// ---------------------------------------------------------------------------

// Created by Tim Eckel - eckel.tim@gmail.com

//

// See NewPing.h for license, purpose, syntax, version history, links, etc.

// ---------------------------------------------------------------------------

#include "NewPing.h"

// ---------------------------------------------------------------------------

// NewPing constructor

// ---------------------------------------------------------------------------

NewPing::NewPing(uint8\_t trigger\_pin, uint8\_t echo\_pin, unsigned int max\_cm\_distance) {

#if DO\_BITWISE == true

\_triggerBit = digitalPinToBitMask(trigger\_pin); // Get the port register bitmask for the trigger pin.

\_echoBit = digitalPinToBitMask(echo\_pin); // Get the port register bitmask for the echo pin.

\_triggerOutput = portOutputRegister(digitalPinToPort(trigger\_pin)); // Get the output port register for the trigger pin.

\_echoInput = portInputRegister(digitalPinToPort(echo\_pin)); // Get the input port register for the echo pin.

\_triggerMode = (uint8\_t \*) portModeRegister(digitalPinToPort(trigger\_pin)); // Get the port mode register for the trigger pin.

#else

\_triggerPin = trigger\_pin;

\_echoPin = echo\_pin;

#endif

\_one\_pin\_mode = (trigger\_pin == echo\_pin); // Automatic one pin mode detection per sensor.

set\_max\_distance(max\_cm\_distance); // Call function to set the max sensor distance.

#if (defined(\_\_arm\_\_) && (defined(TEENSYDUINO) || defined(PARTICLE))) || defined(ARDUINO\_AVR\_YUN) || DO\_BITWISE != true

pinMode(echo\_pin, INPUT); // Set echo pin to input (on Teensy 3.x (ARM), pins default to disabled, at least one pinMode() is needed for GPIO mode).

pinMode(trigger\_pin, OUTPUT); // Set trigger pin to output (on Teensy 3.x (ARM), pins default to disabled, at least one pinMode() is needed for GPIO mode).

#endif

#if DO\_BITWISE == true

\*\_triggerMode |= \_triggerBit; // Set trigger pin to output.

\*\_triggerOutput &= ~\_triggerBit; // Trigger pin should already be low, but set to low to make sure.

#else

digitalWrite(\_triggerPin, LOW); // Trigger pin should already be low, but set to low to make sure.

#endif

}

// ---------------------------------------------------------------------------

// Standard ping methods

// ---------------------------------------------------------------------------

unsigned int NewPing::ping(unsigned int max\_cm\_distance) {

if (max\_cm\_distance > 0) set\_max\_distance(max\_cm\_distance); // Call function to set a new max sensor distance.

if (!ping\_trigger()) return NO\_ECHO; // Trigger a ping, if it returns false, return NO\_ECHO to the calling function.

#if URM37\_ENABLED == true

#if DO\_BITWISE == true

while (!(\*\_echoInput & \_echoBit)) // Wait for the ping echo.

#else

while (!digitalRead(\_echoPin)) // Wait for the ping echo.

#endif

if (micros() > \_max\_time) return NO\_ECHO; // Stop the loop and return NO\_ECHO (false) if we're beyond the set maximum distance.

#else

#if DO\_BITWISE == true

while (\*\_echoInput & \_echoBit) // Wait for the ping echo.

#else

while (digitalRead(\_echoPin)) // Wait for the ping echo.

#endif

if (micros() > \_max\_time) return NO\_ECHO; // Stop the loop and return NO\_ECHO (false) if we're beyond the set maximum distance.

#endif

return (micros() - (\_max\_time - \_maxEchoTime) - PING\_OVERHEAD); // Calculate ping time, include overhead.

}

unsigned long NewPing::ping\_cm(unsigned int max\_cm\_distance) {

unsigned long echoTime = NewPing::ping(max\_cm\_distance); // Calls the ping method and returns with the ping echo distance in uS.

#if ROUNDING\_ENABLED == false

return (echoTime / US\_ROUNDTRIP\_CM); // Call the ping method and returns the distance in centimeters (no rounding).

#else

return NewPingConvert(echoTime, US\_ROUNDTRIP\_CM); // Convert uS to centimeters.

#endif

}

unsigned long NewPing::ping\_in(unsigned int max\_cm\_distance) {

unsigned long echoTime = NewPing::ping(max\_cm\_distance); // Calls the ping method and returns with the ping echo distance in uS.

#if ROUNDING\_ENABLED == false

return (echoTime / US\_ROUNDTRIP\_IN); // Call the ping method and returns the distance in inches (no rounding).

#else

return NewPingConvert(echoTime, US\_ROUNDTRIP\_IN); // Convert uS to inches.

