Beweging, Temperatuur Clock en Buttons

Same setup but improved with three buttons to make it completely independed from PC.

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <DHT.h>
#include <DS3231.h>

// --- Definitions ---
#define DHTPIN 4       // Pin connected to the DHT11 data pin
#define DHTTYPE DHT11  // Specify DHT11 sensor
#define BUTTON1PIN 8
#define BUTTON2PIN 9
#define BUTTON3PIN 10
#define TREND_DISPLAY_DURATION_MS 600000  // 10 minutes, time to keep the sinal (arrow up/down) in display
#define HISTORICAL_DATA_ENTRIES_COUNT 5   // Define the length of the history (min-max) buffers: so we keep the last N hours of min max data.
#define MAX_HISTORY_STRING_LENGTH 100     // Maximum characters for the history string
#define DEBUG true                        // Set to true to enable debug printing, false to disable

DHT dht(DHTPIN, DHTTYPE);

DS3231 myRTC;
bool century = false;
bool h12Flag;
bool pmFlag;

int count = 0;
int pirPin = 12;  // Pin for the HC-S501 sensor
int pirValue;

LiquidCrystal_I2C lcd = LiquidCrystal_I2C(0x27, 16, 2);
```

```
unsigned long previousMillis = 0;
const long dhtInterval = 2000;  // Interval for DHT readings

// Custom characters for the LCD arrows
byte downChar[] = {
  B00000,
  B00000,
  B00100,
  B00100,
  B00100,
  B10101,
  B01110,
  B00100
};

byte upChar[] = {
  B00100,
  B01110,
  B10101,
  B00100,
  B00100,
  B00100,
  B00000,
  B00000
};

byte degreeChar[] = {
  B00110,
  B01001,
  B01001,
  B00110,
  B00000,
  B00000,
  B00000,
  B00000
};

byte maxChar[] = {
  B01110,
  B00000,
```

```
  B00100,
  B01110,
  B10101,
  B00100,
  B00100,
  B00100
};

byte minChar[] = {
  B00100,
  B00100,
  B00100,
  B10101,
  B01110,
  B00100,
  B00000,
  B01110
};

//--------------------------------------------------------------------------------
// Generic MeasurementSensor Class Template
//--------------------------------------------------------------------------------
template<typename T>
class MeasurementSensor {
public:
  // Constructor: initializes with an initial measurement value and history capacity N.
  MeasurementSensor(long sigInterval)
    : measurement(0),
      arrow(0),
      arrowMillis(0),
      lowestMeasurement(0),
      highestMeasurement(0),
      signalInterval(sigInterval),
      historyCount(0),
      historyIndex(0) {}

  // Destructor (no longer needed due to no dynamic memory)
  //~MeasurementSensor() {}

  // Update the measurement reading and determine the arrow indicator.
```

```
void update(T newMeasurement, unsigned long currentMillis) {
  // For the first valid measurement, initialize min/max if needed.
  if (lowestMeasurement == static_cast<T>(0) && highestMeasurement == static_cast<T>(0)) {
    lowestMeasurement = newMeasurement;
    highestMeasurement = newMeasurement;
  }

  // Update the current interval's min and max.
  if (newMeasurement < lowestMeasurement) {
    lowestMeasurement = newMeasurement;
  }
  if (newMeasurement > highestMeasurement) {
    highestMeasurement = newMeasurement;
  }

  // Determine arrow indicator logic.
  if (newMeasurement == measurement && currentMillis - arrowMillis >= signalInterval) {
    arrow = 0;
  } else {
    if (newMeasurement < measurement) {
      arrow = 1;  // Down arrow
      arrowMillis = currentMillis;
    }
    if (newMeasurement > measurement) {
      arrow = 2;  // Up arrow
      arrowMillis = currentMillis;
    }
    if (lowestMeasurement != highestMeasurement) {
      if (newMeasurement == this->getLowest()) {
        arrow = 4;  // Indicator for lowest measurement
        arrowMillis = currentMillis;
      }
      if (newMeasurement == this->getHighest()) {
        arrow = 5;  // Indicator for highest measurement
        arrowMillis = currentMillis;
      }
    }
  }

