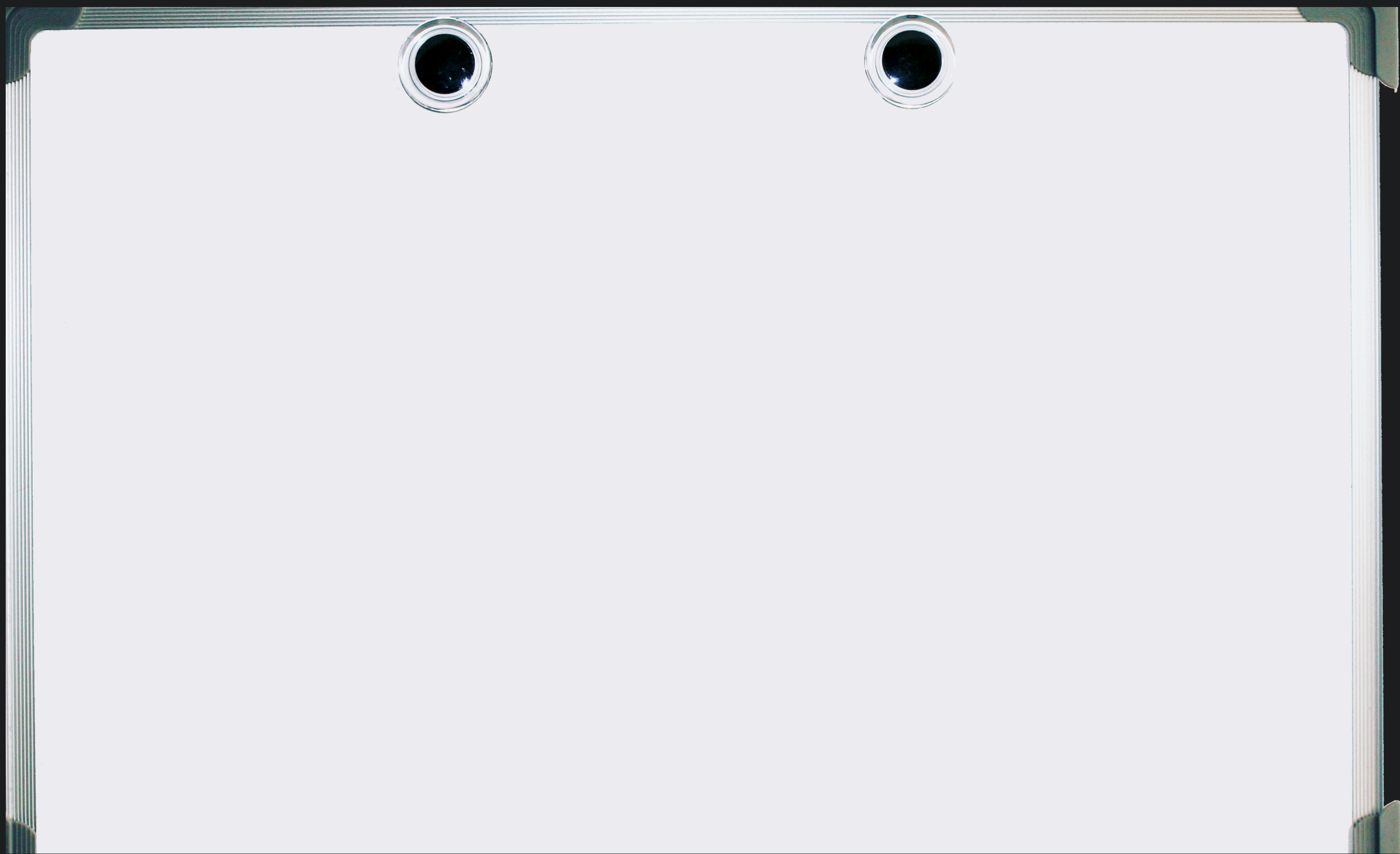
🎨 Systems Design: Orbital Autonomous Control System



"Design a control system that operates 550km above Earth with 240ms latency"

Where do you start?

Most people assume you can remote-control satellites like drones.

I start with the constraint: Earth is too far away.

Here's the thinking process: 👇

📝 Step 1: Why Orbital is DIFFERENT

The Challenge:
Ground control → Satellite
→ 240ms one-way latency (480ms round-trip)
→ Connection windows: 5-15 minutes per orbit
→ 16 orbits per day
→ Physical inaccessibility (can't send a technician)

Questions that determine everything:
→ Can you remote-control with 480ms lag? (No)
→ Can you upload commands during brief windows? (Limited)
→ What if connection drops mid-operation? (System must continue)
→ What if something breaks? (No physical access for repairs)

Ground systems: Low latency, always-connected, accessible
Orbital systems: High latency, intermittent connection, inaccessible

Wrong approach = design like ground system, fail in orbit. Right approach = design for autonomy from day one.

