Arduino Gyroscope Driver

Generated by Doxygen 1.8.11

Contents

1 Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

2 Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

3 File Index

3.1 File List

Here is a list of all files with brief descriptions:

4 Class Documentation

4.1 GyroscopeMPU9250::GYRO_CONFIGbits Union Reference

`#include <GyroscopeMPU9250.h>`

Public Attributes

- struct {
  unsigned char FCHOICE_B:2
  unsigned char:1
  unsigned char GYRO_FS_SEL:2
  unsigned char ZGYRO_CTEN:1
  unsigned char YGYRO_CTEN:1
  unsigned char XGYRO_Cten:1
  };
- unsigned char value

4.1.1 Detailed Description

Gyroscope Configuration (GYRO_CONFIG 0x1b) Serial IF: R/W Reset value: 0x00.

Definition at line 99 of file GyroscopeMPU9250.h.

4.1.2 Member Data Documentation

4.1.2.1 struct { ... }

4.1.2.2 unsigned GyroscopeMPU9250::GYRO_CONFIGbits::char

Definition at line 102 of file GyroscopeMPU9250.h.

4.1.2.3 unsigned char GyroscopeMPU9250::GYRO_CONFIGbits::FCHOICE_B

Definition at line 101 of file GyroscopeMPU9250.h.

4.1.2.4 unsigned char GyroscopeMPU9250::GYRO_CONFIGbits::GYRO_FS_SEL

Definition at line 103 of file GyroscopeMPU9250.h.

4.1.2.5 unsigned char GyroscopeMPU9250::GYRO_CONFIGbits::value

Definition at line 108 of file GyroscopeMPU9250.h.

4.1.2.6 unsigned char GyroscopeMPU9250::GYRO_CONFIGbits::XGYRO_Cten

Definition at line 106 of file GyroscopeMPU9250.h.

4.1.2.7 unsigned char GyroscopeMPU9250::GYRO_CONFIGbits::YGYRO_CTEN

Definition at line 105 of file GyroscopeMPU9250.h.

4.1.2.8 unsigned char GyroscopeMPU9250::GYRO_CONFIGbits::ZGYRO_CTEN

Definition at line 104 of file GyroscopeMPU9250.h.

The documentation for this union was generated from the following file:

- GyroscopeMPU9250.h

4.2 Gyroscope Class Reference

`#include <Gyroscope.h>`

Inheritance diagram for Gyroscope:

Public Member Functions

- virtual ~Gyroscope ()
- virtual float getRotationX ()=0
- virtual float getRotationY ()=0
- virtual float getRotationZ ()=0

4.2.1 Detailed Description

Arduino - Gyroscope Driver.

Abstract interface which all gyroscope should implement.

Author

Dalmir da Silva dalmirdasilva@gmail.com

Definition at line 12 of file Gyroscope.h.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 Gyroscope::~Gyroscope ( ) [virtual]

Definition at line 3 of file Gyroscope.cpp.

4.2.3 Member Function Documentation

4.2.3.1 virtual float Gyroscope::getRotationX ( ) [pure virtual]

Implemented in GyroscopeMPU9250.

4.2.3.2 virtual float Gyroscope::getRotationY ( ) [pure virtual]

Implemented in GyroscopeMPU9250.

4.2.3.3 virtual float Gyroscope::getRotationZ ( ) [pure virtual]

Implemented in GyroscopeMPU9250.

The documentation for this class was generated from the following files:

- Gyroscope.h
- Gyroscope.cpp

4.3 GyroscopeMPU9250 Class Reference

`#include <GyroscopeMPU9250.h>`

Inheritance diagram for GyroscopeMPU9250:

Collaboration diagram for GyroscopeMPU9250:

Classes

- union GYRO_CONFIGbits
- union PWR_MGMT_1bits
- union PWR_MGMT_2bits

Public Types

- enum Register {
  PWR_MGMT_1 = 0x6b, PWR_MGMT_2 = 0x6c, GYRO_CONFIG = 0x1b, GYRO_XOUT_H = 0x43,
  GYRO_XOUT_L = 0x44, GYRO_YOUT_H = 0x45, GYRO_YOUT_L = 0x46, GYRO_ZOUT_H = 0x47,
  GYRO_ZOUT_L = 0x48 }
- enum FullScaleRange { FS_SEL_250DPS = 0x00, FS_SEL_500DPS = 0x08, FS_SEL_1000DPS = 0x10,
  FS_SEL_2000DPS = 0x18 }
- enum ClockSelection { INTERNAL_20MHZ_OSCILLATOR = 0x00, BEST_AVAILABLE_SOURCE = 0x01,
  STOPS_CLOCK_KEEPS_TIMING = 0x07 }
- enum Axis {
  AXIS_NONE = 0x00, AXIS_X = 0x04, AXIS_Y = 0x02, AXIS_Z = 0x01,
  AXIS_XY = AXIS_X | AXIS_Y, AXIS_XZ = AXIS_X | AXIS_Z, AXIS_YZ = AXIS_Y | AXIS_Z, AXIS_XYZ =
  AXIS_X | AXIS_Y | AXIS_Z }
- enum Mask {
  GYRO_CONFIG_GYRO_FS_SEL = 0x18, PWR_MGMT_2_DISABLE_G = 0x07, PWR_MGMT_1_H_RE↩
  SET = 0x80, PWR_MGMT_1_SLEEP = 0x40,
  PWR_MGMT_1_CYCLE = 0x20, PWR_MGMT_1_GYRO_STANDBY = 0x10, PWR_MGMT_1_CLKSEL =
  0x07 }

Public Member Functions

- GyroscopeMPU9250 (bool ad0)
- float getRotationX ()
- float getRotationY ()
- float getRotationZ ()
- unsigned char readXYZ (unsigned char *buf)
- float readAxisRotation (unsigned char axisRegister)
- void setFullScaleRange (FullScaleRange fsr)
- void selectClock (ClockSelection cs)
- void reset ()
- void sleep ()
- void awake ()
- void enableAxis (Axis axis)
- float convertToDegreePerSeconds (unsigned char buf[2])

Private Attributes

- GYRO_CONFIGbits config

4.3.1 Detailed Description

The MPU-9250 consists of three independent vibratory MEMS rate gyroscopes, which detect rotation about the X-, Y-, and Z- Axes.

When the gyros are rotated about any of the sense axes, the Coriolis Effect causes a vibration that is detected by a capacitive pickoff. The resulting signal is amplified, demodulated, and filtered to produce a voltage that is proportional to t he angular rat e. This voltage is digitized using individual on-chip 16-bit Analog-to-Digital Converters (ADCs) to sample each axis. The full-scale range of the gyro sensors may be digitally programmed to ±250, ±500, ±1000, or ±2000 degrees per second (dps). The ADC sample rate is programmable from 8,000 s amples per second, down to 3.9 samples per second, and us er-s electable low-pass filters enable a wide range of cut-off frequencies.

Definition at line 27 of file GyroscopeMPU9250.h.

4.3.2 Member Enumeration Documentation

4.3.2.1 enum GyroscopeMPU9250::Axis

Axis.

Enumerator

AXIS_NONE

AXIS_X

AXIS_Y

AXIS_Z

AXIS_XY

AXIS_XZ

AXIS_YZ

AXIS_XYZ

Definition at line 70 of file GyroscopeMPU9250.h.

4.3.2.2 enum GyroscopeMPU9250::ClockSelection

Enumerator

INTERNAL_20MHZ_OSCILLATOR

BEST_AVAILABLE_SOURCE

STOPS_CLOCK_KEEPS_TIMING

Definition at line 61 of file GyroscopeMPU9250.h.

4.3.2.3 enum GyroscopeMPU9250::FullScaleRange

Gyroscope Full Scale Select.

```
FS1 FS0  g Range
0    0   +250dps
0    1   +500dps
1    0   +1000dps
1    1   +2000dps
```

Enumerator

FS_SEL_250DPS
FS_SEL_500DPS
FS_SEL_1000DPS
FS_SEL_2000DPS

Definition at line 54 of file GyroscopeMPU9250.h.

4.3.2.4 enum GyroscopeMPU9250::Mask

Some useful masks.

Enumerator

GYRO_CONFIG_GYRO_FS_SEL
PWR_MGMT_2_DISABLE_G
PWR_MGMT_1_H_RESET
PWR_MGMT_1_SLEEP
PWR_MGMT_1_CYCLE
PWR_MGMT_1_GYRO_STANDBY
PWR_MGMT_1_CLKSEL

Definition at line 84 of file GyroscopeMPU9250.h.

4.3.2.5 enum GyroscopeMPU9250::Register

Enumerator

PWR_MGMT_1
PWR_MGMT_2
GYRO_CONFIG
GYRO_XOUT_H
GYRO_XOUT_L
GYRO_YOUT_H
GYRO_YOUT_L
GYRO_ZOUT_H
GYRO_ZOUT_L

Definition at line 31 of file GyroscopeMPU9250.h.

4.3.3 Constructor & Destructor Documentation

4.3.3.1 GyroscopeMPU9250::GyroscopeMPU9250 (bool *ad0*)

Public constructor.

Parameters

ad0	LSBit of the device address.

Definition at line 3 of file GyroscopeMPU9250.cpp.

4.3.4 Member Function Documentation

4.3.4.1 void GyroscopeMPU9250::awake ( )

Definition at line 49 of file GyroscopeMPU9250.cpp.

4.3.4.2 float GyroscopeMPU9250::convertToDegreePerSeconds (unsigned char *buf[2]*)

Definition at line 57 of file GyroscopeMPU9250.cpp.

4.3.4.3 void GyroscopeMPU9250::enableAxis (Axis *axis*)

Definition at line 53 of file GyroscopeMPU9250.cpp.

4.3.4.4 float GyroscopeMPU9250::getRotationX ( ) [virtual]

Implements Gyroscope.

Definition at line 8 of file GyroscopeMPU9250.cpp.

4.3.4.5 float GyroscopeMPU9250::getRotationY ( ) [virtual]

Implements Gyroscope.

Definition at line 12 of file GyroscopeMPU9250.cpp.

4.3.4.6 float GyroscopeMPU9250::getRotationZ ( ) [virtual]

Implements Gyroscope.

Definition at line 16 of file GyroscopeMPU9250.cpp.

4.3.4.7 float GyroscopeMPU9250::readAxisRotation (unsigned char *axisRegister*)

Definition at line 24 of file GyroscopeMPU9250.cpp.

4.3.4.8 unsigned char GyroscopeMPU9250::readXYZ (unsigned char * *buf*)

Definition at line 20 of file GyroscopeMPU9250.cpp.

4.3.4.9 void GyroscopeMPU9250::reset ( )

Definition at line 41 of file GyroscopeMPU9250.cpp.

4.3.4.10 void GyroscopeMPU9250::selectClock (ClockSelection *cs*)

Definition at line 37 of file GyroscopeMPU9250.cpp.

4.3.4.11 void GyroscopeMPU9250::setFullScaleRange (FullScaleRange *fsr*)

Definition at line 32 of file GyroscopeMPU9250.cpp.

4.3.4.12 void GyroscopeMPU9250::sleep ( )

Definition at line 45 of file GyroscopeMPU9250.cpp.

4.3.5 Member Data Documentation

4.3.5.1 GYRO_CONFIGbits GyroscopeMPU9250::config [private]

Definition at line 184 of file GyroscopeMPU9250.h.

The documentation for this class was generated from the following files:

- GyroscopeMPU9250.h
- GyroscopeMPU9250.cpp

4.4 GyroscopeMPU9250::PWR_MGMT_1bits Union Reference

`#include <GyroscopeMPU9250.h>`

Public Attributes

- struct {
  unsigned char CLKSEL:3
  unsigned char PD_PTAT:1
  unsigned char GYRO_STANDBY:1
  unsigned char CYCLE:1
  unsigned char SLEEP:1
  unsigned char H_RESET:1
  };
- unsigned char value

4.4.1 Detailed Description

Power Management 1 Serial IF: R/W Reset value: 0x00.

Definition at line 116 of file GyroscopeMPU9250.h.

4.4.2 Member Data Documentation

4.4.2.1 struct { ... }

4.4.2.2 unsigned char GyroscopeMPU9250::PWR_MGMT_1bits::CLKSEL

Definition at line 118 of file GyroscopeMPU9250.h.

4.4.2.3 unsigned char GyroscopeMPU9250::PWR_MGMT_1bits::CYCLE

Definition at line 121 of file GyroscopeMPU9250.h.

4.4.2.4 unsigned char GyroscopeMPU9250::PWR_MGMT_1bits::GYRO_STANDBY

Definition at line 120 of file GyroscopeMPU9250.h.

4.4.2.5 unsigned char GyroscopeMPU9250::PWR_MGMT_1bits::H_RESET

Definition at line 123 of file GyroscopeMPU9250.h.

4.4.2.6 unsigned char GyroscopeMPU9250::PWR_MGMT_1bits::PD_PTAT

Definition at line 119 of file GyroscopeMPU9250.h.

4.4.2.7 unsigned char GyroscopeMPU9250::PWR_MGMT_1bits::SLEEP

Definition at line 122 of file GyroscopeMPU9250.h.

4.4.2.8 unsigned char GyroscopeMPU9250::PWR_MGMT_1bits::value

Definition at line 125 of file GyroscopeMPU9250.h.

The documentation for this union was generated from the following file:

- GyroscopeMPU9250.h

4.5 GyroscopeMPU9250::PWR_MGMT_2bits Union Reference

```
#include <GyroscopeMPU9250.h>
```

Public Attributes

- struct {
  unsigned char DISABLE_ZG:1
  unsigned char DISABLE_YG:1
  unsigned char DISABLE_XG:1
  unsigned char DISABLE_ZA:1
  unsigned char DISABLE_YA:1
  unsigned char DISABLE_XA:1
  unsigned char:2
  };
- struct {
  unsigned char DISABLE_G:3
  unsigned char DISABLE_A:3
  unsigned char:2
  };
- unsigned char value

4.5.1 Detailed Description

Power Management 2 Serial IF: R/W Reset value: 0x00.

Definition at line 133 of file GyroscopeMPU9250.h.

4.5.2 Member Data Documentation

4.5.2.1 struct { ... }

4.5.2.2 struct { ... }

4.5.2.3 unsigned GyroscopeMPU9250::PWR_MGMT_2bits::char

Definition at line 141 of file GyroscopeMPU9250.h.

4.5.2.4 unsigned char GyroscopeMPU9250::PWR_MGMT_2bits::DISABLE_A

Definition at line 145 of file GyroscopeMPU9250.h.

4.5.2.5 unsigned char GyroscopeMPU9250::PWR_MGMT_2bits::DISABLE_G

Definition at line 144 of file GyroscopeMPU9250.h.

4.5.2.6 unsigned char GyroscopeMPU9250::PWR_MGMT_2bits::DISABLE_XA

Definition at line 140 of file GyroscopeMPU9250.h.

4.5.2.7 unsigned char GyroscopeMPU9250::PWR_MGMT_2bits::DISABLE_XG

Definition at line 137 of file GyroscopeMPU9250.h.

4.5.2.8 unsigned char GyroscopeMPU9250::PWR_MGMT_2bits::DISABLE_YA

Definition at line 139 of file GyroscopeMPU9250.h.

4.5.2.9 unsigned char GyroscopeMPU9250::PWR_MGMT_2bits::DISABLE_YG

Definition at line 136 of file GyroscopeMPU9250.h.

4.5.2.10 unsigned char GyroscopeMPU9250::PWR_MGMT_2bits::DISABLE_ZA

Definition at line 138 of file GyroscopeMPU9250.h.

4.5.2.11 unsigned char GyroscopeMPU9250::PWR_MGMT_2bits::DISABLE_ZG

Definition at line 135 of file GyroscopeMPU9250.h.

4.5.2.12 unsigned char GyroscopeMPU9250::PWR_MGMT_2bits::value

Definition at line 148 of file GyroscopeMPU9250.h.

The documentation for this union was generated from the following file:

- GyroscopeMPU9250.h

5 File Documentation

5.1 Gyroscope.cpp File Reference

```
#include "Gyroscope.h"
```

Include dependency graph for Gyroscope.cpp:

5.2 Gyroscope.cpp

```
00001 #include "Gyroscope.h"
00002
00003 Gyroscope::~Gyroscope() {
00004 }
```

5.3 Gyroscope.h File Reference

This graph shows which files directly or indirectly include this file:

Classes

- class Gyroscope

5.4 Gyroscope.h

```
00001
00009 #ifndef __ARDUINO_DRIVER_GYROSCOPE_H__
00010 #define __ARDUINO_DRIVER_GYROSCOPE_H__ 1
00011
00012 class Gyroscope {
00013
00014 public:
00015
00016     virtual ~Gyroscope();
00017
00018     virtual float getRotationX() = 0;
00019
00020     virtual float getRotationY() = 0;
00021
00022     virtual float getRotationZ() = 0;
00023 };
00024
00025 #endif /* __ARDUINO_DRIVER_GYROSCOPE_H__ */
```

