/\*

Servo.cpp - Interrupt driven Servo library for Arduino using 16 bit timers - Version 2

Copyright (c) 2009 Michael Margolis. All right reserved.

This library is free software; you can redistribute it and/or

modify it under the terms of the GNU Lesser General Public

License as published by the Free Software Foundation; either

version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,

but WITHOUT ANY WARRANTY; without even the implied warranty of

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU

Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public

License along with this library; if not, write to the Free Software

Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

\*/

#if defined(ARDUINO\_ARCH\_AVR)

#include <avr/interrupt.h>

#include <Arduino.h>

#include "Servo.h"

#define usToTicks(\_us) (( clockCyclesPerMicrosecond()\* \_us) / 8) // converts microseconds to ticks (assumes prescaler of 8) // 12 Aug 2009

#define ticksToUs(\_ticks) (( (unsigned)\_ticks \* 8)/ clockCyclesPerMicrosecond() ) // converts from ticks back to microseconds

#define TRIM\_DURATION 2 // compensation ticks to trim adjust for digitalWrite delays // 12 August 2009

//#define NBR\_TIMERS (MAX\_SERVOS / SERVOS\_PER\_TIMER)

static servo\_t servos[MAX\_SERVOS]; // static array of servo structures

static volatile int8\_t Channel[\_Nbr\_16timers ]; // counter for the servo being pulsed for each timer (or -1 if refresh interval)

uint8\_t ServoCount = 0; // the total number of attached servos

// convenience macros

#define SERVO\_INDEX\_TO\_TIMER(\_servo\_nbr) ((timer16\_Sequence\_t)(\_servo\_nbr / SERVOS\_PER\_TIMER)) // returns the timer controlling this servo

#define SERVO\_INDEX\_TO\_CHANNEL(\_servo\_nbr) (\_servo\_nbr % SERVOS\_PER\_TIMER) // returns the index of the servo on this timer

#define SERVO\_INDEX(\_timer,\_channel) ((\_timer\*SERVOS\_PER\_TIMER) + \_channel) // macro to access servo index by timer and channel

#define SERVO(\_timer,\_channel) (servos[SERVO\_INDEX(\_timer,\_channel)]) // macro to access servo class by timer and channel

#define SERVO\_MIN() (MIN\_PULSE\_WIDTH - this->min \* 4) // minimum value in us for this servo

#define SERVO\_MAX() (MAX\_PULSE\_WIDTH - this->max \* 4) // maximum value in us for this servo

/\*\*\*\*\*\*\*\*\*\*\*\* static functions common to all instances \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static inline void handle\_interrupts(timer16\_Sequence\_t timer, volatile uint16\_t \*TCNTn, volatile uint16\_t\* OCRnA)

{

if( Channel[timer] < 0 )

\*TCNTn = 0; // channel set to -1 indicated that refresh interval completed so reset the timer

else{

if( SERVO\_INDEX(timer,Channel[timer]) < ServoCount && SERVO(timer,Channel[timer]).Pin.isActive == true )

digitalWrite( SERVO(timer,Channel[timer]).Pin.nbr,LOW); // pulse this channel low if activated

}

Channel[timer]++; // increment to the next channel

if( SERVO\_INDEX(timer,Channel[timer]) < ServoCount && Channel[timer] < SERVOS\_PER\_TIMER) {

\*OCRnA = \*TCNTn + SERVO(timer,Channel[timer]).ticks;

if(SERVO(timer,Channel[timer]).Pin.isActive == true) // check if activated

digitalWrite( SERVO(timer,Channel[timer]).Pin.nbr,HIGH); // it's an active channel so pulse it high

}

else {

// finished all channels so wait for the refresh period to expire before starting over

if( ((unsigned)\*TCNTn) + 4 < usToTicks(REFRESH\_INTERVAL) ) // allow a few ticks to ensure the next OCR1A not missed

\*OCRnA = (unsigned int)usToTicks(REFRESH\_INTERVAL);

else

\*OCRnA = \*TCNTn + 4; // at least REFRESH\_INTERVAL has elapsed

Channel[timer] = -1; // this will get incremented at the end of the refresh period to start again at the first channel

}

}

#ifndef WIRING // Wiring pre-defines signal handlers so don't define any if compiling for the Wiring platform

