  "safety_limits": {
    "max_power_watts": 7.0,
    "max_duration_seconds": 180
  },
  "steps": [
    {
      "step_number": 1,
```

```
      "duration_seconds": 60,
      "power_start_watts": 5.0,
      "power_end_watts": 5.0,
      "ring_size_mm": 2.5,
      "ramp_type": "constant",
      "notes": "Small ring - concentrated treatment"
    },
    {
      "step_number": 2,
      "duration_seconds": 60,
      "power_start_watts": 5.0,
      "power_end_watts": 5.0,
      "ring_size_mm": 3.5,
      "ramp_type": "constant",
      "notes": "Medium ring"
    },
    {
      "step_number": 3,
      "duration_seconds": 60,
      "power_start_watts": 5.0,
      "power_end_watts": 5.0,
      "ring_size_mm": 4.5,
      "ramp_type": "constant",
      "notes": "Large ring - wide coverage"
    }
  ]
}
```

Protocol Execution Engine

Engine Architecture



Core Implementation

```python
from dataclasses import dataclass
from enum import Enum
import time
import json

class RampType(Enum):
    CONSTANT = "constant"
    LINEAR = "linear"
    LOGARITHMIC = "logarithmic"
    EXPONENTIAL = "exponential"

@dataclass
class ProtocolStep:
    """Single step in treatment protocol"""
    step_number: int
    duration_seconds: float
    power_start_watts: float
    power_end_watts: float
    ring_size_mm: float
    ramp_type: RampType
    notes: str = ""

    def calculate_power_at_time(self, elapsed_seconds: float) -> float:
        """
        Calculate power at given time within this step

        Args:
            elapsed_seconds: Time elapsed since step start

        Returns:
            Power in watts
        """
        if elapsed_seconds >= self.duration_seconds:
            return self.power_end_watts
```

```
        # Progress through step (0.0 to 1.0)
        progress = elapsed_seconds / self.duration_seconds
        power_delta = self.power_end_watts - self.power_start_watts

        if self.ramp_type == RampType.CONSTANT:
            # No ramping - use start power throughout
            return self.power_start_watts

        elif self.ramp_type == RampType.LINEAR:
            # Linear interpolation
            return self.power_start_watts + (progress * power_delta)

        elif self.ramp_type == RampType.LOGARITHMIC:
            # Logarithmic curve (fast increase, then slow)
            import math
            # Map [0,1] to [1, e] using exp, then normalize
            log_progress = (math.exp(progress) - 1) / (math.e - 1)
            return self.power_start_watts + (log_progress * power_delta)

        elif self.ramp_type == RampType.EXPONENTIAL:
            # Exponential curve (slow increase, then fast)
            exp_progress = progress ** 2
            return self.power_start_watts + (exp_progress * power_delta)

        else:
            return self.power_start_watts


class TreatmentProtocol:
    """Complete treatment protocol"""

    def __init__(self, protocol_data: dict):
        self.protocol_name = protocol_data['protocol_name']
        self.version = protocol_data.get('version', '1.0.0')
        self.description = protocol_data.get('description', '')
        self.safety_limits = protocol_data.get('safety_limits', {})

        # Parse steps
        self.steps = []
        for step_data in protocol_data['steps']:
            step = ProtocolStep(
                step_number=step_data['step_number'],
                duration_seconds=step_data['duration_seconds'],
                power_start_watts=step_data['power_start_watts'],
                power_end_watts=step_data['power_end_watts'],
                ring_size_mm=step_data['ring_size_mm'],
                ramp_type=RampType(step_data.get('ramp_type', 'constant')),
                notes=step_data.get('notes', '')
            )
            self.steps.append(step)

        # Calculate total duration
        self.total_duration_seconds = sum(s.duration_seconds for s in self.steps)

    def validate(self) -> tuple[bool, str]:
        """Validate protocol constraints"""

