```
/*
==================================================
Project: SWHController6_5_6
Generated: Tue Mar 10 07:10:32 CST 2026
Files: 11
Total Lines: 999
Folder: SWHController6_5_6
==================================================
TABLE OF CONTENTS
   26 SWHController6_5_6.ino  349 buttonHelper.ino  412 LCD_Helper.h  448 SystemMode.h  471 button.h  491 pins.h
568 pump_control.h  597 temps.h  632 LCD_Helper.cpp  842 pump_control.cpp  892 temps.cpp
--------------------
TABLE OF CONTENTS
   11 SWHController6_5_6.ino
   12 buttonHelper.ino
   13 LCD_Helper.h
   14 SystemMode.h
   15 button.h
   16 pins.h
   17 pump_control.h
   18 temps.h
   19 LCD_Helper.cpp
   20 pump_control.cpp
   21 temps.cpp
*/


// ===== ARDUINO SKETCH =====

#pragma region SWHController6_5_6.ino
#define VERSION "Version 6_5_6"
#define ARDUINO_AVR_NANO_EVERY
/*
SWHController_v6_5_6_STABLE`
  // HARDWARE REV: PCB v2.2.1  -- Modified for LCD connector and pin assignement - use FW v6_5_3 or higher
  // HARDWARE REV: PCB v2.1.1  -- includes a series 100 Ohm resistor for the ADC lines and 1k0 inline with digital switch
  // FIRMWARE BASELINE  v6_5_6 -- Improve the Hysteresis with routine -  shouldPumpRun(bool pumpRunning)
  // FIRMWARE BASELINE  v6_5_5 -- Frost alarm clearing will drop Override into Auto mode and extended serial update for
Button A
  // FIRMWARE BASELINE  v6_5_4 -- Parameters reporting to LCD on startup
  // FIRMWARE BASELINE  v6_5_3 -- LCD Data pins reassigned for PCB layout - reqs HW 2.2.1 or higher
  // FIRMWARE BASELINE: v6_5_2 -- adds a safe ESD approach for unused pins
  // FIRMWARE BASELINE: v6_5_1 -- modifies the algorith to correct for the 100 Ohm resistor

### Digital I/O

* `PIN_PUMP` → relay / driver input
* `BUTTON_A` → pull-up, active LOW
* `LED_FROST` → output (currently unused)
*  MCU to LCD pins: '12-D9-E, 11-D8-RS, 9-D6-DB4, 8-D5-DB5, 7-D4-DB6, 6-D3-DB7, 5-D2-Button'

* ======================== LCD ========================
 * 4-bit HD44780 interface
 *
 * RS  -> D8
 * E   -> D9
 * D4  -> D6
 * D5  -> D5
 * D6  -> D4
 * D7  -> D3
 *
 * PCB: ControllerPCB Rev Jan2026v2
 *
```



```

### Electrical assumptions

* Button uses **INPUT_PULLUP**
* Pump relay logic level (**HIGH = ON to NFET)
* LCD is 4-bit parallel, not I²C
* Timing relies on `millis()` (no blocking)

*/

/* ======== SWHController.ino ========== */
#include <Arduino.h>
#include "pins.h"
#include <LiquidCrystal.h>
#include <math.h>
#include "button.h"
#include "LCD_Helper.h"
#include "SystemMode.h"
#include "temps.h"
#include "pump_control.h"

/* --------- Forward declaration --------- */
void clearPumpFault(PumpState &p, unsigned long now);

/* ======================== LCD ======================== */

// LCD pin mapping per PCB rev Jan2026v2
// lcd(rs, enable, d4, d5, d6, d7)
// initialization according to pins.h file
LiquidCrystal lcd(
  PIN_LCD_RS,  // RS -> D8
  PIN_LCD_EN,  // E  -> D9
  PIN_LCD_D4,  // D4 -> D6
  PIN_LCD_D5,  // D5 -> D5
  PIN_LCD_D6,  // D6 -> D4
  PIN_LCD_D7   // D7 -> D3
);

/* ================== Message Strings ================== */

const char msgStarting[] PROGMEM = "Starting    ";
const char msgAuto[] PROGMEM = "Auto        ";
const char msgOverride[] PROGMEM = "Override    ";
const char msgBackup[] PROGMEM = "Backup      ";

const char *const MessageTable[MSG_COUNT] PROGMEM = {
  msgStarting,
  msgAuto,
  msgOverride,
  msgBackup
};

button::State btnState;

MessageId messageForMode(SystemMode mode) {
  switch (mode) {
    case MODE_AUTO:
      return MSG_AUTO;
    case MODE_MANUAL:
      return MSG_OVERRIDE;
    case MODE_BACKUP:
