Robo Desk Buddy - Sprint 3 Documentation

Sprint 3 (Expanded Robot Action Demo)

Sprint 3 Deliverables:

- ✅ Controller extended with an additional placeholder action (turn + speak)
- ✅ Robot can switch between actions during simulation
- ✅ Documentation of how multiple actions were wired together

Unit Tests:

- Robot executes second action without errors
- Robot can switch between first and second action during a single run

To-Dos:

- ☑ Add second placeholder action
- ☑ Update controller to handle multiple actions
- ☑ Run and verify both actions in sequence
- ☑ Document process

Expanded Robot Action Implementation

Sprint 3 Overview

Sprint 3 builds upon Sprint 2 by adding a **second placeholder action** that combines **turning** and **speaking** functionality. The robot can now switch between multiple actions during a single simulation run, demonstrating more complex behavior coordination.

Key Improvements from Sprint 2:

1. **Additional Action**: Added turn + speak combination behavior

2. **Action Switching**: Robot can switch between different actions during simulation
3. **Improved Controls**: Enhanced keyboard interface for multiple actions
4. **Better Organization**: Cleaner controller structure for handling multiple behaviors

Core Implementation Principles:

1. **Action Separation**: Each behavior is implemented as a separate method
2. **Sequential Execution**: Actions run one at a time (not simultaneously)
3. **Keyboard Control**: Each action mapped to specific keyboard input
4. **State Management**: Robot remembers and can switch between different behaviors

---

Robot Action Methods

Action 1: Wave Gesture (From Sprint 2)

```
def wave(self):
    """Make the robot tilt as a wave gesture."""
    print("👋 Waving...")
    self.motor.setPosition(0.5)    # tilt right
    self.step(500)                 # wait 0.5 sec
    self.motor.setPosition(-0.5)   # tilt left
    self.step(500)
    self.motor.setPosition(0.0)    # reset
    self.step(500)
    print("👋 Wave complete!")
```

Action 2: Blink Lights (From Sprint 2)

```
def blink_lights(self):
    """Blink LED on and off."""
    print("✨ Blinking LEDs...")
    for _ in range(3):
        self.led_left.set(1)    # on
        self.led_right.set(1)    # on
        self.step(300)
        self.led_left.set(0)    # off
```

```
        self.led_right.set(0)   # off
        self.step(300)
    print("✨ Blink complete!")
```

Action 3: Turn and Speak (NEW in Sprint 3)

```
def turn_and_speak(self):
    """
    NEW Sprint 3 Action: Combines turning (head movement) with speak

    This demonstrates:
    - Turn: Head rotation to look around
    - Speak: Verbal communication with user
    - Combined behavior: Coordinated action sequence
    """
    print("🔚 Starting turn and speak action...")

    # Phase 1: Turn to look left
    print("🗣 Looking left...")
    self.motor.setPosition(-0.7)  # Turn head left
    self.step(800)                # Hold position

    # Phase 2: Speak while turned
    print("🗣 Hello from the left side!")
    self.step(500)

    # Phase 3: Turn to look right
    print("🗣 Looking right...")
    self.motor.setPosition(0.7)   # Turn head right
    self.step(800)                # Hold position

    # Phase 4: Speak while turned
    print("🗣 Hello from the right side!")
    self.step(500)

    # Phase 5: Return to center
    print("🗣 Returning to center...")
```

```
        self.motor.setPosition(0.0)    # Reset to center
        self.step(500)

        print("🗣️ Turn and speak complete!")
```

Action Switching and Control System

Enhanced Controller Structure

```
    def run_async(self, func):
        """
        Execute any function in a separate thread with automatic cleanup

        Parameters:
        - func: The function to run in a thread (e.g., self.wave_threade

        Returns:
        - thread: The created thread object

        Threading Flow:
        1. Create wrapper function that handles errors and cleanup
        2. Create new thread with the wrapper
        3. Add thread to active_threads list for tracking
        4. Start the thread (non-blocking)
        5. Return thread object for reference
        """
        def wrapper():
            """Internal wrapper that handles thread lifecycle"""
            try:
                func()  # Execute the actual action function
            except Exception as e:
                print(f"❌ Thread error in {func.__name__}: {e}")
            finally:
                # Clean up: Remove this thread from active list when don
                with self.action_lock:  # Thread-safe list modification
                    if threading.current_thread() in self.active_threads
```

```
                self.active_threads.remove(threading.current_thre
        print(f"🧹 Thread {func.__name__} cleaned up")

    # Create and start the thread
    thread = threading.Thread(target=wrapper, name=f"Thread-{func.__

    # Track the thread
    with self.action_lock:  # Thread-safe list modification
        self.active_threads.append(thread)

    thread.start()  # Start thread (non-blocking)
    print(f"🚀 Started thread for {func.__name__}")
    return thread
```