5.5 GyroscopeMPU6050.cpp File Reference

5.6 GyroscopeMPU6050.cpp

```
00001 //
00012 //
00014 //I2Cdev device library code is placed under the MIT license
00015 //Copyright (c) 2012 Jeff Rowberg
00016 //
00017 //Permission is hereby granted, free of charge, to any person obtaining a copy
00018 //of this software and associated documentation files (the "Software"), to deal
00019 //in the Software without restriction, including without limitation the rights
00020 //to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
00021 //copies of the Software, and to permit persons to whom the Software is
00022 //furnished to do so, subject to the following conditions:
00023 //
00024 //The above copyright notice and this permission notice shall be included in
00025 //all copies or substantial portions of the Software.
00026 //
00027 //THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00028 //IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00029 //FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
00030 //AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
00031 //LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
00032 //OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
00033 //THE SOFTWARE.
00034 //===============================================
00035 //*/
00036 //
00037 //#include "MPU6050.h"
00038 //
00040 // * @see MPU6050_DEFAULT_ADDRESS
00041 // */
00042 //MPU6050::MPU6050() {
00043 //    devAddr = MPU6050_DEFAULT_ADDRESS;
00044 //}
00045 //
00047 // * @param address I2C address
00048 // * @see MPU6050_DEFAULT_ADDRESS
00049 // * @see MPU6050_ADDRESS_AD0_LOW
00050 // * @see MPU6050_ADDRESS_AD0_HIGH
00051 // */
00052 //MPU6050::MPU6050(uint8_t address) {
00053 //    devAddr = address;
00054 //}
00055 //
00057 // * This will activate the device and take it out of sleep mode (which must be done
00058 // * after start-up). This function also sets both the accelerometer and the gyroscope
00059 // * to their most sensitive settings, namely +/- 2g and +/- 250 degrees/sec, and sets
00060 // * the clock source to use the X Gyro for reference, which is slightly better than
00061 // * the default internal clock source.
00062 // */
00063 //void MPU6050::initialize() {
00064 //    setClockSource(MPU6050_CLOCK_PLL_XGYRO);
00065 //    setFullScaleGyroRange(MPU6050_GYRO_FS_250);
00066 //    setFullScaleAccelRange(MPU6050_ACCEL_FS_2);
00067 //    setSleepEnabled(false); // thanks to Jack Elston for pointing this one out!
00068 //}
```

```
00069 //
00071 // * Make sure the device is connected and responds as expected.
00072 // * @return True if connection is valid, false otherwise
00073 // */
00074 //bool MPU6050::testConnection() {
00075 //    return getDeviceID() == 0x34;
00076 //}
00077 //
00079 //
00081 // * When set to 1, the auxiliary I2C bus high logic level is VDD. When cleared to
00082 // * 0, the auxiliary I2C bus high logic level is VLOGIC. This does not apply to
00083 // * the MPU-6000, which does not have a VLOGIC pin.
00084 // * @return I2C supply voltage level (0=VLOGIC, 1=VDD)
00085 // */
00086 //uint8_t MPU6050::getAuxVDDIOLevel() {
00087 //    I2Cdev::readBit(devAddr, MPU6050_RA_YG_OFFS_TC, MPU6050_TC_PWR_MODE_BIT, buffer);
00088 //    return buffer[0];
00089 //}
00091 // * When set to 1, the auxiliary I2C bus high logic level is VDD. When cleared to
00092 // * 0, the auxiliary I2C bus high logic level is VLOGIC. This does not apply to
00093 // * the MPU-6000, which does not have a VLOGIC pin.
00094 // * @param level I2C supply voltage level (0=VLOGIC, 1=VDD)
00095 // */
00096 //void MPU6050::setAuxVDDIOLevel(uint8_t level) {
00097 //    I2Cdev::writeBit(devAddr, MPU6050_RA_YG_OFFS_TC, MPU6050_TC_PWR_MODE_BIT, level);
00098 //}
00099 //
00101 //
00103 // * The sensor register output, FIFO output, DMP sampling, Motion detection, Zero
00104 // * Motion detection, and Free Fall detection are all based on the Sample Rate.
00105 // * The Sample Rate is generated by dividing the gyroscope output rate by
00106 // * SMPLRT_DIV:
00107 // *
00108 // * Sample Rate = Gyroscope Output Rate / (1 + SMPLRT_DIV)
00109 // *
00110 // * where Gyroscope Output Rate = 8kHz when the DLPF is disabled (DLPF_CFG = 0 or
00111 // * 7), and 1kHz when the DLPF is enabled (see Register 26).
00112 // *
00113 // * Note: The accelerometer output rate is 1kHz. This means that for a Sample
00114 // * Rate greater than 1kHz, the same accelerometer sample may be output to the
00115 // * FIFO, DMP, and sensor registers more than once.
00116 // *
00117 // * For a diagram of the gyroscope and accelerometer signal paths, see Section 8
00118 // * of the MPU-6000/MPU-6050 Product Specification document.
00119 // *
00120 // * @return Current sample rate
00121 // * @see MPU6050_RA_SMPLRT_DIV
00122 // */
00123 //uint8_t MPU6050::getRate() {
00124 //    I2Cdev::readByte(devAddr, MPU6050_RA_SMPLRT_DIV, buffer);
00125 //    return buffer[0];
00126 //}
00128 // * @param rate New sample rate divider
00129 // * @see getRate()
00130 // * @see MPU6050_RA_SMPLRT_DIV
00131 // */
00132 //void MPU6050::setRate(uint8_t rate) {
00133 //    I2Cdev::writeByte(devAddr, MPU6050_RA_SMPLRT_DIV, rate);
00134 //}
00135 //
00137 //
00139 // * Configures the external Frame Synchronization (FSYNC) pin sampling. An
00140 // * external signal connected to the FSYNC pin can be sampled by configuring
00141 // * EXT_SYNC_SET. Signal changes to the FSYNC pin are latched so that short
00142 // * strobes may be captured. The latched FSYNC signal will be sampled at the
00143 // * Sampling Rate, as defined in register 25. After sampling, the latch will
00144 // * reset to the current FSYNC signal state.
00145 // *
00146 // * The sampled value will be reported in place of the least significant bit in
00147 // * a sensor data register determined by the value of EXT_SYNC_SET according to
00148 // * the following table.
00149 // *
00150 // * <pre>
00151 // * EXT_SYNC_SET | FSYNC Bit Location
00152 // * -------------+-------------------
00153 // * 0            | Input disabled
00154 // * 1            | TEMP_OUT_L[0]
00155 // * 2            | GYRO_XOUT_L[0]
00156 // * 3            | GYRO_YOUT_L[0]
00157 // * 4            | GYRO_ZOUT_L[0]
00158 // * 5            | ACCEL_XOUT_L[0]
00159 // * 6            | ACCEL_YOUT_L[0]
00160 // * 7            | ACCEL_ZOUT_L[0]
00161 // * </pre>
00162 // *
00163 // * @return FSYNC configuration value
00164 // */
```

```
00165 //uint8_t MPU6050::getExternalFrameSync() {
00166 //    I2Cdev::readBits(devAddr, MPU6050_RA_CONFIG, MPU6050_CFG_EXT_SYNC_SET_BIT,
       MPU6050_CFG_EXT_SYNC_SET_LENGTH, buffer);
00167 //    return buffer[0];
00168 //}
00170 // * @see getExternalFrameSync()
00171 // * @see MPU6050_RA_CONFIG
00172 // * @param sync New FSYNC configuration value
00173 // */
00174 //void MPU6050::setExternalFrameSync(uint8_t sync) {
00175 //    I2Cdev::writeBits(devAddr, MPU6050_RA_CONFIG, MPU6050_CFG_EXT_SYNC_SET_BIT,
       MPU6050_CFG_EXT_SYNC_SET_LENGTH, sync);
00176 //}
00178 // * The DLPF_CFG parameter sets the digital low pass filter configuration. It
00179 // * also determines the internal sampling rate used by the device as shown in
00180 // * the table below.
00181 // *
00182 // * Note: The accelerometer output rate is 1kHz. This means that for a Sample
00183 // * Rate greater than 1kHz, the same accelerometer sample may be output to the
00184 // * FIFO, DMP, and sensor registers more than once.
00185 // *
00186 // * <pre>
00187 // *          |   ACCELEROMETER    |           GYROSCOPE
00188 // * DLPF_CFG | Bandwidth | Delay  | Bandwidth | Delay  | Sample Rate
00189 // * ---------+-----------+--------+-----------+--------+-------------
00190 // * 0        | 260Hz     | 0ms    | 256Hz     | 0.98ms | 8kHz
00191 // * 1        | 184Hz     | 2.0ms  | 188Hz     | 1.9ms  | 1kHz
00192 // * 2        | 94Hz      | 3.0ms  | 98Hz      | 2.8ms  | 1kHz
00193 // * 3        | 44Hz      | 4.9ms  | 42Hz      | 4.8ms  | 1kHz
00194 // * 4        | 21Hz      | 8.5ms  | 20Hz      | 8.3ms  | 1kHz
00195 // * 5        | 10Hz      | 13.8ms | 10Hz      | 13.4ms | 1kHz
00196 // * 6        | 5Hz       | 19.0ms | 5Hz       | 18.6ms | 1kHz
00197 // * 7        |   -- Reserved --   |   -- Reserved --   | Reserved
00198 // * </pre>
00199 // *
00200 // * @return DLFP configuration
00201 // * @see MPU6050_RA_CONFIG
00202 // * @see MPU6050_CFG_DLPF_CFG_BIT
00203 // * @see MPU6050_CFG_DLPF_CFG_LENGTH
00204 // */
00205 //uint8_t MPU6050::getDLPFMode() {
00206 //    I2Cdev::readBits(devAddr, MPU6050_RA_CONFIG, MPU6050_CFG_DLPF_CFG_BIT, MPU6050_CFG_DLPF_CFG_LENGTH,
       buffer);
00207 //    return buffer[0];
00208 //}
00210 // * @param mode New DLFP configuration setting
00211 // * @see getDLPFBandwidth()
00212 // * @see MPU6050_DLPF_BW_256
00213 // * @see MPU6050_RA_CONFIG
00214 // * @see MPU6050_CFG_DLPF_CFG_BIT
00215 // * @see MPU6050_CFG_DLPF_CFG_LENGTH
00216 // */
00217 //void MPU6050::setDLPFMode(uint8_t mode) {
00218 //    I2Cdev::writeBits(devAddr, MPU6050_RA_CONFIG, MPU6050_CFG_DLPF_CFG_BIT, MPU6050_CFG_DLPF_CFG_LENGTH,
       mode);
00219 //}
00220 //
00222 //
00224 // * The FS_SEL parameter allows setting the full-scale range of the gyro sensors,
00225 // * as described in the table below.
00226 // *
00227 // * <pre>
00228 // * 0 = +/- 250 degrees/sec
00229 // * 1 = +/- 500 degrees/sec
00230 // * 2 = +/- 1000 degrees/sec
00231 // * 3 = +/- 2000 degrees/sec
00232 // * </pre>
00233 // *
00234 // * @return Current full-scale gyroscope range setting
00235 // * @see MPU6050_GYRO_FS_250
00236 // * @see MPU6050_RA_GYRO_CONFIG
00237 // * @see MPU6050_GCONFIG_FS_SEL_BIT
00238 // * @see MPU6050_GCONFIG_FS_SEL_LENGTH
00239 // */
00240 //uint8_t MPU6050::getFullScaleGyroRange() {
00241 //    I2Cdev::readBits(devAddr, MPU6050_RA_GYRO_CONFIG, MPU6050_GCONFIG_FS_SEL_BIT,
       MPU6050_GCONFIG_FS_SEL_LENGTH, buffer);
00242 //    return buffer[0];
00243 //}
00245 // * @param range New full-scale gyroscope range value
00246 // * @see getFullScaleRange()
00247 // * @see MPU6050_GYRO_FS_250
00248 // * @see MPU6050_RA_GYRO_CONFIG
00249 // * @see MPU6050_GCONFIG_FS_SEL_BIT
00250 // * @see MPU6050_GCONFIG_FS_SEL_LENGTH
00251 // */
00252 //void MPU6050::setFullScaleGyroRange(uint8_t range) {
```

```
00253 //    I2Cdev::writeBits(devAddr, MPU6050_RA_GYRO_CONFIG, MPU6050_GCONFIG_FS_SEL_BIT,
       MPU6050_GCONFIG_FS_SEL_LENGTH, range);
00254 //}
00255 //
00257 //
00259 // * @return Self-test enabled value
00260 // * @see MPU6050_RA_ACCEL_CONFIG
00261 // */
00262 //bool MPU6050::getAccelXSelfTest() {
00263 //    I2Cdev::readBit(devAddr, MPU6050_RA_ACCEL_CONFIG, MPU6050_ACONFIG_XA_ST_BIT, buffer);
00264 //    return buffer[0];
00265 //}
00267 // * @param enabled Self-test enabled value
00268 // * @see MPU6050_RA_ACCEL_CONFIG
00269 // */
00270 //void MPU6050::setAccelXSelfTest(bool enabled) {
00271 //    I2Cdev::writeBit(devAddr, MPU6050_RA_ACCEL_CONFIG, MPU6050_ACONFIG_XA_ST_BIT, enabled);
00272 //}
00274 // * @return Self-test enabled value
00275 // * @see MPU6050_RA_ACCEL_CONFIG
00276 // */
00277 //bool MPU6050::getAccelYSelfTest() {
00278 //    I2Cdev::readBit(devAddr, MPU6050_RA_ACCEL_CONFIG, MPU6050_ACONFIG_YA_ST_BIT, buffer);
00279 //    return buffer[0];
00280 //}
00282 // * @param enabled Self-test enabled value
00283 // * @see MPU6050_RA_ACCEL_CONFIG
00284 // */
00285 //void MPU6050::setAccelYSelfTest(bool enabled) {
00286 //    I2Cdev::writeBit(devAddr, MPU6050_RA_ACCEL_CONFIG, MPU6050_ACONFIG_YA_ST_BIT, enabled);
00287 //}
00289 // * @return Self-test enabled value
00290 // * @see MPU6050_RA_ACCEL_CONFIG
00291 // */
00292 //bool MPU6050::getAccelZSelfTest() {
00293 //    I2Cdev::readBit(devAddr, MPU6050_RA_ACCEL_CONFIG, MPU6050_ACONFIG_ZA_ST_BIT, buffer);
00294 //    return buffer[0];
00295 //}
00297 // * @param enabled Self-test enabled value
00298 // * @see MPU6050_RA_ACCEL_CONFIG
00299 // */
00300 //void MPU6050::setAccelZSelfTest(bool enabled) {
00301 //    I2Cdev::writeBit(devAddr, MPU6050_RA_ACCEL_CONFIG, MPU6050_ACONFIG_ZA_ST_BIT, enabled);
00302 //}
00304 // * The FS_SEL parameter allows setting the full-scale range of the accelerometer
00305 // * sensors, as described in the table below.
00306 // *
00307 // * <pre>
00308 // * 0 = +/- 2g
00309 // * 1 = +/- 4g
00310 // * 2 = +/- 8g
00311 // * 3 = +/- 16g
00312 // * </pre>
00313 // *
00314 // * @return Current full-scale accelerometer range setting
00315 // * @see MPU6050_ACCEL_FS_2
00316 // * @see MPU6050_RA_ACCEL_CONFIG
00317 // * @see MPU6050_ACONFIG_AFS_SEL_BIT
00318 // * @see MPU6050_ACONFIG_AFS_SEL_LENGTH
00319 // */
00320 //uint8_t MPU6050::getFullScaleAccelRange() {
00321 //    I2Cdev::readBits(devAddr, MPU6050_RA_ACCEL_CONFIG, MPU6050_ACONFIG_AFS_SEL_BIT,
       MPU6050_ACONFIG_AFS_SEL_LENGTH, buffer);
00322 //    return buffer[0];
00323 //}
00325 // * @param range New full-scale accelerometer range setting
00326 // * @see getFullScaleAccelRange()
00327 // */
00328 //void MPU6050::setFullScaleAccelRange(uint8_t range) {
00329 //    I2Cdev::writeBits(devAddr, MPU6050_RA_ACCEL_CONFIG, MPU6050_ACONFIG_AFS_SEL_BIT,
       MPU6050_ACONFIG_AFS_SEL_LENGTH, range);
00330 //}
00332 // * The DHPF is a filter module in the path leading to motion detectors (Free
00333 // * Fall, Motion threshold, and Zero Motion). The high pass filter output is not
00334 // * available to the data registers (see Figure in Section 8 of the MPU-6000/
00335 // * MPU-6050 Product Specification document).
00336 // *
00337 // * The high pass filter has three modes:
00338 // *
00339 // * <pre>
00340 // *    Reset: The filter output settles to zero within one sample. This
00341 // *           effectively disables the high pass filter. This mode may be toggled
00342 // *           to quickly settle the filter.
00343 // *
00344 // *    On:    The high pass filter will pass signals above the cut off frequency.
00345 // *
00346 // *    Hold:  When triggered, the filter holds the present sample. The filter
```

```
00347 // *           output will be the difference between the input sample and the held
00348 // *           sample.
00349 // * </pre>
00350 // *
00351 // * <pre>
00352 // * ACCEL_HPF | Filter Mode | Cut-off Frequency
00353 // * ----------+-------------+------------------
00354 // * 0         | Reset       | None
00355 // * 1         | On          | 5Hz
00356 // * 2         | On          | 2.5Hz
00357 // * 3         | On          | 1.25Hz
00358 // * 4         | On          | 0.63Hz
00359 // * 7         | Hold        | None
00360 // * </pre>
00361 // *
00362 // * @return Current high-pass filter configuration
00363 // * @see MPU6050_DHPF_RESET
00364 // * @see MPU6050_RA_ACCEL_CONFIG
00365 // */
00366 //uint8_t MPU6050::getDHPFMode() {
00367 //    I2Cdev::readBits(devAddr, MPU6050_RA_ACCEL_CONFIG, MPU6050_ACONFIG_ACCEL_HPF_BIT,
       MPU6050_ACONFIG_ACCEL_HPF_LENGTH, buffer);
00368 //    return buffer[0];
00369 //}
00371 // * @param bandwidth New high-pass filter configuration
00372 // * @see setDHPFMode()
00373 // * @see MPU6050_DHPF_RESET
00374 // * @see MPU6050_RA_ACCEL_CONFIG
00375 // */
00376 //void MPU6050::setDHPFMode(uint8_t bandwidth) {
00377 //    I2Cdev::writeBits(devAddr, MPU6050_RA_ACCEL_CONFIG, MPU6050_ACONFIG_ACCEL_HPF_BIT,
       MPU6050_ACONFIG_ACCEL_HPF_LENGTH, bandwidth);
00378 //}
00379 //
00381 //
00383 // * This register configures the detection threshold for Free Fall event
00384 // * detection. The unit of FF_THR is 1LSB = 2mg. Free Fall is detected when the
00385 // * absolute value of the accelerometer measurements for the three axes are each
00386 // * less than the detection threshold. This condition increments the Free Fall
00387 // * duration counter (Register 30). The Free Fall interrupt is triggered when the
00388 // * Free Fall duration counter reaches the time specified in FF_DUR.
00389 // *
00390 // * For more details on the Free Fall detection interrupt, see Section 8.2 of the
00391 // * MPU-6000/MPU-6050 Product Specification document as well as Registers 56 and
00392 // * 58 of this document.
00393 // *
00394 // * @return Current free-fall acceleration threshold value (LSB = 2mg)
00395 // * @see MPU6050_RA_FF_THR
00396 // */
00397 //uint8_t MPU6050::getFreefallDetectionThreshold() {
00398 //    I2Cdev::readByte(devAddr, MPU6050_RA_FF_THR, buffer);
00399 //    return buffer[0];
00400 //}
00402 // * @param threshold New free-fall acceleration threshold value (LSB = 2mg)
00403 // * @see getFreefallDetectionThreshold()
00404 // * @see MPU6050_RA_FF_THR
00405 // */
00406 //void MPU6050::setFreefallDetectionThreshold(uint8_t threshold) {
00407 //    I2Cdev::writeByte(devAddr, MPU6050_RA_FF_THR, threshold);
00408 //}
00409 //
00411 //
00413 // * This register configures the duration counter threshold for Free Fall event
00414 // * detection. The duration counter ticks at 1kHz, therefore FF_DUR has a unit
00415 // * of 1 LSB = 1 ms.
00416 // *
00417 // * The Free Fall duration counter increments while the absolute value of the
00418 // * accelerometer measurements are each less than the detection threshold
00419 // * (Register 29). The Free Fall interrupt is triggered when the Free Fall
00420 // * duration counter reaches the time specified in this register.
00421 // *
00422 // * For more details on the Free Fall detection interrupt, see Section 8.2 of
00423 // * the MPU-6000/MPU-6050 Product Specification document as well as Registers 56
00424 // * and 58 of this document.
00425 // *
00426 // * @return Current free-fall duration threshold value (LSB = 1ms)
00427 // * @see MPU6050_RA_FF_DUR
00428 // */
00429 //uint8_t MPU6050::getFreefallDetectionDuration() {
00430 //    I2Cdev::readByte(devAddr, MPU6050_RA_FF_DUR, buffer);
00431 //    return buffer[0];
00432 //}
00434 // * @param duration New free-fall duration threshold value (LSB = 1ms)
00435 // * @see getFreefallDetectionDuration()
00436 // * @see MPU6050_RA_FF_DUR
00437 // */
00438 //void MPU6050::setFreefallDetectionDuration(uint8_t duration) {
```

```
00439 //    I2Cdev::writeByte(devAddr, MPU6050_RA_FF_DUR, duration);
00440 //}
00441 //
00443 //
00445 // * This register configures the detection threshold for Motion interrupt
00446 // * generation. The unit of MOT_THR is 1LSB = 2mg. Motion is detected when the
