#if defined(ARDUINO\_ARCH\_MEGAAVR)

#include <Arduino.h>

#include <Servo.h>

#define usToTicks(\_us) ((clockCyclesPerMicrosecond() / 16 \* \_us) / 4) // converts microseconds to ticks

#define ticksToUs(\_ticks) (((unsigned) \_ticks \* 16) / (clockCyclesPerMicrosecond() / 4)) // 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

#undef REFRESH\_INTERVAL

#define REFRESH\_INTERVAL 16000

void ServoHandler(int timer)

{

if (currentServoIndex[timer] < 0) {

// Write compare register

\_timer->CCMP = 0;

} 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 = 0; //\_timer->CCMP;

\_timer->CCMP = (uint16\_t) (tcCounterValue + SERVO(timer, currentServoIndex[timer]).ticks);

}

else {

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

// Get the counter value

uint16\_t tcCounterValue = \_timer->CCMP;

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

\_timer->CCMP = (uint16\_t) usToTicks(REFRESH\_INTERVAL);

}

else {

\_timer->CCMP = (uint16\_t) (tcCounterValue + 4UL); // at least REFRESH\_INTERVAL has elapsed

}

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

}

/\* Clear flag \*/

\_timer->INTFLAGS = TCB\_CAPT\_bm;

}

#if defined USE\_TIMERB0

ISR(TCB0\_INT\_vect)

#elif defined USE\_TIMERB1

ISR(TCB1\_INT\_vect)

#elif defined USE\_TIMERB2

ISR(TCB2\_INT\_vect)

#endif

{

ServoHandler(0);

}

static void initISR(timer16\_Sequence\_t timer)

{

//TCA0.SINGLE.CTRLA = (TCA\_SINGLE\_CLKSEL\_DIV16\_gc) | (TCA\_SINGLE\_ENABLE\_bm);

\_timer->CTRLA = TCB\_CLKSEL\_CLKTCA\_gc;

// Timer to Periodic interrupt mode

// This write will also disable any active PWM outputs

\_timer->CTRLB = TCB\_CNTMODE\_INT\_gc;

// Enable interrupt

\_timer->INTCTRL = TCB\_CAPTEI\_bm;

// Enable timer

\_timer->CTRLA |= TCB\_ENABLE\_bm;

}

static void finISR(timer16\_Sequence\_t timer)

{

// Disable interrupt

\_timer->INTCTRL = 0;

}

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