      return MSG_BACKUP;
    default:
      return MSG_STARTING;
```

```
}
}
// Function to print data on the LCD, clearing the line first
void printLCDLine(int line, const char *label, int value) {
  lcd.setCursor(0, line);          // Set cursor to the beginning of the specified line
  lcd.print("                ");   // Print spaces to clear the line (16 spaces for a 16x2 LCD)
  lcd.setCursor(0, line);          // Reset cursor to the beginning of the line
  lcd.print(label);                // Print the label
  if (value > 0) lcd.print(value);  // Print the value if it is there
  if (line == 1) delay(2000);      // give some time to read it
}
/* ============== Setup ================== */
void setup() {
  pinMode(BUTTON_A, INPUT_PULLUP);
  pinMode(PIN_PUMP, OUTPUT);
  pinMode(LED_FROST, OUTPUT);
  digitalWrite(LED_FROST, LOW);  // not used in this version

#include <LiquidCrystal.h>;

  lcd.begin(16, 2);  // Initialize the LCD with 16 columns and 2 rows
  lcd.clear();

  // Display version on the second line
  printLCDLine(1, VERSION, 0);

  // Print Diff On and Diff Off
  printLCDLine(0, "Temp Diff On: ", CYL_DIFF_ON_x10 / 10);   // First line: Diff On
  printLCDLine(1, "Temp Diff Off: ", CYL_DIFF_OFF_x10 / 10);  // Second line: Diff Off

  // Print Frost Warning and Cycle Time
  printLCDLine(0, "Frost : temp ", FROST_TEMP_x10 / 10);              // First line: Frost Temp
  printLCDLine(1, "Cycle t> secs ", (MIN_ON_MS + MIN_OFF_MS) / 1000);  // Second line: Cycle time

  // Print Overrun limit hours
  printLCDLine(0, "Overrun limit ", 0);              // First line: Max time for no fault
  printLCDLine(1, "hrs : ", (MAX_ON_MS / 3600000));  // Second line: Overrun time

  temps::begin();  // Prefill the array of temperatures
  lcd.clear();     // Clear the screen for the next phase

  Serial.begin(115200);
  while (!Serial) { ; }  // safe on USB boards, harmless on UNO
  Serial.println(F("SWHController debug start"));
  initUnusedPins();
}
/*================ Main Loop ==============*/
void loop() {
  static SystemMode mode = MODE_AUTO;    // Initial state of the controller
  static PumpState pump;                 // Pump controller state
  static bool faultLock = false;         // Latched until user clears it
  static bool frostAlarm, cold = false;  // latched if the Frost alsrm has triggered - if the Alarm clears switch to Auto

  LCD_::DisplayModifiers mods{};
  bool requestOn = false;
  const unsigned long now = millis();  // this is the time for this loop iteration
  temps::update();

  cold = temps::frostProtectionRequired();

  if (temps::hasChanged()) {
    const auto &t = temps::avg();
    LCD_::renderBottomRow(
      lcd,
      t.tin_x10,
      t.tcyl_x10,
      t.tstag_x10);
  }

  /* ---------- Read button ---------- */
  if (button::pressed(btnState, now)) {
    clearPumpFault(pump, now);
    faultLock = false;

    switch (btnState) {
      case button::SHORT:
        mode = MODE_MANUAL;
        break;

      case button::LONG:
        mode = MODE_BACKUP;
        break;

      case button::TAP:
      default:
        mode = MODE_AUTO;
        break;
    }
  }

  /* ---------- Decide request ---------- */
  if (!faultLock) {
    switch (mode) {
      case MODE_MANUAL:
        requestOn = true;

        if (cold) frostAlarm = true;  // If the controller has been set for Override (maybe to protect from freezing) and the frost alarm is present or arises,
                                      // the pump will cool the tank and heat the roof - if the roof heats up the system reverst to Auto and Override is cancelled.
