A Fault-Tolerant Firmware Architecture for High-Reliability Embedded Control Systems

Section 1: A Framework for Resilient Embedded Systems

The development of embedded control systems for critical applications, such as commercial refrigeration, demands a paradigm shift from conventional programming practices toward an architecture where resilience is a foundational principle, not an afterthought. The primary objective is to create a system that can not only withstand but also intelligently manage hardware faults, ensuring continuous, predictable operation. A fault in a single, low-cost peripheral must never be permitted to cascade into a total system failure. This report details a comprehensive firmware architecture for the Waveshare ESP32-S3 controller, designed to achieve this level of robustness by systematically eliminating the root causes of system hangs and implementing multiple layers of defense against both predictable and unforeseen failure modes.

1.1 The Anatomy of a System Hang: Blocking Calls as a Critical Vulnerability

In embedded systems, a "blocking" function call is one that halts the execution of the current thread until a specific operation is complete. While simple and convenient for basic applications, this approach is a significant liability in a high-reliability system. If the operation being awaited never completes due to a hardware fault—such as a short circuit, an open circuit, or an unresponsive peripheral—the function call will never return, and the entire task will be permanently frozen. This is commonly referred to as a "hang" or "deadlock."

Many standard Arduino libraries, while popular for their ease of use, are inherently blocking and thus present a critical risk. For instance, the default behavior of the DallasTemperature library's requestTemperatures() function is to block the processor for up to 750 milliseconds while the DS18B20 sensor performs its temperature conversion. Similarly, I2C communication functions within the Wire.h library, such as requestFrom() or endTransmission(), can hang indefinitely if a slave device on the bus holds the data (SDA) or clock (SCL) lines in an invalid state, a common result of a physical short circuit. If such a blocking call is placed within the main setup() function, a faulty peripheral connected at boot time will prevent the controller from ever completing its initialization, rendering the entire unit non-functional.

This initial fault creates a devastating domino effect within a real-time operating system (RTOS) like FreeRTOS, which underpins the ESP32 Arduino core. The freezer controller's architecture correctly dedicates a specific RTOS task on a single CPU core (Core 0) to manage all time-critical control logic. If a blocking call within this task hangs, it does more than just stop the refrigeration logic; it starves the task scheduler. The control task can no longer update timers, process the main state machine, or, most critically, reset the system's supervisory watchdog

timers. The initial peripheral hardware fault rapidly cascades into a complete software lock-up on the control core, which is ultimately detected by a watchdog timer, forcing a system-wide reboot. This cycle of fault, hang, and reboot will continue indefinitely, preventing any useful operation. The core principle of a resilient design is, therefore, the complete elimination of indefinite blocking calls in favor of asynchronous, state-driven, and time-bounded operations.

1.2 The Four Pillars of Firmware Resilience

To systematically address these vulnerabilities, this report proposes a multi-layered defense strategy built upon four core principles. Each pillar provides a specific layer of protection, creating a system that is robust by design.

1. **Non-Blocking Initialization:** The firmware's startup sequence must be designed to probe for the presence and health of all required peripherals using strict, non-negotiable timeouts. The system must be able to determine the status of every component without the risk of hanging if a device is disconnected or faulty.
2. **Asynchronous Runtime Operation:** The main control loop must be architected as a non-blocking state machine. It must never wait for a peripheral to complete an operation. Instead, it will initiate an operation (e.g., a temperature conversion), and then continue executing other logic. It will check for the result of the operation on a subsequent pass of the loop, effectively polling for completion without ever halting.
3. **Granular Fault State Management:** The system must maintain a real-time "health status" for every individual component. This allows the firmware to make intelligent decisions based on the specific nature of a failure, such as using a timed backup for a defrost cycle if the evaporator sensor fails, rather than shutting down completely.
4. **System-Level Supervision:** A hardware-based Task Watchdog Timer (TWDT) will be implemented as the ultimate failsafe. This watchdog's purpose is to detect and recover from unforeseen software hangs (e.g., an infinite loop caused by a logic bug) that are not caught by the peripheral management logic, ensuring the system can always recover to a known state.

