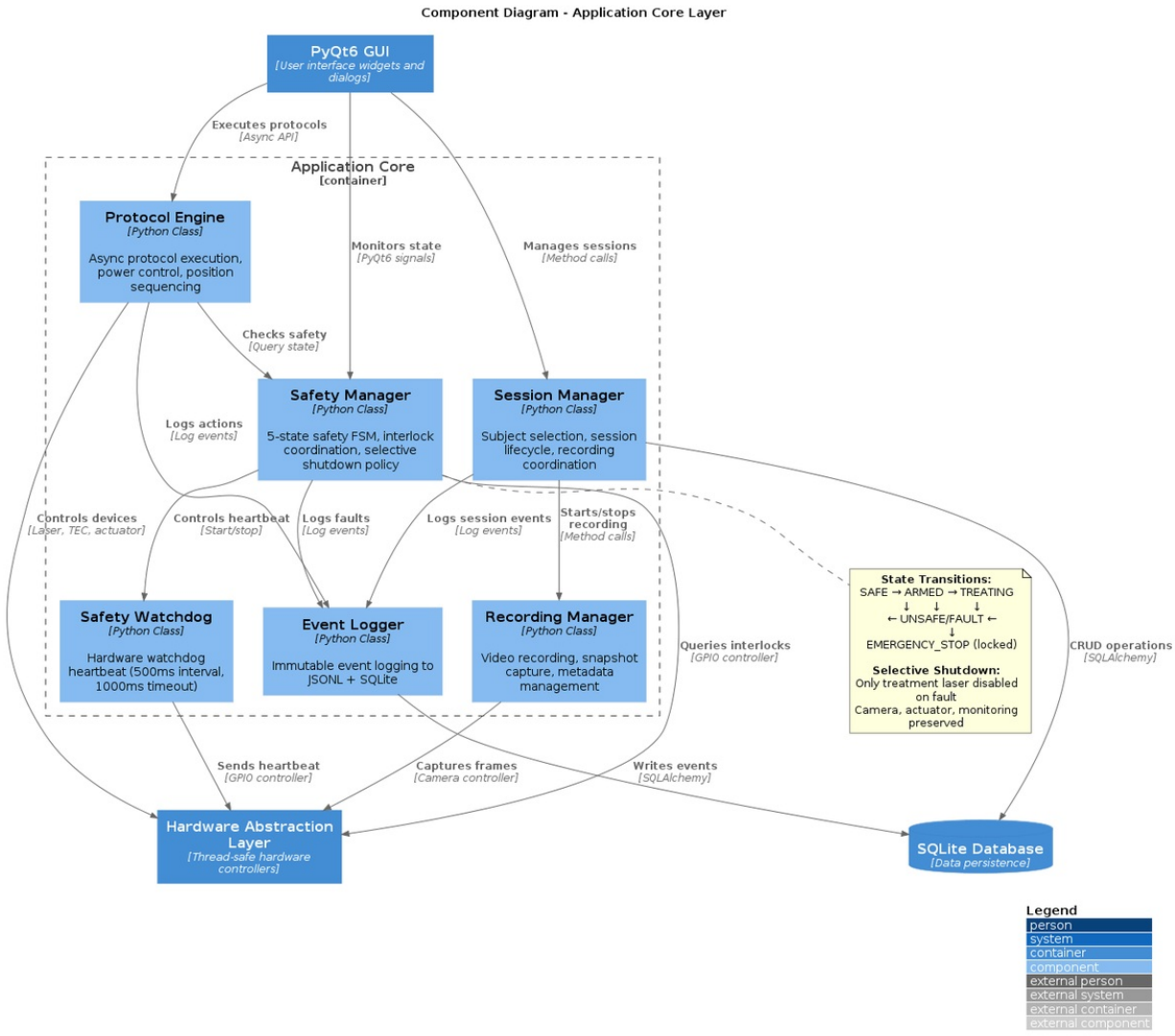
TOSCA Laser Control System - Safety System Architecture

- Architecture Diagrams
  - Figure 1: TOSCA Component Diagram - Application Core
  - Figure 2: safety-state-machine
- Safety Philosophy
- Safety Interlock System
  - Interlock Architecture
  - Hardware Interlocks (GPIO-based)
  - Software Interlocks
  - Safety Manager (Master Controller)
- Safety State Machine
  - State Transition Rules
- Safety Event Logging
  - Logged Events
  - Log Entry Format
- Fault Handling Procedures
  - Fault Detection → Response → Recovery
- Watchdog Timer
- Testing & Validation
  - Safety System Tests
  - Test Documentation
- Regulatory & Compliance Notes