        if (!cold && frostAlarm) {
          frostAlarm = false;
          mode = MODE_AUTO;
        }

        break;

      case MODE_AUTO:
        requestOn = temps::shouldPumpRun(digitalRead(PIN_PUMP));
        // requestOn = temps::shouldPumpRun(requestOn);
        break;

      default:
        requestOn = false;
        break;
    }
  }

  /* ---------- Pump control ---------- */
  bool pumpOn = updatePump(pump, requestOn, now);
  digitalWrite(PIN_PUMP, pumpOn);

  if (pump.maxRunFault) {
    faultLock = true;
    mode = MODE_AUTO;  // force auto
  }

  /* ---------- Display modifiers ---------- */
```

```
  mods.scroll = digitalRead(PIN_PUMP);
  mods.blink = !temps::sensor_inRange();
  mods.caps = (digitalRead(BUTTON_A) == LOW);
  mods.frost = cold;
  mods.lock = pump.lock;  // hold lock indication
  mods.lockBlink = pump.maxRunFault;

  MessageId msg = messageForMode(mode);
  LCD_::renderTopRow(lcd, msg, MessageTable, mods);

  // ---------- Serial debug ----------
static unsigned long lastPrint = 0;
if (now - lastPrint >= 1000) {
  lastPrint = now;

  // Print the current time
  Serial.print(F("now="));
  Serial.print(now);

  // Print the pump status
  Serial.print(F(" pump="));
  Serial.print(pump.running ? F("ON") : F("OFF"));

  // Print the request status
  Serial.print(F(" req="));
  Serial.print(requestOn ? F("ON") : F("OFF"));

  // Print the lock status
  Serial.print(F(" lock="));
  Serial.print(pump.lock ? F("HOLD") : F(" Go "));

  // Print the overrun status
  Serial.print(F("  "));
  Serial.print(pump.maxRunFault ? F("Overrun") : F(" -- "));

  // Print the button state
  Serial.print(F(" BtnA="));

  // Improved switch statement for button state
  const char* BtnAtext;  // Declare BtnAtext as a pointer to const char
  switch (btnState) {
    case button::SHORT:
      BtnAtext = " short ";
      break;

    case button::LONG:
      BtnAtext = " long ";
      break;

    case button::TAP:
      BtnAtext = " tap  ";
      break;

    case button::STUCK:
      BtnAtext = " stuck ";
      break;

    case button::OFF:
      BtnAtext = " off  ";
      break;

    default:
      BtnAtext = " -- ";  // Default case when button state is unknown
      break;
```



```
  }
  Serial.print(BtnAtext);  // Print the button state

  // Print the lock status again (if needed)
  Serial.print(pump.lock ? F("HOLD") : F(" Go "));

  Serial.println();  // End of line
}

}

/* ---------- Clear pump fault ---------- */

void clearPumpFault(PumpState &p, unsigned long /*now*/) {
  p.maxRunFault = false;
  p.lock = false;
  // DO NOT touch lastChange or onStart here
}

// Unused MCU pins are driven OUTPUT LOW to minimise ESD susceptibility
void initUnusedPins() {
  const uint8_t unusedDigitalPins[] = UNUSED_DIGITAL_PINS;
  for (uint8_t pin : unusedDigitalPins) {
    pinMode(pin, OUTPUT);
    digitalWrite(pin, LOW);
  }

  const uint8_t unusedAnalogPins[] = UNUSED_ANALOG_PINS;
  for (uint8_t pin : unusedAnalogPins) {
    pinMode(pin, OUTPUT);
    digitalWrite(pin, LOW);
  }
}
#pragma endregion

#pragma region buttonHelper.ino
/* ==== buttonHelper.ino ==== */
#include "button.h"

namespace button
{
  const unsigned long DEBOUNCE_TIME_MS = 100; // in millisecs
  const unsigned long SHORT_TIME_MS = 1000;
  const unsigned long LONG_TIME_MS = 5000;
  const unsigned long STUCK_TIME_MS = 10000;
  // This routine is non blocking so we need to s=track when the button was pressed

  bool pressed(State &state, unsigned long B_now)
  {
    static unsigned long firstPressedMs = 0;
    const bool isPressed = (digitalRead(BUTTON_A) == LOW);
    //const unsigned long now = millis();
    //  unsigned long now = millis();
    unsigned long now;
  now = B_now; // ....