Section 2: Robust Initialization and Fault Detection for I2C Peripherals (PCF8574)

The Inter-Integrated Circuit (I2C) bus is a common multi-device communication protocol, but its shared-wire nature makes it susceptible to failures where a single misbehaving device can disable the entire bus. For the freezer controller, where the relay module may be driven by a PCF8574 I/O expander, ensuring robust I2C communication is paramount.

2.1 I2C Bus Failure Modes and Detection

The I2C protocol is a master-driven handshake system. The master (ESP32) initiates communication by sending a slave device's address. A functional slave at that address is expected to respond with an acknowledgment (ACK) signal. The Arduino Wire.h library provides a mechanism to detect this handshake: the Wire.endTransmission() function returns a status code. A return value of 0 indicates a successful transmission where the slave acknowledged its address. Non-zero values indicate specific failures: 2 for an address NACK (No Acknowledgment), meaning no device responded at that address; 3 for a data NACK; and 4 for

other, less common errors. This return code is the primary tool for detecting a disconnected or logically failed device in a non-blocking manner.
However, a more severe failure mode exists: a physical short circuit on the SDA or SCL lines to ground or VCC. In this scenario, the I2C hardware peripheral on the ESP32 may be physically unable to drive the bus to the required states to even transmit the address. A naive implementation of Wire.beginTransmission() could block indefinitely, waiting for the bus to become available. This is where a hardware timeout becomes the critical first line of defense. The ESP32's Wire library implementation includes the setTimeOut() function, which configures the underlying hardware driver's timeout in milliseconds. By setting a reasonable timeout (e.g., 50 ms) immediately after initializing the I2C bus with Wire.begin(), any I2C transaction that cannot complete within that timeframe will be aborted by the hardware driver, and the function will return an error instead of hanging the software. This single configuration step is essential to protect against the most catastrophic bus-level failures.
Beyond bus communication, the PCF8574 itself has known failure modes. It is particularly susceptible to electrical noise and power supply transients, which can cause the device to "glitch" and reset its outputs to their default high state. This is especially relevant when the expander is used to control relays switching inductive loads, as the collapsing magnetic fields can induce significant voltage spikes on the power lines. While hardware solutions like bypass capacitors and proper grounding are the primary mitigation for this, the firmware must be able to detect if the PCF8574 has stopped responding to I2C commands entirely.