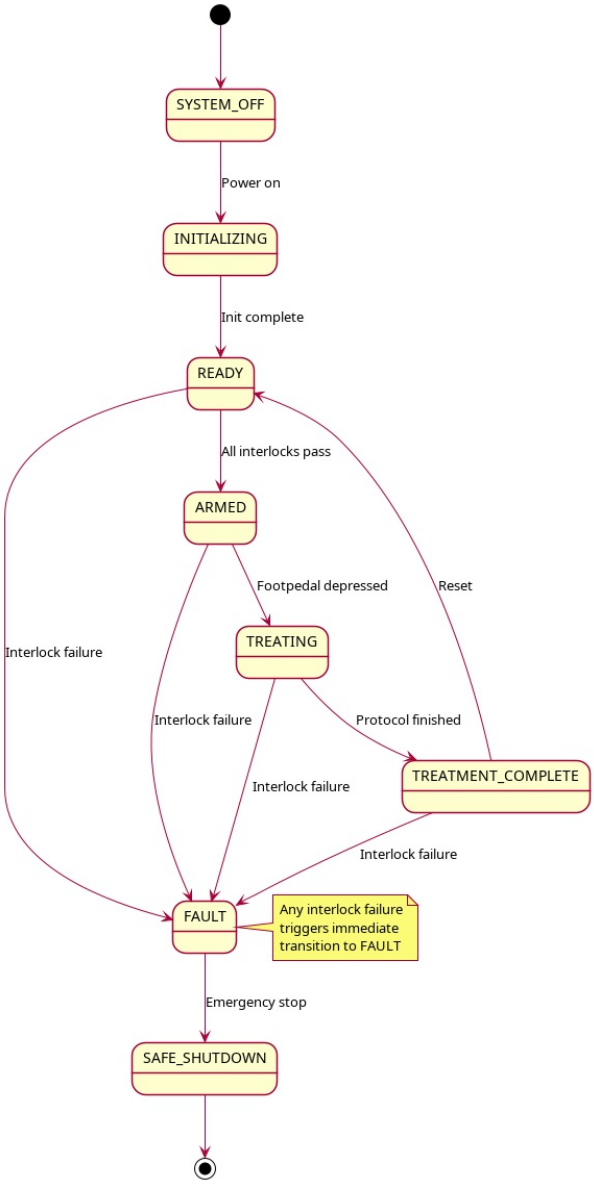
Architecture Diagrams

Figure 1: TOSCA Component Diagram - Application Core



TOSCA Component Diagram - Application Core

Figure 2: safety-state-machine



safety-state-machine

Document Version: 1.0 **Date:** 2025-10-15 **Criticality:** MAXIMUM - Safety Critical System

Safety Philosophy

“Fail-Safe by Default”

This laser system implements multiple redundant safety layers based on these principles:

1. **Defense in Depth**: Multiple independent safety barriers
2. **Fail-Safe Design**: Any component failure defaults to laser OFF
3. **Positive Permission**: Laser requires active permission to fire (not just absence of prohibition)
4. **Continuous Monitoring**: Safety checks at high frequency (≥100Hz)
5. **Immutable Logging**: All safety events are permanently recorded
6. **No Single Point of Failure**: Redundant safety mechanisms

Safety Interlock System

Interlock Architecture

```
                    Safety Manager
                    (Master Controller)



        ▼                ▼                ▼
Hardware Interlocks    Software       Session
- dead man's switch footpedal           Interlocks      Interlocks
- Laser Spot Smoothing Module     - E-stop        - Active sess.
- photodiode laser pickoff measurement          - Power         - Subject ID
                         limits        - Tech auth.
```

```
                    ALL MUST PASS    ► LASER ENABLE
                    ANY FAILS        ► LASER DISABLE
```

Hardware Interlocks (GPIO-based)

1. dead man's switch footpedal Deadman Switch (GPIO-1)

Type: Active-High Requirement (Positive Permission)

Connection: - Arduino Nano Digital Pin (connected via serial protocol) - dead man's switch footpedal switch monitored via custom firmware - Hardware debouncing in firmware

Behavior:

```
dead man's switch footpedal State:
  DEPRESSED (pin HIGH)  → Laser CAN fire (if other interlocks pass)
  RELEASED  (pin LOW)   → Laser CANNOT fire (immediate shutdown)
```

Implementation:

```python
class Dead Man's Switch FootpedalInterlock:
    def __init__(self, gpio_pin):
        self.gpio_pin = gpio_pin
        self.state = False
        self.last_check_time = None

    def check(self) -> tuple[bool, str]:
        """
        Returns: (is_safe, status_message)
        """
        self.state = self.gpio_pin.value  # Read GPIO pin
        self.last_check_time = time.time()

        if self.state:
            return (True, "dead man's switch footpedal depressed - OK")
        else:
            return (False, "dead man's switch footpedal released - LASER DISABLED")

    def is_timeout(self, timeout_seconds=0.1) -> bool:
        """Check if reading is stale"""
        if self.last_check_time is None:
            return True
        return (time.time() - self.last_check_time) > timeout_seconds
```