  // Reset arrow if previous value was an invalid initial value.
```

```
  if (measurement == static_cast<T>(0)) {
    arrow = 0;
  }

  measurement = newMeasurement;
}

// Getters for measurement and arrow.
T getMeasurement() const {
  return measurement;
}

// Returns the lowest measurement seen over the history combined with the current interval.
T getLowest() const {
  T minVal = lowestMeasurement;
  // Loop over the history array.
  for (size_t i = 0; i < historyCount; i++) {
    if (lowestHistory[i] < minVal) {
      minVal = lowestHistory[i];
    }
  }
  return minVal;
}

// Returns the highest measurement seen over the history combined with the current interval.
T getHighest() const {
  T maxVal = highestMeasurement;
  // Loop over the history array.
  for (size_t i = 0; i < historyCount; i++) {
    if (highestHistory[i] > maxVal) {
      maxVal = highestHistory[i];
    }
  }
  return maxVal;
}

int getArrow() const {
  return arrow;
}
```

```
  // Reset the min and max values.
  // The current lowest and highest measurements are stored in the history arrays.
  void resetMinMax(T newMeasurement) {
    // Store the current min and max in the arrays at the current history index.
    lowestHistory[historyIndex] = lowestMeasurement;
    highestHistory[historyIndex] = highestMeasurement;

    lowestMeasurement = newMeasurement;
    highestMeasurement = newMeasurement;

    // Increment historyIndex as a circular buffer.
    historyIndex = (historyIndex + 1) % HISTORICAL_DATA_ENTRIES_COUNT;
    if (historyCount < HISTORICAL_DATA_ENTRIES_COUNT) {
      historyCount++;
    }
  }

  void getHistoryString(bool useHighest, char* buffer, size_t bufferSize) const {
    // Initialize the buffer to an empty string.
    buffer[0] = '\0';

    // Determine the index of the oldest element.
    size_t oldestIndex = (historyCount < HISTORICAL_DATA_ENTRIES_COUNT) ? 0 : historyIndex;

    // Loop over the valid history entries in order.
    for (size_t i = 0; i < historyCount; i++) {
      size_t index = (oldestIndex + i) % HISTORICAL_DATA_ENTRIES_COUNT;
      T value = useHighest ? highestHistory[index] : lowestHistory[index];

      // Temporary buffer for the converted float value.
      char tempValue[10];
      // Convert the float value to a string with width 4 and one decimal point.
      dtostrf(static_cast<double>(value), 4, 1, tempValue);

      // Check if adding the next value will overflow the main buffer.
      size_t remainingSpace = bufferSize - strlen(buffer) - 1;
      size_t charsNeeded = strlen(tempValue) + (i < historyCount - 1 ? 2 : 0);  // value + ",
"

      if (charsNeeded > remainingSpace) {
```

```
        Serial.println("Error: Insufficient buffer space in getHistoryString!");
        return;  // You may choose to handle this differently.
      }

      // Concatenate the value to the buffer.
      strcat(buffer, tempValue);

      // Add the separator if it's not the last element.
      if (i < historyCount - 1) {
        strcat(buffer, ", ");
      }
    }
  }



private:
  T measurement;
  int arrow;
  unsigned long arrowMillis;
  T lowestMeasurement;
  T highestMeasurement;
  long signalInterval;

  // History storage: fixed size arrays
  T lowestHistory[HISTORICAL_DATA_ENTRIES_COUNT];
  T highestHistory[HISTORICAL_DATA_ENTRIES_COUNT];
  size_t historyCount;
  size_t historyIndex;
};



class Button {
private:
  int pin;
  int prevState;
  const char* name;
  unsigned long pressStartTime;
  bool longPressTriggered;
  const unsigned long LONG_PRESS_DURATION = 1000; // 1 second for long press
```

```
public:
  // Constructor: initialize the button pin and its name
  Button(int pin, const char* name)
    : pin(pin), prevState(LOW), name(name), pressStartTime(0), longPressTriggered(false) {
    pinMode(pin, INPUT);
  }