2.2 Implementation: A Timeout-Based I2C Probe Function

To ensure the ESP32 can boot and function regardless of the PCF8574's state, a dedicated, non-blocking probe function must be used during initialization. This function leverages the principles of timeouts and return code checking to safely determine the device's status.
The following code provides a blueprint for a robust setup() sequence and a reusable probe function. This code should be integrated into the main firmware to manage the I2C bus and any connected PCF8574 devices.

```
#include <Wire.h>

// --- Global State and Configuration ---

// Define the I2C address for the PCF8574.
// This address depends on the A0, A1, A2 pin configuration. 0x27 is a
common default.
#define PCF8574_ADDRESS 0x27

// Define a fault flag for the PCF8574.
// This will be set to true if the device fails to respond during
initialization or runtime checks.
bool fault_pcf8574 = false;

// Define a fault code for HMI display
const char* FAULT_CODE_I2C = "I2C_FAULT";

/**
 * @brief Probes the I2C bus for a device at a given address with a
```

```
timeout.
 * This function is non-blocking and safe to call even if the I2C bus
is shorted.
 * It relies on the global I2C timeout set by Wire.setTimeOut().
 *
 * @param address The 7-bit I2C address of the device to check.
 * @return true if the device acknowledged its address, false
otherwise.
 */
bool probeI2CDevice(uint8_t address) {
    byte error;
    // The Wire.beginTransmission() function initiates a transaction.
    Wire.beginTransmission(address);

    // Wire.endTransmission() sends the data and waits for an ACK.
    // It returns 0 on success (ACK received).
    // It will not hang indefinitely due to the hardware timeout.
    error = Wire.endTransmission();

    if (error == 0) {
        // Device is present and responding correctly.
        Serial.printf("I2C device found at address 0x%02X\n",
address);
        return true;
    } else {
        // Device is not present or there was a bus error.
        Serial.printf("Error probing I2C address 0x%02X. Error code:
%d\n", address, error);
        return false;
    }
}

/**
 * @brief Initializes the I2C bus and probes for the PCF8574 I/O
expander.
 * This function should be called from the main setup() routine.
 */
void setup_i2c_devices() {
    // Initialize the I2C bus. On the Waveshare ESP32-S3 board, the
default I2C pins
    // may need to be re-assigned depending on the specific hardware
revision and wiring.
    // Assuming default pins for this example.
    Wire.begin();

    // CRITICAL: Set a hardware timeout for all I2C operations.
    // This prevents the Wire library from hanging indefinitely on a
locked bus.
```

```
    // 50ms is a generous but safe timeout.
    Wire.setTimeOut(50); // Set timeout to 50 milliseconds

    // Probe for the PCF8574 I/O expander.
    if (!probeI2CDevice(PCF8574_ADDRESS)) {
        // If the probe fails, set the global fault flag.
        fault_pcf8574 = true;

        // The HMI can now be updated to show the FAULT_CODE_I2C.
        // The main control logic will see the `fault_pcf8574` flag
and know
        // not to attempt any relay operations via the expander.
        Serial.println("FATAL: PCF8574 I/O expander not found.
Entering fault mode for actuators.");
    } else {
        // Device found, clear the fault flag.
        fault_pcf8574 = false;
        Serial.println("PCF8574 I/O expander initialized
successfully.");
        // Further initialization of the PCF8574 library can proceed
here.
    }
}
```

Section 3: Fault-Tolerant 1-Wire Bus Management (DS18B20 Sensors)

The 1-Wire bus, while simplifying wiring, introduces its own set of unique failure modes that require a multi-stage detection and handling strategy. A robust implementation must validate the bus integrity at the physical layer, handle communication errors for each individual sensor, and perform all operations asynchronously to prevent blocking the main control loop.

3.1 1-Wire Bus and DS18B20 Failure Analysis

Fault detection for the DS18B20 sensor network is a layered process, moving from the general health of the bus down to the validity of data from a single sensor.

1. **Physical Bus Integrity:** The most fundamental operation on the 1-Wire bus is the reset command. The bus master pulls the line low for a specific duration, and in response, all connected slave devices should pull the line low to signal their presence. The OneWire::reset() function encapsulates this and returns true if a "presence pulse" is detected, and false otherwise. A false return value indicates a catastrophic bus failure. This can be caused by an open circuit on the data line, a short circuit to ground (which prevents the pull-up resistor from ever raising the line), or a total power failure to all sensors. Performing this check at startup is the first and most crucial step; if the bus itself is dead, attempting to search for devices or request temperatures is futile and risks a software hang.

2. **Device Communication Errors:** Even on a healthy bus, communication with an individual sensor can fail. The DallasTemperature library reports these failures by returning specific "magic numbers." The most common is -127.0 (defined as DEVICE_DISCONNECTED_C), which is returned if the library fails to read the sensor's scratchpad memory or if the cyclic redundancy check (CRC) on the received data is invalid. This can be caused by intermittent connections, excessive bus noise, or insufficient power, especially in parasite-powered configurations.
3. **Sensor Power-On-Reset State:** According to the DS18B20 datasheet, the internal temperature register is initialized to a value of +85°C upon power-on. If a temperature reading is requested before the sensor has had time to perform its first valid conversion, it may return this default value. This is a common occurrence if a sensor briefly loses power and resets. While +85°C is a physically possible temperature, it is far outside the operational range of a commercial freezer.

A truly resilient system must treat both -127°C and +85°C as invalid readings. Upon detecting either value from a specific sensor, the firmware should immediately discard the reading, flag that specific sensor as faulty (e.g., set fault_p2 = true), and continue to use the last known-good temperature for that sensor in its control algorithms. This prevents a transient glitch from injecting a wildly incorrect temperature into the system, which could cause the controller to make a dangerously wrong decision (e.g., shutting off the compressor because it incorrectly reads +85°C).

