Google Git

Sign in

android / kernel / msm.git / eaf36994a3992b8f918c18e4f7411e8b2320a35f / . / drivers / misc / mpu6050 / mldl_cfg.c

blob: 7f806f33f2e29e8a5e6ff6f75f0e7b3edac55cf5 [file] [log] [blame]

```
/*
      $License:
      Copyright (C) 2011 InvenSense Corporation, All Rights
Reserved.
      Copyright (C) 2012 Sony Mobile Communications AB.
      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.
      You should have received a copy of the GNU General Public
License
      along with this program.  If not, see
<http://www.gnu.org/licenses/>.
      $
 */
/**
 *  @addtogroup MLDL
 *
 *  @{
 *      @file   mldl_cfg.c
 *      @brief  The Motion Library Driver Layer.
 */
/* ------------------------------------------------------------------
----------- */
#include <linux/delay.h>
#include <linux/slab.h>
#include <stddef.h>
#include "mldl_cfg.h"
```

```
#include <linux/mpu.h>
#include "mpu6050.h"
#include "mlsl.h"
#include "mldl_print_cfg.h"
#include "log.h"
#undef MPL_LOG_TAG
#define MPL_LOG_TAG "mldl_cfg:"
/* ---------------------------------------------------------------
----------- */
#define SLEEP   0
#define WAKE_UP 7
#define RESET   1
#define STANDBY 1
#define CHARGEPUMP_WAKE 10
/* ---------------------------------------------------------------
----------- */
/**
 * @brief Stop the DMP running
 *
 * @return INV_SUCCESS or non-zero error code
 */
static int dmp_stop(struct mldl_cfg *mldl_cfg, void *gyro_handle)
{
      unsigned char user_ctrl_reg;
      int result;
      if (mldl_cfg->inv_mpu_state->status & MPU_DMP_IS_SUSPENDED)
            return INV_SUCCESS;
      result = inv_serial_read(gyro_handle, mldl_cfg-
>mpu_chip_info->addr,
                         MPUREG_USER_CTRL, 1, &user_ctrl_reg);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      user_ctrl_reg = (user_ctrl_reg & (~BIT_FIFO_EN)) |
BIT_FIFO_RST;
      user_ctrl_reg = (user_ctrl_reg & (~BIT_DMP_EN)) |
BIT_DMP_RST;
      result = inv_serial_single_write(gyro_handle,
                               mldl_cfg->mpu_chip_info->addr,
                               MPUREG_USER_CTRL, user_ctrl_reg);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      mldl_cfg->inv_mpu_state->status |= MPU_DMP_IS_SUSPENDED;
      return result;
}
/**
 * @brief Starts the DMP running
 *
 * @return INV_SUCCESS or non-zero error code
 */
static int dmp_start(struct mldl_cfg *mldl_cfg, void *mlsl_handle)
{
      unsigned char user_ctrl_reg;
      int result;
      if ((!(mldl_cfg->inv_mpu_state->status &
MPU_DMP_IS_SUSPENDED) &&
          mldl_cfg->mpu_gyro_cfg->dmp_enable)
            ||
         ((mldl_cfg->inv_mpu_state->status & MPU_DMP_IS_SUSPENDED)
&&
               !mldl_cfg->mpu_gyro_cfg->dmp_enable))
            return INV_SUCCESS;
      result = inv_serial_read(mlsl_handle, mldl_cfg-
>mpu_chip_info->addr,
                         MPUREG_USER_CTRL, 1, &user_ctrl_reg);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      result = inv_serial_single_write(
            mlsl_handle, mldl_cfg->mpu_chip_info->addr,
            MPUREG_USER_CTRL,
            ((user_ctrl_reg & (~BIT_FIFO_EN))
                  | BIT_FIFO_RST));
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      result = inv_serial_single_write(
            mlsl_handle, mldl_cfg->mpu_chip_info->addr,
            MPUREG_USER_CTRL, user_ctrl_reg);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      result = inv_serial_read(mlsl_handle, mldl_cfg-
>mpu_chip_info->addr,
                         MPUREG_USER_CTRL, 1, &user_ctrl_reg);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      user_ctrl_reg |= BIT_DMP_EN;
      if (mldl_cfg->mpu_gyro_cfg->fifo_enable)
            user_ctrl_reg |= BIT_FIFO_EN;
      else
            user_ctrl_reg &= ~BIT_FIFO_EN;
      user_ctrl_reg |= BIT_DMP_RST;
      result = inv_serial_single_write(
            mlsl_handle, mldl_cfg->mpu_chip_info->addr,
            MPUREG_USER_CTRL, user_ctrl_reg);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      mldl_cfg->inv_mpu_state->status &= ~MPU_DMP_IS_SUSPENDED;
      return result;
}
/**
 *  @brief  enables/disables the I2C bypass to an external device
 *          connected to MPU's secondary I2C bus.
 *  @param  enable
 *              Non-zero to enable pass through.
 *  @return INV_SUCCESS if successful, a non-zero error code
otherwise.
 */
static int mpu6050b1_set_i2c_bypass(struct mldl_cfg *mldl_cfg,
                            void *mlsl_handle, unsigned char
enable)
{
      unsigned char reg;
      int result;
      unsigned char status = mldl_cfg->inv_mpu_state->status;
      if ((status & MPU_GYRO_IS_BYPASSED && enable) ||
          (!(status & MPU_GYRO_IS_BYPASSED) && !enable))
            return INV_SUCCESS;
      /*---- get current 'USER_CTRL' into b ----*/
      result = inv_serial_read(mlsl_handle, mldl_cfg-
>mpu_chip_info->addr,
                         MPUREG_USER_CTRL, 1, &reg);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      if (!enable) {
            /* setting int_config with the property flag
BIT_BYPASS_EN
               should be done by the setup functions */
            result = inv_serial_single_write(
                  mlsl_handle, mldl_cfg->mpu_chip_info->addr,
                  MPUREG_INT_PIN_CFG,
                  (mldl_cfg->pdata->int_config &
~(BIT_BYPASS_EN)));
            if (!(reg & BIT_I2C_MST_EN)) {
                  result =
                      inv_serial_single_write(
                            mlsl_handle, mldl_cfg->mpu_chip_info-
>addr,
                            MPUREG_USER_CTRL,
                            (reg | BIT_I2C_MST_EN));
                  if (result) {
                        LOG_RESULT_LOCATION(result);
                        return result;
                  }
            }
      } else if (enable) {
            if (reg & BIT_AUX_IF_EN) {
                  result =
                      inv_serial_single_write(
                            mlsl_handle, mldl_cfg->mpu_chip_info-
>addr,
                            MPUREG_USER_CTRL,
                            (reg & (~BIT_I2C_MST_EN)));
                  if (result) {
                        LOG_RESULT_LOCATION(result);
                        return result;
                  }
                  /*******************************************
                   * To avoid hanging the bus we must sleep until
all
                   * slave transactions have been completed.
                   *  24 bytes max slave reads
                   *  +1 byte possible extra write
                   *  +4 max slave address
                   * ---
                   *  33 Maximum bytes
                   *  x9 Approximate bits per byte
                   * ---
                   * 297 bits.
                   * 2.97 ms minimum @ 100kbps
                   * 0.75 ms minimum @ 400kbps.
                   ****************************************/
                  msleep(3);
            }
            result = inv_serial_single_write(
                  mlsl_handle, mldl_cfg->mpu_chip_info->addr,
                  MPUREG_INT_PIN_CFG,
                  (mldl_cfg->pdata->int_config | BIT_BYPASS_EN));
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
      }
      if (enable)
            mldl_cfg->inv_mpu_state->status |=
MPU_GYRO_IS_BYPASSED;
      else
            mldl_cfg->inv_mpu_state->status &=
~MPU_GYRO_IS_BYPASSED;
      return result;
}
/**
 *  @brief  enables/disables the I2C bypass to an external device
 *          connected to MPU's secondary I2C bus.
 *  @param  enable
 *              Non-zero to enable pass through.
 *  @return INV_SUCCESS if successful, a non-zero error code
otherwise.