    if (!isPressed)
    {
      firstPressedMs = now; // reset the timer for the button if not set
      state = OFF;
      return false;
    }

    // So button isPressed (is TRUE)
```

```
385
386     if (firstPressedMs == 0)
387     {
388       firstPressedMs = now;
389       state = OFF;
390       return false;
391     }
392
393     const unsigned long heldTime = now - firstPressedMs;
394     // Button is down so is it debounced?
395     if (heldTime < DEBOUNCE_TIME_MS)
396     {
397       state = OFF;
398       return false;
399     }
400     /* Button is debounced and pressed so measure the length of time
401     // A held button will return each of the following as is meets the criteria for the time held.
402     */
403     if (heldTime >= STUCK_TIME_MS)
404       state = STUCK;
405     else if (heldTime >= LONG_TIME_MS)
406       state = LONG;
407     else if (heldTime >= SHORT_TIME_MS)
408       state = SHORT;
409     else
410       state = TAP;
411
412     return true;
413   }
414 }
415 #pragma endregion
416
417 // ===== HEADERS =====
418
419 #pragma region LCD_Helper.h
420 /* ========== LCD_Helper.h ============ */
421 #pragma once
422 #include <LiquidCrystal.h>
423 #include <stdint.h>
424 #include "SystemMode.h"
425
426 namespace LCD_
427 {
428     struct DisplayModifiers
429     {
430         bool scroll;
431         bool blink;
432         bool caps;
433         bool lock;          // Override active
434         bool frost;         // Frost protection active
435         bool lockBlink;     // Max-run fault (flash lock only)
436     };
437
438     void renderTopRow(
439         LiquidCrystal &lcd,
440         MessageId msg,
441         const char *const *messageTable,
442         const DisplayModifiers &mods);
443
444     void renderBottomRow(
445         LiquidCrystal &lcd,
446         int16_t tin_x10,
447         int16_t tcyl_x10,
448         int16_t tstag_x10);
```



```
449
450     void printTempPos_x10(LiquidCrystal &lcd, int16_t t);
451     void printTempSigned_x10(LiquidCrystal &lcd, int16_t t);
452 }
453 #pragma endregion
454
455 #pragma region SystemMode.h
456 /* ============ SystemMode.h ============== */
457 #pragma once
458 #include <stdint.h>
459
460 enum SystemMode : uint8_t
461 {
462     MODE_STARTING = 0,
463     MODE_AUTO,
464     MODE_MANUAL,
465     MODE_BACKUP
466 };
467
468 enum MessageId : uint8_t
469 {
470     MSG_STARTING = 0,
471     MSG_AUTO,
472     MSG_OVERRIDE,
473     MSG_BACKUP,
474     MSG_COUNT
475 };
476 #pragma endregion
477
478 #pragma region button.h
479 /* ========= button.h ============ */
480 #pragma once
481 #include <Arduino.h>
482
483 namespace button
484 {
485   enum State : uint8_t
486   {
487     OFF = 0,
488     TAP,
489     SHORT,
490     LONG,
491     STUCK
492   };
493
494   bool pressed(State &state, unsigned long B_now);
495 }
496 #pragma endregion
497
498 #pragma region pins.h
499 #pragma once
500 /*
501  * pins.h
502  * ------------------------------------------------------------
503  * Project: Solar Water Heater Controller
504  * MCU:     Arduino Nano Every (ATmega4809)
505  * PCB:     ControllerPCB Rev Jan2026v2 / HW v2.2.1+
506  *
507  * This file is AUTHORITATIVE.
508  * Any pin change must be reflected here and in the schematic.
