```c
/**
* @file tactile_servo_control_v3.c
* @brief Reads 8x4 tactile grid, sends data, receives state AND grasp type
* commands from Python, and controls servo via cogs.
*
* Communication Format from Python: "STATE,GRASP_TYPE\n"
* Example: "STABLE,SOFT\n", "NO_CONTACT,NONE\n", "SLIP,POWER\n"
*
* Connections:
* - ADC CS: P21
* - ADC SCL: P20
* - ADC DO: P19 (MISO)
* - ADC DI: P18 (MOSI)
* - MUX S0: P0
* - MUX S1: P1
* - MUX S2: P2
* - MUX EN: P14 (Active LOW)
* - Row Pins: P4, P5, P6, P7 (Active HIGH)
* - Serial RX: P31 (Connect to Python/PC TX)
* - Serial TX: P30 (Connect to Python/PC RX)
* - Servo: P16
*/
#include "simpletools.h"
#include "adcDCpropab.h" // For ADC reading
#include "fdserial.h" // For full-duplex serial
#include "servo.h" // For servo control
#include "string.h" // For strcmp, strchr
// --- Sensor Grid Configuration ---
#define COLUMNS 8
#define ROWS 4
#define ADC_CH 0 // ADC channel connected to MUX output
// --- Pin Assignments ---
// ADC SPI pins defined implicitly by adcDCpropab library init if default
pins used
#define MUX_S0 0
#define MUX_S1 1
#define MUX_S2 2
#define MUX_EN 14
const int ROW_PINS[ROWS] = {4, 5, 6, 7};
#define SERVO_PIN 16
// --- Timing & Thresholds ---
#define NUM_SAMPLES_READ 3 // Samples per sensor read in main loop (reduce
noise)
#define NUM_SAMPLES_CALIBRATE 50 // Samples per sensor during calibration
#define TOUCH_THRESHOLD 25 // Ignore deltas below this value (noise floor)
#define ADC_SETTLE_TIME_US 100 // Time for MUX/ADC to settle after
switching
#define ROW_STABILIZE_TIME_US 50 // Time for row voltage to stabilize
#define ADC_READ_ERROR -1 // Value to indicate ADC read failure
// --- Serial Port Configuration ---
#define RX_PIN 31
#define TX_PIN 30
#define BAUD_RATE 115200
fdserial *dport; // Serial port handle (global for access from cogs)
// --- Servo Control Parameters ---
#define SERVO_OPEN_POS 500 // Pulse width for fully open (~40 deg) - ADJUST
#define SERVO_CLOSE_POS 1700 // Pulse width for fully closed (~150 deg) -
ADJUST
#define SERVO_STEP_SIZE 20 // Step size for movement (pulse width units)
#define SERVO_SOFT_DELAY 40 // ms delay for smooth/soft movements
#define SERVO_POWER_DELAY 15 // ms delay for fast/power movements
#define SERVO_HOLD_DELAY 50 // ms delay in servo cog loop when holding
// --- State Definitions (Shared between Cogs) ---
// Note: SOFT/POWER GRASP are now handled by grasp_type variable
#define STATE_UNKNOWN -1
#define STATE_NO_CONTACT 0
#define STATE_INITIAL_CONTACT 1 // Moment touch is detected
#define STATE_STABLE 2 // Holding stable contact
#define STATE_MOTION 3 // Object is moving/rolling
#define STATE_SLIP 4 // Object is slipping
#define STATE_RESET 5 // Command to open the gripper
// --- Grasp Type Definitions (Shared between Cogs) ---
#define GRASP_UNKNOWN -1
#define GRASP_NONE 0 // No specific grasp action commanded
#define GRASP_SOFT 1 // Command to close softly