 */
static int mpu_set_i2c_bypass(struct mldl_cfg *mldl_cfg, void
*mlsl_handle,
                        unsigned char enable)
{
      return mpu6050b1_set_i2c_bypass(mldl_cfg, mlsl_handle,
enable);
}
#define NUM_OF_PROD_REVS (ARRAY_SIZE(prod_rev_map))
```

```
#define NOTFOUND_PROD_REVS -1
/* NOTE : when not indicated, product revision
        is considered an 'npp'; non production part */
/* produces an unique identifier for each device based on the
   combination of product version and product revision */
struct prod_rev_map_t {
      unsigned short mpl_product_key;
      unsigned char silicon_rev;
      unsigned short gyro_trim;
      unsigned short accel_trim;
};
/* NOTE: product entries are in chronological order */
static struct prod_rev_map_t prod_rev_map[] = {
      /* prod_ver = 0 */
      {MPL_PROD_KEY(0,  1), MPU_SILICON_REV_A2, 131, 16384},
      {MPL_PROD_KEY(0,  2), MPU_SILICON_REV_A2, 131, 16384},
      {MPL_PROD_KEY(0,  3), MPU_SILICON_REV_A2, 131, 16384},
      {MPL_PROD_KEY(0,  4), MPU_SILICON_REV_A2, 131, 16384},
      {MPL_PROD_KEY(0,  5), MPU_SILICON_REV_A2, 131, 16384},
      {MPL_PROD_KEY(0,  6), MPU_SILICON_REV_A2, 131, 16384},
      /* (A2/C2-1) */
      /* prod_ver = 1, forced to 0 for MPU6050 A2 */
      {MPL_PROD_KEY(0,  7), MPU_SILICON_REV_A2, 131, 16384},
      {MPL_PROD_KEY(0,  8), MPU_SILICON_REV_A2, 131, 16384},
      {MPL_PROD_KEY(0,  9), MPU_SILICON_REV_A2, 131, 16384},
      {MPL_PROD_KEY(0, 10), MPU_SILICON_REV_A2, 131, 16384},
      {MPL_PROD_KEY(0, 11), MPU_SILICON_REV_A2, 131, 16384},
      /* (A2/D2-1) */
      {MPL_PROD_KEY(0, 12), MPU_SILICON_REV_A2, 131, 16384},
      {MPL_PROD_KEY(0, 13), MPU_SILICON_REV_A2, 131, 16384},
      {MPL_PROD_KEY(0, 14), MPU_SILICON_REV_A2, 131, 16384},
      {MPL_PROD_KEY(0, 15), MPU_SILICON_REV_A2, 131, 16384},
      {MPL_PROD_KEY(0, 27), MPU_SILICON_REV_A2, 131, 16384},
      /* (A2/D4)   */
      /* prod_ver = 1 */
      {MPL_PROD_KEY(1, 16), MPU_SILICON_REV_B1, 131, 16384},
      /* (B1/D2-1) */
      {MPL_PROD_KEY(1, 17), MPU_SILICON_REV_B1, 131, 16384},
      /* (B1/D2-2) */
      {MPL_PROD_KEY(1, 18), MPU_SILICON_REV_B1, 131, 16384},
      /* (B1/D2-3) */
      {MPL_PROD_KEY(1, 19), MPU_SILICON_REV_B1, 131, 16384},
      /* (B1/D2-4) */
```

```
{MPL_PROD_KEY(1, 20), MPU_SILICON_REV_B1, 131, 16384},
/* (B1/D2-5) */
{MPL_PROD_KEY(1, 28), MPU_SILICON_REV_B1, 131, 16384},
/* (B1/D4)    */
{MPL_PROD_KEY(1,  1), MPU_SILICON_REV_B1, 131, 16384},
/* (B1/E1-1) */
{MPL_PROD_KEY(1,  2), MPU_SILICON_REV_B1, 131, 16384},
/* (B1/E1-2) */
{MPL_PROD_KEY(1,  3), MPU_SILICON_REV_B1, 131, 16384},
/* (B1/E1-3) */
{MPL_PROD_KEY(1,  4), MPU_SILICON_REV_B1, 131, 16384},
/* (B1/E1-4) */
{MPL_PROD_KEY(1,  5), MPU_SILICON_REV_B1, 131, 16384},
/* (B1/E1-5) */
{MPL_PROD_KEY(1,  6), MPU_SILICON_REV_B1, 131, 16384},
/* (B1/E1-6) */
/* prod_ver = 2 */
{MPL_PROD_KEY(2,  7), MPU_SILICON_REV_B1, 131, 16384},
/* (B2/E1-1) */
{MPL_PROD_KEY(2,  8), MPU_SILICON_REV_B1, 131, 16384},
/* (B2/E1-2) */
{MPL_PROD_KEY(2,  9), MPU_SILICON_REV_B1, 131, 16384},
/* (B2/E1-3) */
{MPL_PROD_KEY(2, 10), MPU_SILICON_REV_B1, 131, 16384},
/* (B2/E1-4) */
{MPL_PROD_KEY(2, 11), MPU_SILICON_REV_B1, 131, 16384},
/* (B2/E1-5) */
{MPL_PROD_KEY(2, 12), MPU_SILICON_REV_B1, 131, 16384},
/* (B2/E1-6) */
{MPL_PROD_KEY(2, 29), MPU_SILICON_REV_B1, 131, 16384},
/* (B2/D4)    */
/* prod_ver = 3 */
{MPL_PROD_KEY(3, 30), MPU_SILICON_REV_B1, 131, 16384},
/* (B2/E2)    */
/* prod_ver = 4 */
{MPL_PROD_KEY(4, 31), MPU_SILICON_REV_B1, 131,  8192},
/* (B2/F1)    */
{MPL_PROD_KEY(4,  1), MPU_SILICON_REV_B1, 131,  8192},
/* (B3/F1)    */
{MPL_PROD_KEY(4,  3), MPU_SILICON_REV_B1, 131,  8192},
/* (B4/F1)    */
/* prod_ver = 6 */
{MPL_PROD_KEY(6, 19), MPU_SILICON_REV_B1, 131, 16384},
/* (B5/E2)    */
```

```
      /* prod_ver = 7 */
      {MPL_PROD_KEY(7, 19), MPU_SILICON_REV_B1, 131, 16384},
      /* (B5/E2)   */
      /* prod_ver = 8 */
      {MPL_PROD_KEY(8, 19), MPU_SILICON_REV_B1, 131, 16384},
      /* (B5/E2)   */
      /* prod_ver = 9*/
      {MPL_PROD_KEY(9, 19), MPU_SILICON_REV_B1, 131, 16384},
      /* (B5/E2)   */
      /* prod_ver = 10 */
      {MPL_PROD_KEY(10, 19), MPU_SILICON_REV_B1, 131, 16384}
      /* (B5/E2)   */
};
/*
   List of product software revisions
   NOTE :
   software revision 0 falls back to the old detection method
   based off the product version and product revision per the
   table above
*/
static struct prod_rev_map_t sw_rev_map[] = {
      {0,              0,   0,     0},
      {1, MPU_SILICON_REV_B1, 131,  8192},      /* rev C */
      {2, MPU_SILICON_REV_B1, 131, 16384} /* rev D */
};
/**
 *  @internal
 *  @brief  Inverse lookup of the index of an MPL product key .
 *  @param  key
 *              the MPL product indentifier also referred to as
'key'.
 *  @return the index position of the key in the array, -1 if not
found.
 */
short index_of_key(unsigned short key)
{
      int i;
      for (i = 0; i < NUM_OF_PROD_REVS; i++)
            if (prod_rev_map[i].mpl_product_key == key)
                  return (short)i;
      return NOTFOUND_PROD_REVS;
}
/**
 *  @internal
 *  @brief  Get the product revision and version for MPU6050 and
 *          extract all per-part specific information.
 *          The product version number is read from the PRODUCT_ID
register in
 *          user space register map.
 *          The product revision number is in read from OTP bank
0, ADDR6[7:2].
 *          These 2 numbers, combined, provide an unique key to be
used to
 *          retrieve some per-device information such as the
silicon revision
 *          and the gyro and accel sensitivity trim values.
 *
 *  @param  mldl_cfg
 *              a pointer to the mldl config data structure.
 *  @param  mlsl_handle
 *              an file handle to the serial communication device
the
 *              device is connected to.
 *
 *  @return 0 on success, a non-zero error code otherwise.
 */
static int inv_get_silicon_rev_mpu6050(
            struct mldl_cfg *mldl_cfg, void *mlsl_handle)
{
      unsigned char prod_ver, prod_rev;
      struct prod_rev_map_t *p_rev;
      unsigned sw_rev;
      unsigned short key;
      unsigned char bank =
          (BIT_PRFTCH_EN | BIT_CFG_USER_BANK |
MPU_MEM_OTP_BANK_0);
      unsigned short mem_addr = ((bank << 8) | 0x06);
      short index;
      unsigned char regs[5];
      struct mpu_chip_info *mpu_chip_info = mldl_cfg-
>mpu_chip_info;
      int result;
      result = inv_serial_read(mlsl_handle, mldl_cfg-
>mpu_chip_info->addr,
                         MPUREG_PRODUCT_ID, 1, &prod_ver);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      prod_ver &= 0xF;
      result = inv_serial_read_mem(mlsl_handle, mldl_cfg-
>mpu_chip_info->addr,
                             mem_addr, 1, &prod_rev);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      prod_rev >>= 2;
      /* clean the prefetch and cfg user bank bits */
      result = inv_serial_single_write(
            mlsl_handle, mldl_cfg->mpu_chip_info->addr,
            MPUREG_BANK_SEL, 0);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      /* get the software-product version */
      result = inv_serial_read(mlsl_handle, mldl_cfg-
>mpu_chip_info->addr,
                         MPUREG_XA_OFFS_L, 5, regs);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      sw_rev = (regs[4] & 0x01) << 2 |    /* 0x0b, bit 0 */
             (regs[2] & 0x01) << 1 |      /* 0x09, bit 0 */
             (regs[0] & 0x01);            /* 0x07, bit 0 */
      /* if 0, use the product key to determine the type of part
*/
      if (sw_rev == 0) {
            key = MPL_PROD_KEY(prod_ver, prod_rev);
            if (key == 0) {
                  MPL_LOGE("Product id read as 0 "
                         "indicates device is either "
                         "incompatible or an MPU3050\n");
                  return INV_ERROR_INVALID_MODULE;
            }
            index = index_of_key(key);
            if (index == -1 || index >= NUM_OF_PROD_REVS) {
                  MPL_LOGE("Unsupported product key %d in MPL\n",
key);
                  return INV_ERROR_INVALID_MODULE;
            }
            /* check MPL is compiled for this device */
            if (prod_rev_map[index].silicon_rev !=
MPU_SILICON_REV_B1) {
                  MPL_LOGE("MPL compiled for MPU6050B1 support "
                         "but device is not MPU6050B1 (%d)\n",
key);
                  return INV_ERROR_INVALID_MODULE;
            }
            p_rev = &prod_rev_map[index];
      /* if valid, use the software product key */
      } else if (sw_rev < ARRAY_SIZE(sw_rev_map)) {
            p_rev = &sw_rev_map[sw_rev];
      } else {
            MPL_LOGE("Software revision key is outside of known "
                "range [0..%d] : %d\n", ARRAY_SIZE(sw_rev_map),
sw_rev);
            return INV_ERROR_INVALID_MODULE;
      }
      mpu_chip_info->product_id = prod_ver;
      mpu_chip_info->product_revision = prod_rev;
      mpu_chip_info->silicon_revision = p_rev->silicon_rev;
      mpu_chip_info->gyro_sens_trim = p_rev->gyro_trim;
      mpu_chip_info->accel_sens_trim = p_rev->accel_trim;
      return result;
}
#define inv_get_silicon_rev inv_get_silicon_rev_mpu6050
/**
 *  @brief  Enable / Disable the use MPU's secondary I2C interface
level
 *          shifters.