3.2 Implementation: An Asynchronous Temperature Reading State Machine

To prevent the 750 ms blocking delay associated with temperature conversions, the firmware must adopt a non-blocking, asynchronous approach. The DallasTemperature library facilitates this with the setWaitForConversion(false) method. When this mode is active, the requestTemperatures() function returns immediately after sending the conversion command. The firmware is then responsible for waiting the appropriate amount of time before calling getTempC() to retrieve the results. This waiting period must be managed using a millis()-based timer to avoid blocking.

The following code provides a complete blueprint for a non-blocking temperature management module. It includes the layered fault detection strategy and the asynchronous state machine for reading values.

```
#include <OneWire.h>
#include <DallasTemperature.h>

// --- Global State and Configuration ---

#define ONE_WIRE_BUS_PIN 15 // GPIO pin for the 1-Wire bus
#define TEMP_RESOLUTION 12  // 12-bit resolution for DS18B20 sensors

// OneWire and DallasTemperature library instances
OneWire oneWire(ONE_WIRE_BUS_PIN);
DallasTemperature sensors(&oneWire);

// Device addresses for the four probes (to be populated by a
```

```
discovery routine)
DeviceAddress p1_addr, p2_addr, p3_addr, p4_addr; // Example addresses

// Fault flags for the bus and individual probes
bool fault_onewire_bus = false;
bool fault_p1 = false;
bool fault_p2 = false;
bool fault_p3 = false;
bool fault_p4 = false;

// Fault codes for HMI display
const char* FAULT_CODE_1W_BUS = "1W_BUS_FAULT";
const char* FAULT_CODE_P1F = "P1F";
const char* FAULT_CODE_P2F = "P2F";
const char* FAULT_CODE_P3F = "P3F";
const char* FAULT_CODE_P4F = "P4F";

// State machine for non-blocking temperature reads
enum TempReaderState {
    TEMP_IDLE,
    TEMP_CONVERSION_IN_PROGRESS
};
TempReaderState tempState = TEMP_IDLE;

// Timer variables for non-blocking delay
unsigned long lastTempRequestTime = 0;
const int CONVERSION_TIMEOUT_MS = 750; // Timeout for 12-bit
resolution

// Variables to store last known good temperatures
float temp_p1 = -999.0, temp_p2 = -999.0, temp_p3 = -999.0, temp_p4 =
-999.0;


/**
 * @brief Initializes the 1-Wire bus and discovers DS18B20 sensors.
 * Performs a bus health check before proceeding.
 */
void setup_onewire_devices() {
    // Start the library
    sensors.begin();

    // 1. Physical Bus Health Check
    if (!oneWire.reset()) {
        Serial.println("FATAL: 1-Wire bus fault detected (no presence
pulse).");
        fault_onewire_bus = true;
        // Do not proceed with sensor discovery if the bus is dead.
```

```
        return;
    }
    fault_onewire_bus = false;

    // 2. Device Discovery (Commissioning)
    // A full implementation would include a routine to search for all
sensors
    // and allow the user to assign them to P1, P2, etc., storing the
addresses in NVS.
    // For this example, we assume addresses are known and hardcoded.
    // e.g., get_p1_address_from_nvs(&p1_addr);

    // Configure sensors
    sensors.setResolution(TEMP_RESOLUTION);
    sensors.setWaitForConversion(false); // Enable non-blocking mode

    Serial.println("1-Wire devices initialized successfully.");
}

/**
 * @brief Reads a temperature from a specific sensor address and
validates it.
 *
 * @param address The DeviceAddress of the sensor.
 * @param last_good_temp Reference to the variable holding the last
valid temperature.
 * @param fault_flag Reference to the fault flag for this sensor.
 */
void read_and_validate_temp(DeviceAddress address, float
&last_good_temp, bool &fault_flag) {
    float tempC = sensors.getTempC(address);

    // 3. Data Validation
    if (tempC == DEVICE_DISCONNECTED_C |

| tempC == 85.0) {
        // Error detected. Do not update the temperature value. Set
the fault flag.
        if (!fault_flag) { // Log the error only on the first
occurrence
            Serial.printf("Error reading sensor. Value: %.2f\n",
tempC);
        }
        fault_flag = true;
    } else {
        // Valid reading. Update the temperature and clear the fault
flag.
        last_good_temp = tempC;
```

```
        if (fault_flag) { // Log the recovery
            Serial.println("Sensor has recovered.");
        }
        fault_flag = false;
    }
}

/**
 * @brief Non-blocking function to manage temperature sensor readings.
 * This should be called in every iteration of the main control loop.
 */
void update_temperatures() {
    // Do not attempt any 1-Wire operations if the bus is in a fault
state.
    if (fault_onewire_bus) {
        return;
    }

    switch (tempState) {
        case TEMP_IDLE:
            // It's time to start a new temperature conversion for all
sensors.
            sensors.requestTemperatures();
            lastTempRequestTime = millis();
            tempState = TEMP_CONVERSION_IN_PROGRESS;
            break;

        case TEMP_CONVERSION_IN_PROGRESS:
            // Check if the conversion time has elapsed.
            if (millis() - lastTempRequestTime >=
CONVERSION_TIMEOUT_MS) {
                // Time is up, read the values from the scratchpad.
                read_and_validate_temp(p1_addr, temp_p1, fault_p1);
                read_and_validate_temp(p2_addr, temp_p2, fault_p2);
                read_and_validate_temp(p3_addr, temp_p3, fault_p3);
                read_and_validate_temp(p4_addr, temp_p4, fault_p4);

                // Return to idle state to await the next reading
cycle.
                tempState = TEMP_IDLE;
            }
            break;
    }
}
```