#define GRASP_POWER 2 // Command to close forcefully/quickly
// --- Global Variables ---
int sensorGrid[ROWS][COLUMNS]; // Holds current raw sensor readings
int baselineGrid[ROWS][COLUMNS]; // Holds calibrated baseline values
// Volatile needed as they are written by one cog and read by another
volatile int current_python_state = STATE_RESET; // State from Python
volatile int current_grasp_type = GRASP_NONE; // Grasp type from Python
volatile int current_servo_pos = SERVO_CLOSE_POS; // Start assuming closed
for RESET
// --- Cog Stack ---
```

```
unsigned int servo_cog_stack[50 + 30]; // Increased stack slightly for
string ops
// --- Function Prototypes ---
// Cog 0 (Main/Sensor)
void readMatrix();
int readSingleSensor(int ch);
void calibrateSensors();
void sendDataSerial();
void setAllRowsLow();
void usleep(int us);
// Cog 1 (Servo/State)
void servoCog(void *par);
int checkForStateCommand(); // Now returns 1 if command parsed, 0 otherwise
void servoControlLogic();
//
=========================================================================
// Main Function (Runs in Cog 0)
//
=========================================================================
int main() {
// Initialize ADC - IMPORTANT: Do this first if pins/power are shared
// CS=P21, SCL=P20, Dout=P19(MISO), Din=P18(MOSI) are defaults for adc_init
adc_init(21, 20, 19, 18);
usleep(10000); // Small delay after ADC init
// Initialize Full Duplex Serial Port
dport = fdserial_open(RX_PIN, TX_PIN, 0, BAUD_RATE);
if (dport == NULL) {
// Critical failure: Serial port didn't open. Indicate error (e.g., blink
LED)
while(1) { high(26); pause(100); low(26); pause(100); } // Example blink
P26
}
// Clear buffers on start
fdserial_rxFlush(dport);
dprint(dport, "Propeller Ready. Serial Port Open at %d baud.\n",
BAUD_RATE);
dprint(dport, "Expected Python Cmd Format: STATE,GRASP_TYPE\\n\n");
// Configure MUX control pins (S0, S1, S2) and Enable pin
set_directions(MUX_S0, 3, 0b111); // P0, P1, P2 as outputs
set_direction(MUX_EN, 1); // P14 as output
high(MUX_EN); // Disable MUX initially (Active LOW)
dprint(dport, "DEBUG: MUX Pins Configured.\n");
// Configure row pins as outputs and set low initially
for(int i = 0; i < ROWS; i++) {
set_direction(ROW_PINS[i], 1);
}
setAllRowsLow();
dprint(dport, "DEBUG: Row Pins Configured.\n");
dprint(dport, "Starting Tactile Sensor System...\n");
dprint(dport, "--------------------------------------\n");
// --- Calibration Phase ---
dprint(dport, "DEBUG: Starting Calibration...\n");
calibrateSensors(); // Calibrate the sensor grid
dprint(dport, "DEBUG: Calibration Complete.\n");
// --- Initialize Global State before starting Servo Cog ---
current_python_state = STATE_RESET; // Start in RESET state
current_grasp_type = GRASP_NONE; // Default grasp type
current_servo_pos = SERVO_CLOSE_POS;// Set initial servo variable so RESET
logic works
// --- Launch Servo/State Cog (Cog 1) ---
// `cogstart` returns cog ID (1-7) or 0 if failed.
