35492 MP

Information including Drawings, Schematics, Links and Code (Software) Supplied or Referenced in this Document is supplied by MPJA inc. as a service to our customers and accuracy or usefulness is not guaranteed nor is it an Endorsement of any particular part, supplier or manufacturer. Use of information and suitability for any application is at users own discretion and user assumes all risk.

Information Subject to Change Without Notice
All rights are retained by the respective Owners/Author(s)

```
/*
 * max30102.c - Support for MAX30102 heart rate and pulse oximeter sensor
 *
 * Copyright (C) 2017 Matt Ranostay <matt@ranostay.consulting>
 *
 * Support for MAX30105 optical particle sensor
 * Copyright (C) 2017 Peter Meerwald-Stadler <pmeerw@pmeerw.net>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program 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 General Public License for more details.
 *
 * 7-bit I2C chip address: 0x57
 * TODO: proximity power saving feature
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <linux/delay.h>
#include <linux/err.h>
#include <linux/irq.h>
#include <linux/i2c.h>
#include <linux/mutex.h>
#include <linux/of.h>
#include <linux/regmap.h>
#include <linux/iio/iio.h>
#include <linux/iio/buffer.h>
#include <linux/iio/kfifo_buf.h>

#define MAX30102_REGMAP_NAME   "max30102_regmap"
#define MAX30102_DRV_NAME      "max30102"
#define MAX30102_PART_NUMBER   0x15

enum max30102_chip_id {
        max30102,
        max30105,
};

enum max3012_led_idx {
        MAX30102_LED_RED,
        MAX30102_LED_IR,
        MAX30105_LED_GREEN,
};

#define MAX30102_REG_INT_STATUS                       0x00
#define MAX30102_REG_INT_STATUS_PWR_RDY               BIT(0)
#define MAX30102_REG_INT_STATUS_PROX_INT      BIT(4)
#define MAX30102_REG_INT_STATUS_ALC_OVF               BIT(5)
#define MAX30102_REG_INT_STATUS_PPG_RDY               BIT(6)
#define MAX30102_REG_INT_STATUS_FIFO_RDY      BIT(7)

#define MAX30102_REG_INT_ENABLE                       0x02
#define MAX30102_REG_INT_ENABLE_PROX_INT_EN   BIT(4)
#define MAX30102_REG_INT_ENABLE_ALC_OVF_EN    BIT(5)
#define MAX30102_REG_INT_ENABLE_PPG_EN        BIT(6)
#define MAX30102_REG_INT_ENABLE_FIFO_EN               BIT(7)
#define MAX30102_REG_INT_ENABLE_MASK          0xf0
#define MAX30102_REG_INT_ENABLE_MASK_SHIFT    4

#define MAX30102_REG_FIFO_WR_PTR              0x04
#define MAX30102_REG_FIFO_OVR_CTR             0x05
#define MAX30102_REG_FIFO_RD_PTR              0x06
#define MAX30102_REG_FIFO_DATA                0x07
#define MAX30102_REG_FIFO_DATA_BYTES          3

#define MAX30102_REG_FIFO_CONFIG              0x08
#define MAX30102_REG_FIFO_CONFIG_AVG_4SAMPLES BIT(1)
#define MAX30102_REG_FIFO_CONFIG_AVG_SHIFT    5
#define MAX30102_REG_FIFO_CONFIG_AFULL        BIT(0)

#define MAX30102_REG_MODE_CONFIG              0x09
#define MAX30102_REG_MODE_CONFIG_MODE_NONE    0x00
#define MAX30102_REG_MODE_CONFIG_MODE_HR      0x02 /* red LED */
#define MAX30102_REG_MODE_CONFIG_MODE_HR_SPO2 0x03 /* red + IR LED */
#define MAX30102_REG_MODE_CONFIG_MODE_MULTI   0x07 /* multi-LED mode */
#define MAX30102_REG_MODE_CONFIG_MODE_MASK    GENMASK(2, 0)
#define MAX30102_REG_MODE_CONFIG_PWR          BIT(7)

#define MAX30102_REG_MODE_CONTROL_SLOT21      0x11 /* multi-LED control */
#define MAX30102_REG_MODE_CONTROL_SLOT43      0x12
#define MAX30102_REG_MODE_CONTROL_SLOT_MASK   (GENMASK(6, 4) | GENMASK(2, 0))
#define MAX30102_REG_MODE_CONTROL_SLOT_SHIFT  4

#define MAX30102_REG_SPO2_CONFIG              0x0a