00447 // * absolute value of any of the accelerometer measurements exceeds this Motion
00448 // * detection threshold. This condition increments the Motion detection duration
00449 // * counter (Register 32). The Motion detection interrupt is triggered when the
00450 // * Motion Detection counter reaches the time count specified in MOT_DUR
00451 // * (Register 32).
00452 // *
00453 // * The Motion interrupt will indicate the axis and polarity of detected motion
00454 // * in MOT_DETECT_STATUS (Register 97).
00455 // *
00456 // * For more details on the Motion detection interrupt, see Section 8.3 of the
00457 // * MPU-6000/MPU-6050 Product Specification document as well as Registers 56 and
00458 // * 58 of this document.
00459 // *
00460 // * @return Current motion detection acceleration threshold value (LSB = 2mg)
00461 // * @see MPU6050_RA_MOT_THR
00462 // */
00463 //uint8_t MPU6050::getMotionDetectionThreshold() {
00464 //    I2Cdev::readByte(devAddr, MPU6050_RA_MOT_THR, buffer);
00465 //    return buffer[0];
00466 //}
00468 // * @param threshold New motion detection acceleration threshold value (LSB = 2mg)
00469 // * @see getMotionDetectionThreshold()
00470 // * @see MPU6050_RA_MOT_THR
00471 // */
00472 //void MPU6050::setMotionDetectionThreshold(uint8_t threshold) {
00473 //    I2Cdev::writeByte(devAddr, MPU6050_RA_MOT_THR, threshold);
00474 //}
00475 //
00477 //
00479 // * This register configures the duration counter threshold for Motion interrupt
00480 // * generation. The duration counter ticks at 1 kHz, therefore MOT_DUR has a unit
00481 // * of 1LSB = 1ms. The Motion detection duration counter increments when the
00482 // * absolute value of any of the accelerometer measurements exceeds the Motion
00483 // * detection threshold (Register 31). The Motion detection interrupt is
00484 // * triggered when the Motion detection counter reaches the time count specified
00485 // * in this register.
00486 // *
00487 // * For more details on the Motion detection interrupt, see Section 8.3 of the
00488 // * MPU-6000/MPU-6050 Product Specification document.
00489 // *
00490 // * @return Current motion detection duration threshold value (LSB = 1ms)
00491 // * @see MPU6050_RA_MOT_DUR
00492 // */
00493 //uint8_t MPU6050::getMotionDetectionDuration() {
00494 //    I2Cdev::readByte(devAddr, MPU6050_RA_MOT_DUR, buffer);
00495 //    return buffer[0];
00496 //}
00498 // * @param duration New motion detection duration threshold value (LSB = 1ms)
00499 // * @see getMotionDetectionDuration()
00500 // * @see MPU6050_RA_MOT_DUR
00501 // */
00502 //void MPU6050::setMotionDetectionDuration(uint8_t duration) {
00503 //    I2Cdev::writeByte(devAddr, MPU6050_RA_MOT_DUR, duration);
00504 //}
00505 //
00507 //
00509 // * This register configures the detection threshold for Zero Motion interrupt
00510 // * generation. The unit of ZRMOT_THR is 1LSB = 2mg. Zero Motion is detected when
00511 // * the absolute value of the accelerometer measurements for the 3 axes are each
00512 // * less than the detection threshold. This condition increments the Zero Motion
00513 // * duration counter (Register 34). The Zero Motion interrupt is triggered when
00514 // * the Zero Motion duration counter reaches the time count specified in
00515 // * ZRMOT_DUR (Register 34).
00516 // *
00517 // * Unlike Free Fall or Motion detection, Zero Motion detection triggers an
00518 // * interrupt both when Zero Motion is first detected and when Zero Motion is no
00519 // * longer detected.
00520 // *
00521 // * When a zero motion event is detected, a Zero Motion Status will be indicated
00522 // * in the MOT_DETECT_STATUS register (Register 97). When a motion-to-zero-motion
00523 // * condition is detected, the status bit is set to 1. When a zero-motion-to-
00524 // * motion condition is detected, the status bit is set to 0.
00525 // *
00526 // * For more details on the Zero Motion detection interrupt, see Section 8.4 of
00527 // * the MPU-6000/MPU-6050 Product Specification document as well as Registers 56
00528 // * and 58 of this document.
00529 // *
00530 // * @return Current zero motion detection acceleration threshold value (LSB = 2mg)
00531 // * @see MPU6050_RA_ZRMOT_THR
00532 // */
00533 //uint8_t MPU6050::getZeroMotionDetectionThreshold() {
```

```
00534 //    I2Cdev::readByte(devAddr, MPU6050_RA_ZRMOT_THR, buffer);
00535 //    return buffer[0];
00536 //}
00538 // * @param threshold New zero motion detection acceleration threshold value (LSB = 2mg)
00539 // * @see getZeroMotionDetectionThreshold()
00540 // * @see MPU6050_RA_ZRMOT_THR
00541 // */
00542 //void MPU6050::setZeroMotionDetectionThreshold(uint8_t threshold) {
00543 //    I2Cdev::writeByte(devAddr, MPU6050_RA_ZRMOT_THR, threshold);
00544 //}
00545 //
00547 //
00549 // * This register configures the duration counter threshold for Zero Motion
00550 // * interrupt generation. The duration counter ticks at 16 Hz, therefore
00551 // * ZRMOT_DUR has a unit of 1 LSB = 64 ms. The Zero Motion duration counter
00552 // * increments while the absolute value of the accelerometer measurements are
00553 // * each less than the detection threshold (Register 33). The Zero Motion
00554 // * interrupt is triggered when the Zero Motion duration counter reaches the time
00555 // * count specified in this register.
00556 // *
00557 // * For more details on the Zero Motion detection interrupt, see Section 8.4 of
00558 // * the MPU-6000/MPU-6050 Product Specification document, as well as Registers 56
00559 // * and 58 of this document.
00560 // *
00561 // * @return Current zero motion detection duration threshold value (LSB = 64ms)
00562 // * @see MPU6050_RA_ZRMOT_DUR
00563 // */
00564 //uint8_t MPU6050::getZeroMotionDetectionDuration() {
00565 //    I2Cdev::readByte(devAddr, MPU6050_RA_ZRMOT_DUR, buffer);
00566 //    return buffer[0];
00567 //}
00569 // * @param duration New zero motion detection duration threshold value (LSB = 1ms)
00570 // * @see getZeroMotionDetectionDuration()
00571 // * @see MPU6050_RA_ZRMOT_DUR
00572 // */
00573 //void MPU6050::setZeroMotionDetectionDuration(uint8_t duration) {
00574 //    I2Cdev::writeByte(devAddr, MPU6050_RA_ZRMOT_DUR, duration);
00575 //}
00576 //
00578 //
00580 // * When set to 1, this bit enables TEMP_OUT_H and TEMP_OUT_L (Registers 65 and
00581 // * 66) to be written into the FIFO buffer.
00582 // * @return Current temperature FIFO enabled value
00583 // * @see MPU6050_RA_FIFO_EN
00584 // */
00585 //bool MPU6050::getTempFIFOEnabled() {
00586 //    I2Cdev::readBit(devAddr, MPU6050_RA_FIFO_EN, MPU6050_TEMP_FIFO_EN_BIT, buffer);
00587 //    return buffer[0];
00588 //}
00590 // * @param enabled New temperature FIFO enabled value
00591 // * @see getTempFIFOEnabled()
00592 // * @see MPU6050_RA_FIFO_EN
00593 // */
00594 //void MPU6050::setTempFIFOEnabled(bool enabled) {
00595 //    I2Cdev::writeBit(devAddr, MPU6050_RA_FIFO_EN, MPU6050_TEMP_FIFO_EN_BIT, enabled);
00596 //}
00598 // * When set to 1, this bit enables GYRO_XOUT_H and GYRO_XOUT_L (Registers 67 and
00599 // * 68) to be written into the FIFO buffer.
00600 // * @return Current gyroscope X-axis FIFO enabled value
00601 // * @see MPU6050_RA_FIFO_EN
00602 // */
00603 //bool MPU6050::getXGyroFIFOEnabled() {
00604 //    I2Cdev::readBit(devAddr, MPU6050_RA_FIFO_EN, MPU6050_XG_FIFO_EN_BIT, buffer);
00605 //    return buffer[0];
00606 //}
00608 // * @param enabled New gyroscope X-axis FIFO enabled value
00609 // * @see getXGyroFIFOEnabled()
00610 // * @see MPU6050_RA_FIFO_EN
00611 // */
00612 //void MPU6050::setXGyroFIFOEnabled(bool enabled) {
00613 //    I2Cdev::writeBit(devAddr, MPU6050_RA_FIFO_EN, MPU6050_XG_FIFO_EN_BIT, enabled);
00614 //}
00616 // * When set to 1, this bit enables GYRO_YOUT_H and GYRO_YOUT_L (Registers 69 and
00617 // * 70) to be written into the FIFO buffer.
00618 // * @return Current gyroscope Y-axis FIFO enabled value
00619 // * @see MPU6050_RA_FIFO_EN
00620 // */
00621 //bool MPU6050::getYGyroFIFOEnabled() {
00622 //    I2Cdev::readBit(devAddr, MPU6050_RA_FIFO_EN, MPU6050_YG_FIFO_EN_BIT, buffer);
00623 //    return buffer[0];
00624 //}
00626 // * @param enabled New gyroscope Y-axis FIFO enabled value
00627 // * @see getYGyroFIFOEnabled()
00628 // * @see MPU6050_RA_FIFO_EN
00629 // */
00630 //void MPU6050::setYGyroFIFOEnabled(bool enabled) {
00631 //    I2Cdev::writeBit(devAddr, MPU6050_RA_FIFO_EN, MPU6050_YG_FIFO_EN_BIT, enabled);
```

```
00632 //}
00634 // * When set to 1, this bit enables GYRO_ZOUT_H and GYRO_ZOUT_L (Registers 71 and
00635 // * 72) to be written into the FIFO buffer.
00636 // * @return Current gyroscope Z-axis FIFO enabled value
00637 // * @see MPU6050_RA_FIFO_EN
00638 // */
00639 //bool MPU6050::getZGyroFIFOEnabled() {
00640 //    I2Cdev::readBit(devAddr, MPU6050_RA_FIFO_EN, MPU6050_ZG_FIFO_EN_BIT, buffer);
00641 //    return buffer[0];
00642 //}
00644 // * @param enabled New gyroscope Z-axis FIFO enabled value
00645 // * @see getZGyroFIFOEnabled()
00646 // * @see MPU6050_RA_FIFO_EN
00647 // */
00648 //void MPU6050::setZGyroFIFOEnabled(bool enabled) {
00649 //    I2Cdev::writeBit(devAddr, MPU6050_RA_FIFO_EN, MPU6050_ZG_FIFO_EN_BIT, enabled);
00650 //}
00652 // * When set to 1, this bit enables ACCEL_XOUT_H, ACCEL_XOUT_L, ACCEL_YOUT_H,
00653 // * ACCEL_YOUT_L, ACCEL_ZOUT_H, and ACCEL_ZOUT_L (Registers 59 to 64) to be
00654 // * written into the FIFO buffer.
00655 // * @return Current accelerometer FIFO enabled value
00656 // * @see MPU6050_RA_FIFO_EN
00657 // */
00658 //bool MPU6050::getAccelFIFOEnabled() {
00659 //    I2Cdev::readBit(devAddr, MPU6050_RA_FIFO_EN, MPU6050_ACCEL_FIFO_EN_BIT, buffer);
00660 //    return buffer[0];
00661 //}
00663 // * @param enabled New accelerometer FIFO enabled value
00664 // * @see getAccelFIFOEnabled()
00665 // * @see MPU6050_RA_FIFO_EN
00666 // */
00667 //void MPU6050::setAccelFIFOEnabled(bool enabled) {
00668 //    I2Cdev::writeBit(devAddr, MPU6050_RA_FIFO_EN, MPU6050_ACCEL_FIFO_EN_BIT, enabled);
00669 //}
00671 // * When set to 1, this bit enables EXT_SENS_DATA registers (Registers 73 to 96)
00672 // * associated with Slave 2 to be written into the FIFO buffer.
00673 // * @return Current Slave 2 FIFO enabled value
00674 // * @see MPU6050_RA_FIFO_EN
00675 // */
00676 //bool MPU6050::getSlave2FIFOEnabled() {
00677 //    I2Cdev::readBit(devAddr, MPU6050_RA_FIFO_EN, MPU6050_SLV2_FIFO_EN_BIT, buffer);
00678 //    return buffer[0];
00679 //}
00681 // * @param enabled New Slave 2 FIFO enabled value
00682 // * @see getSlave2FIFOEnabled()
00683 // * @see MPU6050_RA_FIFO_EN
00684 // */
00685 //void MPU6050::setSlave2FIFOEnabled(bool enabled) {
00686 //    I2Cdev::writeBit(devAddr, MPU6050_RA_FIFO_EN, MPU6050_SLV2_FIFO_EN_BIT, enabled);
00687 //}
00689 // * When set to 1, this bit enables EXT_SENS_DATA registers (Registers 73 to 96)
00690 // * associated with Slave 1 to be written into the FIFO buffer.
00691 // * @return Current Slave 1 FIFO enabled value
00692 // * @see MPU6050_RA_FIFO_EN
00693 // */
00694 //bool MPU6050::getSlave1FIFOEnabled() {
00695 //    I2Cdev::readBit(devAddr, MPU6050_RA_FIFO_EN, MPU6050_SLV1_FIFO_EN_BIT, buffer);
00696 //    return buffer[0];
00697 //}
00699 // * @param enabled New Slave 1 FIFO enabled value
00700 // * @see getSlave1FIFOEnabled()
00701 // * @see MPU6050_RA_FIFO_EN
00702 // */
00703 //void MPU6050::setSlave1FIFOEnabled(bool enabled) {
00704 //    I2Cdev::writeBit(devAddr, MPU6050_RA_FIFO_EN, MPU6050_SLV1_FIFO_EN_BIT, enabled);
00705 //}
00707 // * When set to 1, this bit enables EXT_SENS_DATA registers (Registers 73 to 96)
00708 // * associated with Slave 0 to be written into the FIFO buffer.
00709 // * @return Current Slave 0 FIFO enabled value
00710 // * @see MPU6050_RA_FIFO_EN
00711 // */
00712 //bool MPU6050::getSlave0FIFOEnabled() {
00713 //    I2Cdev::readBit(devAddr, MPU6050_RA_FIFO_EN, MPU6050_SLV0_FIFO_EN_BIT, buffer);
00714 //    return buffer[0];
00715 //}
00717 // * @param enabled New Slave 0 FIFO enabled value
00718 // * @see getSlave0FIFOEnabled()
00719 // * @see MPU6050_RA_FIFO_EN
00720 // */
00721 //void MPU6050::setSlave0FIFOEnabled(bool enabled) {
00722 //    I2Cdev::writeBit(devAddr, MPU6050_RA_FIFO_EN, MPU6050_SLV0_FIFO_EN_BIT, enabled);
00723 //}
00724 //
00726 //
00728 // * Multi-master capability allows multiple I2C masters to operate on the same
00729 // * bus. In circuits where multi-master capability is required, set MULT_MST_EN
00730 // * to 1. This will increase current drawn by approximately 30uA.
```

```
00731 // *
00732 // * In circuits where multi-master capability is required, the state of the I2C
00733 // * bus must always be monitored by each separate I2C Master. Before an I2C
00734 // * Master can assume arbitration of the bus, it must first confirm that no other
00735 // * I2C Master has arbitration of the bus. When MULT_MST_EN is set to 1, the
00736 // * MPU-60X0's bus arbitration detection logic is turned on, enabling it to
00737 // * detect when the bus is available.
00738 // *
00739 // * @return Current multi-master enabled value
00740 // * @see MPU6050_RA_I2C_MST_CTRL
00741 // */
00742 //bool MPU6050::getMultiMasterEnabled() {
00743 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_MST_CTRL, MPU6050_MULT_MST_EN_BIT, buffer);
00744 //    return buffer[0];
00745 //}
00747 // * @param enabled New multi-master enabled value
00748 // * @see getMultiMasterEnabled()
00749 // * @see MPU6050_RA_I2C_MST_CTRL
00750 // */
00751 //void MPU6050::setMultiMasterEnabled(bool enabled) {
00752 //    I2Cdev::writeBit(devAddr, MPU6050_RA_I2C_MST_CTRL, MPU6050_MULT_MST_EN_BIT, enabled);
00753 //}
00755 // * When the WAIT_FOR_ES bit is set to 1, the Data Ready interrupt will be
00756 // * delayed until External Sensor data from the Slave Devices are loaded into the
00757 // * EXT_SENS_DATA registers. This is used to ensure that both the internal sensor
00758 // * data (i.e. from gyro and accel) and external sensor data have been loaded to
00759 // * their respective data registers (i.e. the data is synced) when the Data Ready
00760 // * interrupt is triggered.
00761 // *
00762 // * @return Current wait-for-external-sensor-data enabled value
00763 // * @see MPU6050_RA_I2C_MST_CTRL
00764 // */
00765 //bool MPU6050::getWaitForExternalSensorEnabled() {
00766 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_MST_CTRL, MPU6050_WAIT_FOR_ES_BIT, buffer);
00767 //    return buffer[0];
00768 //}
00770 // * @param enabled New wait-for-external-sensor-data enabled value
00771 // * @see getWaitForExternalSensorEnabled()
00772 // * @see MPU6050_RA_I2C_MST_CTRL
00773 // */
00774 //void MPU6050::setWaitForExternalSensorEnabled(bool enabled) {
00775 //    I2Cdev::writeBit(devAddr, MPU6050_RA_I2C_MST_CTRL, MPU6050_WAIT_FOR_ES_BIT, enabled);
00776 //}
00778 // * When set to 1, this bit enables EXT_SENS_DATA registers (Registers 73 to 96)
00779 // * associated with Slave 3 to be written into the FIFO buffer.
00780 // * @return Current Slave 3 FIFO enabled value
00781 // * @see MPU6050_RA_MST_CTRL
00782 // */
00783 //bool MPU6050::getSlave3FIFOEnabled() {
00784 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_MST_CTRL, MPU6050_SLV_3_FIFO_EN_BIT, buffer);
00785 //    return buffer[0];
00786 //}
00788 // * @param enabled New Slave 3 FIFO enabled value
00789 // * @see getSlave3FIFOEnabled()
00790 // * @see MPU6050_RA_MST_CTRL
00791 // */
00792 //void MPU6050::setSlave3FIFOEnabled(bool enabled) {
00793 //    I2Cdev::writeBit(devAddr, MPU6050_RA_I2C_MST_CTRL, MPU6050_SLV_3_FIFO_EN_BIT, enabled);
00794 //}
00796 // * The I2C_MST_P_NSR bit configures the I2C Master's transition from one slave
00797 // * read to the next slave read. If the bit equals 0, there will be a restart
00798 // * between reads. If the bit equals 1, there will be a stop followed by a start
00799 // * of the following read. When a write transaction follows a read transaction,
00800 // * the stop followed by a start of the successive write will be always used.
00801 // *
00802 // * @return Current slave read/write transition enabled value
00803 // * @see MPU6050_RA_I2C_MST_CTRL
00804 // */
00805 //bool MPU6050::getSlaveReadWriteTransitionEnabled() {
00806 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_MST_CTRL, MPU6050_I2C_MST_P_NSR_BIT, buffer);
00807 //    return buffer[0];
00808 //}
00810 // * @param enabled New slave read/write transition enabled value
00811 // * @see getSlaveReadWriteTransitionEnabled()
00812 // * @see MPU6050_RA_I2C_MST_CTRL
00813 // */
00814 //void MPU6050::setSlaveReadWriteTransitionEnabled(bool enabled) {
00815 //    I2Cdev::writeBit(devAddr, MPU6050_RA_I2C_MST_CTRL, MPU6050_I2C_MST_P_NSR_BIT, enabled);
00816 //}
00818 // * I2C_MST_CLK is a 4 bit unsigned value which configures a divider on the
00819 // * MPU-60X0 internal 8MHz clock. It sets the I2C master clock speed according to
00820 // * the following table:
00821 // *
00822 // * <pre>
00823 // * I2C_MST_CLK | I2C Master Clock Speed | 8MHz Clock Divider
00824 // * ------------+-----------------------+------------------
00825 // * 0           | 348kHz                | 23
```

```
00826 // * 1           | 333kHz                | 24
00827 // * 2           | 320kHz                | 25
00828 // * 3           | 308kHz                | 26
00829 // * 4           | 296kHz                | 27
00830 // * 5           | 286kHz                | 28
00831 // * 6           | 276kHz                | 29
00832 // * 7           | 267kHz                | 30
00833 // * 8           | 258kHz                | 31
00834 // * 9           | 500kHz                | 16
00835 // * 10          | 471kHz                | 17
00836 // * 11          | 444kHz                | 18
00837 // * 12          | 421kHz                | 19
00838 // * 13          | 400kHz                | 20
00839 // * 14          | 381kHz                | 21
00840 // * 15          | 364kHz                | 22
00841 // * </pre>
00842 // *
00843 // * @return Current I2C master clock speed
00844 // * @see MPU6050_RA_I2C_MST_CTRL
00845 // */
00846 //uint8_t MPU6050::getMasterClockSpeed() {
00847 //    I2Cdev::readBits(devAddr, MPU6050_RA_I2C_MST_CTRL, MPU6050_I2C_MST_CLK_BIT,
       MPU6050_I2C_MST_CLK_LENGTH, buffer);
00848 //    return buffer[0];
00849 //}