Section 4: Supervising the Control Logic with the Task

Watchdog Timer (TWDT)

While robust peripheral handling prevents hangs from predictable hardware faults, the Task Watchdog Timer (TWDT) serves as the ultimate safety net. It is a hardware-based supervisor designed to recover the system from unexpected and catastrophic software failures, such as a logic error causing an infinite loop or a third-party library bug that leads to a deadlock.

4.1 The Role of a Hardware Watchdog in an RTOS Environment

The ESP32's TWDT is specifically designed to integrate with the FreeRTOS environment. Its primary function is to detect if a task monopolizes a CPU core for too long without yielding control to the scheduler. In the freezer controller's dual-core architecture, the control_logic_task is pinned to Core 0. By "subscribing" this task to the TWDT, the firmware establishes a contract: the control task must periodically "feed" (or reset) the watchdog timer to signal that it is still executing correctly. If the task hangs for any reason and fails to feed the watchdog within a pre-configured timeout period, the TWDT's hardware will trigger a full system reset, forcing the controller back to a known-good state.
It is critical to distinguish the role of the TWDT from the peripheral fault-handling logic. The non-blocking I2C and 1-Wire handlers are the *first line of defense*, designed to prevent common hardware failures from ever causing a software hang. The TWDT is the *second line of defense*, a failsafe mechanism to catch what the first line misses. A well-designed system should never rely on the TWDT to recover from a disconnected sensor; the peripheral handlers should manage that gracefully. The TWDT's purpose is to guard against the "unknown unknowns"—unforeseen bugs or complex race conditions that could otherwise permanently disable the controller.