Emergency Stop System

```
def stop_all_actions(self):
    """
    Emergency stop - reset all devices to neutral state

    Note: Python threading doesn't allow forceful thread termination
    so this only resets the hardware devices. Threads will complete
    their current sleep cycles naturally.

    Best Practice: Design actions to check a "stop_flag" periodicall
    """
    print("🛑 Stopping all actions...")

    # Reset all devices to neutral/off state
    with self.action_lock:
        self.motor.setPosition(0.0)  # Head to center
        self.led_left.set(0)         # LEDs off
        self.led_right.set(0)        # LEDs off

    print(f"🧹 Active threads: {len(self.active_threads)}")
    print("⚠️ Note: Threads will complete their current sleep cycles
```

Main Control Loop (Threading Coordinator)

Threading Architecture in Main Loop

```
def main():
    """
    Main simulation loop - handles keyboard input and coordinates ac

    Threading Architecture:
    - This function runs in the MAIN THREAD
    - Only the main thread calls bot.step() - critical for simulatio
    - All actions are executed in separate worker threads
    - Keyboard input is processed in the main thread for responsiven
    """
    # Initialize robot and devices
    bot = DeskBuddy()
    keyboard = bot.getKeyboard()
    keyboard.enable(TIME_STEP)

    # Display controls
    print("=" * 60)
    print("🎮 THREADING DEMO CONTROLS:")
    print("  W = Wave (head tilt)")
    print("  B = Blink LEDs")
    print("  H = Say Hello (text messages)")
    print("  Y = ALL ACTIONS SIMULTANEOUSLY! 🎭")
    print("  S = Stop/Reset all actions")
    print("  Q = Quit simulation")
    print("=" * 60)

    # ==========================================
    # MAIN SIMULATION LOOP
    # ==========================================
    while bot.step(TIME_STEP) != -1:  # CRITICAL: Only main thread ca
        key = keyboard.getKey()
```

```
        # Process individual actions (each starts a new thread)
        if key == ord('W'):
            print("🔚 Starting wave action in new thread...")
            bot.run_async(bot.wave_threaded)

        elif key == ord('B'):
            print("🔚 Starting blink action in new thread...")
            bot.run_async(bot.blink_lights_threaded)

        elif key == ord('H'):
            print("🔚 Starting hello action in new thread...")
            bot.run_async(bot.say_hello_threaded)

        # ==========================================
        # SIMULTANEOUS ACTIONS DEMO
        # ==========================================
        elif key == ord('Y'):
            print("🎭 SIMULTANEOUS ACTIONS DEMO!")
            print("🚀 Starting ALL actions in parallel threads...")

            # Start all three actions simultaneously
            # Each runs in its own thread with independent timing
            thread1 = bot.run_async(bot.say_hello_threaded)    # ~0.
            thread2 = bot.run_async(bot.blink_lights_threaded) # ~1.
            thread3 = bot.run_async(bot.wave_threaded)         # ~1.

            print(f"✅ Started {len(bot.active_threads)} simultaneou
            print("👀 Watch: Head waves + LEDs blink + hello message

        # ==========================================
        # EMERGENCY CONTROLS
        # ==========================================
        elif key == ord('S'):
            bot.stop_all_actions()

        elif key == ord('Q'):
            print("🛑 Quitting simulation...")
            bot.stop_all_actions()
```

```
            break

        # Debug: Show unknown keys
        elif key != -1:
            print(f"❓ Unknown key pressed: {chr(key)} (code: {key})

    print("🤖 Desk Buddy shutting down...")
    print(f"🧹 Final cleanup: {len(bot.active_threads)} threads stil
```

Controls and Usage

Threading Demo Controls

- **W** = Wave (head tilt) - Single threaded action
- **B** = Blink LEDs - Single threaded action
- **H** = Say Hello - Single threaded action
- **Y** = **ALL ACTIONS SIMULTANEOUSLY!** 🎭
- **S** = Stop/Reset all actions
- **Q** = Quit simulation

Key Threading Behaviors

Single Actions (W, B, H)

- Each creates **one thread** → you see both text AND physical action
- **Non-blocking** → main loop continues processing input
- **Thread-safe** → multiple single actions can overlap

Simultaneous Actions (Y)

- Creates **multiple threads** → all actions happen at once
- **Independent timing** → each action runs at its own pace
- **True parallelism** → head waves while LEDs blink while messages print

Emergency Stop (S)

- **Immediate device reset** → all motors/LEDs return to neutral
- **Thread tracking** → shows how many threads are still running
- **Graceful handling** → threads complete their current sleep cycles

---

Verification and Testing

Sprint 3 Complete When:

- [ ] Individual actions execute in threads without blocking main loop
- [ ] Simultaneous actions (Y key) run all three actions at once
- [ ] Emergency stop immediately resets all devices
- [ ] No simulation crashes during any threading operation
- [ ] Console output clearly shows thread lifecycle (creation, execution, cleanup)
- [ ] Threading system is fully documented for team recreation