  // Update the button state and return true if the button was pressed (transition from LOW to
HIGH)
  bool update() {
    int currentState = digitalRead(pin);
    bool pressed = false;

    if (currentState != prevState) {
      // Button state changed
      if (currentState == HIGH) {
        // Button was just pressed
        pressStartTime = millis();
        longPressTriggered = false;
      } else {
        // Button was just released
        if ((millis() - pressStartTime < LONG_PRESS_DURATION) && !longPressTriggered) {
          // Short press detected
          pressed = true;
        }
        pressStartTime = 0;
      }
      prevState = currentState;
    }

    return pressed;
  }

  // Check for long press and return true if long press is detected
  bool checkLongPress() {
    int currentState = digitalRead(pin);

    if (currentState == HIGH && prevState == HIGH) {
      // Button is being held down
```

```
      if (millis() - pressStartTime > LONG_PRESS_DURATION && !longPressTriggered) {
        longPressTriggered = true;
        return true;
      }
    }

    return false;
  }

  // Getter for the button's name
  const char* getName() {
    return name;
  }
};

class DateTimeSettings {
private:
  // RTC reference
  DS3231& rtc;

  // LCD reference
  LiquidCrystal_I2C& lcd;

  // Setting state variables
  bool isActive;
  int settingIndex; // 0=day, 1=month, 2=hour, 3=minute
  int currentDay, currentMonth, currentHour, currentMinute;

  // Blinking effect variables
  unsigned long lastBlinkTime;
  const unsigned long BLINK_INTERVAL = 500; // 500ms blink interval
  bool showField;

  // Reference to century flag needed for RTC
  bool& centuryFlag;
  bool h12Flag, pmFlag; // RTC flags

public:
  // Constructor
  DateTimeSettings(DS3231& rtcInstance, LiquidCrystal_I2C& lcdInstance, bool& century)
```

```
  : rtc(rtcInstance),
    lcd(lcdInstance),
    isActive(false),
    settingIndex(0),
    lastBlinkTime(0),
    showField(true),
    centuryFlag(century),
    h12Flag(false),
    pmFlag(false) {
}

// Start the settings mode
void start() {
  isActive = true;
  settingIndex = 0; // Start with day

  // Get current values from RTC
  currentDay = rtc.getDate();
  currentMonth = rtc.getMonth(centuryFlag);
  currentHour = rtc.getHour(h12Flag, pmFlag);
  currentMinute = rtc.getMinute();

  showField = true;
  lastBlinkTime = millis();

  displaySettings();
}

// Check if settings mode is active
bool isSettingActive() const {
  return isActive;
}

// Exit settings mode
void exit() {
  isActive = false;
  lcd.clear();
}

// Move to the next field
```

```
void nextField() {
  settingIndex++;
  showField = true; // Reset blink state

  if (settingIndex > 3) {
    // All fields set, save the time
    rtc.setDate(currentDay);
    rtc.setMonth(currentMonth);
    rtc.setHour(currentHour);
    rtc.setMinute(currentMinute);
    rtc.setSecond(0); // Reset seconds to 00

    // Exit setting mode
    exit();
    return;
  }

  displaySettings();
}

// Increase the current field value
void increaseValue() {
  switch (settingIndex) {
    case 0: // Day
      currentDay++;
      if (currentDay > 31) currentDay = 1;
      break;
    case 1: // Month
      currentMonth++;
      if (currentMonth > 12) currentMonth = 1;
      break;
    case 2: // Hour
      currentHour++;
      if (currentHour > 23) currentHour = 0;
      break;
    case 3: // Minute
      currentMinute++;
      if (currentMinute > 59) currentMinute = 0;
      break;
  }
```

```
  showField = true; // Reset blink state
  displaySettings();
}

// Decrease the current field value
void decreaseValue() {
  switch (settingIndex) {
    case 0: // Day
      currentDay--;
      if (currentDay < 1) currentDay = 31;
      break;
    case 1: // Month
      currentMonth--;
      if (currentMonth < 1) currentMonth = 12;
      break;
    case 2: // Hour
      currentHour--;
      if (currentHour < 0) currentHour = 23;
      break;
    case 3: // Minute
      currentMinute--;
      if (currentMinute < 0) currentMinute = 59;
      break;
  }
  showField = true; // Reset blink state
  displaySettings();
}