00851 // * @reparam speed Current I2C master clock speed
00852 // * @see MPU6050_RA_I2C_MST_CTRL
00853 // */
00854 //void MPU6050::setMasterClockSpeed(uint8_t speed) {
00855 //    I2Cdev::writeBits(devAddr, MPU6050_RA_I2C_MST_CTRL, MPU6050_I2C_MST_CLK_BIT,
       MPU6050_I2C_MST_CLK_LENGTH, speed);
00856 //}
00857 //
00859 //
00861 // * Note that Bit 7 (MSB) controls read/write mode. If Bit 7 is set, it's a read
00862 // * operation, and if it is cleared, then it's a write operation. The remaining
00863 // * bits (6-0) are the 7-bit device address of the slave device.
00864 // *
00865 // * In read mode, the result of the read is placed in the lowest available
00866 // * EXT_SENS_DATA register. For further information regarding the allocation of
00867 // * read results, please refer to the EXT_SENS_DATA register description
00868 // * (Registers 73 - 96).
00869 // *
00870 // * The MPU-6050 supports a total of five slaves, but Slave 4 has unique
00871 // * characteristics, and so it has its own functions (getSlave4* and setSlave4*).
00872 // *
00873 // * I2C data transactions are performed at the Sample Rate, as defined in
00874 // * Register 25. The user is responsible for ensuring that I2C data transactions
00875 // * to and from each enabled Slave can be completed within a single period of the
00876 // * Sample Rate.
00877 // *
00878 // * The I2C slave access rate can be reduced relative to the Sample Rate. This
00879 // * reduced access rate is determined by I2C_MST_DLY (Register 52). Whether a
00880 // * slave's access rate is reduced relative to the Sample Rate is determined by
00881 // * I2C_MST_DELAY_CTRL (Register 103).
00882 // *
00883 // * The processing order for the slaves is fixed. The sequence followed for
00884 // * processing the slaves is Slave 0, Slave 1, Slave 2, Slave 3 and Slave 4. If a
00885 // * particular Slave is disabled it will be skipped.
00886 // *
00887 // * Each slave can either be accessed at the sample rate or at a reduced sample
00888 // * rate. In a case where some slaves are accessed at the Sample Rate and some
00889 // * slaves are accessed at the reduced rate, the sequence of accessing the slaves
00890 // * (Slave 0 to Slave 4) is still followed. However, the reduced rate slaves will
00891 // * be skipped if their access rate dictates that they should not be accessed
00892 // * during that particular cycle. For further information regarding the reduced
00893 // * access rate, please refer to Register 52. Whether a slave is accessed at the
00894 // * Sample Rate or at the reduced rate is determined by the Delay Enable bits in
00895 // * Register 103.
00896 // *
00897 // * @param num Slave number (0-3)
00898 // * @return Current address for specified slave
00899 // * @see MPU6050_RA_I2C_SLV0_ADDR
00900 // */
00901 //uint8_t MPU6050::getSlaveAddress(uint8_t num) {
00902 //    if (num > 3) return 0;
00903 //    I2Cdev::readByte(devAddr, MPU6050_RA_I2C_SLV0_ADDR + num*3, buffer);
00904 //    return buffer[0];
00905 //}
00907 // * @param num Slave number (0-3)
00908 // * @param address New address for specified slave
00909 // * @see getSlaveAddress()
00910 // * @see MPU6050_RA_I2C_SLV0_ADDR
00911 // */
00912 //void MPU6050::setSlaveAddress(uint8_t num, uint8_t address) {
00913 //    if (num > 3) return;
00914 //    I2Cdev::writeByte(devAddr, MPU6050_RA_I2C_SLV0_ADDR + num*3, address);
```

```
00915 //}
00917 // * Read/write operations for this slave will be done to whatever internal
00918 // * register address is stored in this MPU register.
00919 // *
00920 // * The MPU-6050 supports a total of five slaves, but Slave 4 has unique
00921 // * characteristics, and so it has its own functions.
00922 // *
00923 // * @param num Slave number (0-3)
00924 // * @return Current active register for specified slave
00925 // * @see MPU6050_RA_I2C_SLV0_REG
00926 // */
00927 //uint8_t MPU6050::getSlaveRegister(uint8_t num) {
00928 //    if (num > 3) return 0;
00929 //    I2Cdev::readByte(devAddr, MPU6050_RA_I2C_SLV0_REG + num*3, buffer);
00930 //    return buffer[0];
00931 //}
00933 // * @param num Slave number (0-3)
00934 // * @param reg New active register for specified slave
00935 // * @see getSlaveRegister()
00936 // * @see MPU6050_RA_I2C_SLV0_REG
00937 // */
00938 //void MPU6050::setSlaveRegister(uint8_t num, uint8_t reg) {
00939 //    if (num > 3) return;
00940 //    I2Cdev::writeByte(devAddr, MPU6050_RA_I2C_SLV0_REG + num*3, reg);
00941 //}
00943 // * When set to 1, this bit enables Slave 0 for data transfer operations. When
00944 // * cleared to 0, this bit disables Slave 0 from data transfer operations.
00945 // * @param num Slave number (0-3)
00946 // * @return Current enabled value for specified slave
00947 // * @see MPU6050_RA_I2C_SLV0_CTRL
00948 // */
00949 //bool MPU6050::getSlaveEnabled(uint8_t num) {
00950 //    if (num > 3) return 0;
00951 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_SLV0_CTRL + num*3, MPU6050_I2C_SLV_EN_BIT, buffer);
00952 //    return buffer[0];
00953 //}
00955 // * @param num Slave number (0-3)
00956 // * @param enabled New enabled value for specified slave
00957 // * @see getSlaveEnabled()
00958 // * @see MPU6050_RA_I2C_SLV0_CTRL
00959 // */
00960 //void MPU6050::setSlaveEnabled(uint8_t num, bool enabled) {
00961 //    if (num > 3) return;
00962 //    I2Cdev::writeBit(devAddr, MPU6050_RA_I2C_SLV0_CTRL + num*3, MPU6050_I2C_SLV_EN_BIT, enabled);
00963 //}
00965 // * When set to 1, this bit enables byte swapping. When byte swapping is enabled,
00966 // * the high and low bytes of a word pair are swapped. Please refer to
00967 // * I2C_SLV0_GRP for the pairing convention of the word pairs. When cleared to 0,
00968 // * bytes transferred to and from Slave 0 will be written to EXT_SENS_DATA
00969 // * registers in the order they were transferred.
00970 // *
00971 // * @param num Slave number (0-3)
00972 // * @return Current word pair byte-swapping enabled value for specified slave
00973 // * @see MPU6050_RA_I2C_SLV0_CTRL
00974 // */
00975 //bool MPU6050::getSlaveWordByteSwap(uint8_t num) {
00976 //    if (num > 3) return 0;
00977 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_SLV0_CTRL + num*3, MPU6050_I2C_SLV_BYTE_SW_BIT, buffer);
00978 //    return buffer[0];
00979 //}
00981 // * @param num Slave number (0-3)
00982 // * @param enabled New word pair byte-swapping enabled value for specified slave
00983 // * @see getSlaveWordByteSwap()
00984 // * @see MPU6050_RA_I2C_SLV0_CTRL
00985 // */
00986 //void MPU6050::setSlaveWordByteSwap(uint8_t num, bool enabled) {
00987 //    if (num > 3) return;
00988 //    I2Cdev::writeBit(devAddr, MPU6050_RA_I2C_SLV0_CTRL + num*3, MPU6050_I2C_SLV_BYTE_SW_BIT, enabled);
00989 //}
00991 // * When set to 1, the transaction will read or write data only. When cleared to
00992 // * 0, the transaction will write a register address prior to reading or writing
00993 // * data. This should equal 0 when specifying the register address within the
00994 // * Slave device to/from which the ensuing data transaction will take place.
00995 // *
00996 // * @param num Slave number (0-3)
00997 // * @return Current write mode for specified slave (0 = register address + data, 1 = data only)
00998 // * @see MPU6050_RA_I2C_SLV0_CTRL
00999 // */
01000 //bool MPU6050::getSlaveWriteMode(uint8_t num) {
01001 //    if (num > 3) return 0;
01002 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_SLV0_CTRL + num*3, MPU6050_I2C_SLV_REG_DIS_BIT, buffer);
01003 //    return buffer[0];
01004 //}
01006 // * @param num Slave number (0-3)
01007 // * @param mode New write mode for specified slave (0 = register address + data, 1 = data only)
01008 // * @see getSlaveWriteMode()
01009 // * @see MPU6050_RA_I2C_SLV0_CTRL
```

```
01010 // */
01011 //void MPU6050::setSlaveWriteMode(uint8_t num, bool mode) {
01012 //    if (num > 3) return;
01013 //    I2Cdev::writeBit(devAddr, MPU6050_RA_I2C_SLV0_CTRL + num*3, MPU6050_I2C_SLV_REG_DIS_BIT, mode);
01014 //}
01016 // * This sets specifies the grouping order of word pairs received from registers.
01017 // * When cleared to 0, bytes from register addresses 0 and 1, 2 and 3, etc (even,
01018 // * then odd register addresses) are paired to form a word. When set to 1, bytes
01019 // * from register addresses are paired 1 and 2, 3 and 4, etc. (odd, then even
01020 // * register addresses) are paired to form a word.
01021 // *
01022 // * @param num Slave number (0-3)
01023 // * @return Current word pair grouping order offset for specified slave
01024 // * @see MPU6050_RA_I2C_SLV0_CTRL
01025 // */
01026 //bool MPU6050::getSlaveWordGroupOffset(uint8_t num) {
01027 //    if (num > 3) return 0;
01028 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_SLV0_CTRL + num*3, MPU6050_I2C_SLV_GRP_BIT, buffer);
01029 //    return buffer[0];
01030 //}
01032 // * @param num Slave number (0-3)
01033 // * @param enabled New word pair grouping order offset for specified slave
01034 // * @see getSlaveWordGroupOffset()
01035 // * @see MPU6050_RA_I2C_SLV0_CTRL
01036 // */
01037 //void MPU6050::setSlaveWordGroupOffset(uint8_t num, bool enabled) {
01038 //    if (num > 3) return;
01039 //    I2Cdev::writeBit(devAddr, MPU6050_RA_I2C_SLV0_CTRL + num*3, MPU6050_I2C_SLV_GRP_BIT, enabled);
01040 //}
01042 // * Specifies the number of bytes transferred to and from Slave 0. Clearing this
01043 // * bit to 0 is equivalent to disabling the register by writing 0 to I2C_SLV0_EN.
01044 // * @param num Slave number (0-3)
01045 // * @return Number of bytes to read for specified slave
01046 // * @see MPU6050_RA_I2C_SLV0_CTRL
01047 // */
01048 //uint8_t MPU6050::getSlaveDataLength(uint8_t num) {
01049 //    if (num > 3) return 0;
01050 //    I2Cdev::readBits(devAddr, MPU6050_RA_I2C_SLV0_CTRL + num*3, MPU6050_I2C_SLV_LEN_BIT,
       MPU6050_I2C_SLV_LEN_LENGTH, buffer);
01051 //    return buffer[0];
01052 //}
01054 // * @param num Slave number (0-3)
01055 // * @param length Number of bytes to read for specified slave
01056 // * @see getSlaveDataLength()
01057 // * @see MPU6050_RA_I2C_SLV0_CTRL
01058 // */
01059 //void MPU6050::setSlaveDataLength(uint8_t num, uint8_t length) {
01060 //    if (num > 3) return;
01061 //    I2Cdev::writeBits(devAddr, MPU6050_RA_I2C_SLV0_CTRL + num*3, MPU6050_I2C_SLV_LEN_BIT,
       MPU6050_I2C_SLV_LEN_LENGTH, length);
01062 //}
01063 //
01065 //
01067 // * Note that Bit 7 (MSB) controls read/write mode. If Bit 7 is set, it's a read
01068 // * operation, and if it is cleared, then it's a write operation. The remaining
01069 // * bits (6-0) are the 7-bit device address of the slave device.
01070 // *
01071 // * @return Current address for Slave 4
01072 // * @see getSlaveAddress()
01073 // * @see MPU6050_RA_I2C_SLV4_ADDR
01074 // */
01075 //uint8_t MPU6050::getSlave4Address() {
01076 //    I2Cdev::readByte(devAddr, MPU6050_RA_I2C_SLV4_ADDR, buffer);
01077 //    return buffer[0];
01078 //}
01080 // * @param address New address for Slave 4
01081 // * @see getSlave4Address()
01082 // * @see MPU6050_RA_I2C_SLV4_ADDR
01083 // */
01084 //void MPU6050::setSlave4Address(uint8_t address) {
01085 //    I2Cdev::writeByte(devAddr, MPU6050_RA_I2C_SLV4_ADDR, address);
01086 //}
01088 // * Read/write operations for this slave will be done to whatever internal
01089 // * register address is stored in this MPU register.
01090 // *
01091 // * @return Current active register for Slave 4
01092 // * @see MPU6050_RA_I2C_SLV4_REG
01093 // */
01094 //uint8_t MPU6050::getSlave4Register() {
01095 //    I2Cdev::readByte(devAddr, MPU6050_RA_I2C_SLV4_REG, buffer);
01096 //    return buffer[0];
01097 //}
01099 // * @param reg New active register for Slave 4
01100 // * @see getSlave4Register()
01101 // * @see MPU6050_RA_I2C_SLV4_REG
01102 // */
01103 //void MPU6050::setSlave4Register(uint8_t reg) {
```

```
01104 //    I2Cdev::writeByte(devAddr, MPU6050_RA_I2C_SLV4_REG, reg);
01105 //}
01107 // * This register stores the data to be written into the Slave 4. If I2C_SLV4_RW
01108 // * is set 1 (set to read), this register has no effect.
01109 // * @param data New byte to write to Slave 4
01110 // * @see MPU6050_RA_I2C_SLV4_DO
01111 // */
01112 //void MPU6050::setSlave4OutputByte(uint8_t data) {
01113 //    I2Cdev::writeByte(devAddr, MPU6050_RA_I2C_SLV4_DO, data);
01114 //}
01116 // * When set to 1, this bit enables Slave 4 for data transfer operations. When
01117 // * cleared to 0, this bit disables Slave 4 from data transfer operations.
01118 // * @return Current enabled value for Slave 4
01119 // * @see MPU6050_RA_I2C_SLV4_CTRL
01120 // */
01121 //bool MPU6050::getSlave4Enabled() {
01122 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_SLV4_CTRL, MPU6050_I2C_SLV4_EN_BIT, buffer);
01123 //    return buffer[0];
01124 //}
01126 // * @param enabled New enabled value for Slave 4
01127 // * @see getSlave4Enabled()
01128 // * @see MPU6050_RA_I2C_SLV4_CTRL
01129 // */
01130 //void MPU6050::setSlave4Enabled(bool enabled) {
01131 //    I2Cdev::writeBit(devAddr, MPU6050_RA_I2C_SLV4_CTRL, MPU6050_I2C_SLV4_EN_BIT, enabled);
01132 //}
01134 // * When set to 1, this bit enables the generation of an interrupt signal upon
01135 // * completion of a Slave 4 transaction. When cleared to 0, this bit disables the
01136 // * generation of an interrupt signal upon completion of a Slave 4 transaction.
01137 // * The interrupt status can be observed in Register 54.
01138 // *
01139 // * @return Current enabled value for Slave 4 transaction interrupts.
01140 // * @see MPU6050_RA_I2C_SLV4_CTRL
01141 // */
01142 //bool MPU6050::getSlave4InterruptEnabled() {
01143 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_SLV4_CTRL, MPU6050_I2C_SLV4_INT_EN_BIT, buffer);
01144 //    return buffer[0];
01145 //}
01147 // * @param enabled New enabled value for Slave 4 transaction interrupts.
01148 // * @see getSlave4InterruptEnabled()
01149 // * @see MPU6050_RA_I2C_SLV4_CTRL
01150 // */
01151 //void MPU6050::setSlave4InterruptEnabled(bool enabled) {
01152 //    I2Cdev::writeBit(devAddr, MPU6050_RA_I2C_SLV4_CTRL, MPU6050_I2C_SLV4_INT_EN_BIT, enabled);
01153 //}
01155 // * When set to 1, the transaction will read or write data only. When cleared to
01156 // * 0, the transaction will write a register address prior to reading or writing
01157 // * data. This should equal 0 when specifying the register address within the
01158 // * Slave device to/from which the ensuing data transaction will take place.
01159 // *
01160 // * @return Current write mode for Slave 4 (0 = register address + data, 1 = data only)
01161 // * @see MPU6050_RA_I2C_SLV4_CTRL
01162 // */
01163 //bool MPU6050::getSlave4WriteMode() {
01164 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_SLV4_CTRL, MPU6050_I2C_SLV4_REG_DIS_BIT, buffer);
01165 //    return buffer[0];
01166 //}
01168 // * @param mode New write mode for Slave 4 (0 = register address + data, 1 = data only)
01169 // * @see getSlave4WriteMode()
01170 // * @see MPU6050_RA_I2C_SLV4_CTRL
01171 // */
01172 //void MPU6050::setSlave4WriteMode(bool mode) {
01173 //    I2Cdev::writeBit(devAddr, MPU6050_RA_I2C_SLV4_CTRL, MPU6050_I2C_SLV4_REG_DIS_BIT, mode);
01174 //}
01176 // * This configures the reduced access rate of I2C slaves relative to the Sample
01177 // * Rate. When a slave's access rate is decreased relative to the Sample Rate,
01178 // * the slave is accessed every:
01179 // *
01180 // *     1 / (1 + I2C_MST_DLY) samples
01181 // *
01182 // * This base Sample Rate in turn is determined by SMPLRT_DIV (register 25) and
01183 // * DLPF_CFG (register 26). Whether a slave's access rate is reduced relative to
01184 // * the Sample Rate is determined by I2C_MST_DELAY_CTRL (register 103). For
01185 // * further information regarding the Sample Rate, please refer to register 25.
01186 // *
01187 // * @return Current Slave 4 master delay value
01188 // * @see MPU6050_RA_I2C_SLV4_CTRL
01189 // */
01190 //uint8_t MPU6050::getSlave4MasterDelay() {
01191 //    I2Cdev::readBits(devAddr, MPU6050_RA_I2C_SLV4_CTRL, MPU6050_I2C_SLV4_MST_DLY_BIT,
      MPU6050_I2C_SLV4_MST_DLY_LENGTH, buffer);
01192 //    return buffer[0];
01193 //}
01195 // * @param delay New Slave 4 master delay value
01196 // * @see getSlave4MasterDelay()
01197 // * @see MPU6050_RA_I2C_SLV4_CTRL
01198 // */
```

```
01199 //void MPU6050::setSlave4MasterDelay(uint8_t delay) {
01200 //    I2Cdev::writeBits(devAddr, MPU6050_RA_I2C_SLV4_CTRL, MPU6050_I2C_SLV4_MST_DLY_BIT,
      MPU6050_I2C_SLV4_MST_DLY_LENGTH, delay);
01201 //}
01203 // * This register stores the data read from Slave 4. This field is populated
01204 // * after a read transaction.
01205 // * @return Last available byte read from to Slave 4
01206 // * @see MPU6050_RA_I2C_SLV4_DI
01207 // */
01208 //uint8_t MPU6050::getSlate4InputByte() {
01209 //    I2Cdev::readByte(devAddr, MPU6050_RA_I2C_SLV4_DI, buffer);
01210 //    return buffer[0];
01211 //}
01212 //
01214 //
01216 // * This bit reflects the status of the FSYNC interrupt from an external device
01217 // * into the MPU-60X0. This is used as a way to pass an external interrupt
01218 // * through the MPU-60X0 to the host application processor. When set to 1, this
01219 // * bit will cause an interrupt if FSYNC_INT_EN is asserted in INT_PIN_CFG
01220 // * (Register 55).
01221 // * @return FSYNC interrupt status
01222 // * @see MPU6050_RA_I2C_MST_STATUS
01223 // */
01224 //bool MPU6050::getPassthroughStatus() {
01225 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_MST_STATUS, MPU6050_MST_PASS_THROUGH_BIT, buffer);
01226 //    return buffer[0];
01227 //}
01229 // * Automatically sets to 1 when a Slave 4 transaction has completed. This
01230 // * triggers an interrupt if the I2C_MST_INT_EN bit in the INT_ENABLE register
01231 // * (Register 56) is asserted and if the SLV_4_DONE_INT bit is asserted in the
01232 // * I2C_SLV4_CTRL register (Register 52).
01233 // * @return Slave 4 transaction done status
01234 // * @see MPU6050_RA_I2C_MST_STATUS
01235 // */
01236 //bool MPU6050::getSlave4IsDone() {
01237 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_MST_STATUS, MPU6050_MST_I2C_SLV4_DONE_BIT, buffer);
01238 //    return buffer[0];
01239 //}
01241 // * This bit automatically sets to 1 when the I2C Master has lost arbitration of