int servoCogID = cogstart(&servoCog, NULL, servo_cog_stack,
sizeof(servo_cog_stack));
if (servoCogID <= 0) { // Check if cog failed to start
dprint(dport, "FATAL: Failed to start Servo Cog! ID: %d\n", servoCogID);
// Handle failure (e.g., blink LED differently)
while(1) { high(27); pause(50); low(27); pause(50); } // Example blink P27
} else {
dprint(dport, "DEBUG: Servo/State Cog started (ID: %d).\n", servoCogID);
}
// --- Main Loop (Cog 0: Sensor Reading & Sending) ---
dprint(dport, "Starting data stream...\n");
pause(500); // Brief pause before continuous stream
while(1) {
readMatrix(); // Read the latest sensor values
sendDataSerial(); // Send the data to Python
// Brief pause to control loop rate and allow other cog to run
pause(20); // Adjust as needed (e.g., 10-50ms)
}
// fdserial_close(dport); // Unreachable, but good practice if loop could
exit
return 0; // Should not be reached
}
//
========================================================================
// Servo / State Cog Function (Runs in Cog 1)
```

```
//
==========================================================================
void servoCog(void *par) {
dprint(dport, "DEBUG: Servo Cog alive.\n");
pause(100); // Short pause after start
// Initial state is RESET, logic below will handle opening the servo
// No need to explicitly set angle here, servoControlLogic handles it
while(1) {
// 1. Check for new state command (updates globals directly if valid)
checkForStateCommand();
// 2. Execute servo logic based on current state & grasp type
servoControlLogic();
// 3. Pause to yield time and control servo loop rate
// Delays within servoControlLogic determine movement speed.
// This pause prevents hogging CPU when idle (e.g., HOLDING state).
pause(10);
}
}
//
==========================================================================
// Helper Functions (Called by Cog 1)
//
==========================================================================
/**
* @brief Checks serial buffer for a complete "STATE,GRASP_TYPE\n" command.
* Updates global volatile state variables if a valid command is parsed.
* @return 1 if a valid command was successfully parsed this cycle, 0
otherwise.
*/
int checkForStateCommand() {
static char rx_buffer[30]; // Buffer for command string
static int rx_index = 0; // Current write position in buffer
int received_char;
int parsed_state = STATE_UNKNOWN;
int parsed_grasp = GRASP_UNKNOWN;
char *comma_pos;
char *state_str;
char *grasp_str;
int success = 0; // Assume failure until proven otherwise
// Process all available characters currently in the serial buffer
while (fdserial_rxReady(dport) > 0) {
received_char = fdserial_rxChar(dport); // Read one character
if (received_char == '\n') { // Newline marks the end of a command
if (rx_index > 0) { // Check if we received anything before newline
rx_buffer[rx_index] = '\0'; // Null-terminate the received string
// --- Parse the combined string "STATE,GRASP_TYPE" ---
comma_pos = strchr(rx_buffer, ','); // Find the separating comma
if (comma_pos != NULL) { // Check if comma exists
// Comma found, split the string into two parts
*comma_pos = '\0'; // Replace comma with null to terminate STATE string
state_str = rx_buffer; // First part is the state
grasp_str = comma_pos + 1; // Part after comma is grasp type
// --- Parse STATE part ---
if (strcmp(state_str, "NO_CONTACT") == 0) parsed_state = STATE_NO_CONTACT;
else if (strcmp(state_str, "INITIAL_CONTACT") == 0) parsed_state =
STATE_INITIAL_CONTACT;
else if (strcmp(state_str, "STABLE") == 0) parsed_state = STATE_STABLE;
else if (strcmp(state_str, "MOTION") == 0) parsed_state = STATE_MOTION;
else if (strcmp(state_str, "SLIP") == 0) parsed_state = STATE_SLIP;
else if (strcmp(state_str, "RESET") == 0) parsed_state = STATE_RESET;
else parsed_state = STATE_UNKNOWN; // Unrecognized state string
// --- Parse GRASP_TYPE part ---
if (strcmp(grasp_str, "NONE") == 0) parsed_grasp = GRASP_NONE;
else if (strcmp(grasp_str, "SOFT") == 0) parsed_grasp = GRASP_SOFT;
else if (strcmp(grasp_str, "POWER") == 0) parsed_grasp = GRASP_POWER;
else parsed_grasp = GRASP_UNKNOWN; // Unrecognized grasp string (treat as
NONE?)