        # Check steps are sequential
        for i, step in enumerate(self.steps, 1):
            if step.step_number != i:
                return (False, f"Step numbering error: expected {i}, got {step.step_number}")

        # Check safety limits
        max_power = max(max(s.power_start_watts, s.power_end_watts) for s in self.steps)
        if 'max_power_watts' in self.safety_limits:
            if max_power > self.safety_limits['max_power_watts']:
                return (False, f"Protocol exceeds max power: {max_power}W > {self.safety_limits['max_power_watts']}W")

        if 'max_duration_seconds' in self.safety_limits:
            if self.total_duration_seconds > self.safety_limits['max_duration_seconds']:
                return (False, f"Protocol exceeds max duration: {self.total_duration_seconds}s > {self.safety_limits['max_duration_seconds']}s")

        return (True, "Protocol valid")


class ProtocolEngine:
    """
    Executes treatment protocols with real-time control
    """

    def __init__(self, hardware_manager, safety_manager, session_manager):
        self.hardware = hardware_manager
        self.safety = safety_manager
        self.session = session_manager

        self.protocol: TreatmentProtocol = None
        self.current_step_index = 0
        self.step_start_time = None
        self.treatment_start_time = None

        self.is_running = False
        self.is_paused = False

        self.update_interval_seconds = 0.1  # 10Hz update rate

    def load_protocol(self, protocol_data: dict) -> tuple[bool, str]:
        """Load and validate protocol"""
        try:
            self.protocol = TreatmentProtocol(protocol_data)
            is_valid, message = self.protocol.validate()

            if not is_valid:
                return (False, f"Protocol validation failed: {message}")

            return (True, f"Protocol '{self.protocol.protocol_name}' loaded successfully")

        except Exception as e:
            return (False, f"Protocol load error: {str(e)}")
```

```
def start(self) -> tuple[bool, str]:
    """Start protocol execution"""

    if not self.protocol:
        return (False, "No protocol loaded")

    # Safety pre-checks
    if not self.session.has_active_session():
        return (False, "No active session")

    # Reset state
    self.current_step_index = 0
    self.is_running = True
    self.is_paused = False
    self.treatment_start_time = time.time()
    self.step_start_time = time.time()

    # Log protocol start
    self._log_event('protocol_start', {
        'protocol_name': self.protocol.protocol_name,
        'total_steps': len(self.protocol.steps),
        'total_duration': self.protocol.total_duration_seconds
    })

    return (True, "Protocol started")

def update(self):
    """
    Main control loop - call this at regular intervals (e.g., 10Hz)

    Returns:
        (commanded_power, ring_size, status)
    """
    if not self.is_running or self.is_paused:
        return (0.0, 0.0, "Not running")

    # Get current step
    if self.current_step_index >= len(self.protocol.steps):
        # Protocol complete
        self.stop("Protocol completed")
        return (0.0, 0.0, "Complete")

    current_step = self.protocol.steps[self.current_step_index]

    # Calculate time within current step
    step_elapsed = time.time() - self.step_start_time

    # Check if step is complete
    if step_elapsed >= current_step.duration_seconds:
        self._advance_to_next_step()
        return self.update()  # Recurse to get next step values

    # Calculate current power
    commanded_power = current_step.calculate_power_at_time(step_elapsed)

    # Get ring size (constant per step)
    ring_size = current_step.ring_size_mm

    # Calculate overall progress
    treatment_elapsed = time.time() - self.treatment_start_time
    progress_percent = (treatment_elapsed / self.protocol.total_duration_seconds) * 100

    status = (f"Step {current_step.step_number}/{len(self.protocol.steps)} - "
              f"{progress_percent:.1f}% complete")

    return (commanded_power, ring_size, status)

def _advance_to_next_step(self):
    """Transition to next protocol step"""
    old_step = self.protocol.steps[self.current_step_index]

    self.current_step_index += 1

    if self.current_step_index < len(self.protocol.steps):
        new_step = self.protocol.steps[self.current_step_index]
        self.step_start_time = time.time()