#define MAX30102_REG_SPO2_CONFIG_PULSE_411_US 0x03
#define MAX30102_REG_SPO2_CONFIG_SR_400HZ     0x03
#define MAX30102_REG_SPO2_CONFIG_SR_MASK      0x07
#define MAX30102_REG_SPO2_CONFIG_SR_MASK_SHIFT        2
#define MAX30102_REG_SPO2_CONFIG_ADC_4096_STEPS       BIT(0)
#define MAX30102_REG_SPO2_CONFIG_ADC_MASK_SHIFT       5

#define MAX30102_REG_RED_LED_CONFIG           0x0c
#define MAX30102_REG_IR_LED_CONFIG            0x0d
#define MAX30105_REG_GREEN_LED_CONFIG         0x0e

#define MAX30102_REG_TEMP_CONFIG              0x21
#define MAX30102_REG_TEMP_CONFIG_TEMP_EN      BIT(0)

#define MAX30102_REG_TEMP_INTEGER             0x1f
#define MAX30102_REG_TEMP_FRACTION            0x20

#define MAX30102_REG_REV_ID                   0xfe
```

```
#define MAX30102_REG_PART_ID                  0xff

struct max30102_data {
        struct i2c_client *client;
        struct iio_dev *indio_dev;
        struct mutex lock;
        struct regmap *regmap;
        enum max30102_chip_id chip_id;

        u8 buffer[12];
        __be32 processed_buffer[3]; /* 3 x 18-bit (padded to 32-bits) */
};

static const struct regmap_config max30102_regmap_config = {
        .name = MAX30102_REGMAP_NAME,

        .reg_bits = 8,
        .val_bits = 8,
};

static const unsigned long max30102_scan_masks[] = {
        BIT(MAX30102_LED_RED) | BIT(MAX30102_LED_IR),
        0
};

static const unsigned long max30105_scan_masks[] = {
        BIT(MAX30102_LED_RED) | BIT(MAX30102_LED_IR),
        BIT(MAX30102_LED_RED) | BIT(MAX30102_LED_IR) |
                BIT(MAX30105_LED_GREEN),
        0
};

#define MAX30102_INTENSITY_CHANNEL(_si, _mod) { \
                .type = IIO_INTENSITY, \
                .channel2 = _mod, \
                .modified = 1, \
                .scan_index = _si, \
                .scan_type = { \
                        .sign = 'u', \
                        .shift = 8, \
                        .realbits = 18, \
                        .storagebits = 32, \
                        .endianness = IIO_BE, \
                }, \
        }

static const struct iio_chan_spec max30102_channels[] = {
        MAX30102_INTENSITY_CHANNEL(MAX30102_LED_RED, IIO_MOD_LIGHT_RED),
        MAX30102_INTENSITY_CHANNEL(MAX30102_LED_IR, IIO_MOD_LIGHT_IR),
        {
                .type = IIO_TEMP,
                .info_mask_separate =
                        BIT(IIO_CHAN_INFO_RAW) | BIT(IIO_CHAN_INFO_SCALE),
                .scan_index = -1,
        },
};

static const struct iio_chan_spec max30105_channels[] = {
        MAX30102_INTENSITY_CHANNEL(MAX30102_LED_RED, IIO_MOD_LIGHT_RED),
        MAX30102_INTENSITY_CHANNEL(MAX30102_LED_IR, IIO_MOD_LIGHT_IR),
        MAX30102_INTENSITY_CHANNEL(MAX30105_LED_GREEN, IIO_MOD_LIGHT_GREEN),
        {
                .type = IIO_TEMP,
                .info_mask_separate =
                        BIT(IIO_CHAN_INFO_RAW) | BIT(IIO_CHAN_INFO_SCALE),
                .scan_index = -1,
        },
};

static int max30102_set_power(struct max30102_data *data, bool en)
{
        return regmap_update_bits(data->regmap, MAX30102_REG_MODE_CONFIG,
                                  MAX30102_REG_MODE_CONFIG_PWR,
                                  en ? 0 : MAX30102_REG_MODE_CONFIG_PWR);
}

static int max30102_set_powermode(struct max30102_data *data, u8 mode, bool en)
{
        u8 reg = mode;

        if (!en)
                reg |= MAX30102_REG_MODE_CONFIG_PWR;

        return regmap_update_bits(data->regmap, MAX30102_REG_MODE_CONFIG,