Safety Features: - Poll rate: ≥100Hz (10ms intervals) - Timeout detection: If no update in 100ms, trigger fault - Debouncing: 20ms debounce to avoid false triggers - Fail-safe: Pin defaults to LOW (laser disabled) on disconnect

Fault Conditions: - dead man's switch footpedal released during treatment → Immediate laser shutdown - GPIO read timeout → Fault state, laser disabled - GPIO communication error → Fault state, laser disabled

---

2. Hotspot Laser Spot Smoothing Module Interlock (GPIO-1)

Type: Signal Health Monitoring (Positive Permission)

Connection: - Arduino Nano Digital Pins (D2 motor control, D3 vibration sensor) - Motor control output via D2 - Vibration detection input via D3 - Both signals required for safety interlock

Behavior:

```
Laser Spot Smoothing Module State:
  SIGNAL PRESENT & HEALTHY → Laser CAN fire (if other interlocks pass)
  SIGNAL ABSENT OR FAULT   → Laser CANNOT fire (immediate shutdown)
```

Implementation:

```python
class SmoothingDeviceInterlock:
    def __init__(self, gpio_pin, adc_pin=None):
        self.gpio_pin = gpio_pin
        self.adc_pin = adc_pin
        self.state = False
        self.signal_voltage = 0.0
        self.last_check_time = None

        # Thresholds (configure per device)
        self.min_voltage = 2.5  # Minimum voltage for "healthy"
        self.max_voltage = 5.0  # Maximum expected voltage

    def check(self) -> tuple[bool, str]:
        """
        Returns: (is_safe, status_message)
        """
        digital_signal = self.gpio_pin.value
        self.last_check_time = time.time()

        # Check digital signal presence
        if not digital_signal:
            self.state = False
            return (False, "Smoothing device signal ABSENT")

        # If ADC available, check signal quality
        if self.adc_pin:
            self.signal_voltage = self.adc_pin.voltage

            if self.signal_voltage < self.min_voltage:
                self.state = False
                return (False, f"Smoothing device voltage LOW ({self.signal_voltage:.2f}V)")

            if self.signal_voltage > self.max_voltage:
                self.state = False
                return (False, f"Smoothing device voltage HIGH ({self.signal_voltage:.2f}V)")
```

```
        # All checks passed
        self.state = True
        return (True, "Smoothing device healthy - OK")

    def is_timeout(self, timeout_seconds=0.1) -> bool:
        if self.last_check_time is None:
            return True
        return (time.time() - self.last_check_time) > timeout_seconds
```

Safety Features: - Poll rate: ≥100Hz - Voltage range validation (if ADC available) - Timeout detection - Fail-safe: Device fault → Laser disabled

Fault Conditions: - Signal loss during treatment → Immediate laser shutdown - Voltage out of range → Warning → Fault if persists >500ms - GPIO/ADC communication error → Fault state

---

3. photodiode laser pickoff measurement Feedback Monitor (GPIO-2 ADC)

Type: Output Power Verification (Continuous Monitoring)

Connection: - Arduino Nano Analog Input (A0) - photodiode laser pickoff measurement voltage from laser pickoff (0-5V typical) - Voltage proportional to laser output power - 10-bit ADC resolution (0-1023)

Purpose: - Verify commanded laser power matches actual output - Detect laser malfunction or optical path blockage - Provide real-time power feedback