// Update method to be called in the main loop
void update() {
  // Return if not in setting mode
  if (!isActive) return;

  // Handle blinking effect
  if (millis() - lastBlinkTime >= BLINK_INTERVAL) {
    lastBlinkTime = millis();
    showField = !showField;
    displaySettings();
  }
}
```

```
private:
  // Display the settings screen
  void displaySettings() {
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print("Set Date & Time");

    lcd.setCursor(1, 1);

    // Display day - blinking when being set
    if (settingIndex != 0 || showField) {
      print2digits(currentDay);
    } else {
      lcd.print("  ");
    }

    lcd.print("-");

    // Display month - blinking when being set
    if (settingIndex != 1 || showField) {
      print2digits(currentMonth);
    } else {
      lcd.print("  ");
    }

    lcd.setCursor(8, 1);

    // Display hour - blinking when being set
    if (settingIndex != 2 || showField) {
      print2digits(currentHour);
    } else {
      lcd.print("  ");
    }

    lcd.print(":");

    // Display minute - blinking when being set
    if (settingIndex != 3 || showField) {
      print2digits(currentMinute);
```

```
    } else {
      lcd.print("  ");
    }
  }

  // Helper function to print 2 digits with leading zero
  void print2digits(int number) {
    if (number < 10) {
      lcd.print("0");
    }
    lcd.print(number, DEC);
  }
};



//--------------------------------------------------------------------------------
// Global Instances for Temperature and Humidity
//--------------------------------------------------------------------------------
MeasurementSensor<float> tempSensor(TREND_DISPLAY_DURATION_MS);
MeasurementSensor<int> humiditySensor(TREND_DISPLAY_DURATION_MS);

void setup() {
  Serial.begin(9600);
  dht.begin();
  delay(2000);

  lcd.init();
  lcd.backlight();
  lcd.clear();
  lcd.createChar(1, downChar);
  lcd.createChar(2, upChar);
  lcd.createChar(3, degreeChar);
  lcd.createChar(4, minChar);
  lcd.createChar(5, maxChar);

  pinMode(pirPin, INPUT);

  pinMode(BUTTON1PIN, INPUT);
  pinMode(BUTTON2PIN, INPUT);
  pinMode(BUTTON3PIN, INPUT);
```

```
  // myRTC.setYear(2025);
  // myRTC.setMonth(3);
  // myRTC.setDate(20);
  // myRTC.setHour(10);
  // myRTC.setMinute(49);
  // myRTC.setSecond(0);
}

void print2digits(int number) {
  if (number < 10) {
    lcd.print("0");
  }
  lcd.print(number, DEC);
}



int lastResetHour = -1;  // Initialize to an invalid hour

// Create instances for each button
Button button1(BUTTON1PIN, "Button 1");
Button button2(BUTTON2PIN, "Button 2");
Button button3(BUTTON3PIN, "Button 3");

int displayModus = 1;  // we want temp range when 2 we want humidity range

// Add this near the top of your code, with your other global instances
DateTimeSettings dateTimeSettings(myRTC, lcd, century);

void loop() {
  delay(20);

  // Check if we are in setting mode
  if (dateTimeSettings.isSettingActive()) {
    dateTimeSettings.update();

    // Check for long press on button 1 to exit setting mode
    if (button1.checkLongPress()) {
      dateTimeSettings.exit();
      return;
```

```
  }

  // Handle button 1 (short press - move to next field)
  if (button1.update()) {
    dateTimeSettings.nextField();
  }

  // Handle button 2 (decrease value)
  if (button2.update()) {
    dateTimeSettings.decreaseValue();
  }

  // Handle button 3 (increase value)
  if (button3.update()) {
    dateTimeSettings.increaseValue();
  }

  return; // Skip the normal loop when in setting mode
}

// Check for long press on button 1 to enter setting mode
if (button1.checkLongPress()) {
  dateTimeSettings.start();
  return;
}