// --- Update Global Variables only if BOTH parts were valid ---
// Check both parsed state and grasp type before updating globals
if (parsed_state != STATE_UNKNOWN && parsed_grasp != GRASP_UNKNOWN) {
current_python_state = parsed_state; // Update volatile global
current_grasp_type = parsed_grasp; // Update volatile global
success = 1; // Mark as successful parse for this line
// Optional debug print
// dprint(dport, "DEBUG: Parsed State=%d, Grasp=%d\n", parsed_state,
parsed_grasp);
} else {
// Log if either part was invalid
dprint(dport, "WARN: Invalid state or grasp type received: State='%s',
Grasp='%s'\n", state_str, grasp_str);
}
} else {
// Comma not found - invalid command format
dprint(dport, "WARN: Invalid command format (no comma): '%s'\n",
rx_buffer);
}
// --- End Parsing ---
rx_index = 0; // Reset buffer index for the next command
// Return status immediately after processing a complete line
return success;
} else {
// Empty line received (just '\n'), reset buffer
rx_index = 0;
}
} else if (received_char == '\r') {
/* Ignore Carriage Return character */
} else if (received_char >= ' ' && received_char <= '~') { // Check if it's
a printable ASCII character
// Add character to buffer if there is space
if (rx_index < (sizeof(rx_buffer) - 1)) {
rx_buffer[rx_index++] = (char)received_char;
} else {
// Buffer overflow! Reset buffer index to prevent writing out of bounds.
// Optionally print a warning.
// dprint(dport, "WARN: RX command buffer overflow!\n");
rx_index = 0;
}
}
// Ignore any other non-printable characters
} // End while (fdserial_rxReady)
// If the loop finishes without finding a newline, no complete command was
parsed.
return 0; // Indicate no command parsed this cycle
}
/**
* @brief Controls the servo motor based on the global
`current_python_state`
* and `current_grasp_type`. Reads globals at the start for consistency.
*/
void servoControlLogic() {
// Read volatile state variables into local copies for consistent logic
int state = current_python_state;
int grasp = current_grasp_type;
int target_pos; // Target position for the current step/movement
int step_delay; // Delay between steps for current movement
switch(state) {
case STATE_RESET:
// Goal: Move smoothly to fully OPEN position
target_pos = SERVO_OPEN_POS;
```

```
step_delay = SERVO_SOFT_DELAY;
// Check if servo needs to move towards the open position
if (current_servo_pos > target_pos) {
current_servo_pos -= SERVO_STEP_SIZE; // Move one step towards open
// Prevent overshoot
if (current_servo_pos < target_pos) {
current_servo_pos = target_pos;
}
servo_angle(SERVO_PIN, current_servo_pos); // Update servo hardware
pause(step_delay); // Pause for smooth movement
} else {
// Already at or past open position, just hold (or pause)
pause(SERVO_HOLD_DELAY);
}
break;
case STATE_NO_CONTACT:
// Goal: Move smoothly towards fully CLOSED position
// This action continues step-by-step until the state changes or it reaches
the end.
target_pos = SERVO_CLOSE_POS;
step_delay = SERVO_SOFT_DELAY;
// Check if servo needs to move towards the closed position
if (current_servo_pos < target_pos) {
current_servo_pos += SERVO_STEP_SIZE; // Move one step towards close
// Prevent overshoot
if (current_servo_pos > target_pos) {
current_servo_pos = target_pos;
}
servo_angle(SERVO_PIN, current_servo_pos); // Update servo hardware
pause(step_delay); // Pause for smooth movement
} else {
// Already fully closed, just hold (or pause)
pause(SERVO_HOLD_DELAY);
}
break;
case STATE_INITIAL_CONTACT:
// Goal: Hold position firmly upon first contact.
// Python should analyze and send next state/grasp command.
// servo_angle(SERVO_PIN, current_servo_pos); // Optional: Refresh position
pause(SERVO_HOLD_DELAY); // Pause while holding
break;
case STATE_STABLE:
// Goal: Behavior depends on the commanded grasp type.