#endif

}

unsigned long NewPing::ping\_median(uint8\_t it, unsigned int max\_cm\_distance) {

unsigned int uS[it], last;

uint8\_t j, i = 0;

unsigned long t;

uS[0] = NO\_ECHO;

if (max\_cm\_distance > 0) set\_max\_distance(max\_cm\_distance); // Call function to set a new max sensor distance.

while (i < it) {

t = micros(); // Start ping timestamp.

last = ping(); // Send ping.

if (last != NO\_ECHO) { // Ping in range, include as part of median.

if (i > 0) { // Don't start sort till second ping.

for (j = i; j > 0 && uS[j - 1] < last; j--) // Insertion sort loop.

uS[j] = uS[j - 1]; // Shift ping array to correct position for sort insertion.

} else j = 0; // First ping is sort starting point.

uS[j] = last; // Add last ping to array in sorted position.

i++; // Move to next ping.

} else it--; // Ping out of range, skip and don't include as part of median.

if (i < it && micros() - t < PING\_MEDIAN\_DELAY)

delay((PING\_MEDIAN\_DELAY + t - micros()) >> 10); // Millisecond delay between pings.

}

return (uS[it >> 1]); // Return the ping distance median.

}

// ---------------------------------------------------------------------------

// Standard and timer interrupt ping method support functions (not called directly)

// ---------------------------------------------------------------------------

boolean NewPing::ping\_trigger() {

#if DO\_BITWISE == true

\*\_triggerMode |= \_triggerBit; // Set trigger pin to output (only matters if \_one\_pin\_mode is true, but is quicker/smaller than checking \_one\_pin\_mode state).

\*\_triggerOutput |= \_triggerBit; // Set trigger pin high, this tells the sensor to send out a ping.

delayMicroseconds(TRIGGER\_WIDTH); // Wait long enough for the sensor to realize the trigger pin is high.

\*\_triggerOutput &= ~\_triggerBit; // Set trigger pin back to low.

if (\_one\_pin\_mode) \*\_triggerMode &= ~\_triggerBit; // Set trigger pin to input (this is technically setting the echo pin to input as both are tied to the same pin).

#if URM37\_ENABLED == true

if (!(\*\_echoInput & \_echoBit)) return false; // Previous ping hasn't finished, abort.

\_max\_time = micros() + \_maxEchoTime + MAX\_SENSOR\_DELAY; // Maximum time we'll wait for ping to start (most sensors are <450uS, the SRF06 can take up to 34,300uS!)

while (\*\_echoInput & \_echoBit) // Wait for ping to start.

if (micros() > \_max\_time) return false; // Took too long to start, abort.

#else

if (\*\_echoInput & \_echoBit) return false; // Previous ping hasn't finished, abort.

\_max\_time = micros() + \_maxEchoTime + MAX\_SENSOR\_DELAY; // Maximum time we'll wait for ping to start (most sensors are <450uS, the SRF06 can take up to 34,300uS!)

while (!(\*\_echoInput & \_echoBit)) // Wait for ping to start.

if (micros() > \_max\_time) return false; // Took too long to start, abort.

#endif

#else

if (\_one\_pin\_mode) pinMode(\_triggerPin, OUTPUT); // Set trigger pin to output.

digitalWrite(\_triggerPin, HIGH); // Set trigger pin high, this tells the sensor to send out a ping.

delayMicroseconds(TRIGGER\_WIDTH); // Wait long enough for the sensor to realize the trigger pin is high.

digitalWrite(\_triggerPin, LOW); // Set trigger pin back to low.

if (\_one\_pin\_mode) pinMode(\_triggerPin, INPUT); // Set trigger pin to input (this is technically setting the echo pin to input as both are tied to the same pin).

#if URM37\_ENABLED == true

if (!digitalRead(\_echoPin)) return false; // Previous ping hasn't finished, abort.

\_max\_time = micros() + \_maxEchoTime + MAX\_SENSOR\_DELAY; // Maximum time we'll wait for ping to start (most sensors are <450uS, the SRF06 can take up to 34,300uS!)

while (digitalRead(\_echoPin)) // Wait for ping to start.

if (micros() > \_max\_time) return false; // Took too long to start, abort.

#else

if (digitalRead(\_echoPin)) return false; // Previous ping hasn't finished, abort.

\_max\_time = micros() + \_maxEchoTime + MAX\_SENSOR\_DELAY; // Maximum time we'll wait for ping to start (most sensors are <450uS, the SRF06 can take up to 34,300uS!)

while (!digitalRead(\_echoPin)) // Wait for ping to start.

if (micros() > \_max\_time) return false; // Took too long to start, abort.

#endif

#endif

\_max\_time = micros() + \_maxEchoTime; // Ping started, set the time-out.

return true; // Ping started successfully.