🔄 Step 2: Map the Control Loop

Traditional Ground System:

Sensor → Ground Control → Decision → Command → Actuator

(Works with <50ms latency, always connected)

Orbital System:

Sensor → [240ms] → Ground → [240ms] → Actuator

(480ms round-trip = too slow for dynamic control)

Fill in the **[?]:**

→ What decisions can ground make? (High-level only)

→ What must be autonomous? (Everything time-critical)

→ How do we handle disconnection? (System continues safely)

→ How do we update autonomy? (Upload new logic during windows)

This is the skeleton. Now add the layers that make orbital autonomy work.

🤖 Step 3: Autonomous Control Layer

What Must Be Autonomous:

Attitude control:

→ Maintain orientation (solar panels face sun)
→ React to disturbances (drag, solar pressure)
→ Cannot wait 480ms for ground decision

Collision avoidance:

→ Detect debris (radar, visual)
→ Execute avoidance maneuver (seconds to react)
→ Report to ground after

Fault response:

→ Sensor failure (switch to backup)
→ Power loss (enter safe mode)
→ Thermal anomaly (adjust radiators)

Onboard Decision Logic:

Rule-based systems:

→ IF battery < 20% THEN enter power-save mode
→ IF debris detected < 1km THEN execute avoidance
→ IF ground contact lost > 24h THEN enter safe mode

Machine learning (limited):

→ Anomaly detection (is this behavior normal?)
→ Predictive maintenance (will this component fail?)
→ Resource optimization (power, compute, bandwidth)

The system must ASSUME it's alone. Ground control = advisor, not operator

📡 Step 4: Intermittent Connectivity Strategy

Connection Reality:

Orbit period: 90 minutes

Ground station visibility: 5-15 minutes per pass

Passes per day: 16 (not all over friendly ground stations)

Actual contact time: ~2 hours/day (8% of time)

During Contact Window:

Downlink (Satellite → Ground):

→ Telemetry (health, status, errors)

→ Science data (observations, measurements)

→ Logs (what happened since last contact)

Uplink (Ground → Satellite):

→ High-level commands (mission objectives)

→ Software updates (new autonomy logic)

→ Parameter adjustments (thresholds, priorities)

Outside Contact Window (22 hours/day):

Satellite operates FULLY autonomously:

→ Execute pre-planned operations

→ Respond to anomalies

→ Log everything for next downlink

→ Prioritize what to send (bandwidth limited)

Data prioritization:

→ Critical errors: Send immediately when connected

→ Routine telemetry: Batch and compress

→ Science data: Store, send when bandwidth available

Communication = scheduled, not real-time. Design for 92% disconnection.

⚠️ Step 5: Design for Inevitable Failure Part 1

Failure Reality in Orbit:

Single event upsets (cosmic radiation flips bits):

→ Happens constantly
→ Must detect and correct
→ ECC memory, redundant systems

Component degradation:

→ Solar panels degrade (3-5% per year)
→ Batteries cycle (limited charge cycles)
→ Electronics age (radiation damage)

Unrepairable failures:

→ Can't send a technician
→ Must work around failures
→ Graceful degradation

Redundancy:

Triple modular redundancy (TMR):

→ 3 computers vote on decisions
→ Majority wins (fault tolerance)
→ Can survive 1 computer failure

Safe Mode Design:

Trigger conditions:

→ Ground contact lost > 24 hours
→ Battery < critical threshold
→ Multiple sensor failures
→ Unexpected behavior detected

Safe mode actions:

→ Point solar panels at sun (maximize power)
→ Stop all non-essential operations
→ Extend communication antenna (maximize contact chance)
→ Wait for ground intervention

Watchdog timers:

→ System must "heartbeat" every N seconds
→ If heartbeat stops → automatic reboot
→ If reboot fails → safe mode

Failure cascade prevention:

→ Isolate faults (don't let one failure kill everything)
→ Graceful degradation (lose capability, not entire system)
→ Fail-safe defaults (when uncertain, be safe)

1% orbital systems plan for failure. Because failure is guaranteed, eventually

🔋 Step 6: Resource Constraints in Orbit

Power Reality:

Daylight (45 min):
→ Solar panels generate power
→ Charge batteries
→ Run all systems

Eclipse (45 min):
→ No solar power
→ Battery only
→ Reduce consumption

Power budget:
→ Solar panels: 200W (degrades over time)
→ Battery capacity: 100Wh
→ System draw: 80-150W (depending on operations)

Autonomous power management:

Low power operations (eclipse safe):
→ Attitude control (minimal)
→ Housekeeping
→ Data storage

High power operations (daylight only):
→ Data processing
→ High-bandwidth transmission
→ Science instruments

Emergency power (critical only):
→ Survival heaters
→ Core computer
→ Communication receiver (listen for ground)

Thermal Reality:

Sun-facing side: +120°C
Shadow side: -150°C
Temperature swing: 270°C per orbit

Thermal management:
→ Radiators (passive cooling)
→ Heaters (active warming)
→ Thermal mass (batteries, structure)
→ Orientation control (point hot side away from sun

Systems must survive extreme thermal cycling. And manage it autonomously.

