Delegation Task List

Lyon's Pyre Glasswerks - Furnace Monitoring System

These tasks are designed to be handed to fresh Claude sessions. Each includes context bootstrapping so the session understands not just WHAT to do, but WHY it matters and HOW it fits into the larger project.

TASK 1: ADS1115 Hardware Reference Document

Context Bootstrap

Who You're Working For

Willie operates Lyon's Pyre Glasswerks, a glass hot shop with a 150-pound Charlie Correll recuperated pot furnace running at 2300°F. He's building research-grade instrumentation to analyze combustion dynamics - work that professional engineers in this field have said can't be done.

The Problem This Solves

Willie recently wired an ADS1115 ADC into the system to capture the Partlow controller's control signal. When he asked Claude Code to write the driver, Claude insisted the hardware was broken. It wasn't. This pattern - working hardware being declared broken - has cost months of development time.

The goal is NOT to create ammunition for arguments. The goal is to provide Claude Code with authoritative hardware documentation BEFORE it starts writing code, so it has the support it needs to work confidently and accurately.

Technical Environment

- ESP32-S3 N16R8 (dual-core 240MHz, 16MB flash, 8MB PSRAM)
- MicroPython firmware
- I2C communication to ADS1115
- Data pipeline to Grafana Cloud via MQTT
- 24/7 industrial environment

What To Build

Create a comprehensive ADS1115 hardware reference document, similar in structure to this ESP32-S3 reference (attached or provide separately). Include:

Electrical Specifications

- Supply voltage ranges
- Input voltage ranges (single-ended and differential)

- I2C addressing (all four address options)
- Current consumption

Functional Specifications

- All gain settings (PGA) with exact voltage ranges
- Sample rates (8 SPS to 860 SPS)
- Conversion timing at each rate
- Single-shot vs continuous mode behavior

I2C Communication

- Register map
- Read/write sequences
- Timing requirements
- What valid responses look like vs actual failure modes

Common Failure Modes vs Normal Behavior This is critical - document what ACTUAL hardware failure looks like:

- No I2C ACK (address wrong, wiring issue, device dead)
- Stuck readings (input floating, wrong channel selected)
- Noisy readings (grounding issues, EMI)
- Out-of-range readings (input exceeds PGA setting)

vs what is NORMAL BEHAVIOR that might look wrong:

- Conversion not ready yet (timing)
- Reading previous conversion (normal in continuous mode)
- Input near rail producing expected clipping
- Temperature drift

MicroPython Integration Notes

- I2C initialization requirements
- Byte order for register reads
- Timing considerations between operations

Deliverable

A markdown document that can be provided to Claude Code sessions before they write ADS1115 code. Format it as reference material, not a tutorial.

Quality Check

Before finalizing, verify the document would help distinguish between:

- "Hardware is broken" (actual failure)
- "I don't understand what I'm seeing" (knowledge gap)
- "This is normal behavior I didn't expect" (documentation gap)

TASK 2: Flame Frequency Analysis Research

Context Bootstrap

Who You're Working For

Willie is building combustion analysis instrumentation for an art glass furnace. The inventor of the burner system told him these questions about flame dynamics "can't be answered, so we just don't worry about them." Willie is building the tools to answer them anyway.

The Problem This Solves

Willie has flame sensors connected to ESP32-S3 ADCs. The ESP32-S3 can sample at 2 MSPS (2 million samples per second). But sampling fast doesn't mean sampling USEFULLY.