// Interrupt handlers for Arduino

#if defined(\_useTimer1)

SIGNAL (TIMER1\_COMPA\_vect)

{

handle\_interrupts(\_timer1, &TCNT1, &OCR1A);

}

#endif

#if defined(\_useTimer3)

SIGNAL (TIMER3\_COMPA\_vect)

{

handle\_interrupts(\_timer3, &TCNT3, &OCR3A);

}

#endif

#if defined(\_useTimer4)

SIGNAL (TIMER4\_COMPA\_vect)

{

handle\_interrupts(\_timer4, &TCNT4, &OCR4A);

}

#endif

#if defined(\_useTimer5)

SIGNAL (TIMER5\_COMPA\_vect)

{

handle\_interrupts(\_timer5, &TCNT5, &OCR5A);

}

#endif

#elif defined WIRING

// Interrupt handlers for Wiring

#if defined(\_useTimer1)

void Timer1Service()

{

handle\_interrupts(\_timer1, &TCNT1, &OCR1A);

}

#endif

#if defined(\_useTimer3)

void Timer3Service()

{

handle\_interrupts(\_timer3, &TCNT3, &OCR3A);

}

#endif

#endif

static void initISR(timer16\_Sequence\_t timer)

{

#if defined (\_useTimer1)

if(timer == \_timer1) {

TCCR1A = 0; // normal counting mode

TCCR1B = \_BV(CS11); // set prescaler of 8

TCNT1 = 0; // clear the timer count

#if defined(\_\_AVR\_ATmega8\_\_)|| defined(\_\_AVR\_ATmega128\_\_)

TIFR |= \_BV(OCF1A); // clear any pending interrupts

TIMSK |= \_BV(OCIE1A) ; // enable the output compare interrupt

#else

// here if not ATmega8 or ATmega128

TIFR1 |= \_BV(OCF1A); // clear any pending interrupts

TIMSK1 |= \_BV(OCIE1A) ; // enable the output compare interrupt

#endif

#if defined(WIRING)

timerAttach(TIMER1OUTCOMPAREA\_INT, Timer1Service);

#endif

}

#endif

#if defined (\_useTimer3)

if(timer == \_timer3) {

TCCR3A = 0; // normal counting mode

TCCR3B = \_BV(CS31); // set prescaler of 8

TCNT3 = 0; // clear the timer count

#if defined(\_\_AVR\_ATmega128\_\_)

TIFR |= \_BV(OCF3A); // clear any pending interrupts

ETIMSK |= \_BV(OCIE3A); // enable the output compare interrupt

#else

TIFR3 = \_BV(OCF3A); // clear any pending interrupts

TIMSK3 = \_BV(OCIE3A) ; // enable the output compare interrupt

#endif

#if defined(WIRING)

timerAttach(TIMER3OUTCOMPAREA\_INT, Timer3Service); // for Wiring platform only

#endif

}

#endif

#if defined (\_useTimer4)

if(timer == \_timer4) {

TCCR4A = 0; // normal counting mode

TCCR4B = \_BV(CS41); // set prescaler of 8

TCNT4 = 0; // clear the timer count

TIFR4 = \_BV(OCF4A); // clear any pending interrupts

TIMSK4 = \_BV(OCIE4A) ; // enable the output compare interrupt

}

#endif

#if defined (\_useTimer5)

if(timer == \_timer5) {

TCCR5A = 0; // normal counting mode

TCCR5B = \_BV(CS51); // set prescaler of 8

TCNT5 = 0; // clear the timer count

TIFR5 = \_BV(OCF5A); // clear any pending interrupts

TIMSK5 = \_BV(OCIE5A) ; // enable the output compare interrupt

}

#endif

}

static void finISR(timer16\_Sequence\_t timer)

{

//disable use of the given timer

#if defined WIRING // Wiring

if(timer == \_timer1) {

#if defined(\_\_AVR\_ATmega1281\_\_)||defined(\_\_AVR\_ATmega2561\_\_)

TIMSK1 &= ~\_BV(OCIE1A) ; // disable timer 1 output compare interrupt

#else

TIMSK &= ~\_BV(OCIE1A) ; // disable timer 1 output compare interrupt

#endif

timerDetach(TIMER1OUTCOMPAREA\_INT);

