/\*

HCSR04 - Library for arduino, for HC-SR04 ultrasonic distance sensor.

Created by Dirk Sarodnick, 2020.

\*/

#include "Arduino.h"

#include "HCSR04.h"

HCSR04Sensor::HCSR04Sensor() {}

HCSR04Sensor::~HCSR04Sensor() { this->end(); }

void HCSR04Sensor::begin(uint8\_t triggerPin, uint8\_t\* echoPins, uint8\_t echoCount, uint32\_t timeout, uint16\_t triggerTime, uint16\_t triggerWait, eUltraSonicUnlock\_t unlock) {

if (this->echoCount != echoCount) this->end();

this->triggerPin = triggerPin;

pinMode(triggerPin, OUTPUT);

this->timeout = timeout;

this->triggerTime = triggerTime;

this->triggerWait = triggerWait;

this->echoCount = echoCount;

if (this->lastMicroseconds == NULL) this->lastMicroseconds = new long[echoCount];

if (this->lastDistances == NULL) this->lastDistances = new double[echoCount];

if (this->triggerTimes == NULL) this->triggerTimes = new unsigned long[echoCount];

if (this->echoTimes == NULL) this->echoTimes = new unsigned long[echoCount];

if (this->echoStages == NULL) this->echoStages = new int16\_t[echoCount];

if (this->echoInts == NULL) this->echoInts = new int16\_t[echoCount];

if (this->echoPorts == NULL) this->echoPorts = new int16\_t[echoCount];

for (uint8\_t i = 0; i < this->echoCount; i++) {

this->triggerTimes[i] = 0;

this->echoTimes[i] = 0;

int16\_t interrupt = digitalPinToInterrupt(echoPins[i]);

if (interrupt == NOT\_AN\_INTERRUPT) {

this->echoStages[i] = -1;

this->echoInts[i] = -1;

this->echoPorts[i] = echoPins[i];

} else {

this->echoStages[i] = 0;

this->echoInts[i] = interrupt;

this->echoPorts[i] = -1;

}

pinMode(echoPins[i], INPUT);

}

// Unlock sensors that are possibly in a locked state, if this feature is enabled.

this->unlockSensors(unlock, echoPins);

}

void HCSR04Sensor::end() {

if (this->lastMicroseconds != NULL) delete []this->lastMicroseconds;

if (this->lastDistances != NULL) delete []this->lastDistances;

if (this->triggerTimes != NULL) delete []this->triggerTimes;

if (this->echoTimes != NULL) delete []this->echoTimes;

if (this->echoPorts != NULL) delete []this->echoPorts;

if (this->echoInts != NULL) delete []this->echoInts;

if (this->echoStages != NULL) delete []this->echoStages;

this->lastMicroseconds = NULL;

this->lastDistances = NULL;

this->triggerTimes = NULL;

this->echoTimes = NULL;

this->echoPorts = NULL;

this->echoInts = NULL;

this->echoStages = NULL;

}

void HCSR04Sensor::measureMicroseconds(long\* results) {

if (results == NULL) results = this->lastMicroseconds;

bool finished = true;

bool waiting = true;

unsigned long startMicros = micros();

unsigned long currentMicros = 0;

unsigned long elapsedMicros = 0;

// Make sure that trigger pin is LOW.

digitalWrite(triggerPin, LOW);

delayMicroseconds(4);

// Hold trigger HIGH for 10 microseconds (default), which signals the sensor to measure distance.

digitalWrite(triggerPin, HIGH);

delayMicroseconds(this->triggerTime);

// Set trigger LOW again and wait to give the sensor time for sending the signal without interference

digitalWrite(triggerPin, LOW);

delayMicroseconds(this->triggerWait);

// Attach interrupts to echo pins for the starting point

for (uint8\_t i = 0; i < this->echoCount; i++) {

if (this->echoInts[i] >= 0 && this->echoStages[i] == 0) {

this->echoStages[i] = 1;

switch (i) {

case 0: attachInterrupt(this->echoInts[i], &triggerInterrupt0, RISING); break;

case 1: attachInterrupt(this->echoInts[i], &triggerInterrupt1, RISING); break;

case 2: attachInterrupt(this->echoInts[i], &triggerInterrupt2, RISING); break;

case 3: attachInterrupt(this->echoInts[i], &triggerInterrupt3, RISING); break;

case 4: attachInterrupt(this->echoInts[i], &triggerInterrupt4, RISING); break;

case 5: attachInterrupt(this->echoInts[i], &triggerInterrupt5, RISING); break;

case 6: attachInterrupt(this->echoInts[i], &triggerInterrupt6, RISING); break;

case 7: attachInterrupt(this->echoInts[i], &triggerInterrupt7, RISING); break;

case 8: attachInterrupt(this->echoInts[i], &triggerInterrupt8, RISING); break;

case 9: attachInterrupt(this->echoInts[i], &triggerInterrupt9, RISING); break;