                                  MAX30102_REG_MODE_CONFIG_PWR |
                                  MAX30102_REG_MODE_CONFIG_MODE_MASK, reg);
}

#define MAX30102_MODE_CONTROL_LED_SLOTS(slot2, slot1) \
        ((slot2 << MAX30102_REG_MODE_CONTROL_SLOT_SHIFT) | slot1)

static int max30102_buffer_postenable(struct iio_dev *indio_dev)
{
        struct max30102_data *data = iio_priv(indio_dev);
        int ret;
        u8 reg;

        switch (*indio_dev->active_scan_mask) {
        case BIT(MAX30102_LED_RED) | BIT(MAX30102_LED_IR):
                reg = MAX30102_REG_MODE_CONFIG_MODE_HR_SPO2;
                break;
        case BIT(MAX30102_LED_RED) | BIT(MAX30102_LED_IR) |
             BIT(MAX30105_LED_GREEN):
                ret = regmap_update_bits(data->regmap,
                                         MAX30102_REG_MODE_CONTROL_SLOT21,
                                         MAX30102_REG_MODE_CONTROL_SLOT_MASK,
                                         MAX30102_MODE_CONTROL_LED_SLOTS(2, 1));
                if (ret)
                        return ret;

                ret = regmap_update_bits(data->regmap,
                                         MAX30102_REG_MODE_CONTROL_SLOT43,
                                         MAX30102_REG_MODE_CONTROL_SLOT_MASK,
                                         MAX30102_MODE_CONTROL_LED_SLOTS(0, 3));
                if (ret)
                        return ret;

                reg = MAX30102_REG_MODE_CONFIG_MODE_MULTI;
                break;
        default:
                return -EINVAL;
        }
        return max30102_set_powermode(data, reg, true);
}

static int max30102_buffer_predisable(struct iio_dev *indio_dev)
{
        struct max30102_data *data = iio_priv(indio_dev);

        return max30102_set_powermode(data, MAX30102_REG_MODE_CONFIG_MODE_NONE,
                                      false);
}

static const struct iio_buffer_setup_ops max30102_buffer_setup_ops = {
        .postenable = max30102_buffer_postenable,
        .predisable = max30102_buffer_predisable,
};

static inline int max30102_fifo_count(struct max30102_data *data)
{
        unsigned int val;
        int ret;

        ret = regmap_read(data->regmap, MAX30102_REG_INT_STATUS, &val);
        if (ret)
                return ret;

        /* FIFO has one sample slot left */
        if (val & MAX30102_REG_INT_STATUS_FIFO_RDY)
                return 1;

        return 0;
}

#define MAX30102_COPY_DATA(i) \
        memcpy(&data->processed_buffer[(i)], \
               &buffer[(i) * MAX30102_REG_FIFO_DATA_BYTES], \
               MAX30102_REG_FIFO_DATA_BYTES)

static int max30102_read_measurement(struct max30102_data *data,
                                     unsigned int measurements)
{
        int ret;
        u8 *buffer = (u8 *) &data->buffer;

        ret = i2c_smbus_read_i2c_block_data(data->client,
                                            MAX30102_REG_FIFO_DATA,
                                            measurements *
                                            MAX30102_REG_FIFO_DATA_BYTES,
                                            buffer);

        switch (measurements) {
        case 3:
                MAX30102_COPY_DATA(2);
        case 2: /* fall-through */
                MAX30102_COPY_DATA(1);
        case 1: /* fall-through */
                MAX30102_COPY_DATA(0);
                break;
        default:
                return -EINVAL;
        }
        return (ret == measurements * MAX30102_REG_FIFO_DATA_BYTES) ?