Implementation:

```
class Photodiode Laser Pickoff MeasurementMonitor:
    def __init__(self, adc_pin, calibration_data):
        self.adc_pin = adc_pin
        self.calibration = calibration_data  # Maps voltage to watts
        self.measured_voltage = 0.0
        self.measured_power_watts = 0.0
        self.last_check_time = None

        # Thresholds (from config)
        self.warning_threshold_percent = 15.0  # Warn if 15% deviation
        self.fault_threshold_percent = 30.0    # Fault if 30% deviation
        self.min_voltage = 0.05  # Below this is considered "laser off"

    def voltage_to_power(self, voltage: float) -> float:
        """Convert photodiode laser pickoff measurement voltage to watts using calibration"""
        # Example: Linear calibration
        # Actual implementation loads from database
        slope = self.calibration['slope']
        intercept = self.calibration['intercept']
        return slope * voltage + intercept

    def check(self, commanded_power_watts: float) -> tuple[bool, str, str]:
        """
        Returns: (is_safe, severity, status_message)
        severity: 'ok', 'warning', 'fault'
        """
        self.measured_voltage = self.adc_pin.voltage
        self.measured_power_watts = self.voltage_to_power(self.measured_voltage)
        self.last_check_time = time.time()

        # If laser should be off
        if commanded_power_watts < 0.1:
            if self.measured_voltage > self.min_voltage:
                return (False, 'fault', "Laser should be OFF but photodiode laser pickoff measurement detects light")
            else:
                return (True, 'ok', "Laser OFF - OK")

        # Laser should be on - check power match
        deviation_percent = abs(self.measured_power_watts - commanded_power_watts) / commanded_power_watts * 100

        if deviation_percent > self.fault_threshold_percent:
            return (False, 'fault',
                    f"Power mismatch FAULT: commanded {commanded_power_watts:.2f}W, "
                    f"measured {self.measured_power_watts:.2f}W ({deviation_percent:.1f}% deviation)")

        elif deviation_percent > self.warning_threshold_percent:
            return (True, 'warning',
                    f"Power mismatch WARNING: commanded {commanded_power_watts:.2f}W, "
                    f"measured {self.measured_power_watts:.2f}W ({deviation_percent:.1f}% deviation)")

        else:
            return (True, 'ok',
                    f"Power OK: commanded {commanded_power_watts:.2f}W, "
                    f"measured {self.measured_power_watts:.2f}W")

    def is_timeout(self, timeout_seconds=0.1) -> bool:
        if self.last_check_time is None:
            return True
        return (time.time() - self.last_check_time) > timeout_seconds
```

Safety Features: - Sampling rate: ≥100Hz - Calibrated voltage-to-power conversion (from calibration table) - Two-tier alerts: Warning (15% deviation) → Fault (30% deviation) - Validates laser is OFF when commanded OFF - Timeout detection

Fault Conditions: - Power deviation >30% for >500ms → Laser shutdown - photodiode laser pickoff measurement reads power when laser should be off → Fault - ADC communication error → Fault

---

Software Interlocks

4. Software Emergency Stop

Type: User-Initiated Immediate Shutdown

Triggers: - UI button: Large red “EMERGENCY STOP” button - Keyboard shortcut: ESC key (global capture) - Programmatic: Any code can call `SafetyManager.emergency_stop()`

Behavior: - Highest priority interrupt - Bypasses all queues and delays - Immediately: 1. Set laser power to 0 2. Send laser OFF command 3. Transition system to

FAULT state 4. Log event with timestamp 5. Display alert modal (cannot be dismissed without supervisor)

Implementation:

```python
class EmergencyStop:
    def __init__(self, hardware_manager):
        self.hardware = hardware_manager
        self.triggered = False
        self.trigger_time = None
        self.trigger_source = None

    def trigger(self, source: str, reason: str = ""):
        """Activate emergency stop"""
        if self.triggered:
            return  # Already triggered

        self.triggered = True
        self.trigger_time = time.time()
        self.trigger_source = source

        # Execute emergency shutdown sequence
        self._emergency_shutdown(reason)

    def _emergency_shutdown(self, reason: str):
        """
        Selective emergency shutdown - TREATMENT LASER ONLY.

        Per SAFETY_SHUTDOWN_POLICY.md (v1.1):
        - Treatment laser: DISABLED (immediate shutdown)
        - Camera: MAINTAINED (for visual assessment)
        - Actuator: MAINTAINED (for safe repositioning)
        - Aiming laser: MAINTAINED (Class 2, inherently safe)
        - GPIO monitoring: MAINTAINED (for diagnostics)
        - Event logging: MAINTAINED (for audit trail)
        """
        try:
            # 1. IMMEDIATE: Disable treatment laser only
            self.hardware.laser.emergency_stop()
            self.hardware.laser.set_power(0)
            logger.info("Treatment laser disabled (selective shutdown)")

            # 2. MAINTAIN: Other systems remain operational
            # - Camera continues streaming (operator can see treatment area)
            # - Actuator remains controllable (can reposition if needed)
            # - Aiming laser stays available (low power, Class 2 safe)
            # - GPIO monitoring continues (for diagnostics)
            # - Event logging active (for complete audit trail)

            # 3. Log to database
            log_safety_event(
                event_type='emergency_stop',
                severity='emergency',
                description=f"E-stop triggered by {self.trigger_source}: {reason} (selective shutdown - laser only)",
                system_state='fault'
            )

            # 4. Notify UI
            signal_emergency_stop.emit(self.trigger_source, reason)

        except Exception as e:
            # Even if logging fails, laser should be stopped
            logger.critical(f"Emergency stop execution error: {e}")

    def reset(self, supervisor_auth_token: str):
        """Reset e-stop (requires supervisor authorization)"""
        if not verify_supervisor_auth(supervisor_auth_token):
            raise PermissionError("Supervisor authorization required to reset E-stop")

        self.triggered = False
        self.trigger_time = None
        self.trigger_source = None

        log_safety_event(
            event_type='emergency_stop_reset',
            severity='info',
            description="E-stop reset by supervisor"
        )
```

IMPORTANT: This implementation follows the **Selective Shutdown Policy**.