 *          When enabled the secondary I2C interface to which the
external
 *          device is connected runs at VDD voltage (main supply).
 *          When disabled the 2nd interface runs at VDDIO voltage.
 *          See the device specification for more details.
 *
 *  @note   using this API may produce unpredictable results,
depending on how
 *          the MPU and slave device are setup on the target
platform.
 *          Use of this API should entirely be restricted to
system
```

```
 *          integrators. Once the correct value is found, there
should be no
 *          need to change the level shifter at runtime.
 *
 *  @pre    Must be called after inv_serial_start().
 *  @note   Typically called before inv_dmp_open().
 *
 *  @param[in]  enable:
 *                  0 to run at VDDIO (default),
 *                  1 to run at VDD.
 *
 *  @return INV_SUCCESS if successfull, a non-zero error code
otherwise.
 */
static int inv_mpu_set_level_shifter_bit(struct mldl_cfg
*mldl_cfg,
                          void *mlsl_handle, unsigned char enable)
{
      int result;
      unsigned char regval;
      result = inv_serial_read(mlsl_handle, mldl_cfg-
>mpu_chip_info->addr,
                         MPUREG_YG_OFFS_TC, 1, &regval);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      if (enable)
            regval |= BIT_I2C_MST_VDDIO;
      result = inv_serial_single_write(
            mlsl_handle, mldl_cfg->mpu_chip_info->addr,
            MPUREG_YG_OFFS_TC, regval);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      return INV_SUCCESS;
}
/**
 * @internal
 * @brief MPU6050 B1 power management functions.
 * @param mldl_cfg
 *          a pointer to the internal mldl_cfg data structure.
 * @param mlsl_handle
 *          a file handle to the serial device used to communicate
 *          with the MPU6050 B1 device.
 * @param reset
 *          1 to reset hardware.
 * @param sensors
 *          Bitfield of sensors to leave on
 *
 * @return 0 on success, a non-zero error code on error.
 */
static int mpu60xx_pwr_mgmt(struct mldl_cfg *mldl_cfg,
                            void *mlsl_handle,
                            unsigned int reset, unsigned long
sensors)
{
      unsigned char pwr_mgmt[2];
      unsigned char pwr_mgmt_prev[2];
      int result;
      int sleep = !(sensors & (INV_THREE_AXIS_GYRO |
INV_THREE_AXIS_ACCEL
                        | INV_DMP_PROCESSOR));
      if (reset) {
            MPL_LOGI("Reset MPU6050 B1\n");
            result = inv_serial_single_write(
                  mlsl_handle, mldl_cfg->mpu_chip_info->addr,
                  MPUREG_PWR_MGMT_1, BIT_H_RESET);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
            mldl_cfg->inv_mpu_state->status &=
~MPU_GYRO_IS_BYPASSED;
            msleep(100);
      }
      /* NOTE : reading both PWR_MGMT_1 and PWR_MGMT_2 for
efficiency because
              they are accessible even when the device is powered
off */
      result = inv_serial_read(mlsl_handle, mldl_cfg-
>mpu_chip_info->addr,
                         MPUREG_PWR_MGMT_1, 2, pwr_mgmt_prev);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      pwr_mgmt[0] = pwr_mgmt_prev[0];
      pwr_mgmt[1] = pwr_mgmt_prev[1];
      if (sleep) {
            mldl_cfg->inv_mpu_state->status |=
MPU_DEVICE_IS_SUSPENDED;
            pwr_mgmt[0] |= BIT_SLEEP;
      } else {
            mldl_cfg->inv_mpu_state->status &=
~MPU_DEVICE_IS_SUSPENDED;
            pwr_mgmt[0] &= ~BIT_SLEEP;
      }
      if (pwr_mgmt[0] != pwr_mgmt_prev[0]) {
            result = inv_serial_single_write(
                  mlsl_handle, mldl_cfg->mpu_chip_info->addr,
                  MPUREG_PWR_MGMT_1, pwr_mgmt[0]);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
      }
      msleep(CHARGEPUMP_WAKE);
      pwr_mgmt[1] &= ~(BIT_STBY_XG | BIT_STBY_YG | BIT_STBY_ZG);
      if (!(sensors & INV_X_GYRO))
            pwr_mgmt[1] |= BIT_STBY_XG;
      if (!(sensors & INV_Y_GYRO))
            pwr_mgmt[1] |= BIT_STBY_YG;
      if (!(sensors & INV_Z_GYRO))
            pwr_mgmt[1] |= BIT_STBY_ZG;
      if (pwr_mgmt[1] != pwr_mgmt_prev[1]) {
            result = inv_serial_single_write(
                  mlsl_handle, mldl_cfg->mpu_chip_info->addr,
                  MPUREG_PWR_MGMT_2, pwr_mgmt[1]);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
      }
      if ((pwr_mgmt[1] & (BIT_STBY_XG | BIT_STBY_YG |
BIT_STBY_ZG)) ==
          (BIT_STBY_XG | BIT_STBY_YG | BIT_STBY_ZG)) {
            mldl_cfg->inv_mpu_state->status |=
MPU_GYRO_IS_SUSPENDED;
      } else {
            mldl_cfg->inv_mpu_state->status &=
~MPU_GYRO_IS_SUSPENDED;
      }
      return INV_SUCCESS;
}
/**
 *  @brief  sets the clock source for the gyros.
 *  @param  mldl_cfg
 *                a pointer to the struct mldl_cfg data structure.
 *  @param  gyro_handle
 *                an handle to the serial device the gyro is
assigned to.
 *  @return ML_SUCCESS if successful, a non-zero error code
otherwise.