```

```
target_pos = SERVO_CLOSE_POS; // Assume closing if grasp commanded
if (grasp == GRASP_SOFT) {
// Continue closing softly towards the end
step_delay = SERVO_SOFT_DELAY;
if (current_servo_pos < target_pos) {
current_servo_pos += SERVO_STEP_SIZE;
if (current_servo_pos > target_pos) current_servo_pos = target_pos;
servo_angle(SERVO_PIN, current_servo_pos);
pause(step_delay);
} else { pause(SERVO_HOLD_DELAY); } // Reached end, hold
} else if (grasp == GRASP_POWER) {
// Continue closing forcefully/quickly towards the end
step_delay = SERVO_POWER_DELAY;
if (current_servo_pos < target_pos) {
current_servo_pos += SERVO_STEP_SIZE;
if (current_servo_pos > target_pos) current_servo_pos = target_pos;
servo_angle(SERVO_PIN, current_servo_pos);
pause(step_delay);
} else { pause(SERVO_HOLD_DELAY); } // Reached end, hold
} else { // GRASP_NONE or GRASP_UNKNOWN
// If stable but no grasp is actively commanded, hold current position.
// servo_angle(SERVO_PIN, current_servo_pos); // Optional refresh
pause(SERVO_HOLD_DELAY);
}
break;
case STATE_MOTION:
case STATE_SLIP:
// Goal: React quickly - fast close regardless of commanded grasp type.
// Prioritize securing the object.
target_pos = SERVO_CLOSE_POS;
step_delay = SERVO_POWER_DELAY; // Use faster delay
if (current_servo_pos < target_pos) {
current_servo_pos += SERVO_STEP_SIZE;
if (current_servo_pos > target_pos) current_servo_pos = target_pos;
servo_angle(SERVO_PIN, current_servo_pos);
pause(step_delay); // Faster pause
} else {
// Already fully closed, hold
pause(SERVO_HOLD_DELAY);
}
break;
case STATE_UNKNOWN:
default:
```

```
// Goal: Safe default behavior if state is unrecognized - hold current
position.
// servo_angle(SERVO_PIN, current_servo_pos); // Optional refresh
pause(SERVO_HOLD_DELAY);
break;
}
// Final safety check: Ensure servo position variable is always within
bounds
// This protects against logic errors potentially setting invalid values.
if (current_servo_pos < SERVO_OPEN_POS) {
current_servo_pos = SERVO_OPEN_POS;
// servo_angle(SERVO_PIN, current_servo_pos); // Optionally update hardware
immediately
}
if (current_servo_pos > SERVO_CLOSE_POS) {
current_servo_pos = SERVO_CLOSE_POS;
// servo_angle(SERVO_PIN, current_servo_pos); // Optionally update hardware
immediately
}
}
//
========================================================================
// Helper Functions (Cog 0 - Sensor Reading, Calibration, Sending)
//
========================================================================
/**
* @brief Sets all row pins LOW (disables current flow through columns).
*/
void setAllRowsLow() {
for(int i = 0; i < ROWS; i++) {
low(ROW_PINS[i]);
}
}
/**
* @brief Reads the ADC channel multiple times and returns the average.
* @param ch The ADC channel to read (0-N based on ADC chip).
* @return The average ADC reading, or ADC_READ_ERROR if all reads fail.
*/
int readSingleSensor(int ch) {
long sum = 0;
int valid_reads = 0;
int reading;
for (int i = 0; i < NUM_SAMPLES_READ; i++) {
```

```
reading = adc_in(ch); // Read from specified channel
if (reading >= 0) { // Check for valid ADC reading (> -1)
sum += reading;
valid_reads++;
}
// usleep(10); // Short delay IF needed between samples for ADC stability
}
if (valid_reads > 0) {
return (int)(sum / valid_reads); // Return average
} else {
// dprint(dport, "WARN: ADC Read Failed on channel %d\n", ch); // Debug
return ADC_READ_ERROR; // Indicate read failure
}
}
/**
* @brief Reads the entire sensor grid and stores values in `sensorGrid`.
