/\* The MIT License (MIT)

\*

\* Copyright (c) 2022 Arduino SA

\*

\* Permission is hereby granted, free of charge, to any person obtaining a copy

\* of this software and associated documentation files (the "Software"), to deal

\* in the Software without restriction, including without limitation the rights

\* to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

\* copies of the Software, and to permit persons to whom the Software is

\* furnished to do so, subject to the following conditions:

\*

\* The above copyright notice and this permission notice shall be included in

\* all copies or substantial portions of the Software.

\*

\* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

\* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

\* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

\* AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

\* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

\* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN

\* THE SOFTWARE.

\*/

#if defined(ARDUINO\_ARCH\_RENESAS)

#include "Arduino.h"

#include "Servo.h"

#include "ServoTimers.h"

#include "math.h"

#include "FspTimer.h"

#define SERVO\_MAX\_SERVOS (\_Nbr\_16timers \* SERVOS\_PER\_TIMER)

#define SERVO\_INVALID\_INDEX (255)

// Lower the timer ticks for finer resolution.

#define SERVO\_US\_PER\_CYCLE (20000)

#define SERVO\_IO\_PORT\_ADDR(pn) &((R\_PORT0 + ((uint32\_t) (R\_PORT1 - R\_PORT0) \* (pn)))->PCNTR3)

#define SERVO\_MIN\_CYCLE\_OFF\_US 50

// Internal Servo struct to keep track of RA configuration.

typedef struct {

// Servo period in microseconds.

uint32\_t period\_us;

// Store min/max pulse width here, because min/max in

// Servo class are not wide enough for the pulse width.

uint32\_t period\_min;

uint32\_t period\_max;

// Period period\_count in timer ticks.

uint32\_t period\_ticks;

// Internal FSP GPIO port/pin control bits.

volatile uint32\_t \*io\_port;

uint32\_t io\_mask;

} ra\_servo\_t;

// Keep track of the total number of servos attached.

static size\_t n\_servos=0;

static ra\_servo\_t ra\_servos[SERVO\_MAX\_SERVOS];

static FspTimer servo\_timer;

static bool servo\_timer\_started = false;

void servo\_timer\_callback(timer\_callback\_args\_t \*args);

static uint32\_t servo\_ticks\_per\_cycle = 0;

static uint32\_t min\_servo\_cycle\_low = 0;

static uint32\_t active\_servos\_mask = 0;

static uint32\_t active\_servos\_mask\_refresh = 0;

static uint32\_t us\_to\_ticks(uint32\_t time\_us) {

return ((float) servo\_ticks\_per\_cycle / (float) SERVO\_US\_PER\_CYCLE) \* time\_us;

}

static int servo\_timer\_config(uint32\_t period\_us)

{

static bool configured = false;

if (configured == false) {

// Configure and enable the servo timer.

uint8\_t type = 0;

int8\_t channel = FspTimer::get\_available\_timer(type);

if (channel != -1) {

servo\_timer.begin(TIMER\_MODE\_PERIODIC, type, channel,

1000000.0f/period\_us, 50.0f, servo\_timer\_callback, nullptr);

servo\_timer.set\_period\_buffer(false); // disable period buffering

servo\_timer.setup\_overflow\_irq(10);

servo\_timer.open();

servo\_timer.stop();

// Read the timer's period count.

servo\_ticks\_per\_cycle = servo\_timer.get\_period\_raw();

min\_servo\_cycle\_low = us\_to\_ticks(SERVO\_MIN\_CYCLE\_OFF\_US);

configured = true;

}

}

return configured ? 0 : -1;

}

static int servo\_timer\_start()

{

// Start the timer if it's not started

if (servo\_timer\_started == false &&

servo\_timer.start() == false) {

return -1;

}

servo\_timer\_started = true;

return 0;

}

static int servo\_timer\_stop()

{

// Start the timer if it's not started

if (servo\_timer\_started == true &&

servo\_timer.stop() == false) {

return -1;

}

servo\_timer\_started = false;

return 0;

}

inline static void servo\_timer\_set\_period(uint32\_t period) {

servo\_timer.set\_period(period);

}

void servo\_timer\_callback(timer\_callback\_args\_t \*args)

{

(void)args; // remove warning

static uint8\_t channel = SERVO\_MAX\_SERVOS;

static uint8\_t channel\_pin\_set\_high = 0xff;

static uint32\_t ticks\_accum = 0;

// See if we need to set a servo back low

if (channel\_pin\_set\_high != 0xff) {

\*ra\_servos[channel\_pin\_set\_high].io\_port = ra\_servos[channel\_pin\_set\_high].io\_mask << 16;