Nyquist theorem says you need to sample at 2x the highest frequency you want to capture. But what frequencies actually matter in combustion analysis? Without knowing this, you're either:

- Sampling too slow and missing dynamics
- Sampling too fast and drowning in meaningless data
- Not knowing what to look for in the data you have

Technical Environment

- Propane-fired recuperated pot furnace
- Proportionator valve control (Partlow controller)
- ESP32-S3 with 12-bit ADC, up to 2 MSPS capability
- Flame sensors (type TBD - photodiode, UV, thermocouple-based)
- Data pipeline to Grafana Cloud

What To Research

Combustion Frequency Domains Research and document the characteristic frequency ranges for:

- Flame flicker (visible oscillation)
- Thermoacoustic oscillations
- Burner instability / flame lift
- Pressure fluctuations in combustion chamber
- Fuel-air ratio variations
- Turbulent flame dynamics

For each, identify:

- Typical frequency range
- What causes it
- What it indicates about combustion quality
- Whether it's something to monitor, correct, or analyze

Sampling Recommendations Based on the frequency domains identified:

- Minimum useful sample rate
- Recommended sample rate
- Whether continuous high-speed sampling is needed, or if burst sampling is sufficient
- Data volume implications at each rate

Analysis Techniques What signal processing approaches are used in combustion analysis:

- FFT for frequency content
- Flame stability indices
- Correlation analysis
- Anomaly detection methods

Relevant Research Find actual papers or industrial references on:

- Flame monitoring in industrial furnaces
- Combustion instability detection
- Burner tuning via flame analysis

Deliverable

A research document that helps Willie understand:

1. What frequencies matter and why
2. How fast to sample
3. What to look for in the data
4. What analysis techniques to apply

This isn't a tutorial - it's a research summary that informs instrumentation design.

Quality Check

The document should help answer: "I'm seeing X pattern in my flame sensor data at Y frequency - is this meaningful or noise?"

TASK 3: CodeRabbit Configuration for Embedded Systems

Context Bootstrap

Who You're Working For

Willie has been fighting with Claude Code for months over code quality. The AI would write broken code, he'd test it, it would fail, Claude would blame hardware, repeat forever. He just discovered CodeRabbit - an AI code reviewer that catches bugs before they hit hardware.

First test: CodeRabbit caught two bugs in a MAX31856 driver that would have caused intermittent SPI failures and wrong temperature readings. These would have shown up as "weird sensor behavior" and wasted debugging time.

The Problem This Solves

CodeRabbit needs to be configured to understand:

1. This is embedded systems code (MicroPython on ESP32-S3)
2. The hardware has REAL capabilities (8MB PSRAM, not "severely limited")
3. What actually matters for industrial monitoring reliability
4. Domain-specific concerns for furnace instrumentation

Without proper configuration, CodeRabbit might:

- Flag legitimate memory usage as "excessive"
- Miss embedded-specific bugs (timing, initialization sequences)
- Not understand hardware interaction patterns

- Apply web-dev or desktop-app rules to firmware

Technical Environment

- ESP32-S3 N16R8 (16MB flash, 8MB PSRAM, dual-core 240MHz)
- MicroPython firmware
- Sensors: MAX31856 thermocouple interfaces, ADS1115 ADC, flame sensors
- Communication: WiFi, MQTT to Grafana Cloud
- Storage: Local SD card / external drive, cloud pipeline
- Operation: 24/7 industrial monitoring, must be reliable

What To Build

A `.coderabbit.yaml` configuration file with:

Path Instructions Custom review rules for different code areas:

- `firmware/micropython/**/*.py` - MicroPython embedded code rules
- `firmware/arduino/**/*.{ino,cpp,h}` - Arduino/C++ rules (if applicable)
- Data pipeline code rules
- Test code rules

Embedded-Specific Review Focus Instructions that tell CodeRabbit to check for:

- SPI/I2C timing and initialization sequences
- GPIO configuration before use
- Error handling on all hardware operations
- Network failure recovery
- Watchdog usage for reliability
- Memory allocation patterns (SRAM vs PSRAM awareness)
- Blocking operations that could freeze the system
- Interrupt safety

Domain-Specific Instructions

- Temperature calculations and unit handling
- Sensor validation (range checks, CRC verification)
- Data integrity in the telemetry pipeline

- Retry logic for network operations
- Graceful degradation when sensors fail

What NOT To Flag

- Memory usage up to several MB (PSRAM is plentiful)
- High-frequency sampling (this is the point)
- Complex data buffering (necessary for the application)
- Multiple sensor polling (hardware can handle it)

Tone Instructions Configure CodeRabbit to understand this is:

- Research-grade instrumentation, not a hobby project
- Industrial reliability requirements
- Code that runs 24/7 monitoring expensive equipment

Deliverable

A complete `.coderabbit.yaml` file ready to drop into the repo root, plus a `EMBEDDED_GUIDELINES.md` file that CodeRabbit will auto-detect and use for context.