See `SAFETY_SHUTDOWN_POLICY.md` for the complete canonical policy: - **Treatment laser:** DISABLED immediately on safety fault - **Camera:** MAINTAINED (for visual assessment) - **Actuator:** MAINTAINED (for safe repositioning) - **Aiming laser:** MAINTAINED (Class 2, low power, inherently safe) - **GPIO monitoring:** MAINTAINED (for diagnostics)

Only the treatment laser is shut down on safety fault. All other systems remain operational to support safe assessment and recovery.

---

5. Power Limit Enforcer

Type: Software Hard Limits

Purpose: Prevent laser power from exceeding safe limits

Implementation:

```python
class PowerLimitEnforcer:
    def __init__(self):
        # Load from configuration
        self.absolute_max_power_watts = 10.0  # Hardware maximum
        self.protocol_max_power_watts = None  # Set per protocol
        self.session_max_power_watts = None   # Set per session
        self.ramp_rate_limit_watts_per_sec = 2.0  # Max power change rate

    def validate_power_command(self, requested_power: float) -> tuple[bool, float, str]:
        """
        Validate and potentially limit power command
```

```
        Returns:
            (is_valid, limited_power, message)
        """
        # Check absolute hardware limit
        if requested_power > self.absolute_max_power_watts:
            return (False, self.absolute_max_power_watts,
                    f"FAULT: Requested power {requested_power:.2f}W exceeds hardware limit {self.absolute_max_power_watts:.2f}W")

        # Check protocol limit
        if self.protocol_max_power_watts and requested_power > self.protocol_max_power_watts:
            return (False, self.protocol_max_power_watts,
                    f"LIMITED: Requested power {requested_power:.2f}W exceeds protocol limit {self.protocol_max_power_watts:.2f}W")

        # Check session limit (e.g., subject-specific)
        if self.session_max_power_watts and requested_power > self.session_max_power_watts:
            return (False, self.session_max_power_watts,
                    f"LIMITED: Requested power {requested_power:.2f}W exceeds session limit {self.session_max_power_watts:.2f}W")

        # Power is within all limits
        return (True, requested_power, "OK")

    def validate_power_ramp(self, current_power: float, target_power: float, duration_seconds: float) -> tuple[bool, str]:
        """Validate power ramp rate"""
        power_change = abs(target_power - current_power)
        ramp_rate = power_change / duration_seconds

        if ramp_rate > self.ramp_rate_limit_watts_per_sec:
            return (False, f"Ramp rate {ramp_rate:.2f} W/s exceeds limit {self.ramp_rate_limit_watts_per_sec:.2f} W/s")

        return (True, "OK")
```

---

6. Session State Interlock

Type: Logical Interlock (Audit Trail)

Purpose: Ensure laser only fires during active, logged session

Rules: - Laser cannot fire unless: - Active session exists - Subject selected - Tech ID authenticated - Session status = ‘in_progress’

Implementation:

```
class SessionInterlock:
    def __init__(self, session_manager):
        self.session_manager = session_manager

    def check(self) -> tuple[bool, str]:
        """Check if session state allows laser operation"""

        # Check active session
        session = self.session_manager.get_active_session()
        if not session:
            return (False, "No active session - laser disabled")

        # Check session status
        if session.status != 'in_progress':
            return (False, f"Session status '{session.status}' - laser disabled")

        # Check subject assigned
        if not session.subject_id:
            return (False, "No subject assigned - laser disabled")

        # Check tech authenticated
        if not session.tech_id:
            return (False, "No technician authenticated - laser disabled")

        return (True, "Session active - OK")
```

---

7. Camera/Image Validity Interlock

Type: Operational Readiness Check

Purpose: Ensure visual monitoring capability during treatment

Implementation:

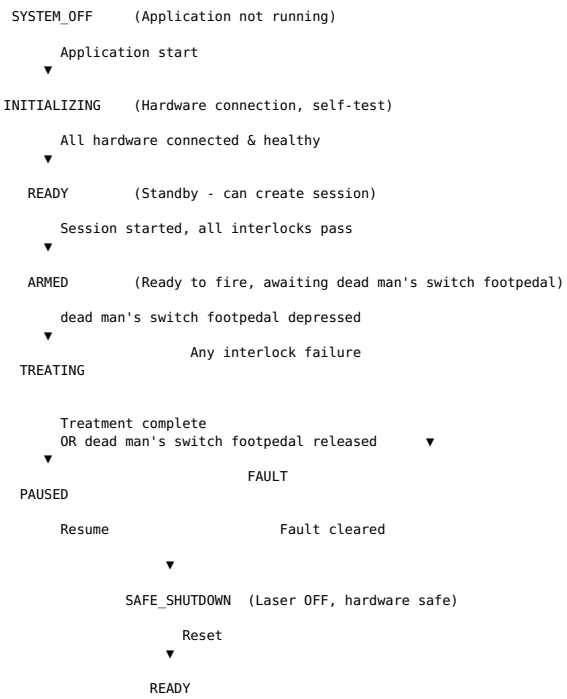
```
class CameraInterlock:
    def __init__(self, camera_controller):
        self.camera = camera_controller
        self.last_valid_frame_time = None
        self.frame_timeout_seconds = 1.0  # Max time between frames

    def check(self) -> tuple[bool, str]:
        """Check if camera is providing valid images"""

        if not self.camera.is_connected():
            return (False, "Camera disconnected")

        if not self.camera.is_streaming():
            return (False, "Camera not streaming")

        # Check frame freshness
        if self.last_valid_frame_time:
            time_since_frame = time.time() - self.last_valid_frame_time
            if time_since_frame > self.frame_timeout_seconds:
                return (False, f"No frame received in {time_since_frame:.1f}s")

        return (True, "Camera active - OK")

    def on_frame_received(self, frame):
        """Called by camera when new frame arrives"""
        self.last_valid_frame_time = time.time()
```

Safety Manager (Master Controller)

The Safety Manager orchestrates all interlocks and makes the final laser enable/disable decision.

```
class SafetyManager:
    """
    Master safety controller - coordinates all interlocks
    """

    def __init__(self, hardware_manager):
        self.hardware = hardware_manager

        # Initialize all interlocks
        self.dead man's switch footpedal = Dead Man's Switch FootpedalInterlock(hardware_manager.gpio1.D5)
        self.smoothing_device = SmoothingDeviceInterlock(hardware_manager.gpio1.D2_D3)
        self.photodiode laser pickoff measurement = Photodiode Laser Pickoff MeasurementMonitor(hardware_manager.gpio2.A0, get_calibration('photodiode laser
        pickoff measurement'))
        self.emergency_stop = EmergencyStop(hardware_manager)
        self.power_limiter = PowerLimitEnforcer()
        self.session_interlock = SessionInterlock(session_manager)
        self.camera_interlock = CameraInterlock(camera_controller)

        self.system_state = SystemState.IDLE
        self.laser_enabled = False
        self.last_safety_check_time = None

    def run_safety_check(self, commanded_power: float = 0.0) -> tuple[bool, dict]:
        """
        Run all safety checks

        Returns:
            (is_safe, status_dict)
        """
        results = {}

        # Check all interlocks
        results['dead man's switch footpedal'] = self.dead man's switch footpedal.check()
        results['smoothing_device'] = self.smoothing_device.check()
        results['photodiode laser pickoff measurement'] = self.photodiode laser pickoff measurement.check(commanded_power)
        results['emergency_stop'] = (not self.emergency_stop.triggered, "OK" if not self.emergency_stop.triggered else "E-STOP ACTIVE")
        results['session'] = self.session_interlock.check()
        results['camera'] = self.camera_interlock.check()

        # Check for timeouts
        if self.dead man's switch footpedal.is_timeout():
            results['dead man's switch footpedal'] = (False, "dead man's switch footpedal read timeout")
        if self.smoothing_device.is_timeout():
            results['smoothing_device'] = (False, "Smoothing device read timeout")
        if self.photodiode laser pickoff measurement.is_timeout():
            results['photodiode laser pickoff measurement'] = (False, 'fault', "photodiode laser pickoff measurement read timeout")

        # Determine overall safety state
        all_safe = all(r[0] for r in results.values() if isinstance(r, tuple))

        self.last_safety_check_time = time.time()

        return (all_safe, results)

    def enable_laser(self, commanded_power: float) -> tuple[bool, str]:
        """
        Attempt to enable laser with given power

        Returns:
            (success, message)
        """
        # Run full safety check
        is_safe, status = self.run_safety_check(commanded_power)

        if not is_safe:
            # Find which interlock failed
            failed_interlocks = [name for name, result in status.items() if not result[0]]
            message = f"Laser DISABLED - Failed interlocks: {', '.join(failed_interlocks)}"

            self.laser_enabled = False
            return (False, message)

        # All interlocks pass - validate power
        valid, limited_power, power_msg = self.power_limiter.validate_power_command(commanded_power)

        if not valid:
            self.laser_enabled = False
            log_safety_event('power_limit_exceeded', 'critical', power_msg)
            return (False, power_msg)

        # Enable laser
        self.hardware.laser.set_power(limited_power)
        self.laser_enabled = True
        self.system_state = SystemState.TREATING

        return (True, f"Laser ENABLED at {limited_power:.2f}W")

    def disable_laser(self, reason: str = "Normal"):
        """
        Disable treatment laser only (selective shutdown).

        Other systems (camera, actuator, GPIO monitoring)
        remain operational per SAFETY_SHUTDOWN_POLICY.md.
        """
        self.hardware.laser.set_power(0)
        self.laser_enabled = False
        self.system_state = SystemState.READY

        log_safety_event('laser_disabled', 'info',
                         f"Treatment laser disabled (selective): {reason}")
```