        # Log step transition
        self._log_event('protocol_step_change', {
            'from_step': old_step.step_number,
            'to_step': new_step.step_number,
            'new_power_start': new_step.power_start_watts,
            'new_ring_size': new_step.ring_size_mm
        })

        # Handle ring size change if needed
        if new_step.ring_size_mm != old_step.ring_size_mm:
            self._change_ring_size(new_step.ring_size_mm)

def _change_ring_size(self, target_ring_size_mm: float):
    """Change actuator position to achieve target ring size"""
    # Map ring size to actuator position using calibration
    target_position = self._ring_size_to_actuator_position(target_ring_size_mm)

    self.hardware.actuator.move_to_position(target_position)

    self._log_event('ring_size_change', {
        'target_ring_size_mm': target_ring_size_mm,
        'actuator_position_um': target_position
    })

def _ring_size_to_actuator_position(self, ring_size_mm: float) -> float:
    """
    Convert ring size to actuator position using calibration data

    Returns:
        Position in micrometers
    """
    # Load calibration from database
    calibration = get_active_calibration('actuator_position_to_ring_size')
```

```
        # Example: Linear interpolation
        # calibration format: [(position_um, ring_size_mm), ...]
        points = calibration['calibration_points']

        # Simple linear interpolation (can be more sophisticated)
        for i in range(len(points) - 1):
            p1, p2 = points[i], points[i + 1]

            if p1['ring_size_mm'] <= ring_size_mm <= p2['ring_size_mm']:
                # Interpolate
                ratio = (ring_size_mm - p1['ring_size_mm']) / (p2['ring_size_mm'] - p1['ring_size_mm'])
                position = p1['actuator_position_um'] + ratio * (p2['actuator_position_um'] - p1['actuator_position_um'])
                return position

        # Out of calibration range - use nearest
        if ring_size_mm < points[0]['ring_size_mm']:
            return points[0]['actuator_position_um']
        else:
            return points[-1]['actuator_position_um']

    def pause(self):
        """Pause protocol execution"""
        if not self.is_running:
            return

        self.is_paused = True

        # Set laser to zero
        self.hardware.laser.set_power(0)

        self._log_event('protocol_paused', {
            'step': self.current_step_index + 1,
            'elapsed_seconds': time.time() - self.treatment_start_time
        })

    def resume(self):
        """Resume paused protocol"""
        if not self.is_paused:
            return

        self.is_paused = False

        self._log_event('protocol_resumed', {
            'step': self.current_step_index + 1
        })

    def stop(self, reason: str = "User stopped"):
        """Stop protocol execution"""
        if not self.is_running:
            return

        self.is_running = False
        self.is_paused = False

        # Set laser to zero
        self.hardware.laser.set_power(0)

        treatment_duration = time.time() - self.treatment_start_time

        self._log_event('protocol_stopped', {
            'reason': reason,
            'completed_steps': self.current_step_index,
            'total_steps': len(self.protocol.steps),
            'duration_seconds': treatment_duration
        })

    def adjust_power(self, new_power_watts: float, reason: str = ""):
        """
        Override current power (deviation from protocol)

        Used for manual adjustments during treatment
        """
        if not self.is_running or self.is_paused:
            return

        current_step = self.protocol.steps[self.current_step_index]

        self._log_event('protocol_power_override', {
            'step': current_step.step_number,
            'protocol_power': current_step.power_start_watts,
            'override_power': new_power_watts,
            'reason': reason,
            'deviation_watts': new_power_watts - current_step.power_start_watts
        })

        # This gets logged as a deviation - important for audit trail

    def _log_event(self, event_type: str, details: dict):
        """Log protocol event to database"""
        log_treatment_event(
            session_id=self.session.get_active_session_id(),
            event_type=event_type,
            event_details=details
        )
```

Ring Size Control

Actuator Calibration

The linear actuator position controls the ring size. Calibration maps actuator position (micrometers) to ring diameter (millimeters).

Calibration Procedure:

1. **Setup:** Position camera to capture ring
2. **Calibration points:** Move actuator to various positions, measure ring size

3. **Data collection:**