}

void NewPing::set\_max\_distance(unsigned int max\_cm\_distance) {

#if ROUNDING\_ENABLED == false

\_maxEchoTime = min(max\_cm\_distance + 1, (unsigned int) MAX\_SENSOR\_DISTANCE + 1) \* US\_ROUNDTRIP\_CM; // Calculate the maximum distance in uS (no rounding).

#else

\_maxEchoTime = min(max\_cm\_distance, (unsigned int) MAX\_SENSOR\_DISTANCE) \* US\_ROUNDTRIP\_CM + (US\_ROUNDTRIP\_CM / 2); // Calculate the maximum distance in uS.

#endif

}

#if TIMER\_ENABLED == true && DO\_BITWISE == true

// ---------------------------------------------------------------------------

// Timer interrupt ping methods (won't work with ATmega128, ATtiny and most non-AVR microcontrollers)

// ---------------------------------------------------------------------------

void NewPing::ping\_timer(void (\*userFunc)(void), unsigned int max\_cm\_distance) {

if (max\_cm\_distance > 0) set\_max\_distance(max\_cm\_distance); // Call function to set a new max sensor distance.

if (!ping\_trigger()) return; // Trigger a ping, if it returns false, return without starting the echo timer.

timer\_us(ECHO\_TIMER\_FREQ, userFunc); // Set ping echo timer check every ECHO\_TIMER\_FREQ uS.

}

boolean NewPing::check\_timer() {

if (micros() > \_max\_time) { // Outside the time-out limit.

timer\_stop(); // Disable timer interrupt

return false; // Cancel ping timer.

}

#if URM37\_ENABLED == false

if (!(\*\_echoInput & \_echoBit)) { // Ping echo received.

#else

if (\*\_echoInput & \_echoBit) { // Ping echo received.

#endif

timer\_stop(); // Disable timer interrupt

ping\_result = (micros() - (\_max\_time - \_maxEchoTime) - PING\_TIMER\_OVERHEAD); // Calculate ping time including overhead.

return true; // Return ping echo true.

}

return false; // Return false because there's no ping echo yet.

}

// ---------------------------------------------------------------------------

// Timer2/Timer4 interrupt methods (can be used for non-ultrasonic needs)

// ---------------------------------------------------------------------------

// Variables used for timer functions

void (\*intFunc)();

void (\*intFunc2)();

unsigned long \_ms\_cnt\_reset;

volatile unsigned long \_ms\_cnt;

#if defined(\_\_arm\_\_) && (defined(TEENSYDUINO) || defined(PARTICLE))

IntervalTimer itimer;

#endif

void NewPing::timer\_us(unsigned int frequency, void (\*userFunc)(void)) {

intFunc = userFunc; // User's function to call when there's a timer event.

timer\_setup(); // Configure the timer interrupt.

#if defined(\_\_AVR\_ATmega32U4\_\_) // Use Timer4 for ATmega32U4 (Teensy/Leonardo).

OCR4C = min((frequency>>2) - 1, 255); // Every count is 4uS, so divide by 4 (bitwise shift right 2) subtract one, then make sure we don't go over 255 limit.

TIMSK4 = (1<<TOIE4); // Enable Timer4 interrupt.

#elif defined(\_\_arm\_\_) && defined(TEENSYDUINO) // Timer for Teensy 3.x

itimer.begin(userFunc, frequency); // Really simple on the Teensy 3.x, calls userFunc every 'frequency' uS.

#elif defined(\_\_arm\_\_) && defined(PARTICLE) // Timer for Particle devices

itimer.begin(userFunc, frequency, uSec); // Really simple on the Particle, calls userFunc every 'frequency' uS.

#else

OCR2A = min((frequency>>2) - 1, 255); // Every count is 4uS, so divide by 4 (bitwise shift right 2) subtract one, then make sure we don't go over 255 limit.

TIMSK2 |= (1<<OCIE2A); // Enable Timer2 interrupt.

#endif

}

void NewPing::timer\_ms(unsigned long frequency, void (\*userFunc)(void)) {

intFunc = NewPing::timer\_ms\_cntdwn; // Timer events are sent here once every ms till user's frequency is reached.

intFunc2 = userFunc; // User's function to call when user's frequency is reached.

\_ms\_cnt = \_ms\_cnt\_reset = frequency; // Current ms counter and reset value.

timer\_setup(); // Configure the timer interrupt.

#if defined(\_\_AVR\_ATmega32U4\_\_) // Use Timer4 for ATmega32U4 (Teensy/Leonardo).

OCR4C = 249; // Every count is 4uS, so 1ms = 250 counts - 1.

TIMSK4 = (1<<TOIE4); // Enable Timer4 interrupt.

#elif defined(\_\_arm\_\_) && defined(TEENSYDUINO) // Timer for Teensy 3.x

itimer.begin(NewPing::timer\_ms\_cntdwn, 1000); // Set timer to 1ms (1000 uS).