01242 // * the auxiliary I2C bus (an error condition). This triggers an interrupt if the
01243 // * I2C_MST_INT_EN bit in the INT_ENABLE register (Register 56) is asserted.
01244 // * @return Master arbitration lost status
01245 // * @see MPU6050_RA_I2C_MST_STATUS
01246 // */
01247 //bool MPU6050::getLostArbitration() {
01248 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_MST_STATUS, MPU6050_MST_I2C_LOST_ARB_BIT, buffer);
01249 //    return buffer[0];
01250 //}
01252 // * This bit automatically sets to 1 when the I2C Master receives a NACK in a
01253 // * transaction with Slave 4. This triggers an interrupt if the I2C_MST_INT_EN
01254 // * bit in the INT_ENABLE register (Register 56) is asserted.
01255 // * @return Slave 4 NACK interrupt status
01256 // * @see MPU6050_RA_I2C_MST_STATUS
01257 // */
01258 //bool MPU6050::getSlave4Nack() {
01259 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_MST_STATUS, MPU6050_MST_I2C_SLV4_NACK_BIT, buffer);
01260 //    return buffer[0];
01261 //}
01263 // * This bit automatically sets to 1 when the I2C Master receives a NACK in a
01264 // * transaction with Slave 3. This triggers an interrupt if the I2C_MST_INT_EN
01265 // * bit in the INT_ENABLE register (Register 56) is asserted.
01266 // * @return Slave 3 NACK interrupt status
01267 // * @see MPU6050_RA_I2C_MST_STATUS
01268 // */
01269 //bool MPU6050::getSlave3Nack() {
01270 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_MST_STATUS, MPU6050_MST_I2C_SLV3_NACK_BIT, buffer);
01271 //    return buffer[0];
01272 //}
01274 // * This bit automatically sets to 1 when the I2C Master receives a NACK in a
01275 // * transaction with Slave 2. This triggers an interrupt if the I2C_MST_INT_EN
01276 // * bit in the INT_ENABLE register (Register 56) is asserted.
01277 // * @return Slave 2 NACK interrupt status
01278 // * @see MPU6050_RA_I2C_MST_STATUS
01279 // */
01280 //bool MPU6050::getSlave2Nack() {
01281 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_MST_STATUS, MPU6050_MST_I2C_SLV2_NACK_BIT, buffer);
01282 //    return buffer[0];
01283 //}
01285 // * This bit automatically sets to 1 when the I2C Master receives a NACK in a
01286 // * transaction with Slave 1. This triggers an interrupt if the I2C_MST_INT_EN
01287 // * bit in the INT_ENABLE register (Register 56) is asserted.
01288 // * @return Slave 1 NACK interrupt status
01289 // * @see MPU6050_RA_I2C_MST_STATUS
01290 // */
01291 //bool MPU6050::getSlave1Nack() {
01292 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_MST_STATUS, MPU6050_MST_I2C_SLV1_NACK_BIT, buffer);
01293 //    return buffer[0];
```

```
01294 //}
01296 // * This bit automatically sets to 1 when the I2C Master receives a NACK in a
01297 // * transaction with Slave 0. This triggers an interrupt if the I2C_MST_INT_EN
01298 // * bit in the INT_ENABLE register (Register 56) is asserted.
01299 // * @return Slave 0 NACK interrupt status
01300 // * @see MPU6050_RA_I2C_MST_STATUS
01301 // */
01302 //bool MPU6050::getSlave0Nack() {
01303 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_MST_STATUS, MPU6050_MST_I2C_SLV0_NACK_BIT, buffer);
01304 //    return buffer[0];
01305 //}
01306 //
01308 //
01310 // * Will be set 0 for active-high, 1 for active-low.
01311 // * @return Current interrupt mode (0=active-high, 1=active-low)
01312 // * @see MPU6050_RA_INT_PIN_CFG
01313 // * @see MPU6050_INTCFG_INT_LEVEL_BIT
01314 // */
01315 //bool MPU6050::getInterruptMode() {
01316 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_PIN_CFG, MPU6050_INTCFG_INT_LEVEL_BIT, buffer);
01317 //    return buffer[0];
01318 //}
01320 // * @param mode New interrupt mode (0=active-high, 1=active-low)
01321 // * @see getInterruptMode()
01322 // * @see MPU6050_RA_INT_PIN_CFG
01323 // * @see MPU6050_INTCFG_INT_LEVEL_BIT
01324 // */
01325 //void MPU6050::setInterruptMode(bool mode) {
01326 //   I2Cdev::writeBit(devAddr, MPU6050_RA_INT_PIN_CFG, MPU6050_INTCFG_INT_LEVEL_BIT, mode);
01327 //}
01329 // * Will be set 0 for push-pull, 1 for open-drain.
01330 // * @return Current interrupt drive mode (0=push-pull, 1=open-drain)
01331 // * @see MPU6050_RA_INT_PIN_CFG
01332 // * @see MPU6050_INTCFG_INT_OPEN_BIT
01333 // */
01334 //bool MPU6050::getInterruptDrive() {
01335 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_PIN_CFG, MPU6050_INTCFG_INT_OPEN_BIT, buffer);
01336 //    return buffer[0];
01337 //}
01339 // * @param drive New interrupt drive mode (0=push-pull, 1=open-drain)
01340 // * @see getInterruptDrive()
01341 // * @see MPU6050_RA_INT_PIN_CFG
01342 // * @see MPU6050_INTCFG_INT_OPEN_BIT
01343 // */
01344 //void MPU6050::setInterruptDrive(bool drive) {
01345 //    I2Cdev::writeBit(devAddr, MPU6050_RA_INT_PIN_CFG, MPU6050_INTCFG_INT_OPEN_BIT, drive);
01346 //}
01348 // * Will be set 0 for 50us-pulse, 1 for latch-until-int-cleared.
01349 // * @return Current latch mode (0=50us-pulse, 1=latch-until-int-cleared)
01350 // * @see MPU6050_RA_INT_PIN_CFG
01351 // * @see MPU6050_INTCFG_LATCH_INT_EN_BIT
01352 // */
01353 //bool MPU6050::getInterruptLatch() {
01354 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_PIN_CFG, MPU6050_INTCFG_LATCH_INT_EN_BIT, buffer);
01355 //    return buffer[0];
01356 //}
01358 // * @param latch New latch mode (0=50us-pulse, 1=latch-until-int-cleared)
01359 // * @see getInterruptLatch()
01360 // * @see MPU6050_RA_INT_PIN_CFG
01361 // * @see MPU6050_INTCFG_LATCH_INT_EN_BIT
01362 // */
01363 //void MPU6050::setInterruptLatch(bool latch) {
01364 //    I2Cdev::writeBit(devAddr, MPU6050_RA_INT_PIN_CFG, MPU6050_INTCFG_LATCH_INT_EN_BIT, latch);
01365 //}
01367 // * Will be set 0 for status-read-only, 1 for any-register-read.
01368 // * @return Current latch clear mode (0=status-read-only, 1=any-register-read)
01369 // * @see MPU6050_RA_INT_PIN_CFG
01370 // * @see MPU6050_INTCFG_INT_RD_CLEAR_BIT
01371 // */
01372 //bool MPU6050::getInterruptLatchClear() {
01373 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_PIN_CFG, MPU6050_INTCFG_INT_RD_CLEAR_BIT, buffer);
01374 //    return buffer[0];
01375 //}
01377 // * @param clear New latch clear mode (0=status-read-only, 1=any-register-read)
01378 // * @see getInterruptLatchClear()
01379 // * @see MPU6050_RA_INT_PIN_CFG
01380 // * @see MPU6050_INTCFG_INT_RD_CLEAR_BIT
01381 // */
01382 //void MPU6050::setInterruptLatchClear(bool clear) {
01383 //    I2Cdev::writeBit(devAddr, MPU6050_RA_INT_PIN_CFG, MPU6050_INTCFG_INT_RD_CLEAR_BIT, clear);
01384 //}
01386 // * @return Current FSYNC interrupt mode (0=active-high, 1=active-low)
01387 // * @see getFSyncInterruptMode()
01388 // * @see MPU6050_RA_INT_PIN_CFG
01389 // * @see MPU6050_INTCFG_FSYNC_INT_LEVEL_BIT
01390 // */
01391 //bool MPU6050::getFSyncInterruptLevel() {
```

```
01392 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_PIN_CFG, MPU6050_INTCFG_FSYNC_INT_LEVEL_BIT, buffer);
01393 //    return buffer[0];
01394 //}
01396 // * @param mode New FSYNC interrupt mode (0=active-high, 1=active-low)
01397 // * @see getFSyncInterruptMode()
01398 // * @see MPU6050_RA_INT_PIN_CFG
01399 // * @see MPU6050_INTCFG_FSYNC_INT_LEVEL_BIT
01400 // */
01401 //void MPU6050::setFSyncInterruptLevel(bool level) {
01402 //    I2Cdev::writeBit(devAddr, MPU6050_RA_INT_PIN_CFG, MPU6050_INTCFG_FSYNC_INT_LEVEL_BIT, level);
01403 //}
01405 // * Will be set 0 for disabled, 1 for enabled.
01406 // * @return Current interrupt enabled setting
01407 // * @see MPU6050_RA_INT_PIN_CFG
01408 // * @see MPU6050_INTCFG_FSYNC_INT_EN_BIT
01409 // */
01410 //bool MPU6050::getFSyncInterruptEnabled() {
01411 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_PIN_CFG, MPU6050_INTCFG_FSYNC_INT_EN_BIT, buffer);
01412 //    return buffer[0];
01413 //}
01415 // * @param enabled New FSYNC pin interrupt enabled setting
01416 // * @see getFSyncInterruptEnabled()
01417 // * @see MPU6050_RA_INT_PIN_CFG
01418 // * @see MPU6050_INTCFG_FSYNC_INT_EN_BIT
01419 // */
01420 //void MPU6050::setFSyncInterruptEnabled(bool enabled) {
01421 //    I2Cdev::writeBit(devAddr, MPU6050_RA_INT_PIN_CFG, MPU6050_INTCFG_FSYNC_INT_EN_BIT, enabled);
01422 //}
01424 // * When this bit is equal to 1 and I2C_MST_EN (Register 106 bit[5]) is equal to
01425 // * 0, the host application processor will be able to directly access the
01426 // * auxiliary I2C bus of the MPU-60X0. When this bit is equal to 0, the host
01427 // * application processor will not be able to directly access the auxiliary I2C
01428 // * bus of the MPU-60X0 regardless of the state of I2C_MST_EN (Register 106
01429 // * bit[5]).
01430 // * @return Current I2C bypass enabled status
01431 // * @see MPU6050_RA_INT_PIN_CFG
01432 // * @see MPU6050_INTCFG_I2C_BYPASS_EN_BIT
01433 // */
01434 //bool MPU6050::getI2CBypassEnabled() {
01435 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_PIN_CFG, MPU6050_INTCFG_I2C_BYPASS_EN_BIT, buffer);
01436 //    return buffer[0];
01437 //}
01439 // * When this bit is equal to 1 and I2C_MST_EN (Register 106 bit[5]) is equal to
01440 // * 0, the host application processor will be able to directly access the
01441 // * auxiliary I2C bus of the MPU-60X0. When this bit is equal to 0, the host
01442 // * application processor will not be able to directly access the auxiliary I2C
01443 // * bus of the MPU-60X0 regardless of the state of I2C_MST_EN (Register 106
01444 // * bit[5]).
01445 // * @param enabled New I2C bypass enabled status
01446 // * @see MPU6050_RA_INT_PIN_CFG
01447 // * @see MPU6050_INTCFG_I2C_BYPASS_EN_BIT
01448 // */
01449 //void MPU6050::setI2CBypassEnabled(bool enabled) {
01450 //    I2Cdev::writeBit(devAddr, MPU6050_RA_INT_PIN_CFG, MPU6050_INTCFG_I2C_BYPASS_EN_BIT, enabled);
01451 //}
01453 // * When this bit is equal to 1, a reference clock output is provided at the
01454 // * CLKOUT pin. When this bit is equal to 0, the clock output is disabled. For
01455 // * further information regarding CLKOUT, please refer to the MPU-60X0 Product
01456 // * Specification document.
01457 // * @return Current reference clock output enabled status
01458 // * @see MPU6050_RA_INT_PIN_CFG
01459 // * @see MPU6050_INTCFG_CLKOUT_EN_BIT
01460 // */
01461 //bool MPU6050::getClockOutputEnabled() {
01462 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_PIN_CFG, MPU6050_INTCFG_CLKOUT_EN_BIT, buffer);
01463 //    return buffer[0];
01464 //}
01466 // * When this bit is equal to 1, a reference clock output is provided at the
01467 // * CLKOUT pin. When this bit is equal to 0, the clock output is disabled. For
01468 // * further information regarding CLKOUT, please refer to the MPU-60X0 Product
01469 // * Specification document.
01470 // * @param enabled New reference clock output enabled status
01471 // * @see MPU6050_RA_INT_PIN_CFG
01472 // * @see MPU6050_INTCFG_CLKOUT_EN_BIT
01473 // */
01474 //void MPU6050::setClockOutputEnabled(bool enabled) {
01475 //    I2Cdev::writeBit(devAddr, MPU6050_RA_INT_PIN_CFG, MPU6050_INTCFG_CLKOUT_EN_BIT, enabled);
01476 //}
01477 //
01479 //
01481 // * Full register byte for all interrupts, for quick reading. Each bit will be
01482 // * set 0 for disabled, 1 for enabled.
01483 // * @return Current interrupt enabled status
01484 // * @see MPU6050_RA_INT_ENABLE
01485 // * @see MPU6050_INTERRUPT_FF_BIT
01486 // **/
01487 //uint8_t MPU6050::getIntEnabled() {
```

```
01488 //    I2Cdev::readByte(devAddr, MPU6050_RA_INT_ENABLE, buffer);
01489 //    return buffer[0];
01490 //}
01492 // * Full register byte for all interrupts, for quick reading. Each bit should be
01493 // * set 0 for disabled, 1 for enabled.
01494 // * @param enabled New interrupt enabled status
01495 // * @see getIntFreefallEnabled()
01496 // * @see MPU6050_RA_INT_ENABLE
01497 // * @see MPU6050_INTERRUPT_FF_BIT
01498 // **/
01499 //void MPU6050::setIntEnabled(uint8_t enabled) {
01500 //    I2Cdev::writeByte(devAddr, MPU6050_RA_INT_ENABLE, enabled);
01501 //}
01503 // * Will be set 0 for disabled, 1 for enabled.
01504 // * @return Current interrupt enabled status
01505 // * @see MPU6050_RA_INT_ENABLE
01506 // * @see MPU6050_INTERRUPT_FF_BIT
01507 // **/
01508 //bool MPU6050::getIntFreefallEnabled() {
01509 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_ENABLE, MPU6050_INTERRUPT_FF_BIT, buffer);
01510 //    return buffer[0];
01511 //}
01513 // * @param enabled New interrupt enabled status
01514 // * @see getIntFreefallEnabled()
01515 // * @see MPU6050_RA_INT_ENABLE
01516 // * @see MPU6050_INTERRUPT_FF_BIT
01517 // **/
01518 //void MPU6050::setIntFreefallEnabled(bool enabled) {
01519 //    I2Cdev::writeBit(devAddr, MPU6050_RA_INT_ENABLE, MPU6050_INTERRUPT_FF_BIT, enabled);
01520 //}
01522 // * Will be set 0 for disabled, 1 for enabled.
01523 // * @return Current interrupt enabled status
01524 // * @see MPU6050_RA_INT_ENABLE
01525 // * @see MPU6050_INTERRUPT_MOT_BIT
01526 // **/
01527 //bool MPU6050::getIntMotionEnabled() {
01528 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_ENABLE, MPU6050_INTERRUPT_MOT_BIT, buffer);
01529 //    return buffer[0];
01530 //}
01532 // * @param enabled New interrupt enabled status
01533 // * @see getIntMotionEnabled()
01534 // * @see MPU6050_RA_INT_ENABLE
01535 // * @see MPU6050_INTERRUPT_MOT_BIT
01536 // **/
01537 //void MPU6050::setIntMotionEnabled(bool enabled) {
01538 //    I2Cdev::writeBit(devAddr, MPU6050_RA_INT_ENABLE, MPU6050_INTERRUPT_MOT_BIT, enabled);
01539 //}
01541 // * Will be set 0 for disabled, 1 for enabled.
01542 // * @return Current interrupt enabled status
01543 // * @see MPU6050_RA_INT_ENABLE
01544 // * @see MPU6050_INTERRUPT_ZMOT_BIT
01545 // **/
01546 //bool MPU6050::getIntZeroMotionEnabled() {
01547 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_ENABLE, MPU6050_INTERRUPT_ZMOT_BIT, buffer);
01548 //    return buffer[0];
01549 //}
01551 // * @param enabled New interrupt enabled status
01552 // * @see getIntZeroMotionEnabled()
01553 // * @see MPU6050_RA_INT_ENABLE
01554 // * @see MPU6050_INTERRUPT_ZMOT_BIT
01555 // **/
01556 //void MPU6050::setIntZeroMotionEnabled(bool enabled) {
01557 //    I2Cdev::writeBit(devAddr, MPU6050_RA_INT_ENABLE, MPU6050_INTERRUPT_ZMOT_BIT, enabled);
01558 //}
01560 // * Will be set 0 for disabled, 1 for enabled.
01561 // * @return Current interrupt enabled status
01562 // * @see MPU6050_RA_INT_ENABLE
01563 // * @see MPU6050_INTERRUPT_FIFO_OFLOW_BIT
01564 // **/
01565 //bool MPU6050::getIntFIFOBufferOverflowEnabled() {
01566 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_ENABLE, MPU6050_INTERRUPT_FIFO_OFLOW_BIT, buffer);
01567 //    return buffer[0];
01568 //}
01570 // * @param enabled New interrupt enabled status
01571 // * @see getIntFIFOBufferOverflowEnabled()
01572 // * @see MPU6050_RA_INT_ENABLE
01573 // * @see MPU6050_INTERRUPT_FIFO_OFLOW_BIT
01574 // **/
01575 //void MPU6050::setIntFIFOBufferOverflowEnabled(bool enabled) {
01576 //    I2Cdev::writeBit(devAddr, MPU6050_RA_INT_ENABLE, MPU6050_INTERRUPT_FIFO_OFLOW_BIT, enabled);
01577 //}
01579 // * This enables any of the I2C Master interrupt sources to generate an
01580 // * interrupt. Will be set 0 for disabled, 1 for enabled.
01581 // * @return Current interrupt enabled status
01582 // * @see MPU6050_RA_INT_ENABLE
01583 // * @see MPU6050_INTERRUPT_I2C_MST_INT_BIT
01584 // **/
```

```
01585 //bool MPU6050::getIntI2CMasterEnabled() {
01586 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_ENABLE, MPU6050_INTERRUPT_I2C_MST_INT_BIT, buffer);
01587 //    return buffer[0];
01588 //}
01590 // * @param enabled New interrupt enabled status
01591 // * @see getIntI2CMasterEnabled()
01592 // * @see MPU6050_RA_INT_ENABLE
01593 // * @see MPU6050_INTERRUPT_I2C_MST_INT_BIT
01594 // **/
01595 //void MPU6050::setIntI2CMasterEnabled(bool enabled) {
01596 //    I2Cdev::writeBit(devAddr, MPU6050_RA_INT_ENABLE, MPU6050_INTERRUPT_I2C_MST_INT_BIT, enabled);
01597 //}
01599 // * This event occurs each time a write operation to all of the sensor registers
01600 // * has been completed. Will be set 0 for disabled, 1 for enabled.
01601 // * @return Current interrupt enabled status
01602 // * @see MPU6050_RA_INT_ENABLE
01603 // * @see MPU6050_INTERRUPT_DATA_RDY_BIT
01604 // */
01605 //bool MPU6050::getIntDataReadyEnabled() {
01606 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_ENABLE, MPU6050_INTERRUPT_DATA_RDY_BIT, buffer);
01607 //    return buffer[0];
01608 //}
01610 // * @param enabled New interrupt enabled status
01611 // * @see getIntDataReadyEnabled()
01612 // * @see MPU6050_RA_INT_CFG
01613 // * @see MPU6050_INTERRUPT_DATA_RDY_BIT
01614 // */
01615 //void MPU6050::setIntDataReadyEnabled(bool enabled) {
01616 //    I2Cdev::writeBit(devAddr, MPU6050_RA_INT_ENABLE, MPU6050_INTERRUPT_DATA_RDY_BIT, enabled);
01617 //}
01618 //
01620 //
01622 // * These bits clear to 0 after the register has been read. Very useful
01623 // * for getting multiple INT statuses, since each single bit read clears
01624 // * all of them because it has to read the whole byte.
01625 // * @return Current interrupt status
01626 // * @see MPU6050_RA_INT_STATUS
01627 // */
01628 //uint8_t MPU6050::getIntStatus() {
01629 //    I2Cdev::readByte(devAddr, MPU6050_RA_INT_STATUS, buffer);
01630 //    return buffer[0];
01631 //}
01633 // * This bit automatically sets to 1 when a Free Fall interrupt has been
01634 // * generated. The bit clears to 0 after the register has been read.
01635 // * @return Current interrupt status
01636 // * @see MPU6050_RA_INT_STATUS
01637 // * @see MPU6050_INTERRUPT_FF_BIT
01638 // */
01639 //bool MPU6050::getIntFreefallStatus() {
01640 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_STATUS, MPU6050_INTERRUPT_FF_BIT, buffer);
01641 //    return buffer[0];
01642 //}
01644 // * This bit automatically sets to 1 when a Motion Detection interrupt has been
01645 // * generated. The bit clears to 0 after the register has been read.
01646 // * @return Current interrupt status
01647 // * @see MPU6050_RA_INT_STATUS
01648 // * @see MPU6050_INTERRUPT_MOT_BIT
01649 // */
01650 //bool MPU6050::getIntMotionStatus() {
01651 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_STATUS, MPU6050_INTERRUPT_MOT_BIT, buffer);
01652 //    return buffer[0];
01653 //}
01655 // * This bit automatically sets to 1 when a Zero Motion Detection interrupt has
01656 // * been generated. The bit clears to 0 after the register has been read.