// The rest of your existing loop code stays the same
// Update each button and print if pressed
if (button1.update()) {
  Serial.print(button1.getName());
  Serial.println(" pressed");
}
if (button2.update()) {
  Serial.print(button2.getName());
  Serial.println(" pressed");
  count = 5;
  previousMillis = 0;
  displayModus = 1;
}
if (button3.update()) {
```

```
  Serial.print(button3.getName());
  Serial.println(" pressed");
  count = 5;
  previousMillis = 0;
  displayModus = 2;
}

// Check for movement
pirValue = digitalRead(pirPin);
// Serial.println(pirValue);

if (pirValue) {
  lcd.backlight();
}

// Do we need to update the display?
unsigned long currentMillis = millis();
if (currentMillis - previousMillis >= dhtInterval) {
  previousMillis = currentMillis;

  if (!pirValue) {
    lcd.noBacklight();
  }

  count++;

  // Read new values from the DHT sensor
  float newTemperature = dht.readTemperature();
  int newHumidity = dht.readHumidity();

  if (isnan(newTemperature) || isnan(newHumidity)) {
    Serial.println("Failed to read from DHT sensor!");
    lcd.clear();
    lcd.print(" *** DHT Error *** ");
    delay(5000);
    return;  // next itteration in the loop
  }

  // Update our measurement sensors
  tempSensor.update(newTemperature, currentMillis);
```

```
humiditySensor.update(newHumidity, currentMillis);

// Get current hour from RTC
int currentHour = myRTC.getHour(h12Flag, pmFlag);
// Check if hour has changed
if (currentHour != lastResetHour) {  // || count%10==0

  tempSensor.resetMinMax(newTemperature);
  humiditySensor.resetMinMax(newHumidity);

  lastResetHour = currentHour;
}

Serial.println(count);
char tempHistoryBuffer[MAX_HISTORY_STRING_LENGTH];
tempSensor.getHistoryString(false, tempHistoryBuffer, sizeof(tempHistoryBuffer));
Serial.print("min temp history: ");
Serial.println(tempHistoryBuffer);

char tempHistoryBuffer2[MAX_HISTORY_STRING_LENGTH];
tempSensor.getHistoryString(true, tempHistoryBuffer2, sizeof(tempHistoryBuffer2));
Serial.print("max temp history: ");
Serial.println(tempHistoryBuffer2);

Serial.println();



// Update the LCD display for temperature
lcd.setCursor(0, 0);
if (!(count % 6) && displayModus == 1) {
  lcd.print("");
  lcd.print(tempSensor.getLowest(), 1);
  lcd.write(3);
  lcd.print(" ");
  lcd.print(tempSensor.getHighest(), 1);
  lcd.write(3);
  lcd.print(" ");
} else {
  if (tempSensor.getArrow()) {
    lcd.write(tempSensor.getArrow());
```

```
      } else {
        lcd.print(" ");
      }
      lcd.print(tempSensor.getMeasurement(), 1);
      lcd.write(3);  // degree symbol
      lcd.print("C     ");
    }

    // Update the LCD display for humidity
    if (!(count % 6) && displayModus == 2) {
      lcd.setCursor(9, 0);
      lcd.print(humiditySensor.getLowest(), 1);
      lcd.print("% ");
      lcd.print(humiditySensor.getHighest(), 1);
      lcd.print("%");
    } else {
      lcd.setCursor(12, 0);
      if (humiditySensor.getArrow()) {
        lcd.write(humiditySensor.getArrow());
      } else {
        lcd.print(" ");
      }
      lcd.print(humiditySensor.getMeasurement());
      lcd.print("%");
    }

    // Update the LCD display second row, date
    lcd.setCursor(1, 1);
    print2digits(myRTC.getDate());
    lcd.print("-");
    print2digits(myRTC.getMonth(century));

    // Update the LCD display second row, time
    lcd.setCursor(10, 1);
    print2digits(myRTC.getHour(h12Flag, pmFlag));
    lcd.print(":");
    print2digits(myRTC.getMinute());
  }
}
```

https://www.circuito.io/app?components=97,97,97,514,10167,11022,417986,821989,931983

Revision #2
Created 2025-04-06 15:12:11 UTC by Max
Updated 2025-04-06 18:38:13 UTC by Max