 */
static int mpu_set_clock_source(void *gyro_handle, struct mldl_cfg
*mldl_cfg)
{
      int result;
      unsigned char cur_clk_src;
      unsigned char reg;
      /* clock source selection */
      result = inv_serial_read(gyro_handle, mldl_cfg-
>mpu_chip_info->addr,
                         MPUREG_PWR_MGM, 1, &reg);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      cur_clk_src = reg & BITS_CLKSEL;
      reg &= ~BITS_CLKSEL;
      result = inv_serial_single_write(
            gyro_handle, mldl_cfg->mpu_chip_info->addr,
            MPUREG_PWR_MGM, mldl_cfg->mpu_gyro_cfg->clk_src |
reg);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      /* ERRATA:
         workaroud to switch from any MPU_CLK_SEL_PLLGYROx to
         MPU_CLK_SEL_INTERNAL and XGyro is powered up:
         1) Select INT_OSC
         2) PD XGyro
         3) PU XGyro
       */
      if ((cur_clk_src == MPU_CLK_SEL_PLLGYROX
             || cur_clk_src == MPU_CLK_SEL_PLLGYROY
             || cur_clk_src == MPU_CLK_SEL_PLLGYROZ)
          && mldl_cfg->mpu_gyro_cfg->clk_src ==
MPU_CLK_SEL_INTERNAL
          && mldl_cfg->inv_mpu_cfg->requested_sensors &
INV_X_GYRO) {
            unsigned char first_result = INV_SUCCESS;
            mldl_cfg->inv_mpu_cfg->requested_sensors &=
~INV_X_GYRO;
            result = mpu60xx_pwr_mgmt(
                  mldl_cfg, gyro_handle,
                  false, mldl_cfg->inv_mpu_cfg-
>requested_sensors);
            ERROR_CHECK_FIRST(first_result, result);
            mldl_cfg->inv_mpu_cfg->requested_sensors |=
INV_X_GYRO;
            result = mpu60xx_pwr_mgmt(
                  mldl_cfg, gyro_handle,
                  false, mldl_cfg->inv_mpu_cfg-
>requested_sensors);
            ERROR_CHECK_FIRST(first_result, result);
            result = first_result;
      }
      return result;
}
/**
 * Configures the MPU I2C Master
 *
 * @mldl_cfg Handle to the configuration data
 * @gyro_handle handle to the gyro communictation interface
 * @slave Can be Null if turning off the slave
 * @slave_pdata Can be null if turning off the slave
 * @slave_id enum ext_slave_type to determine which index to use
 *
 *
 * This fucntion configures the slaves by:
 * 1) Setting up the read
 *    a) Read Register
 *    b) Read Length
 * 2) Set up the data trigger (MPU6050 only)
 *    a) Set trigger write register
 *    b) Set Trigger write value
 * 3) Set up the divider (MPU6050 only)
 * 4) Set the slave bypass mode depending on slave
 *
 * returns INV_SUCCESS or non-zero error code
 */
static int mpu_set_slave_mpu60xx(struct mldl_cfg *mldl_cfg,
                         void *gyro_handle,
                         struct ext_slave_descr *slave,
                         struct ext_slave_platform_data
*slave_pdata,
                         int slave_id)
{
      int result;
      unsigned char reg;
      /* Slave values */
      unsigned char slave_reg;
      unsigned char slave_len;
      unsigned char slave_endian;
      unsigned char slave_address;
      /* Which MPU6050 registers to use */
      unsigned char addr_reg;
      unsigned char reg_reg;
      unsigned char ctrl_reg;
      /* Which MPU6050 registers to use for the trigger */
      unsigned char addr_trig_reg;
      unsigned char reg_trig_reg;
      unsigned char ctrl_trig_reg;
      unsigned char bits_slave_delay = 0;
      /* Divide down rate for the Slave, from the mpu rate */
      unsigned char d0_trig_reg;
      unsigned char delay_ctrl_orig;
      unsigned char delay_ctrl;
      long divider;
      if (NULL == slave || NULL == slave_pdata) {
            slave_reg = 0;
            slave_len = 0;
            slave_endian = 0;
            slave_address = 0;
      } else {
            slave_reg = slave->read_reg;
            slave_len = slave->read_len;
            slave_endian = slave->endian;
            slave_address = slave_pdata->address;
            slave_address |= BIT_I2C_READ;
      }
      switch (slave_id) {
      case EXT_SLAVE_TYPE_ACCEL:
            addr_reg = MPUREG_I2C_SLV1_ADDR;
            reg_reg  = MPUREG_I2C_SLV1_REG;
            ctrl_reg = MPUREG_I2C_SLV1_CTRL;
            addr_trig_reg = 0;
            reg_trig_reg  = 0;
            ctrl_trig_reg = 0;
            bits_slave_delay = BIT_SLV1_DLY_EN;
            break;
      case EXT_SLAVE_TYPE_COMPASS:
            addr_reg = MPUREG_I2C_SLV0_ADDR;
            reg_reg  = MPUREG_I2C_SLV0_REG;
            ctrl_reg = MPUREG_I2C_SLV0_CTRL;
            addr_trig_reg = MPUREG_I2C_SLV2_ADDR;
            reg_trig_reg  = MPUREG_I2C_SLV2_REG;
            ctrl_trig_reg = MPUREG_I2C_SLV2_CTRL;
            d0_trig_reg   = MPUREG_I2C_SLV2_DO;
            bits_slave_delay = BIT_SLV2_DLY_EN | BIT_SLV0_DLY_EN;
            break;
      case EXT_SLAVE_TYPE_PRESSURE:
            addr_reg = MPUREG_I2C_SLV3_ADDR;
            reg_reg  = MPUREG_I2C_SLV3_REG;
            ctrl_reg = MPUREG_I2C_SLV3_CTRL;
            addr_trig_reg = MPUREG_I2C_SLV4_ADDR;
            reg_trig_reg  = MPUREG_I2C_SLV4_REG;
            ctrl_trig_reg = MPUREG_I2C_SLV4_CTRL;
            bits_slave_delay = BIT_SLV4_DLY_EN | BIT_SLV3_DLY_EN;
            break;
      default:
            LOG_RESULT_LOCATION(INV_ERROR_INVALID_PARAMETER);
            return INV_ERROR_INVALID_PARAMETER;
      };
      /* return if this slave has already been set */
      if ((slave_address &&
           ((mldl_cfg->inv_mpu_state->i2c_slaves_enabled &
bits_slave_delay)
                == bits_slave_delay)) ||
          (!slave_address &&
           (mldl_cfg->inv_mpu_state->i2c_slaves_enabled &
bits_slave_delay) ==
                0))
            return 0;
      result = mpu_set_i2c_bypass(mldl_cfg, gyro_handle, true);
      /* Address */
      result = inv_serial_single_write(gyro_handle,
                               mldl_cfg->mpu_chip_info->addr,
                               addr_reg, slave_address);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      /* Register */
      result = inv_serial_single_write(gyro_handle,
                               mldl_cfg->mpu_chip_info->addr,
                               reg_reg, slave_reg);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      /* Length, byte swapping, grouping & enable */
      if (slave_len > BITS_SLV_LENG) {
            MPL_LOGW("Limiting slave burst read length to "
                   "the allowed maximum (15B, req. %d)\n",
slave_len);
            slave_len = BITS_SLV_LENG;
            return INV_ERROR_INVALID_CONFIGURATION;
      }
      reg = slave_len;
      if (slave_endian == EXT_SLAVE_LITTLE_ENDIAN) {
            reg |= BIT_SLV_BYTE_SW;
            if (slave_reg & 1)
                  reg |= BIT_SLV_GRP;
      }
      if (slave_address)
            reg |= BIT_SLV_ENABLE;
      result = inv_serial_single_write(gyro_handle,
                               mldl_cfg->mpu_chip_info->addr,
                               ctrl_reg, reg);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      /* Trigger */
      if (addr_trig_reg) {
            /* If slave address is 0 this clears the trigger */
            result = inv_serial_single_write(gyro_handle,
                                         mldl_cfg->mpu_chip_info-
>addr,
                                         addr_trig_reg,
                                         slave_address &
~BIT_I2C_READ);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
      }
      if (slave && slave->trigger && reg_trig_reg) {
            result = inv_serial_single_write(gyro_handle,
                                         mldl_cfg->mpu_chip_info-
>addr,
                                         reg_trig_reg,
                                         slave->trigger->reg);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
            result = inv_serial_single_write(gyro_handle,
                                         mldl_cfg->mpu_chip_info-
>addr,
                                         ctrl_trig_reg,
                                         BIT_SLV_ENABLE | 0x01);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
            result = inv_serial_single_write(gyro_handle,
                                         mldl_cfg->mpu_chip_info-
>addr,
                                         d0_trig_reg,
                                         slave->trigger->value);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
      } else if (ctrl_trig_reg) {
            result = inv_serial_single_write(gyro_handle,
                                         mldl_cfg->mpu_chip_info-
>addr,
                                         ctrl_trig_reg, 0x00);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
      }
      /* Data rate */
      if (slave) {
            struct ext_slave_config config;
            long data;
            config.key = MPU_SLAVE_CONFIG_ODR_RESUME;
            config.len = sizeof(long);
            config.apply = false;
            config.data = &data;
            if (!(slave->get_config))
                  return INV_ERROR_INVALID_CONFIGURATION;
            result = slave->get_config(NULL, slave, slave_pdata,
&config);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
            MPL_LOGI("Slave %d ODR: %ld Hz\n", slave_id, data /
1000);
            divider = ((1000 * inv_mpu_get_sampling_rate_hz(
                            mldl_cfg->mpu_gyro_cfg))
                  / data) - 1;
      } else {
            divider = 0;
      }
      result = inv_serial_read(gyro_handle,
                        mldl_cfg->mpu_chip_info->addr,
                        MPUREG_I2C_MST_DELAY_CTRL,
                        1, &delay_ctrl_orig);
      delay_ctrl = delay_ctrl_orig;
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      if (divider > 0 && divider <= MASK_I2C_MST_DLY) {
            result = inv_serial_read(gyro_handle,
                               mldl_cfg->mpu_chip_info->addr,
                               MPUREG_I2C_SLV4_CTRL, 1, &reg);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
            if ((reg & MASK_I2C_MST_DLY) &&
                  ((long)(reg & MASK_I2C_MST_DLY) !=
                        (divider & MASK_I2C_MST_DLY))) {
                  MPL_LOGW("Changing slave divider: %ld to %ld\n",
                         (long)(reg & MASK_I2C_MST_DLY),
                         (divider & MASK_I2C_MST_DLY));
            }
            reg |= (unsigned char)(divider & MASK_I2C_MST_DLY);
            result = inv_serial_single_write(gyro_handle,
                                     mldl_cfg->mpu_chip_info-
>addr,

                                     MPUREG_I2C_SLV4_CTRL,
                                     reg);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
            delay_ctrl |= bits_slave_delay;
      } else {
            delay_ctrl &= ~(bits_slave_delay);
      }
      if (delay_ctrl != delay_ctrl_orig) {
            result = inv_serial_single_write(
                  gyro_handle, mldl_cfg->mpu_chip_info->addr,
                  MPUREG_I2C_MST_DELAY_CTRL,
                  delay_ctrl);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
      }
      if (slave_address)
            mldl_cfg->inv_mpu_state->i2c_slaves_enabled |=
                  bits_slave_delay;
      else
            mldl_cfg->inv_mpu_state->i2c_slaves_enabled &=
                  ~bits_slave_delay;
      return result;
}
static int mpu_set_slave(struct mldl_cfg *mldl_cfg,
                   void *gyro_handle,
                   struct ext_slave_descr *slave,
                   struct ext_slave_platform_data *slave_pdata,
                   int slave_id)
{
      return mpu_set_slave_mpu60xx(mldl_cfg, gyro_handle, slave,
                             slave_pdata, slave_id);
}
/**
 * Check to see if the gyro was reset by testing a couple of
registers known
 * to change on reset.
