/\*

Copyright (c) 2015 Arduino LLC. 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\_SAMD)

#include <Arduino.h>

#include <Servo.h>

#define usToTicks(\_us) ((clockCyclesPerMicrosecond() \* \_us) / 16) // converts microseconds to ticks

#define ticksToUs(\_ticks) (((unsigned) \_ticks \* 16) / clockCyclesPerMicrosecond()) // converts from ticks back to microseconds

#define TRIM\_DURATION 5 // compensation ticks to trim adjust for digitalWrite delays

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

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

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

// 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

#define WAIT\_TC16\_REGS\_SYNC(x) while(x->COUNT16.STATUS.bit.SYNCBUSY);

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

void Servo\_Handler(timer16\_Sequence\_t timer, Tc \*pTc, uint8\_t channel, uint8\_t intFlag);

#if defined (\_useTimer1)

void HANDLER\_FOR\_TIMER1(void) {

Servo\_Handler(\_timer1, TC\_FOR\_TIMER1, CHANNEL\_FOR\_TIMER1, INTFLAG\_BIT\_FOR\_TIMER\_1);

}

#endif

#if defined (\_useTimer2)

void HANDLER\_FOR\_TIMER2(void) {

Servo\_Handler(\_timer2, TC\_FOR\_TIMER2, CHANNEL\_FOR\_TIMER2, INTFLAG\_BIT\_FOR\_TIMER\_2);

}

#endif

void Servo\_Handler(timer16\_Sequence\_t timer, Tc \*tc, uint8\_t channel, uint8\_t intFlag)

{

if (currentServoIndex[timer] < 0) {

tc->COUNT16.COUNT.reg = (uint16\_t) 0;

WAIT\_TC16\_REGS\_SYNC(tc)

} else {

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

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

}

}

// Select the next servo controlled by this timer

currentServoIndex[timer]++;

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

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

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

}

// Get the counter value

uint16\_t tcCounterValue = tc->COUNT16.COUNT.reg;

WAIT\_TC16\_REGS\_SYNC(tc)

tc->COUNT16.CC[channel].reg = (uint16\_t) (tcCounterValue + SERVO(timer, currentServoIndex[timer]).ticks);

WAIT\_TC16\_REGS\_SYNC(tc)

}

else {

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

// Get the counter value

uint16\_t tcCounterValue = tc->COUNT16.COUNT.reg;

WAIT\_TC16\_REGS\_SYNC(tc)

if (tcCounterValue + 4UL < usToTicks(REFRESH\_INTERVAL)) { // allow a few ticks to ensure the next OCR1A not missed

tc->COUNT16.CC[channel].reg = (uint16\_t) usToTicks(REFRESH\_INTERVAL);

}

else {

tc->COUNT16.CC[channel].reg = (uint16\_t) (tcCounterValue + 4UL); // at least REFRESH\_INTERVAL has elapsed

}

WAIT\_TC16\_REGS\_SYNC(tc)

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

}

// Clear the interrupt

tc->COUNT16.INTFLAG.reg = intFlag;

}

static inline void resetTC (Tc\* TCx)

{

// Disable TCx

TCx->COUNT16.CTRLA.reg &= ~TC\_CTRLA\_ENABLE;

WAIT\_TC16\_REGS\_SYNC(TCx)

// Reset TCx

TCx->COUNT16.CTRLA.reg = TC\_CTRLA\_SWRST;

WAIT\_TC16\_REGS\_SYNC(TCx)

while (TCx->COUNT16.CTRLA.bit.SWRST);

}

static void \_initISR(Tc \*tc, uint8\_t channel, uint32\_t id, IRQn\_Type irqn, uint8\_t gcmForTimer, uint8\_t intEnableBit)

{

// Enable GCLK for timer 1 (timer counter input clock)

GCLK->CLKCTRL.reg = (uint16\_t) (GCLK\_CLKCTRL\_CLKEN | GCLK\_CLKCTRL\_GEN\_GCLK0 | GCLK\_CLKCTRL\_ID(gcmForTimer));

while (GCLK->STATUS.bit.SYNCBUSY);

// Reset the timer

// TODO this is not the right thing to do if more than one channel per timer is used by the Servo library

resetTC(tc);

// Set timer counter mode to 16 bits

tc->COUNT16.CTRLA.reg |= TC\_CTRLA\_MODE\_COUNT16;

// Set timer counter mode as normal PWM

tc->COUNT16.CTRLA.reg |= TC\_CTRLA\_WAVEGEN\_NPWM;

// Set the prescaler factor to GCLK\_TC/16. At nominal 48 MHz GCLK\_TC this is 3000 ticks per millisecond

tc->COUNT16.CTRLA.reg |= TC\_CTRLA\_PRESCALER\_DIV16;

// Count up

tc->COUNT16.CTRLBCLR.bit.DIR = 1;

WAIT\_TC16\_REGS\_SYNC(tc)

// First interrupt request after 1 ms

tc->COUNT16.CC[channel].reg = (uint16\_t) usToTicks(1000UL);

WAIT\_TC16\_REGS\_SYNC(tc)

// Configure interrupt request

// TODO this should be changed if more than one channel per timer is used by the Servo library

NVIC\_DisableIRQ(irqn);

NVIC\_ClearPendingIRQ(irqn);

NVIC\_SetPriority(irqn, 0);

NVIC\_EnableIRQ(irqn);

// Enable the match channel interrupt request

tc->COUNT16.INTENSET.reg = intEnableBit;

// Enable the timer and start it

tc->COUNT16.CTRLA.reg |= TC\_CTRLA\_ENABLE;

WAIT\_TC16\_REGS\_SYNC(tc)

}

static void initISR(timer16\_Sequence\_t timer)

{

#if defined (\_useTimer1)

if (timer == \_timer1)

\_initISR(TC\_FOR\_TIMER1, CHANNEL\_FOR\_TIMER1, ID\_TC\_FOR\_TIMER1, IRQn\_FOR\_TIMER1, GCM\_FOR\_TIMER\_1, INTENSET\_BIT\_FOR\_TIMER\_1);

#endif

#if defined (\_useTimer2)

if (timer == \_timer2)

\_initISR(TC\_FOR\_TIMER2, CHANNEL\_FOR\_TIMER2, ID\_TC\_FOR\_TIMER2, IRQn\_FOR\_TIMER2, GCM\_FOR\_TIMER\_2, INTENSET\_BIT\_FOR\_TIMER\_2);

#endif

}

static void finISR(timer16\_Sequence\_t timer)

{

#if defined (\_useTimer1)

// Disable the match channel interrupt request

TC\_FOR\_TIMER1->COUNT16.INTENCLR.reg = INTENCLR\_BIT\_FOR\_TIMER\_1;

#endif

#if defined (\_useTimer2)

// Disable the match channel interrupt request

TC\_FOR\_TIMER2->COUNT16.INTENCLR.reg = INTENCLR\_BIT\_FOR\_TIMER\_2;

#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

} 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)

{

timer16\_Sequence\_t timer;

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

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()

{

timer16\_Sequence\_t timer;

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

timer = SERVO\_INDEX\_TO\_TIMER(servoIndex);

if(isTimerActive(timer) == false) {

finISR(timer);

}

}

void Servo::write(int value)

{

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

if (value < MIN\_PULSE\_WIDTH)

{

if (value < 0)

value = 0;

else if (value > 180)

value = 180;

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

}

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

servos[channel].ticks = value;

}

}

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

{

return map(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;

else

pulsewidth = 0;

return pulsewidth;

}

bool Servo::attached()

{

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

}

#endif // ARDUINO\_ARCH\_SAMD