509  * ------------------------------------------------------------
510  */
511
512 /* ====================== Board sanity ====================== */
```

```
#if !defined(ARDUINO_AVR_NANO_EVERY)
  #error "This firmware is intended for Arduino Nano Every only"
#endif

/* ====================== LCD (HD44780, 4-bit) ====================== */
/*
 * LCD Pinout (HD44780):
 *  RS  -> D8
 *  E   -> D9
 *  D4  -> D6
 *  D5  -> D5
 *  D6  -> D4
 *  D7  -> D3
 */
#define PIN_LCD_RS   8
#define PIN_LCD_EN   9
#define PIN_LCD_D4   6
#define PIN_LCD_D5   5
#define PIN_LCD_D6   4
#define PIN_LCD_D7   3

/* ====================== User Inputs ====================== */
// Active LOW, internal pull-up enabled
#define BUTTON_A     2

/* ====================== Outputs ====================== */
// Pump relay / NFET gate (HIGH = ON)
#define PIN_PUMP     7

// Frost LED (currently unused, driven LOW)
#define LED_FROST    10

/* ====================== Serial ====================== */
#define PIN_SERIAL_RX 0
#define PIN_SERIAL_TX 1

/* ====================== Analog Sensors ====================== */
// Temperature sensors handled in temps.h
// Analog pins not explicitly listed here are unused

/* ====================== Unused Pins (EMI / ESD hardened) ====================== */
/*
 * These pins are NOT connected on the PCB.
 * They are driven OUTPUT LOW during init to avoid floating nodes.
 */
#define UNUSED_DIGITAL_PINS { \
  11, /* MOSI */              \
  12, /* MISO */              \
  13  /* SCK  */              \
}

#define UNUSED_ANALOG_PINS { \
  A0                         \
}

/* ====================== Safety checks ====================== */
static_assert(PIN_LCD_RS != PIN_LCD_EN, "LCD RS and EN cannot share a pin");
static_assert(PIN_LCD_D4 != PIN_LCD_D5, "LCD data pins must be unique");
static_assert(PIN_LCD_D5 != PIN_LCD_D6, "LCD data pins must be unique");
static_assert(PIN_LCD_D6 != PIN_LCD_D7, "LCD data pins must be unique");
#pragma endregion

#pragma region pump_control.h
/* ========== pump_control.h ============ */
```



```
#pragma once
// #define TEST_MODE   // <-- MUST be before the #ifdef

#include <Arduino.h>
constexpr unsigned long MIN_ON_MS = 10000UL;   // 10 s - if pump starts it must run for at lease ten seconds
constexpr unsigned long MIN_OFF_MS = 15000UL;  // 15 s - if pump stops it must stop for a least 15 seconds
// therefore min Cycle time is (MIN_ON_MS + MIN_OFF_MS) / 1000 seconds

#ifdef TEST_MODE
constexpr unsigned long MAX_ON_MS = 30UL * 1000;  // 30 seconds
#else
constexpr unsigned long MAX_ON_MS = 12UL * 60 * 60 * 1000;  // 12 hours
#endif

struct PumpState {
  bool running = false;          // actual pump output state
  unsigned long lastChange = 0;  // time we entered the current state (ON or OFF)
  unsigned long onStart = 0;     // time pump last started
  bool maxRunFault = false;      // latched 12h overrun fault
  bool lock = false;             // min on/off or lock
};


bool updatePump(PumpState &ps,
                bool requestOn,  // from temps::shouldPumpRun()
                unsigned long now);#pragma endregion

#pragma region temps.h
/* ============ temps.h ================ */
#pragma once
#include <stdint.h>

// Pump thresholds (×10 °C)
constexpr int16_t CYL_DIFF_ON_x10 = 120;   // 12 °C
constexpr int16_t CYL_DIFF_OFF_x10 = 40;   // 4 °C
constexpr int16_t FROST_TEMP_x10 = 40;    // 4.0 °C

constexpr int16_t maxTemp = 1200;
constexpr int16_t minTemp = -300;