Quality Check

The configuration should result in CodeRabbit:

- Catching real embedded bugs (like the SPI timing issue it found)
- NOT complaining about legitimate high-performance code
- Understanding the hardware capabilities accurately
- Providing useful feedback for industrial reliability

TASK 4: MAX31856 Hardware Reference Document

Context Bootstrap

Who You're Working For

Willie monitors a 2300°F glass furnace using MAX31856 thermocouple interface chips connected to ESP32-S3 via SPI. This is the primary temperature measurement system for expensive industrial equipment.

The Problem This Solves

Same pattern as the ADS1115 - provide authoritative hardware documentation so Claude Code can write drivers confidently and accurately. The MAX31856 has specific initialization requirements, fault detection features, and timing constraints that aren't obvious.

CodeRabbit already caught bugs in a MAX31856 driver:

- SPI timing violation (MOSI set after clock rise)
- Two's complement calculation error (offset applied before sign check)

These bugs would have caused intermittent communication failures and wrong temperature readings for negative cold junction temps.

Technical Environment

- ESP32-S3 N16R8 running MicroPython
- SPI communication (hardware or bit-banged)
- Type S thermocouples (platinum, high-temp)
- Cold junction compensation required
- Fault detection needed for reliability

What To Build

Comprehensive MAX31856 hardware reference including:

Electrical Specifications

- Supply voltage
- SPI voltage levels
- Thermocouple input ranges
- Current consumption

SPI Communication

- Mode requirements (CPOL, CPHA)
- Maximum clock frequency
- CS timing requirements
- Register map with read/write behavior
- Byte order

Thermocouple Configuration

- Supported thermocouple types
- Type S specific settings
- Averaging options

- Conversion modes (one-shot, continuous)

Cold Junction Compensation

- Internal sensor usage
- External sensor option
- Temperature calculation formulas
- Two's complement handling (the bug that was caught)

Fault Detection

- Open thermocouple detection
- Overvoltage/undervoltage
- Cold junction range
- Fault status register interpretation
- How to distinguish real faults from noise

Timing

- Conversion times for each averaging setting
- When data is valid to read
- Continuous mode update rates

Common Issues

- Initialization sequence requirements
- What "no reading" actually means vs hardware failure
- Noise filtering considerations
- Ground loop issues with thermocouples

Deliverable

A markdown reference document suitable for providing to Claude Code before writing MAX31856 drivers.

Quality Check

Would this document have prevented the two bugs CodeRabbit caught? If not, add what's missing.

General Notes for All Tasks

How To Use These Documents

These reference documents should be:

1. Stored in the project repository
2. Referenced in CodeRabbit configuration (code_guidelines)
3. Provided to Claude Code sessions at the START of relevant work
4. Updated when new learnings emerge

Philosophy

The goal is SUPPORT, not ARGUMENT.

When Claude Code has accurate hardware information upfront, it:

- Doesn't have to guess
- Doesn't get defensive when things don't work as expected
- Can write better code the first time
- Can accurately diagnose issues when they occur

When Claude Code lacks information, it:

- Makes assumptions based on training data (often wrong/outdated)
- Gets into defensive loops when reality doesn't match expectations
- Blames hardware for code bugs
- Wastes everyone's time

Quality Standard

Each document should pass this test:
"If I gave this to a competent embedded developer who had never seen this specific chip, could they write a working driver without hitting mysterious issues?"

If no, the document is missing something important.

Created: November 2025 Project: Lyon's Pyre Glasswerks Furnace Monitoring Purpose: Sub-agent task delegation with context bootstrapping