 *
 * @mldl_cfg mldl configuration structure
 * @gyro_handle handle used to communicate with the gyro
 *
 * @return INV_SUCCESS or non-zero error code
 */
static int mpu_was_reset(struct mldl_cfg *mldl_cfg, void
*gyro_handle)
{
      int result = INV_SUCCESS;
      unsigned char reg;
      result = inv_serial_read(gyro_handle, mldl_cfg-
>mpu_chip_info->addr,
                         MPUREG_DMP_CFG_2, 1, &reg);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      if (mldl_cfg->mpu_gyro_cfg->dmp_cfg2 != reg)
            return true;
      if (0 != mldl_cfg->mpu_gyro_cfg->dmp_cfg1)
            return false;
      /* Inconclusive assume it was reset */
      return true;
}
int inv_mpu_set_firmware(struct mldl_cfg *mldl_cfg, void
*mlsl_handle,
                   const unsigned char *data, int size)
{
      int bank, offset, write_size;
      int result;
      unsigned char read[MPU_MEM_BANK_SIZE];
      if (mldl_cfg->inv_mpu_state->status &
MPU_DEVICE_IS_SUSPENDED) {
```

```
#if INV_CACHE_DMP == 1
                memcpy(mldl_cfg->mpu_ram->ram, data, size);
                return INV_SUCCESS;
#else
                LOG_RESULT_LOCATION(INV_ERROR_MEMORY_SET);
                return INV_ERROR_MEMORY_SET;
#endif
        }
        if (!(mldl_cfg->inv_mpu_state->status &
MPU_DMP_IS_SUSPENDED)) {
                LOG_RESULT_LOCATION(INV_ERROR_MEMORY_SET);
                return INV_ERROR_MEMORY_SET;
        }
        /* Write and verify memory */
        for (bank = 0; size > 0; bank++,
                        size -= write_size,
                        data += write_size) {
                if (size > MPU_MEM_BANK_SIZE)
                        write_size = MPU_MEM_BANK_SIZE;
                else
                        write_size = size;
                result = inv_serial_write_mem(mlsl_handle,
                                mldl_cfg->mpu_chip_info->addr,
                                ((bank << 8) | 0x00),
                                write_size,
                                data);
                if (result) {
                        LOG_RESULT_LOCATION(result);
                        MPL_LOGE("Write mem error in bank %d\n", bank);
                        return result;
                }
                result = inv_serial_read_mem(mlsl_handle,
                                mldl_cfg->mpu_chip_info->addr,
                                ((bank << 8) | 0x00),
                                write_size,
                                read);
                if (result) {
                        LOG_RESULT_LOCATION(result);
                        MPL_LOGE("Read mem error in bank %d\n", bank);
                        return result;
                }
#define ML_SKIP_CHECK 38
                for (offset = 0; offset < write_size; offset++) {
                        /* skip the register memory locations */
```

```
                  if (bank == 0 && offset < ML_SKIP_CHECK)
                        continue;
                  if (data[offset] != read[offset]) {
                        result = INV_ERROR_SERIAL_WRITE;
                        break;
                  }
            }
            if (result != INV_SUCCESS) {
                  LOG_RESULT_LOCATION(result);
                  MPL_LOGE("Read data mismatch at bank %d, offset
%d\n",
                        bank, offset);
                  return result;
            }
      }
      return INV_SUCCESS;
}
static int gyro_resume(struct mldl_cfg *mldl_cfg, void
*gyro_handle,
                   unsigned long sensors)
{
      int result;
      int ii;
      unsigned char reg;
      unsigned char regs[7];
      /* Wake up the part */
      result = mpu60xx_pwr_mgmt(mldl_cfg, gyro_handle, false,
sensors);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      /* Always set the INT_ENABLE and DIVIDER as the Accel Only
mode for 6050
         can set these too */
      result = inv_serial_single_write(
            gyro_handle, mldl_cfg->mpu_chip_info->addr,
            MPUREG_INT_ENABLE, (mldl_cfg->mpu_gyro_cfg-
>int_config));
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      result = inv_serial_single_write(
            gyro_handle, mldl_cfg->mpu_chip_info->addr,
            MPUREG_SMPLRT_DIV, mldl_cfg->mpu_gyro_cfg->divider);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      if (!(mldl_cfg->inv_mpu_state->status &
MPU_GYRO_NEEDS_CONFIG) &&
          !mpu_was_reset(mldl_cfg, gyro_handle)) {
            return INV_SUCCESS;
      }
      /* Configure the MPU */
      result = mpu_set_i2c_bypass(mldl_cfg, gyro_handle, 1);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      result = mpu_set_clock_source(gyro_handle, mldl_cfg);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      reg = MPUREG_GYRO_CONFIG_VALUE(0, 0, 0,
                               mldl_cfg->mpu_gyro_cfg-
>full_scale);
      result = inv_serial_single_write(
            gyro_handle, mldl_cfg->mpu_chip_info->addr,
            MPUREG_GYRO_CONFIG, reg);
      reg = MPUREG_CONFIG_VALUE(mldl_cfg->mpu_gyro_cfg->ext_sync,
                          mldl_cfg->mpu_gyro_cfg->lpf);
      result = inv_serial_single_write(
            gyro_handle, mldl_cfg->mpu_chip_info->addr,
            MPUREG_CONFIG, reg);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      result = inv_serial_single_write(
            gyro_handle, mldl_cfg->mpu_chip_info->addr,
            MPUREG_DMP_CFG_1, mldl_cfg->mpu_gyro_cfg->dmp_cfg1);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      result = inv_serial_single_write(
            gyro_handle, mldl_cfg->mpu_chip_info->addr,
            MPUREG_DMP_CFG_2, mldl_cfg->mpu_gyro_cfg->dmp_cfg2);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      /* Write and verify memory */
#if INV_CACHE_DMP != 0
      inv_mpu_set_firmware(mldl_cfg, gyro_handle,
            mldl_cfg->mpu_ram->ram, mldl_cfg->mpu_ram->length);
#endif
      result = inv_serial_single_write(
            gyro_handle, mldl_cfg->mpu_chip_info->addr,
            MPUREG_XG_OFFS_TC,
            ((mldl_cfg->mpu_offsets->tc[0] << 1) &
BITS_XG_OFFS_TC));
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      regs[0] = ((mldl_cfg->mpu_offsets->tc[1] << 1) &
BITS_YG_OFFS_TC);
      result = inv_serial_single_write(
            gyro_handle, mldl_cfg->mpu_chip_info->addr,
            MPUREG_YG_OFFS_TC, regs[0]);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      result = inv_serial_single_write(
            gyro_handle, mldl_cfg->mpu_chip_info->addr,
            MPUREG_ZG_OFFS_TC,
            ((mldl_cfg->mpu_offsets->tc[2] << 1) &
BITS_ZG_OFFS_TC));
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      regs[0] = MPUREG_X_OFFS_USRH;
      for (ii = 0; ii < ARRAY_SIZE(mldl_cfg->mpu_offsets->gyro);
ii++) {
            regs[1 + ii * 2] =
                  (unsigned char)(mldl_cfg->mpu_offsets->gyro[ii]
>> 8)
                  & 0xff;
            regs[1 + ii * 2 + 1] =
                  (unsigned char)(mldl_cfg->mpu_offsets->gyro[ii]
& 0xff);
      }
      result = inv_serial_write(gyro_handle, mldl_cfg-