🔄 Step 7: Update Autonomy Without Bricking

The Problem:

Software bug = satellite is now space junk

→ No physical access
→ No rollback if update fails
→ One shot to get it right

Staged updates:

→ Upload new software (may take multiple passes)
→ Verify checksums (ensure no corruption)
→ Ground command: "Run tests on partition B"
→ Satellite reports test results
→ Ground command: "Switch to partition B"
→ If anything fails: Stay on partition A

Automatic rollback:

→ New software boots
→ Must send "I'm alive" within 10 minutes
→ If no heartbeat: automatic reboot to partition A
→ Ground notified of failure

Update Strategy:

Dual-boot system:

→ Partition A: Current working software
→ Partition B: New software upload
→ Test partition B extensively
→ Only switch if tests pass

Golden image:

→ Partition C: Factory default (last resort)
→ Minimal functionality, guaranteed to work
→ Used if both A and B corrupted

ML model updates (if applicable):

→ Upload new model weights
→ Test against validation set onboard
→ Compare performance vs current model
→ Only deploy if better

Over-the-air updates:

→ Differential updates (only changed bytes)
→ Compressed (bandwidth limited)
→ Encrypted (security)
→ Signed (authenticity)

Updates are the highest-risk operation. Design for them from day one.

📊 Step 8: Debug a System 550km Away

The Debugging Problem:
Can't attach debugger
Can't see logs in real-time
Can't reproduce environment on ground

Log prioritization:
→ Errors: Immediate downlink
→ Warnings: Next contact window
→ Info: Batch and compress
→ Debug: Only if requested

Black box recorder:
→ Last 24 hours of data (high resolution)
→ Survives reboots, failures
→ Downloaded first if anomaly detected

Telemetry Strategy:

What to log:
→ Every state transition (mode changes)
→ Every decision made (and why)
→ Every anomaly detected
→ Resource usage (power, memory, CPU)
→ Environmental conditions (temperature, radiation)

Bandwidth-limited logging:
→ Can't send everything
→ Summarize (statistical aggregates)
→ Sample (1% of routine data)
→ Compress (lossless for critical, lossy for science)

Ground replay:
→ High-fidelity simulator on ground
→ Replay satellite telemetry
→ Reproduce issues
→ Test fixes before uploading

Predictive monitoring:
→ Trend analysis (battery degrading faster than expected?)
→ Anomaly detection (behavior changed?)
→ Predictive alerts (failure likely in 30 days)

You can't debug what you can't see. Log everything. Prioritize transmission.

💡Bonus: Why Launch East?

Earth's rotation: 1,670 km/h at equator (460 m/s eastward)

Launching EAST:

→ Rocket velocity + Earth rotation = COMBINED

→ Free 460 m/s boost (saves ~15% fuel)

→ More payload capacity OR higher orbit

Launching WEST:

→ Must overcome Earth's rotation

→ Lose 460 m/s (need 920 m/s extra)

→ Massive fuel penalty

This is why:

→ Cape Canaveral (Florida) launches east over Atlantic

→ SpaceX Starbase (Texas) launches east over Gulf of Mexico

→ Baikonur (Kazakhstan) launches east

→ Kourou (French Guiana) launches east over ocean

→ No major launch sites face west

SpaceX chose Boca Chica, Texas, specifically for:

→ Southern latitude (closer to equator = more rotational boost)

→ Eastward ocean access (Gulf of Mexico)

→ Minimal air traffic (south Texas remote)

→ Starship orbital test flights

Orbital mechanics = constraint from second zero. You can't fight physics.

✅ The Complete Orbital Autonomous System

Layer 1: Autonomous Decision Making
(Rule-based + ML, real-time response)

Layer 2: Intermittent Communication
(5-15 min windows, 16x/day)

Layer 3: Safe Mode & Failure Handling
(TMR, watchdogs, graceful degradation)

Layer 4: Power & Thermal Management
(45-min day/night cycles, autonomous optimization)

Layer 5: Software Updates
(Dual-boot, staged rollout, automatic rollback)

Layer 6: Observability & Debugging
(Prioritized telemetry, black box, ground replay)

Layer 7: Mission Planning & Optimization
(Multi-orbit schedules, constraint satisfaction)

Bonus: Why we launch from the East

This is orbital autonomous control. Not "remote control from ground."
7 layers of autonomy. Because Earth is 550km away.