namespace temps
{
    struct Averages
    {
        int16_t tin_x10;
        int16_t tout_x10;
        int16_t tcyl_x10;
        int16_t tstag_x10;
    };

    void begin();
    void update();
    bool hasChanged();
    const Averages &avg();
    bool shouldPumpRun(bool pumpRunning);
    bool sensor_inRange();
    bool frostProtectionRequired();
}
#pragma endregion

// ===== SOURCE FILES =====

#pragma region LCD_Helper.cpp
/* ========== LCD_Helper.cpp =============== */
```

```
#include <Arduino.h>  // MUST be first
#include "LCD_Helper.h"
#include <math.h>
#include "temps.h"

// LCD_Helper.cpp (top of file, outside namespace)
static const uint8_t ICON_LOCK[8] = {
  0b01110, 0b10001,
  0b10001, 0b11111,
  0b11011, 0b11011,
  0b11111, 0b00000
};

static const uint8_t ICON_FROST[8] = {
  0b00100, 0b10101,
  0b01110, 0b10101,
  0b00100, 0b10101,
  0b01110, 0b00000
};

namespace LCD_

{
static bool iconsLoaded = false;

static void loadIcons(LiquidCrystal &lcd) {
  lcd.createChar(0, (uint8_t *)ICON_LOCK);
  lcd.createChar(1, (uint8_t *)ICON_FROST);
  iconsLoaded = true;
}

void renderTopRow(
  LiquidCrystal &lcd,
  MessageId msg,
  const char *const *messageTable,
  const DisplayModifiers &mods) {
  static uint8_t scrollIndex = 0;
  static unsigned long lastScroll = 0;
  const unsigned long SCROLL_MS = 300;
  constexpr unsigned long REFRESH_MS = 99;
  static unsigned long last_time = 0;

  static char shadow[17] = "                ";  // clear what LCD currently shows

  char text[24];
  char out[17];
  /*
  [ TEXT TEXT TEXT TEXT ][L][F]
  0.................13 14 15
  last the spaces reserved for icons Lock and Frost
*/
  if (!iconsLoaded) loadIcons(lcd);

  if (millis() - last_time < REFRESH_MS)
    return;

  last_time = millis();

  strncpy_P(text, (PGM_P)pgm_read_ptr(&messageTable[msg]), sizeof(text));
  text[sizeof(text) - 1] = '\0';

  /* ---------- Blink handling (FAULT) ---------- */
  if (mods.blink) {
    static bool visible = true;
```



```
    static unsigned long lastBlink = 0;

    if (millis() - lastBlink > 500) {
      lastBlink = millis();
      visible = !visible;
    }

    if (!visible) {
      // Desired output = blank line
      for (uint8_t i = 0; i < 16; i++)
        out[i] = ' ';
      out[16] = '\0';
    } else {
      goto BUILD_TEXT;
    }
  } else {
BUILD_TEXT:
    /* ---------- CAPS handling ---------- */
    if (mods.caps) {
      for (uint8_t i = 0; text[i]; i++) {
        if (text[i] >= 'a' && text[i] <= 'z')
          text[i] -= 32;
      }
    }

    /* ---------- Scroll handling ---------- */
    if (mods.scroll) {
      if (millis() - lastScroll > SCROLL_MS) {
        lastScroll = millis();
        scrollIndex++;
      }

      uint8_t len = strlen(text);
      if (scrollIndex >= len)
        scrollIndex = 0;

      for (uint8_t i = 0; i < 16; i++)
        out[i] = text[(scrollIndex + i) % len];
    } else {
      scrollIndex = 0;
      strncpy(out, text, 16);
    }

    out[16] = '\0';
  }

  /* ---------- DIFF-BASED WRITE ---------- */
  for (uint8_t col = 0; col < 16; col++) {
    if (out[col] != shadow[col]) {
      lcd.setCursor(col, 0);
      lcd.write(out[col]);
      shadow[col] = out[col];
    }
  }

  /* ---------- ICON RENDERING ---------- */
  static bool lockVisible = true;
  static unsigned long lastLockBlink = 0;

  if (mods.lockBlink) {
    if (millis() - lastLockBlink > 500) {
      lastLockBlink = millis();
      lockVisible = !lockVisible;