01657 // * @return Current interrupt status
01658 // * @see MPU6050_RA_INT_STATUS
01659 // * @see MPU6050_INTERRUPT_ZMOT_BIT
01660 // */
01661 //bool MPU6050::getIntZeroMotionStatus() {
01662 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_STATUS, MPU6050_INTERRUPT_ZMOT_BIT, buffer);
01663 //    return buffer[0];
01664 //}
01666 // * This bit automatically sets to 1 when a Free Fall interrupt has been
01667 // * generated. The bit clears to 0 after the register has been read.
01668 // * @return Current interrupt status
01669 // * @see MPU6050_RA_INT_STATUS
01670 // * @see MPU6050_INTERRUPT_FIFO_OFLOW_BIT
01671 // */
01672 //bool MPU6050::getIntFIFOBufferOverflowStatus() {
01673 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_STATUS, MPU6050_INTERRUPT_FIFO_OFLOW_BIT, buffer);
01674 //    return buffer[0];
01675 //}
01677 // * This bit automatically sets to 1 when an I2C Master interrupt has been
01678 // * generated. For a list of I2C Master interrupts, please refer to Register 54.
01679 // * The bit clears to 0 after the register has been read.
01680 // * @return Current interrupt status
01681 // * @see MPU6050_RA_INT_STATUS
```

```
01682 // * @see MPU6050_INTERRUPT_I2C_MST_INT_BIT
01683 // */
01684 //bool MPU6050::getIntI2CMasterStatus() {
01685 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_STATUS, MPU6050_INTERRUPT_I2C_MST_INT_BIT, buffer);
01686 //    return buffer[0];
01687 //}
01689 // * This bit automatically sets to 1 when a Data Ready interrupt has been
01690 // * generated. The bit clears to 0 after the register has been read.
01691 // * @return Current interrupt status
01692 // * @see MPU6050_RA_INT_STATUS
01693 // * @see MPU6050_INTERRUPT_DATA_RDY_BIT
01694 // */
01695 //bool MPU6050::getIntDataReadyStatus() {
01696 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_STATUS, MPU6050_INTERRUPT_DATA_RDY_BIT, buffer);
01697 //    return buffer[0];
01698 //}
01699 //
01701 //
01703 // * FUNCTION NOT FULLY IMPLEMENTED YET.
01704 // * @param ax 16-bit signed integer container for accelerometer X-axis value
01705 // * @param ay 16-bit signed integer container for accelerometer Y-axis value
01706 // * @param az 16-bit signed integer container for accelerometer Z-axis value
01707 // * @param gx 16-bit signed integer container for gyroscope X-axis value
01708 // * @param gy 16-bit signed integer container for gyroscope Y-axis value
01709 // * @param gz 16-bit signed integer container for gyroscope Z-axis value
01710 // * @param mx 16-bit signed integer container for magnetometer X-axis value
01711 // * @param my 16-bit signed integer container for magnetometer Y-axis value
01712 // * @param mz 16-bit signed integer container for magnetometer Z-axis value
01713 // * @see getMotion6()
01714 // * @see getAcceleration()
01715 // * @see getRotation()
01716 // * @see MPU6050_RA_ACCEL_XOUT_H
01717 // */
01718 //void MPU6050::getMotion9(int16_t* ax, int16_t* ay, int16_t* az, int16_t* gx, int16_t* gy, int16_t* gz,
       int16_t* mx, int16_t* my, int16_t* mz) {
01719 //    getMotion6(ax, ay, az, gx, gy, gz);
01720 //    // TODO: magnetometer integration
01721 //}
01723 // * Retrieves all currently available motion sensor values.
01724 // * @param ax 16-bit signed integer container for accelerometer X-axis value
01725 // * @param ay 16-bit signed integer container for accelerometer Y-axis value
01726 // * @param az 16-bit signed integer container for accelerometer Z-axis value
01727 // * @param gx 16-bit signed integer container for gyroscope X-axis value
01728 // * @param gy 16-bit signed integer container for gyroscope Y-axis value
01729 // * @param gz 16-bit signed integer container for gyroscope Z-axis value
01730 // * @see getAcceleration()
01731 // * @see getRotation()
01732 // * @see MPU6050_RA_ACCEL_XOUT_H
01733 // */
01734 //void MPU6050::getMotion6(int16_t* ax, int16_t* ay, int16_t* az, int16_t* gx, int16_t* gy, int16_t* gz) {
01735 //    I2Cdev::readBytes(devAddr, MPU6050_RA_ACCEL_XOUT_H, 14, buffer);
01736 //    *ax = (((int16_t)buffer[0]) << 8) | buffer[1];
01737 //    *ay = (((int16_t)buffer[2]) << 8) | buffer[3];
01738 //    *az = (((int16_t)buffer[4]) << 8) | buffer[5];
01739 //    *gx = (((int16_t)buffer[8]) << 8) | buffer[9];
01740 //    *gy = (((int16_t)buffer[10]) << 8) | buffer[11];
01741 //    *gz = (((int16_t)buffer[12]) << 8) | buffer[13];
01742 //}
01744 // * These registers store the most recent accelerometer measurements.
01745 // * Accelerometer measurements are written to these registers at the Sample Rate
01746 // * as defined in Register 25.
01747 // *
01748 // * The accelerometer measurement registers, along with the temperature
01749 // * measurement registers, gyroscope measurement registers, and external sensor
01750 // * data registers, are composed of two sets of registers: an internal register
01751 // * set and a user-facing read register set.
01752 // *
01753 // * The data within the accelerometer sensors' internal register set is always
01754 // * updated at the Sample Rate. Meanwhile, the user-facing read register set
01755 // * duplicates the internal register set's data values whenever the serial
01756 // * interface is idle. This guarantees that a burst read of sensor registers will
01757 // * read measurements from the same sampling instant. Note that if burst reads
01758 // * are not used, the user is responsible for ensuring a set of single byte reads
01759 // * correspond to a single sampling instant by checking the Data Ready interrupt.
01760 // *
01761 // * Each 16-bit accelerometer measurement has a full scale defined in ACCEL_FS
01762 // * (Register 28). For each full scale setting, the accelerometers' sensitivity
01763 // * per LSB in ACCEL_xOUT is shown in the table below:
01764 // *
01765 // * <pre>
01766 // * AFS_SEL | Full Scale Range | LSB Sensitivity
01767 // * --------+------------------+----------------
01768 // * 0       | +/- 2g           | 8192 LSB/mg
01769 // * 1       | +/- 4g           | 4096 LSB/mg
01770 // * 2       | +/- 8g           | 2048 LSB/mg
01771 // * 3       | +/- 16g          | 1024 LSB/mg
01772 // * </pre>
```

```
01773 // *
01774 // * @param x 16-bit signed integer container for X-axis acceleration
01775 // * @param y 16-bit signed integer container for Y-axis acceleration
01776 // * @param z 16-bit signed integer container for Z-axis acceleration
01777 // * @see MPU6050_RA_GYRO_XOUT_H
01778 // */
01779 //void MPU6050::getAcceleration(int16_t* x, int16_t* y, int16_t* z) {
01780 //    I2Cdev::readBytes(devAddr, MPU6050_RA_ACCEL_XOUT_H, 6, buffer);
01781 //    *x = (((int16_t)buffer[0]) << 8) | buffer[1];
01782 //    *y = (((int16_t)buffer[2]) << 8) | buffer[3];
01783 //    *z = (((int16_t)buffer[4]) << 8) | buffer[5];
01784 //}
01786 // * @return X-axis acceleration measurement in 16-bit 2's complement format
01787 // * @see getMotion6()
01788 // * @see MPU6050_RA_ACCEL_XOUT_H
01789 // */
01790 //int16_t MPU6050::getAccelerationX() {
01791 //    I2Cdev::readBytes(devAddr, MPU6050_RA_ACCEL_XOUT_H, 2, buffer);
01792 //    return (((int16_t)buffer[0]) << 8) | buffer[1];
01793 //}
01795 // * @return Y-axis acceleration measurement in 16-bit 2's complement format
01796 // * @see getMotion6()
01797 // * @see MPU6050_RA_ACCEL_YOUT_H
01798 // */
01799 //int16_t MPU6050::getAccelerationY() {
01800 //    I2Cdev::readBytes(devAddr, MPU6050_RA_ACCEL_YOUT_H, 2, buffer);
01801 //    return (((int16_t)buffer[0]) << 8) | buffer[1];
01802 //}
01804 // * @return Z-axis acceleration measurement in 16-bit 2's complement format
01805 // * @see getMotion6()
01806 // * @see MPU6050_RA_ACCEL_ZOUT_H
01807 // */
01808 //int16_t MPU6050::getAccelerationZ() {
01809 //    I2Cdev::readBytes(devAddr, MPU6050_RA_ACCEL_ZOUT_H, 2, buffer);
01810 //    return (((int16_t)buffer[0]) << 8) | buffer[1];
01811 //}
01812 //
01814 //
01816 // * @return Temperature reading in 16-bit 2's complement format
01817 // * @see MPU6050_RA_TEMP_OUT_H
01818 // */
01819 //int16_t MPU6050::getTemperature() {
01820 //    I2Cdev::readBytes(devAddr, MPU6050_RA_TEMP_OUT_H, 2, buffer);
01821 //    return (((int16_t)buffer[0]) << 8) | buffer[1];
01822 //}
01823 //
01825 //
01827 // * These gyroscope measurement registers, along with the accelerometer
01828 // * measurement registers, temperature measurement registers, and external sensor
01829 // * data registers, are composed of two sets of registers: an internal register
01830 // * set and a user-facing read register set.
01831 // * The data within the gyroscope sensors' internal register set is always
01832 // * updated at the Sample Rate. Meanwhile, the user-facing read register set
01833 // * duplicates the internal register set's data values whenever the serial
01834 // * interface is idle. This guarantees that a burst read of sensor registers will
01835 // * read measurements from the same sampling instant. Note that if burst reads
01836 // * are not used, the user is responsible for ensuring a set of single byte reads
01837 // * correspond to a single sampling instant by checking the Data Ready interrupt.
01838 // *
01839 // * Each 16-bit gyroscope measurement has a full scale defined in FS_SEL
01840 // * (Register 27). For each full scale setting, the gyroscopes' sensitivity per
01841 // * LSB in GYRO_xOUT is shown in the table below:
01842 // *
01843 // * <pre>
01844 // * FS_SEL | Full Scale Range   | LSB Sensitivity
01845 // * -------+--------------------+----------------
01846 // * 0      | +/- 250 degrees/s  | 131 LSB/deg/s
01847 // * 1      | +/- 500 degrees/s  | 65.5 LSB/deg/s
01848 // * 2      | +/- 1000 degrees/s | 32.8 LSB/deg/s
01849 // * 3      | +/- 2000 degrees/s | 16.4 LSB/deg/s
01850 // * </pre>
01851 // *
01852 // * @param x 16-bit signed integer container for X-axis rotation
01853 // * @param y 16-bit signed integer container for Y-axis rotation
01854 // * @param z 16-bit signed integer container for Z-axis rotation
01855 // * @see getMotion6()
01856 // * @see MPU6050_RA_GYRO_XOUT_H
01857 // */
01858 //void MPU6050::getRotation(int16_t* x, int16_t* y, int16_t* z) {
01859 //    I2Cdev::readBytes(devAddr, MPU6050_RA_GYRO_XOUT_H, 6, buffer);
01860 //    *x = (((int16_t)buffer[0]) << 8) | buffer[1];
01861 //    *y = (((int16_t)buffer[2]) << 8) | buffer[3];
01862 //    *z = (((int16_t)buffer[4]) << 8) | buffer[5];
01863 //}
01864 //
01865 //void MPU6050::getRotationXY(int16_t* x, int16_t* y) {
01866 //    I2Cdev::readBytes(devAddr, MPU6050_RA_GYRO_XOUT_H, 4, buffer);
```

```
01867 //    *x = (((int16_t)buffer[0]) << 8) | buffer[1];
01868 //    *y = (((int16_t)buffer[2]) << 8) | buffer[3];
01869 //}
01870 //
01872 // * @return X-axis rotation measurement in 16-bit 2's complement format
01873 // * @see getMotion6()
01874 // * @see MPU6050_RA_GYRO_XOUT_H
01875 // */
01876 //int16_t MPU6050::getRotationX() {
01877 //    I2Cdev::readBytes(devAddr, MPU6050_RA_GYRO_XOUT_H, 2, buffer);
01878 //    return (((int16_t)buffer[0]) << 8) | buffer[1];
01879 //}
01881 // * @return Y-axis rotation measurement in 16-bit 2's complement format
01882 // * @see getMotion6()
01883 // * @see MPU6050_RA_GYRO_YOUT_H
01884 // */
01885 //int16_t MPU6050::getRotationY() {
01886 //    I2Cdev::readBytes(devAddr, MPU6050_RA_GYRO_YOUT_H, 2, buffer);
01887 //    return (((int16_t)buffer[0]) << 8) | buffer[1];
01888 //}
01890 // * @return Z-axis rotation measurement in 16-bit 2's complement format
01891 // * @see getMotion6()
01892 // * @see MPU6050_RA_GYRO_ZOUT_H
01893 // */
01894 //int16_t MPU6050::getRotationZ() {
01895 //    I2Cdev::readBytes(devAddr, MPU6050_RA_GYRO_ZOUT_H, 2, buffer);
01896 //    return (((int16_t)buffer[0]) << 8) | buffer[1];
01897 //}
01898 //
01900 //
01902 // * These registers store data read from external sensors by the Slave 0, 1, 2,
01903 // * and 3 on the auxiliary I2C interface. Data read by Slave 4 is stored in
01904 // * I2C_SLV4_DI (Register 53).
01905 // *
01906 // * External sensor data is written to these registers at the Sample Rate as
01907 // * defined in Register 25. This access rate can be reduced by using the Slave
01908 // * Delay Enable registers (Register 103).
01909 // *
01910 // * External sensor data registers, along with the gyroscope measurement
01911 // * registers, accelerometer measurement registers, and temperature measurement
01912 // * registers, are composed of two sets of registers: an internal register set
01913 // * and a user-facing read register set.
01914 // *
01915 // * The data within the external sensors' internal register set is always updated
01916 // * at the Sample Rate (or the reduced access rate) whenever the serial interface
01917 // * is idle. This guarantees that a burst read of sensor registers will read
01918 // * measurements from the same sampling instant. Note that if burst reads are not
01919 // * used, the user is responsible for ensuring a set of single byte reads
01920 // * correspond to a single sampling instant by checking the Data Ready interrupt.
01921 // *
01922 // * Data is placed in these external sensor data registers according to
01923 // * I2C_SLV0_CTRL, I2C_SLV1_CTRL, I2C_SLV2_CTRL, and I2C_SLV3_CTRL (Registers 39,
01924 // * 42, 45, and 48). When more than zero bytes are read (I2C_SLVx_LEN > 0) from
01925 // * an enabled slave (I2C_SLVx_EN = 1), the slave is read at the Sample Rate (as
01926 // * defined in Register 25) or delayed rate (if specified in Register 52 and
01927 // * 103). During each Sample cycle, slave reads are performed in order of Slave
01928 // * number. If all slaves are enabled with more than zero bytes to be read, the
01929 // * order will be Slave 0, followed by Slave 1, Slave 2, and Slave 3.
01930 // *
01931 // * Each enabled slave will have EXT_SENS_DATA registers associated with it by
01932 // * number of bytes read (I2C_SLVx_LEN) in order of slave number, starting from
01933 // * EXT_SENS_DATA_00. Note that this means enabling or disabling a slave may
01934 // * change the higher numbered slaves' associated registers. Furthermore, if
01935 // * fewer total bytes are being read from the external sensors as a result of
01936 // * such a change, then the data remaining in the registers which no longer have
01937 // * an associated slave device (i.e. high numbered registers) will remain in
01938 // * these previously allocated registers unless reset.
01939 // *
01940 // * If the sum of the read lengths of all SLVx transactions exceed the number of
01941 // * available EXT_SENS_DATA registers, the excess bytes will be dropped. There
01942 // * are 24 EXT_SENS_DATA registers and hence the total read lengths between all
01943 // * the slaves cannot be greater than 24 or some bytes will be lost.
01944 // *
01945 // * Note: Slave 4's behavior is distinct from that of Slaves 0-3. For further
01946 // * information regarding the characteristics of Slave 4, please refer to
01947 // * Registers 49 to 53.
01948 // *
01949 // * EXAMPLE:
01950 // * Suppose that Slave 0 is enabled with 4 bytes to be read (I2C_SLV0_EN = 1 and
01951 // * I2C_SLV0_LEN = 4) while Slave 1 is enabled with 2 bytes to be read so that
01952 // * I2C_SLV1_EN = 1 and I2C_SLV1_LEN = 2. In such a situation, EXT_SENS_DATA _00
01953 // * through _03 will be associated with Slave 0, while EXT_SENS_DATA _04 and 05
01954 // * will be associated with Slave 1. If Slave 2 is enabled as well, registers
01955 // * starting from EXT_SENS_DATA_06 will be allocated to Slave 2.
01956 // *
01957 // * If Slave 2 is disabled while Slave 3 is enabled in this same situation, then
01958 // * registers starting from EXT_SENS_DATA_06 will be allocated to Slave 3
```

```
01959 // * instead.
01960 // *
01961 // * REGISTER ALLOCATION FOR DYNAMIC DISABLE VS. NORMAL DISABLE:
01962 // * If a slave is disabled at any time, the space initially allocated to the
01963 // * slave in the EXT_SENS_DATA register, will remain associated with that slave.
01964 // * This is to avoid dynamic adjustment of the register allocation.
01965 // *
01966 // * The allocation of the EXT_SENS_DATA registers is recomputed only when (1) all
01967 // * slaves are disabled, or (2) the I2C_MST_RST bit is set (Register 106).
01968 // *
01969 // * This above is also true if one of the slaves gets NACKed and stops
01970 // * functioning.
01971 // *
01972 // * @param position Starting position (0-23)
01973 // * @return Byte read from register
01974 // */
01975 //uint8_t MPU6050::getExternalSensorByte(int position) {
01976 //    I2Cdev::readByte(devAddr, MPU6050_RA_EXT_SENS_DATA_00 + position, buffer);
01977 //    return buffer[0];
01978 //}
01980 // * @param position Starting position (0-21)
01981 // * @return Word read from register
01982 // * @see getExternalSensorByte()
01983 // */
01984 //uint16_t MPU6050::getExternalSensorWord(int position) {
01985 //    I2Cdev::readBytes(devAddr, MPU6050_RA_EXT_SENS_DATA_00 + position, 2, buffer);
01986 //    return (((uint16_t)buffer[0]) << 8) | buffer[1];
01987 //}
01989 // * @param position Starting position (0-20)
01990 // * @return Double word read from registers
01991 // * @see getExternalSensorByte()
01992 // */
01993 //uint32_t MPU6050::getExternalSensorDWord(int position) {
01994 //    I2Cdev::readBytes(devAddr, MPU6050_RA_EXT_SENS_DATA_00 + position, 4, buffer);
01995 //    return (((uint32_t)buffer[0]) << 24) | (((uint32_t)buffer[1]) << 16) | (((uint16_t)buffer[2]) << 8) |
      buffer[3];
01996 //}
01997 //
01999 //
02001 // * @return Motion detection status
02002 // * @see MPU6050_RA_MOT_DETECT_STATUS
02003 // * @see MPU6050_MOTION_MOT_XNEG_BIT
02004 // */
02005 //bool MPU6050::getXNegMotionDetected() {
02006 //    I2Cdev::readBit(devAddr, MPU6050_RA_MOT_DETECT_STATUS, MPU6050_MOTION_MOT_XNEG_BIT, buffer);
02007 //    return buffer[0];
02008 //}
02010 // * @return Motion detection status
02011 // * @see MPU6050_RA_MOT_DETECT_STATUS
02012 // * @see MPU6050_MOTION_MOT_XPOS_BIT
02013 // */
02014 //bool MPU6050::getXPosMotionDetected() {
02015 //    I2Cdev::readBit(devAddr, MPU6050_RA_MOT_DETECT_STATUS, MPU6050_MOTION_MOT_XPOS_BIT, buffer);
02016 //    return buffer[0];
02017 //}
02019 // * @return Motion detection status
02020 // * @see MPU6050_RA_MOT_DETECT_STATUS
02021 // * @see MPU6050_MOTION_MOT_YNEG_BIT
02022 // */