Safety State Machine



State Transition Rules

From State	To State	Condition
SYSTEM_OFF	INITIALIZING	Application starts
INITIALIZING	READY	All hardware connected, self-test pass
INITIALIZING	FAULT	Hardware connection failure
READY	ARMED	Session active, all interlocks pass
ARMED	TREATING	dead man’s switch footpedal depressed
ARMED	READY	Session ended
TREATING	ARMED	dead man’s switch footpedal released (normal)
TREATING	FAULT	Any interlock failure
TREATING	PAUSED	User pauses (dead man’s switch footpedal still required)
PAUSED	TREATING	User resumes, dead man’s switch footpedal depressed
PAUSED	FAULT	Interlock failure during pause
FAULT	SAFE_SHUTDOWN	Fault acknowledged
SAFE_SHUTDOWN	READY	Supervisor resets, interlocks restored
ANY	FAULT	Emergency stop pressed

Safety Event Logging

All safety-related events are logged to: 1. **Database:** `safety_log` table (persistent) 2. **Application log:** `logs/safety_YYYYMMDD.log` 3. **Event stream:** Real-time UI updates

Logged Events

```
# Event types
SAFETY_EVENT_TYPES = [
    # Hardware interlocks
    'footpedal_depressed',
    'footpedal_released',
    'footpedal_timeout',
    'smoothing_device_healthy',
    'smoothing_device_fault',
    'smoothing_device_recovery',
    'photodiode_ok',
    'photodiode_warning',
    'photodiode_fault',

    # Software interlocks
    'emergency_stop_pressed',
    'emergency_stop_reset',
    'power_limit_exceeded',
    'power_ramp_violation',
    'session_interlock_fail',
    'camera_interlock_fail',

    # State changes
    'system_state_change',
    'laser_enabled',
    'laser_disabled',

    # System health
    'watchdog_timeout',
    'hardware_communication_error',
    'self_test_pass',
```

```
    'self_test_fail',
]
```

Log Entry Format

```python
def log_safety_event(event_type: str, severity: str, description: str, **kwargs):
    """
    Log safety event to database and file

    Args:
        event_type: One of SAFETY_EVENT_TYPES
        severity: 'info', 'warning', 'critical', 'emergency'
        description: Human-readable description
        **kwargs: Additional data (stored in JSON 'details' field)
    """
    entry = {
        'timestamp': datetime.now(),
        'event_type': event_type,
        'severity': severity,
        'description': description,
        'session_id': get_active_session_id(),
        'tech_id': get_active_tech_id(),
        'system_state': safety_manager.system_state,
        'laser_state': hardware.laser.get_state(),
        'footpedal_state': safety_manager.dead man's switch footpedal.state,
        'smoothing_device_state': safety_manager.smoothing_device.state,
        'photodiode_voltage': safety_manager.photodiode laser pickoff measurement.measured_voltage,
        'details': json.dumps(kwargs)
    }

    # Write to database
    db.insert_safety_log(entry)

    # Write to log file
    logger.log(entry)

    # Emit to UI
    signal_safety_event.emit(entry)
```