4.2 Implementation: Integrating the TWDT into the Control Task

Integrating the TWDT into the Arduino environment is straightforward using the ESP-IDF functions provided by the core framework. The process involves initializing the watchdog with a specific timeout, adding the control task to its watch list, and then periodically resetting it from within the task's main loop.
The following code provides a complete, self-contained example demonstrating how to configure and use the TWDT within the control_logic_task defined in the system architecture.

```
#include <Arduino.h>
#include <esp_task_wdt.h>

// --- Global State and Configuration ---

// Define the watchdog timeout in seconds.
// This should be longer than the longest expected legitimate
execution time of the control loop.
// A value of 3-5 seconds is typically a safe choice.
#define WDT_TIMEOUT_S 5

TaskHandle_t controlTaskHandle; // Handle for the control logic task
```

```
/**
 * @brief The main control logic task for the freezer.
 * This task runs in a continuous loop on Core 0.
 *
 * @param pvParameters Task parameters (not used).
 */
void control_logic_task(void* pvParameters) {
    Serial.println("Control Logic Task started on Core 0.");

    // Subscribe this task to the Task Watchdog Timer.
    // A call with NULL subscribes the currently running task.
    esp_task_wdt_add(NULL);
    esp_task_wdt_status(NULL); // Check subscription status

    // This is the main, infinite loop for the control logic.
    for (;;) {
        //
---------------------------------------------------------------------
        // 1. READ SENSORS (Non-blocking)
        // update_temperatures(); // As defined in Section 3.2

        // 2. EXECUTE STATE MACHINE
        // run_state_machine(); // As defined in the operational
report

        // 3. UPDATE ACTUATORS
        // update_relay_outputs(); // Based on state machine decisions

        // 4. SHARE STATE WITH UI
        // (Code to send data to UI task via a mutex-protected shared
structure)
        //
---------------------------------------------------------------------

        // 5. FEED THE WATCHDOG
        // This line is the most critical part of the watchdog
implementation.
        // It signals that the entire control loop has completed
successfully.
        // If any of the above functions hang, this line will not be
reached,
        // the watchdog will time out, and the system will reset.
        esp_task_wdt_reset();

        // Delay for the next cycle. This yields control to the RTOS
scheduler.
        vTaskDelay(pdMS_TO_TICKS(1000)); // Run control loop once per
second
```

```
    }
}

/**
 * @brief Standard Arduino setup function. Runs once on boot.
 */
void setup() {
    Serial.begin(115200);
    Serial.println("System Booting...");

    // Configure and initialize the Task Watchdog Timer.
    Serial.printf("Configuring TWDT with a %d second timeout...\n",
WDT_TIMEOUT_S);
    esp_task_wdt_config_t twdt_config = {
       .timeout_ms = WDT_TIMEOUT_S * 1000,
       .idle_core_mask = (1 << portNUM_PROCESSORS) - 1, // Watch idle
tasks on all cores
       .trigger_panic = true, // Trigger a panic (and reboot) on
timeout
    };
    esp_task_wdt_init(&twdt_config);

    // Initialize UI, peripherals, etc.
    // setup_ui();
    // setup_i2c_devices();
    // setup_onewire_devices();

    // Create and pin the control logic task to Core 0.
    xTaskCreatePinnedToCore(
        control_logic_task,   // Function to implement the task
        "ControlLogic",       // Name of the task
        10000,                // Stack size in words
        NULL,                 // Task input parameter
        1,                    // Priority of the task
        &controlTaskHandle,   // Task handle
        0                     // Pin to Core 0
    );

    // The loop() function will run on Core 1 for the UI.
}

/**
 * @brief Standard Arduino loop function. Runs continuously on Core 1.
 */
void loop() {
    // This core is dedicated to the HMI.
    // lv_timer_handler();
    delay(5);
```

```
}
```

Section 5: Integrating Fault States into the System Architecture

With robust detection mechanisms in place for each peripheral, the final step is to integrate this fault information into the controller's core logic. The system must be "fault-aware," meaning its operational state machine can dynamically alter its behavior based on the real-time health of its hardware components. This ensures that a single component failure results in a predictable, safe, and degraded mode of operation rather than a complete shutdown.

5.1 The Device Fault Status Structure

To centralize fault information, a dedicated data structure should be part of the main ControllerState struct. This structure will hold a boolean flag for each potential point of failure, providing a single source of truth for the entire application.

```
// This structure should be part of the main ControllerState struct
struct DeviceFaultStatus {
    bool pcf8574_fault;       // True if the I2C I/O expander is
unresponsive
    bool onewire_bus_fault;   // True if the entire 1-Wire bus is down
    bool probe_p1_fault;      // Cabin probe read error
    bool probe_p2_fault;      // Evaporator probe read error
    bool probe_p3_fault;      // Condenser probe read error
    bool probe_p4_fault;      // Suction line probe read error
};

// Example integration into the main state structure
struct ControllerState {
    float temp_p1, temp_p2, temp_p3, temp_p4;
    SystemState currentState;
    bool relay_state;
    DeviceFaultStatus faults; // Integrated fault status
    //... other timer and state variables
};
```

This faults structure is populated by the initialization routines (e.g., setup_i2c_devices) and updated in real-time by the runtime handlers (e.g., update_temperatures).