               0 : -EINVAL;
}

static irqreturn_t max30102_interrupt_handler(int irq, void *private)
{
        struct iio_dev *indio_dev = private;
        struct max30102_data *data = iio_priv(indio_dev);
        unsigned int measurements = bitmap_weight(indio_dev->active_scan_mask,
                                                indio_dev->masklength);
        int ret, cnt = 0;

        mutex_lock(&data->lock);

        while (cnt || (cnt = max30102_fifo_count(data)) > 0) {
                ret = max30102_read_measurement(data, measurements);
                if (ret)
                        break;

                iio_push_to_buffers(data->indio_dev, data->processed_buffer);
                cnt--;
        }

        mutex_unlock(&data->lock);

        return IRQ_HANDLED;
}

static int max30102_get_current_idx(unsigned int val, int *reg)
{
        /* each step is 0.200 mA */
        *reg = val / 200;

        return *reg > 0xff ? -EINVAL : 0;
}

static int max30102_led_init(struct max30102_data *data)
{
        struct device *dev = &data->client->dev;
        struct device_node *np = dev->of_node;
        unsigned int val;
        int reg, ret;

        ret = of_property_read_u32(np, "maxim,red-led-current-microamp", &val);
        if (ret) {
                dev_info(dev, "no red-led-current-microamp set\n");

                /* Default to 7 mA RED LED */
                val = 7000;
        }

        ret = max30102_get_current_idx(val, &reg);
        if (ret) {
                dev_err(dev, "invalid RED LED current setting %d\n", val);
                return ret;
        }

        ret = regmap_write(data->regmap, MAX30102_REG_RED_LED_CONFIG, reg);
        if (ret)
                return ret;
        if (data->chip_id == max30105) {
                ret = of_property_read_u32(np,
                        "maxim,green-led-current-microamp", &val);
                if (ret) {
                        dev_info(dev, "no green-led-current-microamp set\n");

                        /* Default to 7 mA green LED */
                        val = 7000;
                }

                ret = max30102_get_current_idx(val, &reg);
                if (ret) {
                        dev_err(dev, "invalid green LED current setting %d\n",
                                val);
                        return ret;
                }

                ret = regmap_write(data->regmap, MAX30105_REG_GREEN_LED_CONFIG,
                                   reg);
                if (ret)
                        return ret;
        }

        ret = of_property_read_u32(np, "maxim,ir-led-current-microamp", &val);
        if (ret) {
                dev_info(dev, "no ir-led-current-microamp set\n");

                /* Default to 7 mA IR LED */
                val = 7000;
        }

        ret = max30102_get_current_idx(val, &reg);
        if (ret) {
                dev_err(dev, "invalid IR LED current setting %d\n", val);
                return ret;
        }

        return regmap_write(data->regmap, MAX30102_REG_IR_LED_CONFIG, reg);
}

static int max30102_chip_init(struct max30102_data *data)
{
        int ret;

        /* setup LED current settings */
        ret = max30102_led_init(data);
        if (ret)
                return ret;

        /* configure 18-bit HR + SpO2 readings at 400Hz */
        ret = regmap_write(data->regmap, MAX30102_REG_SPO2_CONFIG,
                                (MAX30102_REG_SPO2_CONFIG_ADC_4096_STEPS
                                 << MAX30102_REG_SPO2_CONFIG_ADC_MASK_SHIFT) |
                                (MAX30102_REG_SPO2_CONFIG_SR_400HZ