    }
```

```
  } else {
    lockVisible = true;
  }

  /* Column 14 = LOCK */
  char desiredLock =
    (mods.lock && lockVisible) ? 0 : ' ';

  /* Column 15 = FROST */
  char desiredFrost =
    mods.frost ? 1 : ' ';

  /* Diff-write icons */
  if (shadow[14] != desiredLock) {
    lcd.setCursor(14, 0);
    lcd.write(desiredLock);
    shadow[14] = desiredLock;
  }

  if (shadow[15] != desiredFrost) {
    lcd.setCursor(15, 0);
    lcd.write(desiredFrost);
    shadow[15] = desiredFrost;
  }
}

void renderBottomRow(
  LiquidCrystal &lcd,
  int16_t tin_x10,
  int16_t tcyl_x10,
  int16_t tstag_x10) {
  lcd.setCursor(0, 1);
  lcd.print("                ");  // remove any old text
  lcd.setCursor(0, 1);            // reposition cursor to start of the second line
  printTempPos_x10(lcd, tin_x10);
  printTempPos_x10(lcd, tcyl_x10);
  printTempSigned_x10(lcd, tstag_x10);
}


void printTempPos_x10(LiquidCrystal &lcd, int16_t t) {
  if (t < 0) {
    lcd.print(" --- ");  // this is out of range print dashes and return
    return;
  }

  int16_t whole = t / 10;
  int16_t frac = abs(t % 10);

  if (whole < 10)
    lcd.print(' '); // leading space if required
  lcd.print(whole);
  lcd.print('.');
  lcd.print(frac);
  lcd.print(' ');
}

void printTempSigned_x10(LiquidCrystal &lcd, int16_t t) {
  int16_t whole = abs(t) / 10;
  int16_t frac = abs(t % 10);

  // Explicit sign column
  if (t < 0)
    lcd.print('-');
```



```
  else
    lcd.print(' '); // leading space if required

  if (whole < 10)
    lcd.print(' ');
  lcd.print(whole);

  lcd.print('.');
  lcd.print(frac);
  lcd.print(' ');
}


}  // namespace LCD_
#pragma endregion

#pragma region pump_control.cpp
#include <Arduino.h>  // MUST be first
#include "pump_control.h"

bool updatePump(PumpState &ps, bool requestOn, unsigned long now)
{
    // ----- HARD FAULT LOCKOUT -----
    if (ps.maxRunFault) {
        ps.running = false;
        return false;
    }

    ps.lock = false;  // default: no lock

    // ----- STATE CHANGE REQUESTED? -----
    if (ps.running != requestOn) {

        // Minimum time for the CURRENT state
        unsigned long minLimit =
            ps.running ? MIN_ON_MS : MIN_OFF_MS;

        // Too early to change?
        if (now - ps.lastChange < minLimit) {
            ps.lock = true;         // show lock icon
            return ps.running;      // refuse transition
        }

        // Transition allowed
        ps.running = requestOn;
        ps.lastChange = now;

        if (ps.running)
            ps.onStart = now;

        return ps.running;
    }

    // ----- MAX RUN TIME -----
    if (ps.running && (now - ps.onStart >= MAX_ON_MS)) {
        ps.running = false;
        ps.lastChange = now;
        ps.maxRunFault = true;
        ps.lock = true;
        return false;
    }

    return ps.running;
}
```

```
#pragma endregion

#pragma region temps.cpp
/* ============== temps.cpp ============= */
#include <Arduino.h>
#include <math.h>
#include "temps.h"

namespace temps {

/* ==========================
   Configuration
   ========================== */

constexpr uint8_t PIN_TOUT = A1;
constexpr uint8_t PIN_TIN = A2;
constexpr uint8_t PIN_TCYL = A3;
constexpr uint8_t PIN_TSTAG = A4;

constexpr uint8_t NUM_READINGS = 40;
constexpr uint32_t SAMPLE_INTERVAL_MS = 200;  // 40 × 200 ms = 8 s



/* ==========================
   ADC ring buffers
   ========================== */

static uint16_t tinBuf[NUM_READINGS];