>mpu_chip_info->addr,
                          7, regs);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      /* Configure slaves */
      result = inv_mpu_set_level_shifter_bit(mldl_cfg,
gyro_handle,
                                     mldl_cfg->pdata-
>level_shifter);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      mldl_cfg->inv_mpu_state->status &= ~MPU_GYRO_NEEDS_CONFIG;
      return result;
}
int gyro_config(void *mlsl_handle,
            struct mldl_cfg *mldl_cfg,
            struct ext_slave_config *data)
{
      struct mpu_gyro_cfg *mpu_gyro_cfg = mldl_cfg->mpu_gyro_cfg;
      struct mpu_chip_info *mpu_chip_info = mldl_cfg-
>mpu_chip_info;
      struct mpu_offsets *mpu_offsets = mldl_cfg->mpu_offsets;
      int ii;
      if (!data->data)
            return INV_ERROR_INVALID_PARAMETER;
      switch (data->key) {
      case MPU_SLAVE_INT_CONFIG:
            mpu_gyro_cfg->int_config = *((__u8 *)data->data);
            break;
      case MPU_SLAVE_EXT_SYNC:
            mpu_gyro_cfg->ext_sync = *((__u8 *)data->data);
            break;
      case MPU_SLAVE_FULL_SCALE:
            mpu_gyro_cfg->full_scale = *((__u8 *)data->data);
            break;
      case MPU_SLAVE_LPF:
            mpu_gyro_cfg->lpf = *((__u8 *)data->data);
            break;
      case MPU_SLAVE_CLK_SRC:
            mpu_gyro_cfg->clk_src = *((__u8 *)data->data);
            break;
      case MPU_SLAVE_DIVIDER:
            mpu_gyro_cfg->divider = *((__u8 *)data->data);
            break;
      case MPU_SLAVE_DMP_ENABLE:
            mpu_gyro_cfg->dmp_enable = *((__u8 *)data->data);
            break;
      case MPU_SLAVE_FIFO_ENABLE:
            mpu_gyro_cfg->fifo_enable = *((__u8 *)data->data);
            break;
      case MPU_SLAVE_DMP_CFG1:
            mpu_gyro_cfg->dmp_cfg1 = *((__u8 *)data->data);
            break;
      case MPU_SLAVE_DMP_CFG2:
            mpu_gyro_cfg->dmp_cfg2 = *((__u8 *)data->data);
            break;
      case MPU_SLAVE_TC:
            for (ii = 0; ii < GYRO_NUM_AXES; ii++)
                  mpu_offsets->tc[ii] = ((__u8 *)data->data)[ii];
            break;
      case MPU_SLAVE_GYRO:
            for (ii = 0; ii < GYRO_NUM_AXES; ii++)
                  mpu_offsets->gyro[ii] = ((__u16 *)data-
>data)[ii];
            break;
      case MPU_SLAVE_ADDR:
            mpu_chip_info->addr = *((__u8 *)data->data);
            break;
      case MPU_SLAVE_PRODUCT_REVISION:
            mpu_chip_info->product_revision = *((__u8 *)data-
>data);
            break;
      case MPU_SLAVE_SILICON_REVISION:
            mpu_chip_info->silicon_revision = *((__u8 *)data-
>data);
            break;
```

```
    case MPU_SLAVE_PRODUCT_ID:
        mpu_chip_info->product_id = *((__u8 *)data->data);
        break;
    case MPU_SLAVE_GYRO_SENS_TRIM:
        mpu_chip_info->gyro_sens_trim = *((__u16 *)data-
>data);
        break;
    case MPU_SLAVE_ACCEL_SENS_TRIM:
        mpu_chip_info->accel_sens_trim = *((__u16 *)data-
>data);
        break;
    case MPU_SLAVE_RAM:
        if (data->len != mldl_cfg->mpu_ram->length)
            return INV_ERROR_INVALID_PARAMETER;
        memcpy(mldl_cfg->mpu_ram->ram, data->data, data->len);
        break;
    default:

    LOG_RESULT_LOCATION(INV_ERROR_FEATURE_NOT_IMPLEMENTED);
        return INV_ERROR_FEATURE_NOT_IMPLEMENTED;
    };
    mldl_cfg->inv_mpu_state->status |= MPU_GYRO_NEEDS_CONFIG;
    return INV_SUCCESS;
}
int gyro_get_config(void *mlsl_handle,
        struct mldl_cfg *mldl_cfg,
        struct ext_slave_config *data)
{
    struct mpu_gyro_cfg *mpu_gyro_cfg = mldl_cfg->mpu_gyro_cfg;
    struct mpu_chip_info *mpu_chip_info = mldl_cfg-
>mpu_chip_info;
    struct mpu_offsets *mpu_offsets = mldl_cfg->mpu_offsets;
    int ii;
    if (!data->data)
        return INV_ERROR_INVALID_PARAMETER;
    switch (data->key) {
    case MPU_SLAVE_INT_CONFIG:
        *((__u8 *)data->data) = mpu_gyro_cfg->int_config;
        break;
    case MPU_SLAVE_EXT_SYNC:
        *((__u8 *)data->data) = mpu_gyro_cfg->ext_sync;
        break;
    case MPU_SLAVE_FULL_SCALE:
        *((__u8 *)data->data) = mpu_gyro_cfg->full_scale;
```

```
            break;
      case MPU_SLAVE_LPF:
            *((__u8 *)data->data) = mpu_gyro_cfg->lpf;
            break;
      case MPU_SLAVE_CLK_SRC:
            *((__u8 *)data->data) = mpu_gyro_cfg->clk_src;
            break;
      case MPU_SLAVE_DIVIDER:
            *((__u8 *)data->data) = mpu_gyro_cfg->divider;
            break;
      case MPU_SLAVE_DMP_ENABLE:
            *((__u8 *)data->data) = mpu_gyro_cfg->dmp_enable;
            break;
      case MPU_SLAVE_FIFO_ENABLE:
            *((__u8 *)data->data) = mpu_gyro_cfg->fifo_enable;
            break;
      case MPU_SLAVE_DMP_CFG1:
            *((__u8 *)data->data) = mpu_gyro_cfg->dmp_cfg1;
            break;
      case MPU_SLAVE_DMP_CFG2:
            *((__u8 *)data->data) = mpu_gyro_cfg->dmp_cfg2;
            break;
      case MPU_SLAVE_TC:
            for (ii = 0; ii < GYRO_NUM_AXES; ii++)
                  ((__u8 *)data->data)[ii] = mpu_offsets->tc[ii];
            break;
      case MPU_SLAVE_GYRO:
            for (ii = 0; ii < GYRO_NUM_AXES; ii++)
                  ((__u16 *)data->data)[ii] = mpu_offsets-
>gyro[ii];
            break;
      case MPU_SLAVE_ADDR:
            *((__u8 *)data->data) = mpu_chip_info->addr;
            break;
      case MPU_SLAVE_PRODUCT_REVISION:
            *((__u8 *)data->data) = mpu_chip_info-
>product_revision;
            break;
      case MPU_SLAVE_SILICON_REVISION:
            *((__u8 *)data->data) = mpu_chip_info-
>silicon_revision;
            break;
      case MPU_SLAVE_PRODUCT_ID:
            *((__u8 *)data->data) = mpu_chip_info->product_id;
            break;
      case MPU_SLAVE_GYRO_SENS_TRIM:
            *((__u16 *)data->data) = mpu_chip_info-
>gyro_sens_trim;
            break;
      case MPU_SLAVE_ACCEL_SENS_TRIM:
            *((__u16 *)data->data) = mpu_chip_info-
>accel_sens_trim;
            break;
      case MPU_SLAVE_RAM:
            if (data->len != mldl_cfg->mpu_ram->length)
                  return INV_ERROR_INVALID_PARAMETER;
            memcpy(data->data, mldl_cfg->mpu_ram->ram, data->len);
            break;
      default:

      LOG_RESULT_LOCATION(INV_ERROR_FEATURE_NOT_IMPLEMENTED);
            return INV_ERROR_FEATURE_NOT_IMPLEMENTED;
      };
      return INV_SUCCESS;
}
/*****************************************************************
**************

******************************************************************
*************
 * Exported functions

******************************************************************
*************

******************************************************************
************/
/**
 * Initializes the pdata structure to defaults.
 *
 * Opens the device to read silicon revision, product id and
whoami.
 *
 * @mldl_cfg
 *          The internal device configuration data structure.
 * @mlsl_handle
 *          The serial communication handle.
 *
 * @return INV_SUCCESS if silicon revision, product id and woami
are supported
 *         by this software.