                                 << MAX30102_REG_SPO2_CONFIG_SR_MASK_SHIFT) |
                                 MAX30102_REG_SPO2_CONFIG_PULSE_411_US);
        if (ret)
                return ret;

        /* average 4 samples + generate FIFO interrupt */
        ret = regmap_write(data->regmap, MAX30102_REG_FIFO_CONFIG,
                               (MAX30102_REG_FIFO_CONFIG_AVG_4SAMPLES
                                << MAX30102_REG_FIFO_CONFIG_AVG_SHIFT) |
                                MAX30102_REG_FIFO_CONFIG_AFULL);
        if (ret)
                return ret;

        /* enable FIFO interrupt */
        return regmap_update_bits(data->regmap, MAX30102_REG_INT_ENABLE,
                                MAX30102_REG_INT_ENABLE_MASK,
                                MAX30102_REG_INT_ENABLE_FIFO_EN);
}

static int max30102_read_temp(struct max30102_data *data, int *val)
{
        int ret;
        unsigned int reg;

        ret = regmap_read(data->regmap, MAX30102_REG_TEMP_INTEGER, &reg);
        if (ret < 0)
                return ret;
        *val = reg << 4;

        ret = regmap_read(data->regmap, MAX30102_REG_TEMP_FRACTION, &reg);
        if (ret < 0)
                return ret;

        *val |= reg & 0xf;
        *val = sign_extend32(*val, 11);

        return 0;
}

static int max30102_get_temp(struct max30102_data *data, int *val, bool en)
{
        int ret;

        if (en) {
                ret = max30102_set_power(data, true);
                if (ret)
                        return ret;
        }

        /* start acquisition */
        ret = regmap_update_bits(data->regmap, MAX30102_REG_TEMP_CONFIG,
                                MAX30102_REG_TEMP_CONFIG_TEMP_EN,
                                MAX30102_REG_TEMP_CONFIG_TEMP_EN);
        if (ret)
                goto out;

        msleep(35);
        ret = max30102_read_temp(data, val);

out:
        if (en)
                max30102_set_power(data, false);

        return ret;
}

static int max30102_read_raw(struct iio_dev *indio_dev,
                             struct iio_chan_spec const *chan,
                             int *val, int *val2, long mask)
{
        struct max30102_data *data = iio_priv(indio_dev);
        int ret = -EINVAL;

        switch (mask) {
        case IIO_CHAN_INFO_RAW:
                /*
                 * Temperature reading can only be acquired when not in
                 * shutdown; leave shutdown briefly when buffer not running
                 */
                mutex_lock(&indio_dev->mlock);
                if (!iio_buffer_enabled(indio_dev))
                        ret = max30102_get_temp(data, val, true);
                else
                        ret = max30102_get_temp(data, val, false);
                mutex_unlock(&indio_dev->mlock);
                if (ret)
                        return ret;

                ret = IIO_VAL_INT;
                break;
        case IIO_CHAN_INFO_SCALE:
                *val = 1000;  /* 62.5 */
                *val2 = 16;
                ret = IIO_VAL_FRACTIONAL;
                break;
        }

        return ret;
}

static const struct iio_info max30102_info = {
        .read_raw = max30102_read_raw,
};

static int max30102_probe(struct i2c_client *client,
                          const struct i2c_device_id *id)
{
        struct max30102_data *data;
        struct iio_buffer *buffer;
        struct iio_dev *indio_dev;
        int ret;
        unsigned int reg;

        indio_dev = devm_iio_device_alloc(&client->dev, sizeof(*data));
        if (!indio_dev)
                return -ENOMEM;