*/
void readMatrix() {
low(MUX_EN); // Enable MUX (Active LOW)
usleep(5); // Short delay for MUX enable
for(int row = 0; row < ROWS; row++) {
// Activate only the current row
setAllRowsLow(); // Ensure others are off
high(ROW_PINS[row]);
usleep(ROW_STABILIZE_TIME_US); // Wait for row voltage to stabilize
// Read all columns for the active row
for(int col = 0; col < COLUMNS; col++) {
// Set MUX address (S0, S1, S2) for the current column
// MUX pins P0, P1, P2 correspond to bits 0, 1, 2
set_outputs(MUX_S0, 3, col & 0b111); // Set P0, P1, P2 based on column
index
usleep(ADC_SETTLE_TIME_US); // Wait for MUX and ADC input to settle
// Read the sensor value
sensorGrid[row][col] = readSingleSensor(ADC_CH);
// Optional: Check for error immediately for debugging
// if (sensorGrid[row][col] == ADC_READ_ERROR) {
// dprint(dport, "ERROR: ADC read failed at R%d C%d\n", row, col);
// }
}
low(ROW_PINS[row]); // Deactivate the current row
}
high(MUX_EN); // Disable MUX when done reading
}
```

```
/**
* @brief Performs baseline calibration by reading the sensor grid multiple
times.
* Stores the average readings in `baselineGrid`.
*/
void calibrateSensors() {
dprint(dport, "Calibrating Sensor Grid (%d samples)...\n",
NUM_SAMPLES_CALIBRATE);
dprint(dport, "!!! DO NOT TOUCH THE SENSOR !!!\n");
pause(1500); // Give user time to react
// Use long long for sums to prevent overflow with many samples/high values
long long calibrationSum[ROWS][COLUMNS] = {0};
int valid_samples[ROWS][COLUMNS] = {0};
int failed_reads_total = 0;
dprint(dport, "Calibration Progress:
[....................]\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b"); // 20
dots
int progress_dots = 0;
int progress_interval = NUM_SAMPLES_CALIBRATE / 20;
if (progress_interval == 0) progress_interval = 1;
for (int sample = 0; sample < NUM_SAMPLES_CALIBRATE; sample++) {
low(MUX_EN); // Enable MUX
usleep(5);
for(int r = 0; r < ROWS; r++) {
setAllRowsLow();
high(ROW_PINS[r]);
usleep(ROW_STABILIZE_TIME_US);
for(int c = 0; c < COLUMNS; c++) {
set_outputs(MUX_S0, 3, c & 0b111); // Set MUX address
usleep(ADC_SETTLE_TIME_US);
// Single read for calibration is usually sufficient per sample
int reading = adc_in(ADC_CH);
if (reading >= 0) { // Check for valid read
calibrationSum[r][c] += reading;
valid_samples[r][c]++;
} else {
failed_reads_total++;
// Optionally log the first failure for a cell during calibration
}
}
low(ROW_PINS[r]); // Deactivate row
}
high(MUX_EN); // Disable MUX between samples (optional, might reduce noise)
if (NUM_SAMPLES_CALIBRATE > 1) pause(5); // Small delay between calibration
scans
// Update progress bar
if ((sample + 1) % progress_interval == 0 && progress_dots < 20) {
dprint(dport, "#");
progress_dots++;
}
}
dprint(dport, "] Done.\n"); // Finish progress bar
// Calculate average baseline values
dprint(dport, "Calculating baseline averages...\n");
int calibration_failures = 0;
for(int r = 0; r < ROWS; r++) {
for(int c = 0; c < COLUMNS; c++) {
// Require a reasonable number of valid samples for calibration
if (valid_samples[r][c] >= NUM_SAMPLES_CALIBRATE / 2) {
baselineGrid[r][c] = (int)(calibrationSum[r][c] / valid_samples[r][c]);
} else {
dprint(dport, "ERROR: Calibration failed for R%d C%d (Only %d valid
samples)\n", r, c, valid_samples[r][c]);
baselineGrid[r][c] = ADC_READ_ERROR; // Mark baseline as invalid
calibration_failures++;
}
}
}
if (failed_reads_total > 0) {
dprint(dport, "Warning: %d total ADC read failures encountered during
calibration.\n", failed_reads_total);
}
if (calibration_failures > 0) {
dprint(dport, "ERROR: %d sensor cells failed calibration!\n",
calibration_failures);
}
dprint(dport, "Calibration Finished.\n");
}
/**
* @brief Sends the current sensor grid data (as deltas from baseline)
* over serial, framed by "START" and "END" markers.