static uint16_t toutBuf[NUM_READINGS];
static uint16_t tcylBuf[NUM_READINGS];
static uint16_t tstagBuf[NUM_READINGS];

static uint32_t tinSum, toutSum, tcylSum, tstagSum;
static uint8_t index;
static uint32_t lastSampleMs;

static Averages current{};
static Averages previous{};

/* ==========================
   Thermistor conversion
   ========================== */

static double calcPTC_2K(uint16_t adc) {
  constexpr double R_PULLUP = 4700.0;  // 4k7
  constexpr double R_SERIES = 100.0;   // series resistor

  if (adc <= 1)
    return -273.15;
  if (adc >= 1022)
    return 150.0;

  // Correct divider math including series resistor
  double rTherm =
    (R_PULLUP * adc / (1023.0 - adc)) - R_SERIES;

  // Clamp to sane minimum
  if (rTherm < 100.0)
    rTherm = 100.0;

  // Empirical fit (your sensor)
  constexpr double A = 203.58;
  constexpr double B = -1529.9;
```



```

  return A * ::log(rTherm) + B;
}

static int16_t adcToTemp_x10(uint32_t adcAvg) {
  double temp = calcPTC_2K((uint16_t)adcAvg);
  return (int16_t)(temp * 10.0 + (temp >= 0 ? 0.5 : -0.5));
}

/* ==========================
   Public API
   ========================== */

void begin() {
  constexpr uint16_t ADC_25C = 316;

  for (uint8_t i = 0; i < NUM_READINGS; i++) {
    tinBuf[i]  = ADC_25C;
    toutBuf[i] = ADC_25C;
    tcylBuf[i] = ADC_25C;
    tstagBuf[i] = ADC_25C;

    tinSum  += ADC_25C;
    toutSum += ADC_25C;
    tcylSum += ADC_25C;
    tstagSum += ADC_25C;
  }

  current = { 250, 250, 250, 250 };
  previous = current;
}

void update() {
  uint32_t now = millis();
  if (now - lastSampleMs < SAMPLE_INTERVAL_MS)
    return;

  lastSampleMs = now;

  // Remove old sample
  tinSum  -= tinBuf[index];
  toutSum -= toutBuf[index];
  tcylSum -= tcylBuf[index];
  tstagSum -= tstagBuf[index];

  // Read ADCs
  tinBuf[index]  = analogRead(PIN_TIN);
  toutBuf[index] = analogRead(PIN_TOUT);
  tcylBuf[index] = analogRead(PIN_TCYL);
  tstagBuf[index] = analogRead(PIN_TSTAG);

  // Add new
  tinSum  += tinBuf[index];
  toutSum += toutBuf[index];
  tcylSum += tcylBuf[index];
  tstagSum += tstagBuf[index];

  index = (index + 1) % NUM_READINGS;

  // Convert averaged ADC → temperature
  current.tin_x10  = adcToTemp_x10(tinSum / NUM_READINGS);
  current.tout_x10  = adcToTemp_x10(toutSum / NUM_READINGS);
  current.tcyl_x10  = adcToTemp_x10(tcylSum / NUM_READINGS);
  current.tstag_x10 = adcToTemp_x10(tstagSum / NUM_READINGS);
```

```
}

const Averages &avg() {
  return current;
}


bool shouldPumpRun(bool pumpRunning) {

  int16_t diff = current.tstag_x10 - current.tin_x10;
// Hysteresis - Use OFF threshold if already running, otherwise use ON threshold.
  int16_t threshold = pumpRunning ? CYL_DIFF_OFF_x10  : CYL_DIFF_ON_x10;

  return diff >= threshold;
}

bool frostProtectionRequired() {
  return current.tstag_x10 <= FROST_TEMP_x10;
}

bool hasChanged() {
  bool changed =
    current.tin_x10 != previous.tin_x10
    || current.tout_x10  != previous.tout_x10
    || current.tcyl_x10  != previous.tcyl_x10
    || current.tstag_x10 != previous.tstag_x10;

  if (changed)
    previous = current;

  return changed;
}

bool sensor_inRange() {
  return current.tin_x10 >= minTemp && current.tin_x10
  <= maxTemp && current.tout_x10  >= minTemp && current.tout_x10
  <= maxTemp && current.tcyl_x10  >= minTemp && current.tcyl_x10
  <= maxTemp && current.tstag_x10 >= minTemp && current.tstag_x10
  <= maxTemp;
}

}  // namespace temps
#pragma endregion

```