        buffer = devm_iio_kfifo_allocate(&client->dev);
        if (!buffer)
                return -ENOMEM;

        iio_device_attach_buffer(indio_dev, buffer);

        indio_dev->name = MAX30102_DRV_NAME;
        indio_dev->info = &max30102_info;
        indio_dev->modes = (INDIO_BUFFER_SOFTWARE | INDIO_DIRECT_MODE);
        indio_dev->setup_ops = &max30102_buffer_setup_ops;
        indio_dev->dev.parent = &client->dev;
```

```
data = iio_priv(indio_dev);
data->indio_dev = indio_dev;
data->client = client;
data->chip_id = id->driver_data;

mutex_init(&data->lock);
i2c_set_clientdata(client, indio_dev);

switch (data->chip_id) {
case max30105:
        indio_dev->channels = max30105_channels;
        indio_dev->num_channels = ARRAY_SIZE(max30105_channels);
        indio_dev->available_scan_masks = max30105_scan_masks;
        break;
case max30102:
        indio_dev->channels = max30102_channels;
        indio_dev->num_channels = ARRAY_SIZE(max30102_channels);
        indio_dev->available_scan_masks = max30102_scan_masks;
        break;
default:
        return -ENODEV;
}

data->regmap = devm_regmap_init_i2c(client, &max30102_regmap_config);
if (IS_ERR(data->regmap)) {
        dev_err(&client->dev, "regmap initialization failed\n");
        return PTR_ERR(data->regmap);
}

/* check part ID */
ret = regmap_read(data->regmap, MAX30102_REG_PART_ID, &reg);
if (ret)
        return ret;
if (reg != MAX30102_PART_NUMBER)
        return -ENODEV;

/* show revision ID */
ret = regmap_read(data->regmap, MAX30102_REG_REV_ID, &reg);
if (ret)
        return ret;
dev_dbg(&client->dev, "max3010x revision %02x\n", reg);

/* clear mode setting, chip shutdown */
ret = max30102_set_powermode(data, MAX30102_REG_MODE_CONFIG_MODE_NONE,
                            false);
if (ret)
        return ret;

ret = max30102_chip_init(data);
if (ret)
        return ret;

if (client->irq <= 0) {
        dev_err(&client->dev, "no valid irq defined\n");
        return -EINVAL;
}

ret = devm_request_threaded_irq(&client->dev, client->irq,
                               NULL, max30102_interrupt_handler,
                               IRQF_TRIGGER_FALLING | IRQF_ONESHOT,
                                      "max30102_irq", indio_dev);
        if (ret) {
                dev_err(&client->dev, "request irq (%d) failed\n", client-
>irq);
                return ret;
        }

        return iio_device_register(indio_dev);
}

static int max30102_remove(struct i2c_client *client)
{
        struct iio_dev *indio_dev = i2c_get_clientdata(client);
        struct max30102_data *data = iio_priv(indio_dev);

        iio_device_unregister(indio_dev);
        max30102_set_power(data, false);

        return 0;
}

static const struct i2c_device_id max30102_id[] = {
        { "max30102", max30102 },
        { "max30105", max30105 },
        {}
};
MODULE_DEVICE_TABLE(i2c, max30102_id);

static const struct of_device_id max30102_dt_ids[] = {
        { .compatible = "maxim,max30102" },
        { .compatible = "maxim,max30105" },
        { }
};
MODULE_DEVICE_TABLE(of, max30102_dt_ids);

static struct i2c_driver max30102_driver = {
        .driver = {
                .name   = MAX30102_DRV_NAME,
                .of_match_table= of_match_ptr(max30102_dt_ids),
        },
        .probe          = max30102_probe,
        .remove         = max30102_remove,
        .id_table       = max30102_id,
};
module_i2c_driver(max30102_driver);

MODULE_AUTHOR("Matt Ranostay <matt@ranostay.consulting>");
MODULE_DESCRIPTION("MAX30102 heart rate/pulse oximeter and MAX30105 particle
sensor driver");
MODULE_LICENSE("GPL");
```