}

}

}

// Wait until all echos are returned or timed out.

while(true) {

delayMicroseconds(1);

finished = true;

waiting = true;

currentMicros = micros();

elapsedMicros = currentMicros - startMicros;

for (uint8\_t i = 0; i < this->echoCount; i++) {

waiting &= elapsedMicros < this->timeout || (this->triggerTimes[i] > 0 && this->echoTimes[i] == 0 && (currentMicros - this->triggerTimes[i]) < this->timeout);

if (this->echoPorts[i] >= 0 && this->triggerTimes[i] == 0) {

if (digitalRead(this->echoPorts[i]) == HIGH) this->triggerTimes[i] = micros();

}

if (this->triggerTimes[i] > 0 || !waiting) {

if (this->echoInts[i] >= 0 && (this->echoStages[i] == 1 || !waiting)) {

if (this->echoStages[i] == 1) this->echoStages[i] = 2;

detachInterrupt(this->echoInts[i]);

}

} else finished &= false;

if (this->echoInts[i] >= 0 && this->triggerTimes[i] > 0 && this->echoStages[i] == 2 && waiting) {

this->echoStages[i] = 3;

switch (i) {

case 0: attachInterrupt(this->echoInts[i], &echoInterrupt0, FALLING); break;

case 1: attachInterrupt(this->echoInts[i], &echoInterrupt1, FALLING); break;

case 2: attachInterrupt(this->echoInts[i], &echoInterrupt2, FALLING); break;

case 3: attachInterrupt(this->echoInts[i], &echoInterrupt3, FALLING); break;

case 4: attachInterrupt(this->echoInts[i], &echoInterrupt4, FALLING); break;

case 5: attachInterrupt(this->echoInts[i], &echoInterrupt5, FALLING); break;

case 6: attachInterrupt(this->echoInts[i], &echoInterrupt6, FALLING); break;

case 7: attachInterrupt(this->echoInts[i], &echoInterrupt7, FALLING); break;

case 8: attachInterrupt(this->echoInts[i], &echoInterrupt8, FALLING); break;

case 9: attachInterrupt(this->echoInts[i], &echoInterrupt9, FALLING); break;

}

}

if (this->echoPorts[i] >= 0 && this->triggerTimes[i] > 0 && this->echoTimes[i] == 0) {

if (digitalRead(this->echoPorts[i]) == LOW) this->echoTimes[i] = micros();

}

if ((this->triggerTimes[i] > 0 && this->echoTimes[i] > 0) || !waiting) {

if (this->echoInts[i] >= 0 && (this->echoStages[i] == 3 || !waiting)) {

if (this->echoStages[i] == 3) this->echoStages[i] = 4;

detachInterrupt(this->echoInts[i]);

}

} else finished &= false;

}

if (!waiting || finished) break;

}

// Determine the durations of each sensor.

for (uint8\_t i = 0; i < this->echoCount; i++) {

if (this->echoInts[i] >= 0) this->echoStages[i] = 0;

if (this->triggerTimes[i] > 0 && this->echoTimes[i] > 0) {

long resultTime = this->echoTimes[i] - this->triggerTimes[i];

results[i] = resultTime > 0 ? resultTime : HCSR04\_INVALID\_RESULT;

} else if (this->triggerTimes[i] > 0) {

results[i] = HCSR04\_NO\_ECHO;

} else {

results[i] = HCSR04\_NO\_TRIGGER;

}

this->triggerTimes[i] = 0;

this->echoTimes[i] = 0;

}

}

void HCSR04Sensor::measureDistanceMm(float temperature, double\* results) {

if (results == NULL) results = this->lastDistances;

double speedOfSoundInMmPerMs = (331.3 + 0.606 \* temperature) / 1000; // Cair ≈ (331.3 + 0.606 ⋅ ϑ) m/s

long\* times = measureMicroseconds();

// Calculate the distance in mm for each result.

for (uint8\_t i = 0; i < this->echoCount; i++) {

double distanceMm = times[i] / 2.0 \* speedOfSoundInMmPerMs;

if (distanceMm < 10 || distanceMm > 4000) {

results[i] = HCSR04\_INVALID\_RESULT;

} else {

results[i] = distanceMm;