02023 //bool MPU6050::getYNegMotionDetected() {
02024 //    I2Cdev::readBit(devAddr, MPU6050_RA_MOT_DETECT_STATUS, MPU6050_MOTION_MOT_YNEG_BIT, buffer);
02025 //    return buffer[0];
02026 //}
02028 // * @return Motion detection status
02029 // * @see MPU6050_RA_MOT_DETECT_STATUS
02030 // * @see MPU6050_MOTION_MOT_YPOS_BIT
02031 // */
02032 //bool MPU6050::getYPosMotionDetected() {
02033 //    I2Cdev::readBit(devAddr, MPU6050_RA_MOT_DETECT_STATUS, MPU6050_MOTION_MOT_YPOS_BIT, buffer);
02034 //    return buffer[0];
02035 //}
02037 // * @return Motion detection status
02038 // * @see MPU6050_RA_MOT_DETECT_STATUS
02039 // * @see MPU6050_MOTION_MOT_ZNEG_BIT
02040 // */
02041 //bool MPU6050::getZNegMotionDetected() {
02042 //    I2Cdev::readBit(devAddr, MPU6050_RA_MOT_DETECT_STATUS, MPU6050_MOTION_MOT_ZNEG_BIT, buffer);
02043 //    return buffer[0];
02044 //}
02046 // * @return Motion detection status
02047 // * @see MPU6050_RA_MOT_DETECT_STATUS
02048 // * @see MPU6050_MOTION_MOT_ZPOS_BIT
02049 // */
02050 //bool MPU6050::getZPosMotionDetected() {
02051 //    I2Cdev::readBit(devAddr, MPU6050_RA_MOT_DETECT_STATUS, MPU6050_MOTION_MOT_ZPOS_BIT, buffer);
02052 //    return buffer[0];
02053 //}
```

```
02055 // * @return Motion detection status
02056 // * @see MPU6050_RA_MOT_DETECT_STATUS
02057 // * @see MPU6050_MOTION_MOT_ZRMOT_BIT
02058 // */
02059 //bool MPU6050::getZeroMotionDetected() {
02060 //    I2Cdev::readBit(devAddr, MPU6050_RA_MOT_DETECT_STATUS, MPU6050_MOTION_MOT_ZRMOT_BIT, buffer);
02061 //    return buffer[0];
02062 //}
02063 //
02065 //
02067 // * This register holds the output data written into Slave when Slave is set to
02068 // * write mode. For further information regarding Slave control, please
02069 // * refer to Registers 37 to 39 and immediately following.
02070 // * @param num Slave number (0-3)
02071 // * @param data Byte to write
02072 // * @see MPU6050_RA_I2C_SLV0_DO
02073 // */
02074 //void MPU6050::setSlaveOutputByte(uint8_t num, uint8_t data) {
02075 //    if (num > 3) return;
02076 //    I2Cdev::writeByte(devAddr, MPU6050_RA_I2C_SLV0_DO + num, data);
02077 //}
02078 //
02080 //
02082 // * This register is used to specify the timing of external sensor data
02083 // * shadowing. When DELAY_ES_SHADOW is set to 1, shadowing of external
02084 // * sensor data is delayed until all data has been received.
02085 // * @return Current external data shadow delay enabled status.
02086 // * @see MPU6050_RA_I2C_MST_DELAY_CTRL
02087 // * @see MPU6050_DELAYCTRL_DELAY_ES_SHADOW_BIT
02088 // */
02089 //bool MPU6050::getExternalShadowDelayEnabled() {
02090 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_MST_DELAY_CTRL, MPU6050_DELAYCTRL_DELAY_ES_SHADOW_BIT,
      buffer);
02091 //    return buffer[0];
02092 //}
02094 // * @param enabled New external data shadow delay enabled status.
02095 // * @see getExternalShadowDelayEnabled()
02096 // * @see MPU6050_RA_I2C_MST_DELAY_CTRL
02097 // * @see MPU6050_DELAYCTRL_DELAY_ES_SHADOW_BIT
02098 // */
02099 //void MPU6050::setExternalShadowDelayEnabled(bool enabled) {
02100 //    I2Cdev::writeBit(devAddr, MPU6050_RA_I2C_MST_DELAY_CTRL, MPU6050_DELAYCTRL_DELAY_ES_SHADOW_BIT,
      enabled);
02101 //}
02103 // * When a particular slave delay is enabled, the rate of access for the that
02104 // * slave device is reduced. When a slave's access rate is decreased relative to
02105 // * the Sample Rate, the slave is accessed every:
02106 // *
02107 // *     1 / (1 + I2C_MST_DLY) Samples
02108 // *
02109 // * This base Sample Rate in turn is determined by SMPLRT_DIV (register  * 25)
02110 // * and DLPF_CFG (register 26).
02111 // *
02112 // * For further information regarding I2C_MST_DLY, please refer to register 52.
02113 // * For further information regarding the Sample Rate, please refer to register 25.
02114 // *
02115 // * @param num Slave number (0-4)
02116 // * @return Current slave delay enabled status.
02117 // * @see MPU6050_RA_I2C_MST_DELAY_CTRL
02118 // * @see MPU6050_DELAYCTRL_I2C_SLV0_DLY_EN_BIT
02119 // */
02120 //bool MPU6050::getSlaveDelayEnabled(uint8_t num) {
02121 //    // MPU6050_DELAYCTRL_I2C_SLV4_DLY_EN_BIT is 4, SLV3 is 3, etc.
02122 //    if (num > 4) return 0;
02123 //    I2Cdev::readBit(devAddr, MPU6050_RA_I2C_MST_DELAY_CTRL, num, buffer);
02124 //    return buffer[0];
02125 //}
02127 // * @param num Slave number (0-4)
02128 // * @param enabled New slave delay enabled status.
02129 // * @see MPU6050_RA_I2C_MST_DELAY_CTRL
02130 // * @see MPU6050_DELAYCTRL_I2C_SLV0_DLY_EN_BIT
02131 // */
02132 //void MPU6050::setSlaveDelayEnabled(uint8_t num, bool enabled) {
02133 //    I2Cdev::writeBit(devAddr, MPU6050_RA_I2C_MST_DELAY_CTRL, num, enabled);
02134 //}
02135 //
02137 //
02139 // * The reset will revert the signal path analog to digital converters and
02140 // * filters to their power up configurations.
02141 // * @see MPU6050_RA_SIGNAL_PATH_RESET
02142 // * @see MPU6050_PATHRESET_GYRO_RESET_BIT
02143 // */
02144 //void MPU6050::resetGyroscopePath() {
02145 //    I2Cdev::writeBit(devAddr, MPU6050_RA_SIGNAL_PATH_RESET, MPU6050_PATHRESET_GYRO_RESET_BIT, true);
02146 //}
02148 // * The reset will revert the signal path analog to digital converters and
02149 // * filters to their power up configurations.
```

```
02150 // * @see MPU6050_RA_SIGNAL_PATH_RESET
02151 // * @see MPU6050_PATHRESET_ACCEL_RESET_BIT
02152 // */
02153 //void MPU6050::resetAccelerometerPath() {
02154 //    I2Cdev::writeBit(devAddr, MPU6050_RA_SIGNAL_PATH_RESET, MPU6050_PATHRESET_ACCEL_RESET_BIT, true);
02155 //}
02157 // * The reset will revert the signal path analog to digital converters and
02158 // * filters to their power up configurations.
02159 // * @see MPU6050_RA_SIGNAL_PATH_RESET
02160 // * @see MPU6050_PATHRESET_TEMP_RESET_BIT
02161 // */
02162 //void MPU6050::resetTemperaturePath() {
02163 //    I2Cdev::writeBit(devAddr, MPU6050_RA_SIGNAL_PATH_RESET, MPU6050_PATHRESET_TEMP_RESET_BIT, true);
02164 //}
02165 //
02167 //
02169 // * The accelerometer data path provides samples to the sensor registers, Motion
02170 // * detection, Zero Motion detection, and Free Fall detection modules. The
02171 // * signal path contains filters which must be flushed on wake-up with new
02172 // * samples before the detection modules begin operations. The default wake-up
02173 // * delay, of 4ms can be lengthened by up to 3ms. This additional delay is
02174 // * specified in ACCEL_ON_DELAY in units of 1 LSB = 1 ms. The user may select
02175 // * any value above zero unless instructed otherwise by InvenSense. Please refer
02176 // * to Section 8 of the MPU-6000/MPU-6050 Product Specification document for
02177 // * further information regarding the detection modules.
02178 // * @return Current accelerometer power-on delay
02179 // * @see MPU6050_RA_MOT_DETECT_CTRL
02180 // * @see MPU6050_DETECT_ACCEL_ON_DELAY_BIT
02181 // */
02182 //uint8_t MPU6050::getAccelerometerPowerOnDelay() {
02183 //    I2Cdev::readBits(devAddr, MPU6050_RA_MOT_DETECT_CTRL, MPU6050_DETECT_ACCEL_ON_DELAY_BIT,
       MPU6050_DETECT_ACCEL_ON_DELAY_LENGTH, buffer);
02184 //    return buffer[0];
02185 //}
02187 // * @param delay New accelerometer power-on delay (0-3)
02188 // * @see getAccelerometerPowerOnDelay()
02189 // * @see MPU6050_RA_MOT_DETECT_CTRL
02190 // * @see MPU6050_DETECT_ACCEL_ON_DELAY_BIT
02191 // */
02192 //void MPU6050::setAccelerometerPowerOnDelay(uint8_t delay) {
02193 //    I2Cdev::writeBits(devAddr, MPU6050_RA_MOT_DETECT_CTRL, MPU6050_DETECT_ACCEL_ON_DELAY_BIT,
       MPU6050_DETECT_ACCEL_ON_DELAY_LENGTH, delay);
02194 //}
02196 // * Detection is registered by the Free Fall detection module after accelerometer
02197 // * measurements meet their respective threshold conditions over a specified
02198 // * number of samples. When the threshold conditions are met, the corresponding
02199 // * detection counter increments by 1. The user may control the rate at which the
02200 // * detection counter decrements when the threshold condition is not met by
02201 // * configuring FF_COUNT. The decrement rate can be set according to the
02202 // * following table:
02203 // *
02204 // * <pre>
02205 // * FF_COUNT | Counter Decrement
02206 // * ---------+------------------
02207 // * 0        | Reset
02208 // * 1        | 1
02209 // * 2        | 2
02210 // * 3        | 4
02211 // * </pre>
02212 // *
02213 // * When FF_COUNT is configured to 0 (reset), any non-qualifying sample will
02214 // * reset the counter to 0. For further information on Free Fall detection,
02215 // * please refer to Registers 29 to 32.
02216 // *
02217 // * @return Current decrement configuration
02218 // * @see MPU6050_RA_MOT_DETECT_CTRL
02219 // * @see MPU6050_DETECT_FF_COUNT_BIT
02220 // */
02221 //uint8_t MPU6050::getFreefallDetectionCounterDecrement() {
02222 //    I2Cdev::readBits(devAddr, MPU6050_RA_MOT_DETECT_CTRL, MPU6050_DETECT_FF_COUNT_BIT,
       MPU6050_DETECT_FF_COUNT_LENGTH, buffer);
02223 //    return buffer[0];
02224 //}
02226 // * @param decrement New decrement configuration value
02227 // * @see getFreefallDetectionCounterDecrement()
02228 // * @see MPU6050_RA_MOT_DETECT_CTRL
02229 // * @see MPU6050_DETECT_FF_COUNT_BIT
02230 // */
02231 //void MPU6050::setFreefallDetectionCounterDecrement(uint8_t decrement) {
02232 //    I2Cdev::writeBits(devAddr, MPU6050_RA_MOT_DETECT_CTRL, MPU6050_DETECT_FF_COUNT_BIT,
       MPU6050_DETECT_FF_COUNT_LENGTH, decrement);
02233 //}
02235 // * Detection is registered by the Motion detection module after accelerometer
02236 // * measurements meet their respective threshold conditions over a specified
02237 // * number of samples. When the threshold conditions are met, the corresponding
02238 // * detection counter increments by 1. The user may control the rate at which the
02239 // * detection counter decrements when the threshold condition is not met by
```

```
02240 // * configuring MOT_COUNT. The decrement rate can be set according to the
02241 // * following table:
02242 // *
02243 // * <pre>
02244 // * MOT_COUNT | Counter Decrement
02245 // * ----------+------------------
02246 // * 0         | Reset
02247 // * 1         | 1
02248 // * 2         | 2
02249 // * 3         | 4
02250 // * </pre>
02251 // *
02252 // * When MOT_COUNT is configured to 0 (reset), any non-qualifying sample will
02253 // * reset the counter to 0. For further information on Motion detection,
02254 // * please refer to Registers 29 to 32.
02255 // *
02256 // */
02257 //uint8_t MPU6050::getMotionDetectionCounterDecrement() {
02258 //    I2Cdev::readBits(devAddr, MPU6050_RA_MOT_DETECT_CTRL, MPU6050_DETECT_MOT_COUNT_BIT,
       MPU6050_DETECT_MOT_COUNT_LENGTH, buffer);
02259 //    return buffer[0];
02260 //}
02262 // * @param decrement New decrement configuration value
02263 // * @see getMotionDetectionCounterDecrement()
02264 // * @see MPU6050_RA_MOT_DETECT_CTRL
02265 // * @see MPU6050_DETECT_MOT_COUNT_BIT
02266 // */
02267 //void MPU6050::setMotionDetectionCounterDecrement(uint8_t decrement) {
02268 //    I2Cdev::writeBits(devAddr, MPU6050_RA_MOT_DETECT_CTRL, MPU6050_DETECT_MOT_COUNT_BIT,
       MPU6050_DETECT_MOT_COUNT_LENGTH, decrement);
02269 //}
02270 //
02272 //
02274 // * When this bit is set to 0, the FIFO buffer is disabled. The FIFO buffer
02275 // * cannot be written to or read from while disabled. The FIFO buffer's state
02276 // * does not change unless the MPU-60X0 is power cycled.
02277 // * @return Current FIFO enabled status
02278 // * @see MPU6050_RA_USER_CTRL
02279 // * @see MPU6050_USERCTRL_FIFO_EN_BIT
02280 // */
02281 //bool MPU6050::getFIFOEnabled() {
02282 //    I2Cdev::readBit(devAddr, MPU6050_RA_USER_CTRL, MPU6050_USERCTRL_FIFO_EN_BIT, buffer);
02283 //    return buffer[0];
02284 //}
02286 // * @param enabled New FIFO enabled status
02287 // * @see getFIFOEnabled()
02288 // * @see MPU6050_RA_USER_CTRL
02289 // * @see MPU6050_USERCTRL_FIFO_EN_BIT
02290 // */
02291 //void MPU6050::setFIFOEnabled(bool enabled) {
02292 //    I2Cdev::writeBit(devAddr, MPU6050_RA_USER_CTRL, MPU6050_USERCTRL_FIFO_EN_BIT, enabled);
02293 //}
02295 // * When this mode is enabled, the MPU-60X0 acts as the I2C Master to the
02296 // * external sensor slave devices on the auxiliary I2C bus. When this bit is
02297 // * cleared to 0, the auxiliary I2C bus lines (AUX_DA and AUX_CL) are logically
02298 // * driven by the primary I2C bus (SDA and SCL). This is a precondition to
02299 // * enabling Bypass Mode. For further information regarding Bypass Mode, please
02300 // * refer to Register 55.
02301 // * @return Current I2C Master Mode enabled status
02302 // * @see MPU6050_RA_USER_CTRL
02303 // * @see MPU6050_USERCTRL_I2C_MST_EN_BIT
02304 // */
02305 //bool MPU6050::getI2CMasterModeEnabled() {
02306 //    I2Cdev::readBit(devAddr, MPU6050_RA_USER_CTRL, MPU6050_USERCTRL_I2C_MST_EN_BIT, buffer);
02307 //    return buffer[0];
02308 //}
02310 // * @param enabled New I2C Master Mode enabled status
02311 // * @see getI2CMasterModeEnabled()
02312 // * @see MPU6050_RA_USER_CTRL
02313 // * @see MPU6050_USERCTRL_I2C_MST_EN_BIT
02314 // */
02315 //void MPU6050::setI2CMasterModeEnabled(bool enabled) {
02316 //    I2Cdev::writeBit(devAddr, MPU6050_RA_USER_CTRL, MPU6050_USERCTRL_I2C_MST_EN_BIT, enabled);
02317 //}
02319 // * If this is set, the primary SPI interface will be enabled in place of the
02320 // * disabled primary I2C interface.
02321 // */
02322 //void MPU6050::switchSPIEnabled(bool enabled) {
02323 //    I2Cdev::writeBit(devAddr, MPU6050_RA_USER_CTRL, MPU6050_USERCTRL_I2C_IF_DIS_BIT, enabled);
02324 //}
02326 // * This bit resets the FIFO buffer when set to 1 while FIFO_EN equals 0. This
02327 // * bit automatically clears to 0 after the reset has been triggered.
02328 // * @see MPU6050_RA_USER_CTRL
02329 // * @see MPU6050_USERCTRL_FIFO_RESET_BIT
02330 // */
02331 //void MPU6050::resetFIFO() {
02332 //    I2Cdev::writeBit(devAddr, MPU6050_RA_USER_CTRL, MPU6050_USERCTRL_FIFO_RESET_BIT, true);
```

```
02333 //}
02335 // * This bit resets the I2C Master when set to 1 while I2C_MST_EN equals 0.
02336 // * This bit automatically clears to 0 after the reset has been triggered.
02337 // * @see MPU6050_RA_USER_CTRL
02338 // * @see MPU6050_USERCTRL_I2C_MST_RESET_BIT
02339 // */
02340 //void MPU6050::resetI2CMaster() {
02341 //    I2Cdev::writeBit(devAddr, MPU6050_RA_USER_CTRL, MPU6050_USERCTRL_I2C_MST_RESET_BIT, true);
02342 //}
02344 // * When set to 1, this bit resets the signal paths for all sensors (gyroscopes,
02345 // * accelerometers, and temperature sensor). This operation will also clear the
02346 // * sensor registers. This bit automatically clears to 0 after the reset has been
02347 // * triggered.
02348 // *
02349 // * When resetting only the signal path (and not the sensor registers), please
02350 // * use Register 104, SIGNAL_PATH_RESET.
02351 // *
02352 // * @see MPU6050_RA_USER_CTRL
02353 // * @see MPU6050_USERCTRL_SIG_COND_RESET_BIT
02354 // */
02355 //void MPU6050::resetSensors() {
02356 //    I2Cdev::writeBit(devAddr, MPU6050_RA_USER_CTRL, MPU6050_USERCTRL_SIG_COND_RESET_BIT, true);
02357 //}
02358 //
02360 //
02362 // * A small delay of ~50ms may be desirable after triggering a reset.
02363 // * @see MPU6050_RA_PWR_MGMT_1
02364 // * @see MPU6050_PWR1_DEVICE_RESET_BIT
02365 // */
02366 //void MPU6050::reset() {
02367 //    I2Cdev::writeBit(devAddr, MPU6050_RA_PWR_MGMT_1, MPU6050_PWR1_DEVICE_RESET_BIT, true);
02368 //}
02370 // * Setting the SLEEP bit in the register puts the device into very low power
02371 // * sleep mode. In this mode, only the serial interface and internal registers
02372 // * remain active, allowing for a very low standby current. Clearing this bit
02373 // * puts the device back into normal mode. To save power, the individual standby
02374 // * selections for each of the gyros should be used if any gyro axis is not used
02375 // * by the application.
02376 // * @return Current sleep mode enabled status
02377 // * @see MPU6050_RA_PWR_MGMT_1
02378 // * @see MPU6050_PWR1_SLEEP_BIT
02379 // */
02380 //bool MPU6050::getSleepEnabled() {
02381 //    I2Cdev::readBit(devAddr, MPU6050_RA_PWR_MGMT_1, MPU6050_PWR1_SLEEP_BIT, buffer);
02382 //    return buffer[0];
02383 //}
02385 // * @param enabled New sleep mode enabled status
02386 // * @see getSleepEnabled()
02387 // * @see MPU6050_RA_PWR_MGMT_1
02388 // * @see MPU6050_PWR1_SLEEP_BIT
02389 // */
02390 //void MPU6050::setSleepEnabled(bool enabled) {
02391 //    I2Cdev::writeBit(devAddr, MPU6050_RA_PWR_MGMT_1, MPU6050_PWR1_SLEEP_BIT, enabled);
02392 //}
02394 // * When this bit is set to 1 and SLEEP is disabled, the MPU-60X0 will cycle
02395 // * between sleep mode and waking up to take a single sample of data from active
02396 // * sensors at a rate determined by LP_WAKE_CTRL (register 108).
02397 // * @return Current sleep mode enabled status
02398 // * @see MPU6050_RA_PWR_MGMT_1
02399 // * @see MPU6050_PWR1_CYCLE_BIT
02400 // */
02401 //bool MPU6050::getWakeCycleEnabled() {
02402 //    I2Cdev::readBit(devAddr, MPU6050_RA_PWR_MGMT_1, MPU6050_PWR1_CYCLE_BIT, buffer);
02403 //    return buffer[0];
02404 //}
02406 // * @param enabled New sleep mode enabled status
02407 // * @see getWakeCycleEnabled()
02408 // * @see MPU6050_RA_PWR_MGMT_1
02409 // * @see MPU6050_PWR1_CYCLE_BIT
02410 // */
02411 //void MPU6050::setWakeCycleEnabled(bool enabled) {
02412 //    I2Cdev::writeBit(devAddr, MPU6050_RA_PWR_MGMT_1, MPU6050_PWR1_CYCLE_BIT, enabled);
02413 //}
02415 // * Control the usage of the internal temperature sensor.
02416 // *
02417 // * Note: this register stores the *disabled* value, but for consistency with the
02418 // * rest of the code, the function is named and used with standard true/false
02419 // * values to indicate whether the sensor is enabled or disabled, respectively.
02420 // *
02421 // * @return Current temperature sensor enabled status
02422 // * @see MPU6050_RA_PWR_MGMT_1
02423 // * @see MPU6050_PWR1_TEMP_DIS_BIT
02424 // */
02425 //bool MPU6050::getTempSensorEnabled() {
02426 //    I2Cdev::readBit(devAddr, MPU6050_RA_PWR_MGMT_1, MPU6050_PWR1_TEMP_DIS_BIT, buffer);
02427 //    return buffer[0] == 0; // 1 is actually disabled here
02428 //}
```

```