 */
int inv_mpu_open(struct mldl_cfg *mldl_cfg,
             void *gyro_handle,
             void *accel_handle,
             void *compass_handle, void *pressure_handle)
{
      int result;
      void *slave_handle[EXT_SLAVE_NUM_TYPES];
      int ii;
      /* Default is Logic HIGH, pushpull, latch disabled, anyread
to clear */
      ii = 0;
      mldl_cfg->inv_mpu_cfg->ignore_system_suspend = false;
      mldl_cfg->mpu_gyro_cfg->int_config = BIT_DMP_INT_EN;
      mldl_cfg->mpu_gyro_cfg->clk_src = MPU_CLK_SEL_PLLGYROZ;
      mldl_cfg->mpu_gyro_cfg->lpf = MPU_FILTER_42HZ;
      mldl_cfg->mpu_gyro_cfg->full_scale = MPU_FS_2000DPS;
      mldl_cfg->mpu_gyro_cfg->divider = 4;
      mldl_cfg->mpu_gyro_cfg->dmp_enable = 1;
      mldl_cfg->mpu_gyro_cfg->fifo_enable = 1;
      mldl_cfg->mpu_gyro_cfg->ext_sync = 0;
      mldl_cfg->mpu_gyro_cfg->dmp_cfg1 = 0;
      mldl_cfg->mpu_gyro_cfg->dmp_cfg2 = 0;
      mldl_cfg->inv_mpu_state->status =
            MPU_DMP_IS_SUSPENDED |
            MPU_GYRO_IS_SUSPENDED |
            MPU_ACCEL_IS_SUSPENDED |
            MPU_COMPASS_IS_SUSPENDED |
            MPU_PRESSURE_IS_SUSPENDED |
            MPU_DEVICE_IS_SUSPENDED;
      mldl_cfg->inv_mpu_state->i2c_slaves_enabled = 0;
      slave_handle[EXT_SLAVE_TYPE_GYROSCOPE] = gyro_handle;
      slave_handle[EXT_SLAVE_TYPE_ACCEL] = accel_handle;
      slave_handle[EXT_SLAVE_TYPE_COMPASS] = compass_handle;
      slave_handle[EXT_SLAVE_TYPE_PRESSURE] = pressure_handle;
      if (mldl_cfg->mpu_chip_info->addr == 0) {
            LOG_RESULT_LOCATION(INV_ERROR_INVALID_PARAMETER);
            return INV_ERROR_INVALID_PARAMETER;
      }
      /*
       * Reset,
       * Take the DMP out of sleep, and
       * read the product_id, sillicon rev and whoami
       */
      mldl_cfg->inv_mpu_state->status &= ~MPU_GYRO_IS_BYPASSED;
      result = mpu60xx_pwr_mgmt(mldl_cfg, gyro_handle, true,
                                INV_THREE_AXIS_GYRO);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      result = inv_get_silicon_rev(mldl_cfg, gyro_handle);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      /* Get the factory temperature compensation offsets */
      result = inv_serial_read(gyro_handle, mldl_cfg-
>mpu_chip_info->addr,
                               MPUREG_XG_OFFS_TC, 1,
                               &mldl_cfg->mpu_offsets->tc[0]);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      result = inv_serial_read(gyro_handle, mldl_cfg-
>mpu_chip_info->addr,
                               MPUREG_YG_OFFS_TC, 1,
                               &mldl_cfg->mpu_offsets->tc[1]);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      result = inv_serial_read(gyro_handle, mldl_cfg-
>mpu_chip_info->addr,
                               MPUREG_ZG_OFFS_TC, 1,
                               &mldl_cfg->mpu_offsets->tc[2]);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      /* Into bypass mode before sleeping and calling the slaves
init */
      result = mpu_set_i2c_bypass(mldl_cfg, gyro_handle, true);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      result = inv_mpu_set_level_shifter_bit(mldl_cfg,
gyro_handle,
                  mldl_cfg->pdata->level_shifter);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      for (ii = 0; ii < GYRO_NUM_AXES; ii++) {
            mldl_cfg->mpu_offsets->tc[ii] =
                (mldl_cfg->mpu_offsets->tc[ii] & BITS_XG_OFFS_TC)
>> 1;
      }
#if INV_CACHE_DMP != 0
      result = mpu60xx_pwr_mgmt(mldl_cfg, gyro_handle, false, 0);
#endif
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      return result;
}
/**
 * Close the mpu interface
 *
 * @mldl_cfg pointer to the configuration structure
 * @mlsl_handle pointer to the serial layer handle
 *
 * @return INV_SUCCESS or non-zero error code
 */
int inv_mpu_close(struct mldl_cfg *mldl_cfg,
              void *gyro_handle,
              void *accel_handle,
              void *compass_handle,
              void *pressure_handle)
{
      return 0;
}
/**
 *  @brief  resume the MPU device and all the other sensor
 *          devices from their low power state.
 *
 *  @mldl_cfg
 *              pointer to the configuration structure
 *  @gyro_handle
 *              the main file handle to the MPU device.
 *  @accel_handle
 *              an handle to the accelerometer device, if sitting
 *              onto a separate bus. Can match mlsl_handle if
 *              the accelerometer device operates on the same
 *              primary bus of MPU.
 *  @compass_handle
 *              an handle to the compass device, if sitting
 *              onto a separate bus. Can match mlsl_handle if
 *              the compass device operates on the same
 *              primary bus of MPU.
 *  @pressure_handle
 *              an handle to the pressure sensor device, if
sitting
 *              onto a separate bus. Can match mlsl_handle if
 *              the pressure sensor device operates on the same
 *              primary bus of MPU.
 *  @resume_gyro
 *              whether resuming the gyroscope device is
 *              actually needed (if the device supports low power
 *              mode of some sort).
 *  @resume_accel
 *              whether resuming the accelerometer device is
 *              actually needed (if the device supports low power
 *              mode of some sort).
 *  @resume_compass
 *              whether resuming the compass device is
 *              actually needed (if the device supports low power
 *              mode of some sort).
 *  @resume_pressure
 *              whether resuming the pressure sensor device is
 *              actually needed (if the device supports low power
 *              mode of some sort).
 *  @return  INV_SUCCESS or a non-zero error code.
 */
int inv_mpu_resume(struct mldl_cfg *mldl_cfg,
               void *gyro_handle,
               void *accel_handle,
               void *compass_handle,
               void *pressure_handle,
               unsigned long sensors)
```

```
{
      int result = INV_SUCCESS;
      int ii;
      bool resume_slave[EXT_SLAVE_NUM_TYPES];
      bool resume_dmp = sensors & INV_DMP_PROCESSOR;
      void *slave_handle[EXT_SLAVE_NUM_TYPES];
      resume_slave[EXT_SLAVE_TYPE_GYROSCOPE] =
            (sensors & (INV_X_GYRO | INV_Y_GYRO | INV_Z_GYRO));
      resume_slave[EXT_SLAVE_TYPE_ACCEL] =
            sensors & INV_THREE_AXIS_ACCEL;
      resume_slave[EXT_SLAVE_TYPE_COMPASS] =
            sensors & INV_THREE_AXIS_COMPASS;
      resume_slave[EXT_SLAVE_TYPE_PRESSURE] =
            sensors & INV_THREE_AXIS_PRESSURE;
      slave_handle[EXT_SLAVE_TYPE_GYROSCOPE] = gyro_handle;
      slave_handle[EXT_SLAVE_TYPE_ACCEL] = accel_handle;
      slave_handle[EXT_SLAVE_TYPE_COMPASS] = compass_handle;
      slave_handle[EXT_SLAVE_TYPE_PRESSURE] = pressure_handle;
      mldl_print_cfg(mldl_cfg);
      /* Skip the Gyro since slave[EXT_SLAVE_TYPE_GYROSCOPE] is
NULL */
      for (ii = EXT_SLAVE_TYPE_ACCEL; ii < EXT_SLAVE_NUM_TYPES;
ii++) {
            if (resume_slave[ii] &&
                ((!mldl_cfg->slave[ii]) ||
                  (!mldl_cfg->slave[ii]->resume))) {

      LOG_RESULT_LOCATION(INV_ERROR_INVALID_PARAMETER);
                  return INV_ERROR_INVALID_PARAMETER;
            }
      }
      if ((resume_slave[EXT_SLAVE_TYPE_GYROSCOPE] || resume_dmp)
          && ((mldl_cfg->inv_mpu_state->status &
MPU_GYRO_IS_SUSPENDED) ||
            (mldl_cfg->inv_mpu_state->status &
MPU_GYRO_NEEDS_CONFIG))) {
            result = mpu_set_i2c_bypass(mldl_cfg, gyro_handle, 1);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
            result = dmp_stop(mldl_cfg, gyro_handle);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
            result = gyro_resume(mldl_cfg, gyro_handle, sensors);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
      }
      for (ii = 0; ii < EXT_SLAVE_NUM_TYPES; ii++) {
            if (!mldl_cfg->slave[ii] ||
                !mldl_cfg->pdata_slave[ii] ||
                !resume_slave[ii] ||
                !(mldl_cfg->inv_mpu_state->status & (1 << ii)))
                  continue;
            if (EXT_SLAVE_BUS_SECONDARY ==
                mldl_cfg->pdata_slave[ii]->bus) {
                  result = mpu_set_i2c_bypass(mldl_cfg,
gyro_handle,
                                        true);
                  if (result) {
                        LOG_RESULT_LOCATION(result);
                        return result;
                  }
            }
            result = mldl_cfg->slave[ii]->resume(slave_handle[ii],
                                    mldl_cfg->slave[ii],
                                    mldl_cfg->pdata_slave[ii]);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
            mldl_cfg->inv_mpu_state->status &= ~(1 << ii);
      }
      for (ii = 0; ii < EXT_SLAVE_NUM_TYPES; ii++) {
            if (resume_dmp &&
                !(mldl_cfg->inv_mpu_state->status & (1 << ii)) &&
                mldl_cfg->pdata_slave[ii] &&
                EXT_SLAVE_BUS_SECONDARY == mldl_cfg-
>pdata_slave[ii]->bus) {
                  result = mpu_set_slave(mldl_cfg,
                              gyro_handle,
                              mldl_cfg->slave[ii],
                              mldl_cfg->pdata_slave[ii],
                              mldl_cfg->slave[ii]->type);
                  if (result) {
                        LOG_RESULT_LOCATION(result);
                        return result;
                  }
            }
      }
      /* Turn on the master i2c iterface if necessary */
      if (resume_dmp) {
            result = mpu_set_i2c_bypass(
                  mldl_cfg, gyro_handle,
                  !(mldl_cfg->inv_mpu_state->i2c_slaves_enabled));
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
            /* Now start */
            result = dmp_start(mldl_cfg, gyro_handle);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
      }
      mldl_cfg->inv_mpu_cfg->requested_sensors = sensors;
      return result;
}
/**
 *  @brief  suspend the MPU device and all the other sensor
 *          devices into their low power state.