}

}

}

void HCSR04Sensor::measureDistanceCm(float temperature, double\* results) {

if (results == NULL) results = this->lastDistances;

double speedOfSoundInCmPerMs = (331.3 + 0.606 \* temperature) / 1000 / 10; // Cair ≈ (331.3 + 0.606 ⋅ ϑ) m/s

long\* times = measureMicroseconds();

// Calculate the distance in cm for each result.

for (uint8\_t i = 0; i < this->echoCount; i++) {

double distanceCm = times[i] / 2.0 \* speedOfSoundInCmPerMs;

if (distanceCm < 1 || distanceCm > 400) {

results[i] = HCSR04\_INVALID\_RESULT;

} else {

results[i] = distanceCm;

}

}

}

void HCSR04Sensor::measureDistanceIn(float temperature, double\* results) {

if (results == NULL) results = this->lastDistances;

double speedOfSoundInCmPerMs = (331.3 + 0.606 \* temperature) \* 39.37007874 / 1000 / 1000; // Cair ≈ (331.3 + 0.606 ⋅ ϑ) m/s

long\* times = measureMicroseconds();

// Calculate the distance in cm for each result.

for (uint8\_t i = 0; i < this->echoCount; i++) {

double distanceIn = times[i] / 2.0 \* speedOfSoundInCmPerMs;

if (distanceIn < 1 || distanceIn > 157.4804) {

results[i] = HCSR04\_INVALID\_RESULT;

}

else {

results[i] = distanceIn;

}

}

}

void HCSR04Sensor::unlockSensors(eUltraSonicUnlock\_t unlock, uint8\_t\* echoPins) {

if (unlock == eUltraSonicUnlock\_t::unlockSkip) return;

bool hasLocked = false;

// Check if any sensor is in a locked state and unlock it if necessary.

for (uint8\_t i = 0; echoPins[i] != 0; i++) {

if (unlock == eUltraSonicUnlock\_t::unlockMaybe && digitalRead(echoPins[i]) == LOW) continue;

pinMode(echoPins[i], OUTPUT);

digitalWrite(echoPins[i], LOW);

hasLocked = true;

}

if (hasLocked) delay(100);

// Revert the pinMode after potential unlocking.

for (uint8\_t i = 0; echoPins[i] != 0; i++) {

pinMode(echoPins[i], INPUT);

}

if (hasLocked) delay(100);

}

void HCSR04Sensor::triggerInterrupt(uint8\_t index) {

if (this->triggerTimes[index] == 0) this->triggerTimes[index] = micros();

}

void HCSR04Sensor::echoInterrupt(uint8\_t index) {

if (this->triggerTimes[index] > 0 && this->echoTimes[index] == 0) this->echoTimes[index] = micros();

}

void HCSR04Sensor::triggerInterrupt0() { HCSR04.triggerInterrupt(0); }

void HCSR04Sensor::triggerInterrupt1() { HCSR04.triggerInterrupt(1); }

void HCSR04Sensor::triggerInterrupt2() { HCSR04.triggerInterrupt(2); }

void HCSR04Sensor::triggerInterrupt3() { HCSR04.triggerInterrupt(3); }

void HCSR04Sensor::triggerInterrupt4() { HCSR04.triggerInterrupt(4); }

void HCSR04Sensor::triggerInterrupt5() { HCSR04.triggerInterrupt(5); }

void HCSR04Sensor::triggerInterrupt6() { HCSR04.triggerInterrupt(6); }

void HCSR04Sensor::triggerInterrupt7() { HCSR04.triggerInterrupt(7); }

void HCSR04Sensor::triggerInterrupt8() { HCSR04.triggerInterrupt(8); }

void HCSR04Sensor::triggerInterrupt9() { HCSR04.triggerInterrupt(9); }

void HCSR04Sensor::echoInterrupt0() { HCSR04.echoInterrupt(0); }

void HCSR04Sensor::echoInterrupt1() { HCSR04.echoInterrupt(1); }

void HCSR04Sensor::echoInterrupt2() { HCSR04.echoInterrupt(2); }

void HCSR04Sensor::echoInterrupt3() { HCSR04.echoInterrupt(3); }

void HCSR04Sensor::echoInterrupt4() { HCSR04.echoInterrupt(4); }

void HCSR04Sensor::echoInterrupt5() { HCSR04.echoInterrupt(5); }

void HCSR04Sensor::echoInterrupt6() { HCSR04.echoInterrupt(6); }

void HCSR04Sensor::echoInterrupt7() { HCSR04.echoInterrupt(7); }

void HCSR04Sensor::echoInterrupt8() { HCSR04.echoInterrupt(8); }

void HCSR04Sensor::echoInterrupt9() { HCSR04.echoInterrupt(9); }

HCSR04Sensor HCSR04;