}

// Find the next servo to set high

while (active\_servos\_mask\_refresh) {

channel = \_\_builtin\_ctz(active\_servos\_mask\_refresh);

if (ra\_servos[channel].period\_us) {

\*ra\_servos[channel].io\_port = ra\_servos[channel].io\_mask;

servo\_timer\_set\_period(ra\_servos[channel].period\_ticks);

channel\_pin\_set\_high = channel;

ticks\_accum += ra\_servos[channel].period\_ticks;

active\_servos\_mask\_refresh &= ~(1 << channel);

return;

}

active\_servos\_mask\_refresh &= ~(1 << channel);

}

// Finished processing all servos, now delay to start of next pass.

ticks\_accum += min\_servo\_cycle\_low;

uint32\_t time\_to\_next\_cycle;

if (servo\_ticks\_per\_cycle > ticks\_accum) {

time\_to\_next\_cycle = servo\_ticks\_per\_cycle - ticks\_accum;

} else {

time\_to\_next\_cycle = min\_servo\_cycle\_low;

}

ticks\_accum = 0;

servo\_timer\_set\_period(time\_to\_next\_cycle);

channel\_pin\_set\_high = 0xff;

active\_servos\_mask\_refresh = active\_servos\_mask;

}

Servo::Servo()

{

servoIndex = SERVO\_INVALID\_INDEX;

}

uint8\_t Servo::attach(int pin)

{

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

}

bool Servo::attached()

{

return (servoIndex != SERVO\_INVALID\_INDEX);

}

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

{

//assert(pin < NUM\_DIGITAL\_PINS); ?

if (n\_servos == SERVO\_MAX\_SERVOS) {

return 0;

}

// Configure the servo timer.

if (servo\_timer\_config(SERVO\_US\_PER\_CYCLE) != 0) {

return 0;

}

// Try to find a free servo slot.

ra\_servo\_t \*servo = NULL;

bsp\_io\_port\_pin\_t io\_pin = g\_pin\_cfg[pin].pin;

for (size\_t i=0; i<SERVO\_MAX\_SERVOS; i++) {

servo = &ra\_servos[i];

if (servo->period\_us == 0) {

n\_servos++;

servoIndex = i;

servo->period\_min = min;

servo->period\_max = max;

servo->io\_mask = (1U << (io\_pin & 0xFF));

servo->io\_port = SERVO\_IO\_PORT\_ADDR(((io\_pin >> 8U) & 0xFF));

active\_servos\_mask |= (1 << i); // update mask of servos that are active.

writeMicroseconds(DEFAULT\_PULSE\_WIDTH);

break;

}

}

if (servoIndex == SERVO\_INVALID\_INDEX) {

return 0;

}

// Configure GPIO pin for the servo.

R\_IOPORT\_PinCfg(&g\_ioport\_ctrl, io\_pin,

IOPORT\_CFG\_PORT\_DIRECTION\_OUTPUT | IOPORT\_CFG\_PORT\_OUTPUT\_HIGH);

// Start the timer if it's not started.

if (servo\_timer\_start() != 0) {

return 0;

}

return 1;

}

void Servo::detach()

{

if (servoIndex != SERVO\_INVALID\_INDEX) {

ra\_servo\_t \*servo = &ra\_servos[servoIndex];

servo\_timer\_stop();

servo->period\_us = 0;

active\_servos\_mask &= ~(1 << servoIndex); // update mask of servos that are active.

servoIndex = SERVO\_INVALID\_INDEX;

if (--n\_servos) {

servo\_timer\_start();

}

}

}

void Servo::write(int angle)

{

if (servoIndex != SERVO\_INVALID\_INDEX) {

ra\_servo\_t \*servo = &ra\_servos[servoIndex];

angle = constrain(angle, 0, 180);

writeMicroseconds(map(angle, 0, 180, servo->period\_min, servo->period\_max));

}

}

int Servo::read()

{

if (servoIndex != SERVO\_INVALID\_INDEX) {

ra\_servo\_t \*servo = &ra\_servos[servoIndex];

return map(servo->period\_us, servo->period\_min, servo->period\_max, 0, 180);

}

return 0;

}

void Servo::writeMicroseconds(int us)

{

if (servoIndex != SERVO\_INVALID\_INDEX) {

ra\_servo\_t \*servo = &ra\_servos[servoIndex];

servo->period\_us = constrain(us, servo->period\_min, servo->period\_max);

servo->period\_ticks = us\_to\_ticks(servo->period\_us);

}

}

int Servo::readMicroseconds()

{

if (servoIndex != SERVO\_INVALID\_INDEX) {

ra\_servo\_t \*servo = &ra\_servos[servoIndex];

return servo->period\_us;

}

return 0;

}

#endif // defined(ARDUINO\_ARCH\_RENESAS)