 *  @mldl_cfg
 *              a pointer to the struct mldl_cfg internal data
 *              structure.
 *  @gyro_handle
 *              the main file handle to the MPU device.
 *  @accel_handle
 *              an handle to the accelerometer device, if sitting
 *              onto a separate bus. Can match gyro_handle if
 *              the accelerometer device operates on the same
 *              primary bus of MPU.
 *  @compass_handle
 *              an handle to the compass device, if sitting
 *              onto a separate bus. Can match gyro_handle if
 *              the compass device operates on the same
 *              primary bus of MPU.
 *  @pressure_handle
 *              an handle to the pressure sensor device, if
sitting
 *              onto a separate bus. Can match gyro_handle if
 *              the pressure sensor device operates on the same
 *              primary bus of MPU.
 *  @accel
 *              whether suspending the accelerometer device is
 *              actually needed (if the device supports low power
 *              mode of some sort).
 *  @compass
 *              whether suspending the compass device is
 *              actually needed (if the device supports low power
 *              mode of some sort).
 *  @pressure
 *              whether suspending the pressure sensor device is
 *              actually needed (if the device supports low power
 *              mode of some sort).
 *  @return  INV_SUCCESS or a non-zero error code.
 */
int inv_mpu_suspend(struct mldl_cfg *mldl_cfg,
                void *gyro_handle,
                void *accel_handle,
                void *compass_handle,
                void *pressure_handle,
                unsigned long sensors)
{
      int result = INV_SUCCESS;
      int ii;
      struct ext_slave_descr **slave = mldl_cfg->slave;
      struct ext_slave_platform_data **pdata_slave = mldl_cfg-
>pdata_slave;
      bool suspend_dmp = ((sensors & INV_DMP_PROCESSOR) ==
INV_DMP_PROCESSOR);
      bool suspend_slave[EXT_SLAVE_NUM_TYPES];
      void *slave_handle[EXT_SLAVE_NUM_TYPES];
      suspend_slave[EXT_SLAVE_TYPE_GYROSCOPE] =
            ((sensors & (INV_X_GYRO | INV_Y_GYRO | INV_Z_GYRO))
                  == (INV_X_GYRO | INV_Y_GYRO | INV_Z_GYRO));
      suspend_slave[EXT_SLAVE_TYPE_ACCEL] =
            ((sensors & INV_THREE_AXIS_ACCEL) ==
INV_THREE_AXIS_ACCEL);
      suspend_slave[EXT_SLAVE_TYPE_COMPASS] =
            ((sensors & INV_THREE_AXIS_COMPASS) ==
INV_THREE_AXIS_COMPASS);
```

```
      suspend_slave[EXT_SLAVE_TYPE_PRESSURE] =
            ((sensors & INV_THREE_AXIS_PRESSURE) ==
                  INV_THREE_AXIS_PRESSURE);
      slave_handle[EXT_SLAVE_TYPE_GYROSCOPE] = gyro_handle;
      slave_handle[EXT_SLAVE_TYPE_ACCEL] = accel_handle;
      slave_handle[EXT_SLAVE_TYPE_COMPASS] = compass_handle;
      slave_handle[EXT_SLAVE_TYPE_PRESSURE] = pressure_handle;
      if (suspend_dmp) {
            result = mpu_set_i2c_bypass(mldl_cfg, gyro_handle, 1);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
            result = dmp_stop(mldl_cfg, gyro_handle);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
      }
      /* Gyro */
      if (suspend_slave[EXT_SLAVE_TYPE_GYROSCOPE] &&
          !(mldl_cfg->inv_mpu_state->status &
MPU_GYRO_IS_SUSPENDED)) {
            result = mpu60xx_pwr_mgmt(mldl_cfg, gyro_handle,
false,
                              ((~sensors) & INV_ALL_SENSORS));
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
      }
      for (ii = 0; ii < EXT_SLAVE_NUM_TYPES; ii++) {
            bool is_suspended = mldl_cfg->inv_mpu_state->status &
(1 << ii);
            if (!slave[ii]   || !pdata_slave[ii] ||
                is_suspended || !suspend_slave[ii])
                  continue;
            if (EXT_SLAVE_BUS_SECONDARY == pdata_slave[ii]->bus) {
                  result = mpu_set_i2c_bypass(mldl_cfg,
gyro_handle, 1);
                  if (result) {
                        LOG_RESULT_LOCATION(result);
                        return result;
                  }
            }
            result = slave[ii]->suspend(slave_handle[ii],
                                      slave[ii],
                                      pdata_slave[ii]);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
            if (EXT_SLAVE_BUS_SECONDARY == pdata_slave[ii]->bus) {
                  result = mpu_set_slave(mldl_cfg, gyro_handle,
                                     NULL, NULL,
                                     slave[ii]->type);
                  if (result) {
                        LOG_RESULT_LOCATION(result);
                        return result;
                  }
            }
            mldl_cfg->inv_mpu_state->status |= (1 << ii);
      }
      /* Re-enable the i2c master if there are configured slaves and DMP */
      if (!suspend_dmp) {
            result = mpu_set_i2c_bypass(
                  mldl_cfg, gyro_handle,
                  !(mldl_cfg->inv_mpu_state->i2c_slaves_enabled));
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
      }
      mldl_cfg->inv_mpu_cfg->requested_sensors = (~sensors) & INV_ALL_SENSORS;
      return result;
}
int inv_mpu_slave_read(struct mldl_cfg *mldl_cfg,
                   void *gyro_handle,
                   void *slave_handle,
                   struct ext_slave_descr *slave,
                   struct ext_slave_platform_data *pdata,
                   unsigned char *data)
{
      int result;
      int bypass_result;
      int remain_bypassed = true;
      if (NULL == slave || NULL == slave->read) {
            LOG_RESULT_LOCATION(INV_ERROR_INVALID_CONFIGURATION);
            return INV_ERROR_INVALID_CONFIGURATION;
      }
      if ((EXT_SLAVE_BUS_SECONDARY == pdata->bus)
          && (!(mldl_cfg->inv_mpu_state->status &
MPU_GYRO_IS_BYPASSED))) {
            remain_bypassed = false;
            result = mpu_set_i2c_bypass(mldl_cfg, gyro_handle, 1);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
      }
      result = slave->read(slave_handle, slave, pdata, data);
      if (!remain_bypassed) {
            bypass_result = mpu_set_i2c_bypass(mldl_cfg,
gyro_handle, 0);
            if (bypass_result) {
                  LOG_RESULT_LOCATION(bypass_result);
                  return bypass_result;
            }
      }
      return result;
}
int inv_mpu_slave_config(struct mldl_cfg *mldl_cfg,
                   void *gyro_handle,
                   void *slave_handle,
                   struct ext_slave_config *data,
                   struct ext_slave_descr *slave,
                   struct ext_slave_platform_data *pdata)
{
      int result;
      int remain_bypassed = true;
      if (NULL == slave || NULL == slave->config) {
            LOG_RESULT_LOCATION(INV_ERROR_INVALID_CONFIGURATION);
            return INV_ERROR_INVALID_CONFIGURATION;
      }
      if (data->apply && (EXT_SLAVE_BUS_SECONDARY == pdata->bus)
          && (!(mldl_cfg->inv_mpu_state->status &
MPU_GYRO_IS_BYPASSED))) {
            remain_bypassed = false;
            result = mpu_set_i2c_bypass(mldl_cfg, gyro_handle, 1);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
      }
      result = slave->config(slave_handle, slave, pdata, data);
      if (result) {
            LOG_RESULT_LOCATION(result);
            return result;
      }
      if (!remain_bypassed) {
            result = mpu_set_i2c_bypass(mldl_cfg, gyro_handle, 0);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
      }
      return result;
}
int inv_mpu_get_slave_config(struct mldl_cfg *mldl_cfg,
                       void *gyro_handle,
                       void *slave_handle,
                       struct ext_slave_config *data,
                       struct ext_slave_descr *slave,
                       struct ext_slave_platform_data *pdata)
{
      int result;
      int remain_bypassed = true;
      if (NULL == slave || NULL == slave->get_config) {
            LOG_RESULT_LOCATION(INV_ERROR_INVALID_CONFIGURATION);
            return INV_ERROR_INVALID_CONFIGURATION;
      }
      if (data->apply && (EXT_SLAVE_BUS_SECONDARY == pdata->bus)
          && (!(mldl_cfg->inv_mpu_state->status &
MPU_GYRO_IS_BYPASSED))) {
            remain_bypassed = false;
            result = mpu_set_i2c_bypass(mldl_cfg, gyro_handle, 1);
            if (result) {
                  LOG_RESULT_LOCATION(result);
                  return result;
            }
      }
      result = slave->get_config(slave_handle, slave, pdata,
data);
      if (result) {
```

```
                LOG_RESULT_LOCATION(result);
                return result;
        }
        if (!remain_bypassed) {
                result = mpu_set_i2c_bypass(mldl_cfg, gyro_handle, 0);
                if (result) {
                        LOG_RESULT_LOCATION(result);
                        return result;
                }
        }
        return result;
}
```