Fault Handling Procedures

Fault Detection → Response → Recovery

```python
class FaultHandler:
    def handle_fault(self, fault_source: str, fault_description: str):
        """
        Execute fault response procedure

        1. Immediate: Disable laser
        2. Log: Record fault event
        3. Alert: Notify operator
        4. Analyze: Determine if recoverable
        5. Guide: Provide recovery steps
        """

        # 1. Immediate laser disable
        safety_manager.disable_laser(f"FAULT: {fault_source}")

        # 2. Log fault
        log_safety_event(
            event_type=f'{fault_source}_fault',
            severity='critical',
            description=fault_description,
            fault_source=fault_source
        )

        # 3. Transition to FAULT state
        safety_manager.system_state = SystemState.FAULT

        # 4. Display alert
        ui.display_fault_alert(fault_source, fault_description)

        # 5. Determine recovery procedure
        recovery_steps = self.get_recovery_steps(fault_source)
        ui.display_recovery_procedure(recovery_steps)

    def get_recovery_steps(self, fault_source: str) -> list:
        """Return recovery procedure for fault type"""
        recovery_procedures = {
            'dead man's switch footpedal': [
                "1. Check dead man's switch footpedal connection",
                "2. Test dead man's switch footpedal operation (depress and release)",
                "3. Verify GPIO pin reading in diagnostics",
                "4. If OK, supervisor reset to clear fault"
            ],
            'smoothing_device': [
                "1. Check laser spot smoothing module power supply",
                "2. Verify device status indicator",
                "3. Check GPIO connection",
                "4. If device fault, do NOT resume treatment",
                "5. Contact service technician"
            ],
            'photodiode laser pickoff measurement': [
                "1. Check optical pickoff alignment",
                "2. Verify photodiode laser pickoff measurement connection",
                "3. Run photodiode laser pickoff measurement calibration test",
                "4. If persistent, contact service technician"
            ],
            'camera': [
                "1. Check camera USB connection",
                "2. Check camera power",
                "3. Restart camera in diagnostics menu",
                "4. If unresolved, treatment cannot continue"
            ],
        }
```

```
        return recovery_procedures.get(fault_source, [
            "1. Document fault details",
            "2. Contact system administrator",
            "3. Do not attempt to resume treatment"
        ])
```

Watchdog Timer

Ensures control loop is running and responsive.

```
class SafetyWatchdog:
    def __init__(self, timeout_seconds=1.0):
        self.timeout_seconds = timeout_seconds
        self.last_heartbeat = time.time()
        self.enabled = False
        self.watchdog_thread = None

    def start(self):
        """Start watchdog monitoring"""
        self.enabled = True
        self.last_heartbeat = time.time()
        self.watchdog_thread = threading.Thread(target=self._watchdog_loop, daemon=True)
        self.watchdog_thread.start()

    def heartbeat(self):
        """Called by main control loop to signal alive"""
        self.last_heartbeat = time.time()

    def _watchdog_loop(self):
        """Monitor for heartbeat timeout"""
        while self.enabled:
            time.sleep(0.1)  # Check every 100ms

            time_since_heartbeat = time.time() - self.last_heartbeat

            if time_since_heartbeat > self.timeout_seconds:
                # WATCHDOG TIMEOUT - Critical fault
                self._trigger_watchdog_fault()

    def _trigger_watchdog_fault(self):
        """Execute emergency shutdown due to watchdog timeout"""
        log_safety_event(
            event_type='watchdog_timeout',
            severity='emergency',
            description=f"Control loop unresponsive for {self.timeout_seconds}s"
        )

        # Emergency hardware shutdown
        safety_manager.emergency_stop.trigger('watchdog', 'Control loop timeout')

    def stop(self):
        """Stop watchdog (application shutdown)"""
        self.enabled = False
```

Testing & Validation

Safety System Tests

1. **dead man's switch footpedal Test**
   - Verify laser disables immediately on pedal release (<50ms)
   - Test pedal timeout detection
   - Test debouncing (no false triggers)
2. **Laser Spot Smoothing Module Test**
   - Verify laser disables on signal loss
   - Test voltage threshold detection
   - Test recovery behavior
3. **photodiode laser pickoff measurement Test**
   - Verify power mismatch detection
   - Test warning threshold (15% deviation)
   - Test fault threshold (30% deviation)
   - Test response time (<100ms)
4. **Emergency Stop Test**
   - Verify immediate response (<50ms)
   - Test UI button and keyboard shortcut
   - Verify laser shutdown and state transition
5. **Integration Test**
   - Simulate multiple simultaneous faults
   - Verify fault priority handling
   - Test recovery procedures

Test Documentation

Location: `tests/test_safety/`

Each test logs results and must pass before system deployment.

Regulatory & Compliance Notes

IMPORTANT: This is a safety-critical laser system. Before deployment:

1. **Risk Analysis:** Perform FMEA (Failure Mode and Effects Analysis)
2. **Testing:** Comprehensive safety testing with documented results

3. **Validation:** Independent safety review
4. **Documentation:** Complete safety manual
5. **Training:** Operator training on safety systems
6. **Maintenance:** Regular safety system verification

Document Owner: Safety Engineer **Last Updated:** 2025-10-26 **Next Review:** Before each development phase