5.2 Creating a Fault-Aware State Machine

The core run_state_machine() function must be modified to consult the DeviceFaultStatus structure before making any state transitions or decisions. This makes the logic resilient to missing or invalid data.

Example 1: Handling Cabin Probe (P1) Failure The cabin probe is the primary input for thermostatic control. If it fails, the system must switch to its timed failsafe cycle (Cy/Cn).

Original Logic (Implicit): if (state.temp_p1 >= (params.St + params.Hy)) { state.currentState = SystemState::COOLING; }
Fault-Aware Logic:

```
// In state SystemState::IDLE:
if (state.faults.probe_p1_fault) {
    // P1 has failed, cannot use temperature for control.
    // Transition to the dedicated FAULT state.
    state.currentState = SystemState::FAULT;
} else if (state.temp_p1 >= (params.St + params.Hy) &&
ac_timer_expired) {
    // P1 is healthy, proceed with normal temperature-based control.
    state.currentState = SystemState::COOLING;
}
```

Example 2: Handling Evaporator Probe (P2) Failure The evaporator probe is used for temperature-terminated defrost. If it fails, the system must rely solely on the maximum defrost time (MdF) as a failsafe.
Original Logic (Implicit): `if (state.temp_p2 >= params.dtE |
| mdf_timer_expired) { exit_defrost_state(); }`
Fault-Aware Logic:

```
// In state SystemState::DEFROST:
bool terminate_defrost = false;
if (mdf_timer_expired) {
    // Failsafe time termination always applies.
    terminate_defrost = true;
} else if (!state.faults.probe_p2_fault) {
    // Only check temperature termination if the P2 probe is healthy.
    if (state.temp_p2 >= params.dtE) {
        terminate_defrost = true;
    }
}

if (terminate_defrost) {
    exit_defrost_state();
}
```

5.3 The Device Fault Code Dictionary

To provide clear diagnostics to the operator, the internal firmware fault flags must be mapped to the user-facing HMI codes specified in the system design. This mapping forms the basis for the alarm and fault display logic.

Firmware Fault Flag	HMI Display Code	Trigger Condition	System Response / Failsafe Action
faults.probe_p1_fault	"P1F"	DS18B20 at P1 address returns -127 or +85, or fails CRC	Enters FAULT state; compressor cycles based on Cy (ON time)

Firmware Fault Flag	HMI Display Code	Trigger Condition	System Response / Failsafe Action
		check.	and Cn (OFF time) parameters.
faults.probe_p2_fault	"P2F"	DS18B20 at P2 address returns -127 or +85, or fails CRC check.	Defrost terminates on time (MdF) only. Post-defrost fan delay uses timed failsafe (Fnd) instead of temperature (FSt).
faults.probe_p3_fault	"P3F"	DS18B20 at P3 address returns -127 or +85, or fails CRC check.	Disables high condenser temperature alarm (HCA).
faults.pcf8574_fault	"I2C_FAULT"	PCF8574 fails to acknowledge its I2C address during probe.	All relay outputs are disabled. No actuator control is possible. The system effectively halts refrigeration.
faults.onewire_bus_fault	"1W_BUS_FAULT"	oneWire.reset() returns false, indicating no presence pulse on the bus.	All temperature readings are considered invalid. Triggers faults for P1, P2, P3, and P4 simultaneously. System enters FAULT state.

5.4 Dynamic Fault Recovery

Hardware faults are not always permanent. A loose connection, transient electrical noise, or a temporary power sag to a peripheral can cause an intermittent failure. A truly advanced system should not require a manual reboot to recover from such events. The fault-tolerant architecture should include a mechanism for dynamic recovery by periodically and safely re-probing for failed devices.