```
calibration_points = [
    {"actuator_position_um": 0,    "ring_size_mm": 2.0},
    {"actuator_position_um": 500,  "ring_size_mm": 2.5},
    {"actuator_position_um": 1000, "ring_size_mm": 3.0},
    {"actuator_position_um": 1500, "ring_size_mm": 3.5},
    {"actuator_position_um": 2000, "ring_size_mm": 4.0},
    {"actuator_position_um": 2500, "ring_size_mm": 4.5},
]
```

4. **Fit curve:** Linear or polynomial fit
5. **Store in database:** `calibrations` table

Ring Size Measurement: Use image processing (circle detection) to measure actual ring diameter during calibration.

Protocol Builder UI

UI Components

1. **Protocol List Panel**
   - Browse saved protocols
   - Search/filter
   - Load protocol for editing
2. **Protocol Editor Panel**
   - Protocol name, description
   - Add/remove/reorder steps
   - Edit step parameters:
     - Duration slider (0-300s)
     - Power start/end (5W)
     - Ring size (2-5mm)
     - Ramp type dropdown
3. **Step Timeline Visualization**
   - Graphical view of power vs. time
   - Ring size indicator per step
   - Hover for details
4. **Validation Panel**
   - Real-time validation as you edit
   - Show safety limit warnings
   - Total treatment time/energy
5. **Protocol Actions**
   - Save protocol
   - Test protocol (dry run without laser)
   - Export to JSON
   - Import from JSON

Example UI Mockup (Conceptual)

```
1. **Protocol Builder                                    [X]**
2. **Protocol Name** - [Custom Treatment A________]
3. **Description** - [Multi-step therapeutic treatment___]
4. **Steps** -
5. **Step 1** - Warm-up                    [Edit] [Delete] [^][v]
6. **Duration** - 30s | Power: 1W  3W | Ring: 4.0mm
7. **Ramp** - Linear
8. **Step 2** - Treatment                  [Edit] [Delete] [^][v]
9. **Duration** - 90s | Power: 5W  5W | Ring: 3.0mm
10. **Ramp** - Constant
11. **Step 3** - Cool-down                  [Edit] [Delete] [^][v]
12. **Duration** - 30s | Power: 5W  2W | Ring: 3.5mm
13. **Ramp** - Linear
14. **[Add Step]**
15. **Power Profile** -
16. **8W            /         \**
17. **6W          /           \**
18. **4W        /**
19. **2W      /                    \___**
20. **0W                                  > Time**
21. **0s    30s   60s   90s  120s  150s**
22. **Safety Check** -
23. **[DONE] All steps within power limits (max 5W)**
24. **[DONE] Total duration 150s (within 300s limit)**
25. **[DONE] Ring sizes within range (2-5mm)**
26. **Total Energy** - 675 Joules
27. **[Cancel]  [Save Protocol]  [Test Run]  [Apply to Session]**
```

Treatment Execution Monitoring

Real-Time Display

During treatment execution, display:

1. **Current Step Info**
   - Step number / total steps
   - Step elapsed time / duration
   - Current power / target power
   - Current ring size
2. **Protocol Progress**
   - Overall progress bar
   - Time remaining
   - Total energy delivered

3. **Live Graphs** (pyqtgraph)
    - Commanded power (blue line)
    - Actual power from photodiode (red line)
    - Power deviation (shaded area)
4. **Safety Status**
    - All interlock states (green/red indicators)
    - Photodiode voltage
    - Footpedal state
5. **Controls**
    - Pause button
    - Stop button
    - Manual power adjustment slider (logs as deviation)

Document Owner: Protocol Engineer **Last Updated:** 2025-10-26