#elif defined(\_\_arm\_\_) && defined(PARTICLE) // Timer for Particle

itimer.begin(NewPing::timer\_ms\_cntdwn, 1000, uSec); // Set timer to 1ms (1000 uS).

#else

OCR2A = 249; // Every count is 4uS, so 1ms = 250 counts - 1.

TIMSK2 |= (1<<OCIE2A); // Enable Timer2 interrupt.

#endif

}

void NewPing::timer\_stop() { // Disable timer interrupt.

#if defined(\_\_AVR\_ATmega32U4\_\_) // Use Timer4 for ATmega32U4 (Teensy/Leonardo).

TIMSK4 = 0;

#elif defined(\_\_arm\_\_) && (defined(TEENSYDUINO) || defined(PARTICLE)) // Timer for Teensy 3.x & Particle

itimer.end();

#else

TIMSK2 &= ~(1<<OCIE2A);

#endif

}

// ---------------------------------------------------------------------------

// Timer2/Timer4 interrupt method support functions (not called directly)

// ---------------------------------------------------------------------------

void NewPing::timer\_setup() {

#if defined(\_\_AVR\_ATmega32U4\_\_) // Use Timer4 for ATmega32U4 (Teensy/Leonardo).

timer\_stop(); // Disable Timer4 interrupt.

TCCR4A = TCCR4C = TCCR4D = TCCR4E = 0;

TCCR4B = (1<<CS42) | (1<<CS41) | (1<<CS40) | (1<<PSR4); // Set Timer4 prescaler to 64 (4uS/count, 4uS-1020uS range).

TIFR4 = (1<<TOV4);

TCNT4 = 0; // Reset Timer4 counter.

#elif defined(\_\_AVR\_ATmega8\_\_) || defined(\_\_AVR\_ATmega16\_\_) || defined(\_\_AVR\_ATmega32\_\_) || defined(\_\_AVR\_ATmega8535\_\_) // Alternate timer commands for certain microcontrollers.

timer\_stop(); // Disable Timer2 interrupt.

ASSR &= ~(1<<AS2); // Set clock, not pin.

TCCR2 = (1<<WGM21 | 1<<CS22); // Set Timer2 to CTC mode, prescaler to 64 (4uS/count, 4uS-1020uS range).

TCNT2 = 0; // Reset Timer2 counter.

#elif defined(\_\_arm\_\_) && (defined(TEENSYDUINO) || defined(PARTICLE))

timer\_stop(); // Stop the timer.

#else

timer\_stop(); // Disable Timer2 interrupt.

ASSR &= ~(1<<AS2); // Set clock, not pin.

TCCR2A = (1<<WGM21); // Set Timer2 to CTC mode.

TCCR2B = (1<<CS22); // Set Timer2 prescaler to 64 (4uS/count, 4uS-1020uS range).

TCNT2 = 0; // Reset Timer2 counter.

#endif

}

void NewPing::timer\_ms\_cntdwn() {

if (!\_ms\_cnt--) { // Count down till we reach zero.

intFunc2(); // Scheduled time reached, run the main timer event function.

\_ms\_cnt = \_ms\_cnt\_reset; // Reset the ms timer.

}

}

#if defined(\_\_AVR\_ATmega32U4\_\_) // Use Timer4 for ATmega32U4 (Teensy/Leonardo).

ISR(TIMER4\_OVF\_vect) {

intFunc(); // Call wrapped function.

}

#elif defined(\_\_AVR\_ATmega8\_\_) || defined(\_\_AVR\_ATmega16\_\_) || defined(\_\_AVR\_ATmega32\_\_) || defined(\_\_AVR\_ATmega8535\_\_) // Alternate timer commands for certain microcontrollers.

ISR(TIMER2\_COMP\_vect) {

intFunc(); // Call wrapped function.

}

#elif defined(\_\_arm\_\_)

// Do nothing...

#else

ISR(TIMER2\_COMPA\_vect) {

intFunc(); // Call wrapped function.

}

#endif

#endif

// ---------------------------------------------------------------------------

// Conversion methods (rounds result to nearest cm or inch).

// ---------------------------------------------------------------------------

unsigned int NewPing::convert\_cm(unsigned int echoTime) {

#if ROUNDING\_ENABLED == false

return (echoTime / US\_ROUNDTRIP\_CM); // Convert uS to centimeters (no rounding).

#else

return NewPingConvert(echoTime, US\_ROUNDTRIP\_CM); // Convert uS to centimeters.

#endif

}

unsigned int NewPing::convert\_in(unsigned int echoTime) {

#if ROUNDING\_ENABLED == false

return (echoTime / US\_ROUNDTRIP\_IN); // Convert uS to inches (no rounding).

#else

return NewPingConvert(echoTime, US\_ROUNDTRIP\_IN); // Convert uS to inches.

#endif

}