This can be implemented with a simple millis()-based timer in the main control loop. For example, every 60 seconds, the code can check the status of any flagged faults. If faults.pcf8574_fault is true, it can re-run the non-blocking probeI2CDevice() function. If the device now responds successfully, the fault flag can be cleared, the "I2C_FAULT" message can be removed from the HMI, and the system can resume normal actuator control. Similarly, if faults.onewire_bus_fault is true, it can attempt another oneWire.reset(). If successful, it can clear the bus fault and allow the asynchronous temperature reading state machine to resume its operation. This transforms the system from being merely fault-tolerant to being self-recovering, significantly increasing its uptime and reliability in real-world conditions.

Works cited

1. Dallas Temperature libary non blocking question - Sensors - Arduino Forum, https://forum.arduino.cc/t/dallas-temperature-libary-non-blocking-question/153430 2. DS18B20

750ms delay bypass? - Sensors - Arduino Forum, https://forum.arduino.cc/t/ds18b20-750ms-delay-bypass/61650 3. ESP32 hangs at the line Wire.requestFrom() - Sensors - Arduino Forum, https://forum.arduino.cc/t/esp32-hangs-at-the-line-wire-requestfrom/1080993 4. Arduino IDE, ESP32, I2C Timeout problem - Networking, Protocols, and Devices, https://forum.arduino.cc/t/arduino-ide-esp32-i2c-timeout-problem/1040744 5. ESP32 I2C Issue: Interrupt wdt timeout on CPU0 - Arduino Forum, https://forum.arduino.cc/t/esp32-i2c-issue-interrupt-wdt-timeout-on-cpu0/694858 6. ESP32: I2C Scanner (Arduino) | Random Nerd Tutorials, https://randomnerdtutorials.com/esp32-i2c-scanner-arduino/ 7. I2C - - — Arduino ESP32 latest documentation - Espressif Systems, https://docs.espressif.com/projects/arduino-esp32/en/latest/api/i2c.html 8. PCF8574 relay glitches on HA IoT - Electrical Engineering Stack ..., https://electronics.stackexchange.com/questions/567434/pcf8574-relay-glitches-on-ha-iot 9. 1-Wire Search Algorithm - Analog Devices, https://www.analog.com/en/resources/app-notes/1wire-search-algorithm.html 10. M16C/26 APPLICATION NOTE Interfacing with 1-Wire. Devices - Renesas, https://www.renesas.com/kr/en/document/apn/interfacing-1-wire-devices 11. OneWire Library Bug / Proposed FIX (June 2019) Affects DS18B20 and other OneWire devices - Particle Community, https://community.particle.io/t/onewire-library-bug-proposed-fix-june-2019-affects-ds18b20-and-other-onewire-devices/50589 12. ds18b20 issue show -127°c - Sensors - Arduino Forum, https://forum.arduino.cc/t/ds18b20-issue-show-127-c/535268 13. ESP32 Environmental Sensor DS18B20 (Temperature) | by AndroidCrypto - Medium, https://medium.com/@androidcrypto/esp32-environmental-sensor-ds18b20-temperature-53dda02250a7 14. Can a temp sensor cause the Arduino to lock up? - Programming, https://forum.arduino.cc/t/can-a-temp-sensor-cause-the-arduino-to-lock-up/137688 15. DS18B20 - Programmable Resolution 1-Wire Digital Thermometer - Analog Devices, https://www.analog.com/media/en/technical-documentation/data-sheets/ds18b20.pdf 16. How to handle DallasTemperature 85 °C - Sensors - Arduino Forum, https://forum.arduino.cc/t/how-to-handle-dallastemperature-85-c/463988 17. DS18B20 temperature sensor sends 85 °C error value - OpenEnergyMonitor Community, https://community.openenergymonitor.org/t/ds18b20-temperature-sensor-sends-85-c-error-value/21573 18. Watchdogs - ESP32 - — ESP-IDF Programming Guide v5.5 documentation, https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/wdts.html