*/
void sendDataSerial() {
// Send START marker
dprint(dport, "START\n");
// Send sensor data row by row
```

```
for(int row = 0; row < ROWS; row++) {
for(int col = 0; col < COLUMNS; col++) {
int value_to_send;
// Check if baseline or current reading is invalid
if (baselineGrid[row][col] == ADC_READ_ERROR || sensorGrid[row][col] ==
ADC_READ_ERROR) {
value_to_send = -1; // Send -1 to indicate sensor error to Python
} else {
// Calculate delta: baseline - current reading
value_to_send = baselineGrid[row][col] - sensorGrid[row][col];
// Apply threshold: If delta is below threshold (or negative), send 0
if (value_to_send <= TOUCH_THRESHOLD) {
value_to_send = 0;
}
// Optional: Clamp maximum value if needed
// if (value_to_send > 999) value_to_send = 999;
}
// Print the value
dprint(dport, "%d", value_to_send);
// Add comma separator if not the last column
if (col < COLUMNS - 1) {
dprint(dport, ",");
}
}
// Add newline at the end of each row
dprint(dport, "\n");
}
// Send END marker
dprint(dport, "END\n");
// fdserial_txFlush(dport); // Optional: Force buffer flush if timing is
critical
}
/**
* @brief Provides a microsecond delay using waitcnt.
* @param us The number of microseconds to delay.
*/
void usleep(int us) {
if (us <= 0) return;
// Calculate ticks needed based on clock frequency (CLKFREQ from
simpletools)
unsigned int ticks = us * (CLKFREQ / 1000000);
// Ensure at least minimal delay for small non-zero us values
if (ticks == 0 && us > 0) ticks = 1;
```

```
waitcnt(ticks + CNT); // Wait for specified number of clock ticks
}
```

```
void readMatrix() {
  low(MUX_EN); // Enable MUX (Active LOW)
  usleep(5); // Short delay for MUX enable

  for(int row = 0; row < ROWS; row++) {

    // Activate only the current row
    setAllRowsLow(); // Ensure others are off
    high(ROW_PINS[row]);

    usleep(ROW_STABILIZE_TIME_US); // Wait for row voltage to stabilize

    // Read all columns for the active row

    for(int col = 0; col < COLUMNS; col++) {

      // Set MUX address (S0, S1, S2) for the current column
      // MUX pins P0, P1, P2 correspond to bits 0, 1, 2
      set_outputs(MUX_S0, 3, col & 0b111); // Set P0, P1, P2 based on
column index

      usleep(ADC_SETTLE_TIME_US); // Wait for MUX and ADC input to settle

      // Read the sensor value
      sensorGrid[row][col] = readSingleSensor(ADC_CH);

      // Optional: Check for error immediately for debugging
      // if (sensorGrid[row][col] == ADC_READ_ERROR) {
        // dprint(dport, "ERROR: ADC read failed at R%d C%d\n", row, col);
        // }
    }
    low(ROW_PINS[row]); // Deactivate the current row
  }
  high(MUX_EN); // Disable MUX when done reading
}
```