Testing Procedures

Test 1: Single Action Threading

1. Run simulation and press **W** (Wave)
2. **Verify**: Head tilts while console shows thread messages
3. **Verify**: Can immediately press **B** (Blink) before wave completes
4. **Result**: Both actions run simultaneously without conflicts

Test 2: Simultaneous Action Demo

1. Press **Y** (All actions simultaneously)
2. **Verify**: Head waves AND LEDs blink AND hello messages print **at the same time**
3. **Verify**: Console shows all 3 threads starting
4. **Result**: True multitasking behavior demonstrated

Test 3: Emergency Stop

1. Start any action(s) with **W**, **B**, **H**, or **Y**
2. Immediately press **S** (Stop)
3. **Verify**: All devices reset to neutral (head center, LEDs off)
4. **Verify**: Console shows active thread count

5. **Result**: Emergency stop works regardless of running actions

Test 4: Thread Safety

1. Rapidly press multiple keys (**W**, **B**, **H**) in quick succession
2. **Verify**: No crashes or device conflicts occur
3. **Verify**: Each action completes properly
4. **Result**: Threading locks prevent device conflicts successfully

---

Technical Implementation Notes

Why This Threading Approach Works

1. **Simulation Stability**

- Only main thread calls `bot.step()` → prevents Webots crashes
- Worker threads use `time.sleep()` → safe for background execution
- Clean thread lifecycle management → no resource leaks

2. **Device Safety**

- `threading.Lock()` prevents simultaneous device access
- Atomic operations within lock blocks → consistent device states
- Emergency stop can override any thread's device settings

3. **User Experience**

- **Responsive controls** → main loop never blocks on actions
- **Visual feedback** → can see multiple actions happening at once
- **Predictable behavior** → threading is transparent to user

4. **Code Maintainability**

- **Comprehensive documentation** → team can understand and extend
- **Consistent patterns** → all actions follow same threading model
- **Error handling** → graceful degradation when issues occur

---

Troubleshooting Threading Issues

Common Threading Problems

Actions Don't Appear Visual

- **Cause**: Calling `self.step()` in threaded function
- **Solution**: Replace all `self.step()` with `time.sleep()`
- **Check**: Ensure only main thread calls `bot.step()`

Device Conflicts/Erratic Behavior

- **Cause**: Multiple threads accessing devices without locks
- **Solution**: Wrap all device calls with `with self.action_lock:`
- **Check**: Every `self.motor` and `self.led_*` call should be in lock block

Simulation Crashes

- **Cause**: Multiple threads calling `step()` simultaneously
- **Solution**: Remove `self.step()` from all action methods
- **Check**: Only main `while` loop should contain `bot.step()`

Actions Don't Run Simultaneously

- **Cause**: Calling actions directly instead of via `run_async()`
- **Solution**: Use `bot.run_async(bot.action_method)` for all actions
- **Check**: Individual key presses (W, B, H) should use `run_async()`

Sprint Status

Sprint 1 Status: ✅ Complete (Setup + Repo + Scaffolding)
Sprint 2 Status: ✅ Complete (Manual Trigger Actions)
Sprint 3 Status: ✅ Complete (Threading Implementation)
Next Sprint: Sprint 4 - Expanded Robot Action Demo

Files Modified in Sprint 3

Controller Updates

- `controllers/desk_buddy_controller/desk_buddy_controller.py`
  - Added threading imports (`threading` , `time`)
  - Implemented thread-safe action methods
  - Created threading management system (`run_async()` , `stop_all_actions()`)
  - Added comprehensive threading documentation
  - Enhanced main loop with threading coordination

Key Code Additions

- **Threading Architecture**: Main thread vs worker thread separation
- **Thread Safety**: Device access locks and atomic operations
- **Thread Management**: Creation, tracking, and cleanup systems
- **Error Handling**: Exception management in threaded environment
- **Documentation**: Extensive comments for team recreation

Team Recreation Instructions

Step 1: Understanding Threading Architecture

1. **Study the threading principles** outlined in this document
2. **Understand thread separation** between main and worker threads
3. **Learn the critical rules** about `step()` vs `time.sleep()`

Step 2: Implementing Thread-Safe Actions

1. **Convert existing actions** to use `time.sleep()` instead of `self.step()`
2. **Add threading locks** around all device access (`with self.action_lock:`)
3. **Test each action individually** to ensure thread safety

Step 3: Adding Threading Management

1. **Implement `run_async()` method** with error handling and cleanup

2. **Add thread tracking** with `active_threads` list
3. **Create emergency stop** functionality for device reset

Step 4: Integration and Testing

1. **Update main loop** to use `run_async()` for all actions
2. **Add simultaneous action demo** (Y key functionality)
3. **Test thoroughly** following verification procedures above
4. **Document any customizations** for your specific robot setup

This threading foundation enables all future sprints that require concurrent robot behaviors, making it a critical milestone in the development process.