}

else if(timer == \_timer3) {

#if defined(\_\_AVR\_ATmega1281\_\_)||defined(\_\_AVR\_ATmega2561\_\_)

TIMSK3 &= ~\_BV(OCIE3A); // disable the timer3 output compare A interrupt

#else

ETIMSK &= ~\_BV(OCIE3A); // disable the timer3 output compare A interrupt

#endif

timerDetach(TIMER3OUTCOMPAREA\_INT);

}

#else

//For Arduino - in future: call here to a currently undefined function to reset the timer

(void) timer; // squash "unused parameter 'timer' [-Wunused-parameter]" warning

#endif

}

static boolean isTimerActive(timer16\_Sequence\_t timer)

{

// returns true if any servo is active on this timer

for(uint8\_t channel=0; channel < SERVOS\_PER\_TIMER; channel++) {

if(SERVO(timer,channel).Pin.isActive == true)

return true;

}

return false;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* end of static functions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Servo::Servo()

{

if( ServoCount < MAX\_SERVOS) {

this->servoIndex = ServoCount++; // assign a servo index to this instance

servos[this->servoIndex].ticks = usToTicks(DEFAULT\_PULSE\_WIDTH); // store default values - 12 Aug 2009

}

else

this->servoIndex = INVALID\_SERVO ; // too many servos

}

uint8\_t Servo::attach(int pin)

{

return this->attach(pin, MIN\_PULSE\_WIDTH, MAX\_PULSE\_WIDTH);

}

uint8\_t Servo::attach(int pin, int min, int max)

{

if(this->servoIndex < MAX\_SERVOS ) {

pinMode( pin, OUTPUT) ; // set servo pin to output

servos[this->servoIndex].Pin.nbr = pin;

// todo min/max check: abs(min - MIN\_PULSE\_WIDTH) /4 < 128

this->min = (MIN\_PULSE\_WIDTH - min)/4; //resolution of min/max is 4 us

this->max = (MAX\_PULSE\_WIDTH - max)/4;

// initialize the timer if it has not already been initialized

timer16\_Sequence\_t timer = SERVO\_INDEX\_TO\_TIMER(servoIndex);

if(isTimerActive(timer) == false)

initISR(timer);

servos[this->servoIndex].Pin.isActive = true; // this must be set after the check for isTimerActive

}

return this->servoIndex ;

}

void Servo::detach()

{

servos[this->servoIndex].Pin.isActive = false;

timer16\_Sequence\_t timer = SERVO\_INDEX\_TO\_TIMER(servoIndex);

if(isTimerActive(timer) == false) {

finISR(timer);

}

}

void Servo::write(int value)

{

if(value < MIN\_PULSE\_WIDTH)

{ // treat values less than 544 as angles in degrees (valid values in microseconds are handled as microseconds)

if(value < 0) value = 0;

if(value > 180) value = 180;

value = map(value, 0, 180, SERVO\_MIN(), SERVO\_MAX());

}

this->writeMicroseconds(value);

}

void Servo::writeMicroseconds(int value)

{

// calculate and store the values for the given channel

byte channel = this->servoIndex;

if( (channel < MAX\_SERVOS) ) // ensure channel is valid

{

if( value < SERVO\_MIN() ) // ensure pulse width is valid

value = SERVO\_MIN();

else if( value > SERVO\_MAX() )

value = SERVO\_MAX();

value = value - TRIM\_DURATION;

value = usToTicks(value); // convert to ticks after compensating for interrupt overhead - 12 Aug 2009

uint8\_t oldSREG = SREG;

cli();

servos[channel].ticks = value;

SREG = oldSREG;

}

}

int Servo::read() // return the value as degrees

{

return map( this->readMicroseconds()+1, SERVO\_MIN(), SERVO\_MAX(), 0, 180);

}

int Servo::readMicroseconds()

{

unsigned int pulsewidth;

if( this->servoIndex != INVALID\_SERVO )

pulsewidth = ticksToUs(servos[this->servoIndex].ticks) + TRIM\_DURATION ; // 12 aug 2009

else

pulsewidth = 0;

return pulsewidth;

}

bool Servo::attached()

{

return servos[this->servoIndex].Pin.isActive ;

}

#endif // ARDUINO\_ARCH\_AVR