02430 // * Note: this register stores the *disabled* value, but for consistency with the
02431 // * rest of the code, the function is named and used with standard true/false
02432 // * values to indicate whether the sensor is enabled or disabled, respectively.
02433 // *
02434 // * @param enabled New temperature sensor enabled status
02435 // * @see getTempSensorEnabled()
02436 // * @see MPU6050_RA_PWR_MGMT_1
02437 // * @see MPU6050_PWR1_TEMP_DIS_BIT
02438 // */
02439 //void MPU6050::setTempSensorEnabled(bool enabled) {
02440 //    // 1 is actually disabled here
02441 //    I2Cdev::writeBit(devAddr, MPU6050_RA_PWR_MGMT_1, MPU6050_PWR1_TEMP_DIS_BIT, !enabled);
02442 //}
02444 // * @return Current clock source setting
02445 // * @see MPU6050_RA_PWR_MGMT_1
02446 // * @see MPU6050_PWR1_CLKSEL_BIT
02447 // * @see MPU6050_PWR1_CLKSEL_LENGTH
02448 // */
02449 //uint8_t MPU6050::getClockSource() {
02450 //    I2Cdev::readBits(devAddr, MPU6050_RA_PWR_MGMT_1, MPU6050_PWR1_CLKSEL_BIT, MPU6050_PWR1_CLKSEL_LENGTH,
       buffer);
02451 //    return buffer[0];
02452 //}
02454 // * An internal 8MHz oscillator, gyroscope based clock, or external sources can
02455 // * be selected as the MPU-60X0 clock source. When the internal 8 MHz oscillator
02456 // * or an external source is chosen as the clock source, the MPU-60X0 can operate
02457 // * in low power modes with the gyroscopes disabled.
02458 // *
02459 // * Upon power up, the MPU-60X0 clock source defaults to the internal oscillator.
02460 // * However, it is highly recommended that the device be configured to use one of
02461 // * the gyroscopes (or an external clock source) as the clock reference for
02462 // * improved stability. The clock source can be selected according to the following table:
02463 // *
02464 // * <pre>
02465 // * CLK_SEL | Clock Source
02466 // * --------+--------------------------------------
02467 // * 0       | Internal oscillator
02468 // * 1       | PLL with X Gyro reference
02469 // * 2       | PLL with Y Gyro reference
02470 // * 3       | PLL with Z Gyro reference
02471 // * 4       | PLL with external 32.768kHz reference
02472 // * 5       | PLL with external 19.2MHz reference
02473 // * 6       | Reserved
02474 // * 7       | Stops the clock and keeps the timing generator in reset
02475 // * </pre>
02476 // *
02477 // * @param source New clock source setting
02478 // * @see getClockSource()
02479 // * @see MPU6050_RA_PWR_MGMT_1
02480 // * @see MPU6050_PWR1_CLKSEL_BIT
02481 // * @see MPU6050_PWR1_CLKSEL_LENGTH
02482 // */
02483 //void MPU6050::setClockSource(uint8_t source) {
02484 //    I2Cdev::writeBits(devAddr, MPU6050_RA_PWR_MGMT_1, MPU6050_PWR1_CLKSEL_BIT,
       MPU6050_PWR1_CLKSEL_LENGTH, source);
02485 //}
02486 //
02488 //
02490 // * The MPU-60X0 can be put into Accerlerometer Only Low Power Mode by setting
02491 // * PWRSEL to 1 in the Power Management 1 register (Register 107). In this mode,
02492 // * the device will power off all devices except for the primary I2C interface,
02493 // * waking only the accelerometer at fixed intervals to take a single
02494 // * measurement. The frequency of wake-ups can be configured with LP_WAKE_CTRL
02495 // * as shown below:
02496 // *
02497 // * <pre>
02498 // * LP_WAKE_CTRL | Wake-up Frequency
02499 // * -------------+------------------
02500 // * 0            | 1.25 Hz
02501 // * 1            | 2.5 Hz
02502 // * 2            | 5 Hz
02503 // * 3            | 10 Hz
02504 // * <pre>
02505 // *
02506 // * For further information regarding the MPU-60X0's power modes, please refer to
02507 // * Register 107.
02508 // *
02509 // * @return Current wake frequency
02510 // * @see MPU6050_RA_PWR_MGMT_2
02511 // */
02512 //uint8_t MPU6050::getWakeFrequency() {
02513 //    I2Cdev::readBits(devAddr, MPU6050_RA_PWR_MGMT_2, MPU6050_PWR2_LP_WAKE_CTRL_BIT,
       MPU6050_PWR2_LP_WAKE_CTRL_LENGTH, buffer);
02514 //    return buffer[0];
02515 //}
02517 // * @param frequency New wake frequency
02518 // * @see MPU6050_RA_PWR_MGMT_2
```

```
02519 // */
02520 //void MPU6050::setWakeFrequency(uint8_t frequency) {
02521 //    I2Cdev::writeBits(devAddr, MPU6050_RA_PWR_MGMT_2, MPU6050_PWR2_LP_WAKE_CTRL_BIT,
      MPU6050_PWR2_LP_WAKE_CTRL_LENGTH, frequency);
02522 //}
02523 //
02525 // * If enabled, the X-axis will not gather or report data (or use power).
02526 // * @return Current X-axis standby enabled status
02527 // * @see MPU6050_RA_PWR_MGMT_2
02528 // * @see MPU6050_PWR2_STBY_XA_BIT
02529 // */
02530 //bool MPU6050::getStandbyXAccelEnabled() {
02531 //    I2Cdev::readBit(devAddr, MPU6050_RA_PWR_MGMT_2, MPU6050_PWR2_STBY_XA_BIT, buffer);
02532 //    return buffer[0];
02533 //}
02535 // * @param New X-axis standby enabled status
02536 // * @see getStandbyXAccelEnabled()
02537 // * @see MPU6050_RA_PWR_MGMT_2
02538 // * @see MPU6050_PWR2_STBY_XA_BIT
02539 // */
02540 //void MPU6050::setStandbyXAccelEnabled(bool enabled) {
02541 //    I2Cdev::writeBit(devAddr, MPU6050_RA_PWR_MGMT_2, MPU6050_PWR2_STBY_XA_BIT, enabled);
02542 //}
02544 // * If enabled, the Y-axis will not gather or report data (or use power).
02545 // * @return Current Y-axis standby enabled status
02546 // * @see MPU6050_RA_PWR_MGMT_2
02547 // * @see MPU6050_PWR2_STBY_YA_BIT
02548 // */
02549 //bool MPU6050::getStandbyYAccelEnabled() {
02550 //    I2Cdev::readBit(devAddr, MPU6050_RA_PWR_MGMT_2, MPU6050_PWR2_STBY_YA_BIT, buffer);
02551 //    return buffer[0];
02552 //}
02554 // * @param New Y-axis standby enabled status
02555 // * @see getStandbyYAccelEnabled()
02556 // * @see MPU6050_RA_PWR_MGMT_2
02557 // * @see MPU6050_PWR2_STBY_YA_BIT
02558 // */
02559 //void MPU6050::setStandbyYAccelEnabled(bool enabled) {
02560 //    I2Cdev::writeBit(devAddr, MPU6050_RA_PWR_MGMT_2, MPU6050_PWR2_STBY_YA_BIT, enabled);
02561 //}
02563 // * If enabled, the Z-axis will not gather or report data (or use power).
02564 // * @return Current Z-axis standby enabled status
02565 // * @see MPU6050_RA_PWR_MGMT_2
02566 // * @see MPU6050_PWR2_STBY_ZA_BIT
02567 // */
02568 //bool MPU6050::getStandbyZAccelEnabled() {
02569 //    I2Cdev::readBit(devAddr, MPU6050_RA_PWR_MGMT_2, MPU6050_PWR2_STBY_ZA_BIT, buffer);
02570 //    return buffer[0];
02571 //}
02573 // * @param New Z-axis standby enabled status
02574 // * @see getStandbyZAccelEnabled()
02575 // * @see MPU6050_RA_PWR_MGMT_2
02576 // * @see MPU6050_PWR2_STBY_ZA_BIT
02577 // */
02578 //void MPU6050::setStandbyZAccelEnabled(bool enabled) {
02579 //    I2Cdev::writeBit(devAddr, MPU6050_RA_PWR_MGMT_2, MPU6050_PWR2_STBY_ZA_BIT, enabled);
02580 //}
02582 // * If enabled, the X-axis will not gather or report data (or use power).
02583 // * @return Current X-axis standby enabled status
02584 // * @see MPU6050_RA_PWR_MGMT_2
02585 // * @see MPU6050_PWR2_STBY_XG_BIT
02586 // */
02587 //bool MPU6050::getStandbyXGyroEnabled() {
02588 //    I2Cdev::readBit(devAddr, MPU6050_RA_PWR_MGMT_2, MPU6050_PWR2_STBY_XG_BIT, buffer);
02589 //    return buffer[0];
02590 //}
02592 // * @param New X-axis standby enabled status
02593 // * @see getStandbyXGyroEnabled()
02594 // * @see MPU6050_RA_PWR_MGMT_2
02595 // * @see MPU6050_PWR2_STBY_XG_BIT
02596 // */
02597 //void MPU6050::setStandbyXGyroEnabled(bool enabled) {
02598 //    I2Cdev::writeBit(devAddr, MPU6050_RA_PWR_MGMT_2, MPU6050_PWR2_STBY_XG_BIT, enabled);
02599 //}
02601 // * If enabled, the Y-axis will not gather or report data (or use power).
02602 // * @return Current Y-axis standby enabled status
02603 // * @see MPU6050_RA_PWR_MGMT_2
02604 // * @see MPU6050_PWR2_STBY_YG_BIT
02605 // */
02606 //bool MPU6050::getStandbyYGyroEnabled() {
02607 //    I2Cdev::readBit(devAddr, MPU6050_RA_PWR_MGMT_2, MPU6050_PWR2_STBY_YG_BIT, buffer);
02608 //    return buffer[0];
02609 //}
02611 // * @param New Y-axis standby enabled status
02612 // * @see getStandbyYGyroEnabled()
02613 // * @see MPU6050_RA_PWR_MGMT_2
02614 // * @see MPU6050_PWR2_STBY_YG_BIT
```

```
02615 // */
02616 //void MPU6050::setStandbyYGyroEnabled(bool enabled) {
02617 //    I2Cdev::writeBit(devAddr, MPU6050_RA_PWR_MGMT_2, MPU6050_PWR2_STBY_YG_BIT, enabled);
02618 //}
02620 // * If enabled, the Z-axis will not gather or report data (or use power).
02621 // * @return Current Z-axis standby enabled status
02622 // * @see MPU6050_RA_PWR_MGMT_2
02623 // * @see MPU6050_PWR2_STBY_ZG_BIT
02624 // */
02625 //bool MPU6050::getStandbyZGyroEnabled() {
02626 //    I2Cdev::readBit(devAddr, MPU6050_RA_PWR_MGMT_2, MPU6050_PWR2_STBY_ZG_BIT, buffer);
02627 //    return buffer[0];
02628 //}
02630 // * @param New Z-axis standby enabled status
02631 // * @see getStandbyZGyroEnabled()
02632 // * @see MPU6050_RA_PWR_MGMT_2
02633 // * @see MPU6050_PWR2_STBY_ZG_BIT
02634 // */
02635 //void MPU6050::setStandbyZGyroEnabled(bool enabled) {
02636 //    I2Cdev::writeBit(devAddr, MPU6050_RA_PWR_MGMT_2, MPU6050_PWR2_STBY_ZG_BIT, enabled);
02637 //}
02638 //
02640 //
02642 // * This value indicates the number of bytes stored in the FIFO buffer. This
02643 // * number is in turn the number of bytes that can be read from the FIFO buffer
02644 // * and it is directly proportional to the number of samples available given the
02645 // * set of sensor data bound to be stored in the FIFO (register 35 and 36).
02646 // * @return Current FIFO buffer size
02647 // */
02648 //uint16_t MPU6050::getFIFOCount() {
02649 //    I2Cdev::readBytes(devAddr, MPU6050_RA_FIFO_COUNTH, 2, buffer);
02650 //    return (((uint16_t)buffer[0]) << 8) | buffer[1];
02651 //}
02652 //
02654 //
02656 // * This register is used to read and write data from the FIFO buffer. Data is
02657 // * written to the FIFO in order of register number (from lowest to highest). If
02658 // * all the FIFO enable flags (see below) are enabled and all External Sensor
02659 // * Data registers (Registers 73 to 96) are associated with a Slave device, the
02660 // * contents of registers 59 through 96 will be written in order at the Sample
02661 // * Rate.
02662 // *
02663 // * The contents of the sensor data registers (Registers 59 to 96) are written
02664 // * into the FIFO buffer when their corresponding FIFO enable flags are set to 1
02665 // * in FIFO_EN (Register 35). An additional flag for the sensor data registers
02666 // * associated with I2C Slave 3 can be found in I2C_MST_CTRL (Register 36).
02667 // *
02668 // * If the FIFO buffer has overflowed, the status bit FIFO_OFLOW_INT is
02669 // * automatically set to 1. This bit is located in INT_STATUS (Register 58).
02670 // * When the FIFO buffer has overflowed, the oldest data will be lost and new
02671 // * data will be written to the FIFO.
02672 // *
02673 // * If the FIFO buffer is empty, reading this register will return the last byte
02674 // * that was previously read from the FIFO until new data is available. The user
02675 // * should check FIFO_COUNT to ensure that the FIFO buffer is not read when
02676 // * empty.
02677 // *
02678 // * @return Byte from FIFO buffer
02679 // */
02680 //uint8_t MPU6050::getFIFOByte() {
02681 //    I2Cdev::readByte(devAddr, MPU6050_RA_FIFO_R_W, buffer);
02682 //    return buffer[0];
02683 //}
02684 //void MPU6050::getFIFOBytes(uint8_t *data, uint8_t length) {
02685 //    I2Cdev::readBytes(devAddr, MPU6050_RA_FIFO_R_W, length, data);
02686 //}
02688 // * @see getFIFOByte()
02689 // * @see MPU6050_RA_FIFO_R_W
02690 // */
02691 //void MPU6050::setFIFOByte(uint8_t data) {
02692 //    I2Cdev::writeByte(devAddr, MPU6050_RA_FIFO_R_W, data);
02693 //}
02694 //
02696 //
02698 // * This register is used to verify the identity of the device (0b110100, 0x34).
02699 // * @return Device ID (6 bits only! should be 0x34)
02700 // * @see MPU6050_RA_WHO_AM_I
02701 // * @see MPU6050_WHO_AM_I_BIT
02702 // * @see MPU6050_WHO_AM_I_LENGTH
02703 // */
02704 //uint8_t MPU6050::getDeviceID() {
02705 //    I2Cdev::readBits(devAddr, MPU6050_RA_WHO_AM_I, MPU6050_WHO_AM_I_BIT, MPU6050_WHO_AM_I_LENGTH,
      buffer);
02706 //    return buffer[0];
02707 //}
02709 // * Write a new ID into the WHO_AM_I register (no idea why this should ever be
02710 // * necessary though).
```

```
02711 // * @param id New device ID to set.
02712 // * @see getDeviceID()
02713 // * @see MPU6050_RA_WHO_AM_I
02714 // * @see MPU6050_WHO_AM_I_BIT
02715 // * @see MPU6050_WHO_AM_I_LENGTH
02716 // */
02717 //void MPU6050::setDeviceID(uint8_t id) {
02718 //    I2Cdev::writeBits(devAddr, MPU6050_RA_WHO_AM_I, MPU6050_WHO_AM_I_BIT, MPU6050_WHO_AM_I_LENGTH, id);
02719 //}
02720 //
02722 //
02724 //
02725 //uint8_t MPU6050::getOTPBankValid() {
02726 //    I2Cdev::readBit(devAddr, MPU6050_RA_XG_OFFS_TC, MPU6050_TC_OTP_BNK_VLD_BIT, buffer);
02727 //    return buffer[0];
02728 //}
02729 //void MPU6050::setOTPBankValid(bool enabled) {
02730 //    I2Cdev::writeBit(devAddr, MPU6050_RA_XG_OFFS_TC, MPU6050_TC_OTP_BNK_VLD_BIT, enabled);
02731 //}
02732 //int8_t MPU6050::getXGyroOffset() {
02733 //    I2Cdev::readBits(devAddr, MPU6050_RA_XG_OFFS_TC, MPU6050_TC_OFFSET_BIT, MPU6050_TC_OFFSET_LENGTH,
       buffer);
02734 //    return buffer[0];
02735 //}
02736 //void MPU6050::setXGyroOffset(int8_t offset) {
02737 //    I2Cdev::writeBits(devAddr, MPU6050_RA_XG_OFFS_TC, MPU6050_TC_OFFSET_BIT, MPU6050_TC_OFFSET_LENGTH,
       offset);
02738 //}
02739 //
02741 //
02742 //int8_t MPU6050::getYGyroOffset() {
02743 //    I2Cdev::readBits(devAddr, MPU6050_RA_YG_OFFS_TC, MPU6050_TC_OFFSET_BIT, MPU6050_TC_OFFSET_LENGTH,
       buffer);
02744 //    return buffer[0];
02745 //}
02746 //void MPU6050::setYGyroOffset(int8_t offset) {
02747 //    I2Cdev::writeBits(devAddr, MPU6050_RA_YG_OFFS_TC, MPU6050_TC_OFFSET_BIT, MPU6050_TC_OFFSET_LENGTH,
       offset);
02748 //}
02749 //
02751 //
02752 //int8_t MPU6050::getZGyroOffset() {
02753 //    I2Cdev::readBits(devAddr, MPU6050_RA_ZG_OFFS_TC, MPU6050_TC_OFFSET_BIT, MPU6050_TC_OFFSET_LENGTH,
       buffer);
02754 //    return buffer[0];
02755 //}
02756 //void MPU6050::setZGyroOffset(int8_t offset) {
02757 //    I2Cdev::writeBits(devAddr, MPU6050_RA_ZG_OFFS_TC, MPU6050_TC_OFFSET_BIT, MPU6050_TC_OFFSET_LENGTH,
       offset);
02758 //}
02759 //
02761 //
02762 //int8_t MPU6050::getXFineGain() {
02763 //    I2Cdev::readByte(devAddr, MPU6050_RA_X_FINE_GAIN, buffer);
02764 //    return buffer[0];
02765 //}
02766 //void MPU6050::setXFineGain(int8_t gain) {
02767 //    I2Cdev::writeByte(devAddr, MPU6050_RA_X_FINE_GAIN, gain);
02768 //}
02769 //
02771 //
02772 //int8_t MPU6050::getYFineGain() {
02773 //    I2Cdev::readByte(devAddr, MPU6050_RA_Y_FINE_GAIN, buffer);
02774 //    return buffer[0];
02775 //}
02776 //void MPU6050::setYFineGain(int8_t gain) {
02777 //    I2Cdev::writeByte(devAddr, MPU6050_RA_Y_FINE_GAIN, gain);
02778 //}
02779 //
02781 //
02782 //int8_t MPU6050::getZFineGain() {
02783 //    I2Cdev::readByte(devAddr, MPU6050_RA_Z_FINE_GAIN, buffer);
02784 //    return buffer[0];
02785 //}
02786 //void MPU6050::setZFineGain(int8_t gain) {
02787 //    I2Cdev::writeByte(devAddr, MPU6050_RA_Z_FINE_GAIN, gain);
02788 //}
02789 //
02791 //
02792 //int16_t MPU6050::getXAccelOffset() {
02793 //    I2Cdev::readBytes(devAddr, MPU6050_RA_XA_OFFS_H, 2, buffer);
02794 //    return (((int16_t)buffer[0]) << 8) | buffer[1];
02795 //}
02796 //void MPU6050::setXAccelOffset(int16_t offset) {
02797 //    I2Cdev::writeWord(devAddr, MPU6050_RA_XA_OFFS_H, offset);
02798 //}
02799 //
```

```
02801 //
02802 //int16_t MPU6050::getYAccelOffset() {
02803 //    I2Cdev::readBytes(devAddr, MPU6050_RA_YA_OFFS_H, 2, buffer);
02804 //    return (((int16_t)buffer[0]) << 8) | buffer[1];
02805 //}
02806 //void MPU6050::setYAccelOffset(int16_t offset) {
02807 //    I2Cdev::writeWord(devAddr, MPU6050_RA_YA_OFFS_H, offset);
02808 //}
02809 //
02811 //
02812 //int16_t MPU6050::getZAccelOffset() {
02813 //    I2Cdev::readBytes(devAddr, MPU6050_RA_ZA_OFFS_H, 2, buffer);
02814 //    return (((int16_t)buffer[0]) << 8) | buffer[1];
02815 //}
02816 //void MPU6050::setZAccelOffset(int16_t offset) {
02817 //    I2Cdev::writeWord(devAddr, MPU6050_RA_ZA_OFFS_H, offset);
02818 //}
02819 //
02821 //
02822 //int16_t MPU6050::getXGyroOffsetUser() {
02823 //    I2Cdev::readBytes(devAddr, MPU6050_RA_XG_OFFS_USRH, 2, buffer);
02824 //    return (((int16_t)buffer[0]) << 8) | buffer[1];
02825 //}
02826 //void MPU6050::setXGyroOffsetUser(int16_t offset) {
02827 //    I2Cdev::writeWord(devAddr, MPU6050_RA_XG_OFFS_USRH, offset);
02828 //}
02829 //
02831 //
02832 //int16_t MPU6050::getYGyroOffsetUser() {
02833 //    I2Cdev::readBytes(devAddr, MPU6050_RA_YG_OFFS_USRH, 2, buffer);
02834 //    return (((int16_t)buffer[0]) << 8) | buffer[1];
02835 //}
02836 //void MPU6050::setYGyroOffsetUser(int16_t offset) {
02837 //    I2Cdev::writeWord(devAddr, MPU6050_RA_YG_OFFS_USRH, offset);
02838 //}
02839 //
02841 //
02842 //int16_t MPU6050::getZGyroOffsetUser() {
02843 //    I2Cdev::readBytes(devAddr, MPU6050_RA_ZG_OFFS_USRH, 2, buffer);
02844 //    return (((int16_t)buffer[0]) << 8) | buffer[1];
02845 //}
02846 //void MPU6050::setZGyroOffsetUser(int16_t offset) {
02847 //    I2Cdev::writeWord(devAddr, MPU6050_RA_ZG_OFFS_USRH, offset);
02848 //}
02849 //
02851 //
02852 //bool MPU6050::getIntPLLReadyEnabled() {
02853 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_ENABLE, MPU6050_INTERRUPT_PLL_RDY_INT_BIT, buffer);
02854 //    return buffer[0];
02855 //}
02856 //void MPU6050::setIntPLLReadyEnabled(bool enabled) {
02857 //    I2Cdev::writeBit(devAddr, MPU6050_RA_INT_ENABLE, MPU6050_INTERRUPT_PLL_RDY_INT_BIT, enabled);
02858 //}
02859 //bool MPU6050::getIntDMPEnabled() {
02860 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_ENABLE, MPU6050_INTERRUPT_DMP_INT_BIT, buffer);
02861 //    return buffer[0];
02862 //}
02863 //void MPU6050::setIntDMPEnabled(bool enabled) {
02864 //    I2Cdev::writeBit(devAddr, MPU6050_RA_INT_ENABLE, MPU6050_INTERRUPT_DMP_INT_BIT, enabled);
02865 //}
02866 //
02868 //
02869 //bool MPU6050::getDMPInt5Status() {
02870 //    I2Cdev::readBit(devAddr, MPU6050_RA_DMP_INT_STATUS, MPU6050_DMPINT_5_BIT, buffer);
02871 //    return buffer[0];
02872 //}
02873 //bool MPU6050::getDMPInt4Status() {
02874 //    I2Cdev::readBit(devAddr, MPU6050_RA_DMP_INT_STATUS, MPU6050_DMPINT_4_BIT, buffer);
02875 //    return buffer[0];
02876 //}
02877 //bool MPU6050::getDMPInt3Status() {
02878 //    I2Cdev::readBit(devAddr, MPU6050_RA_DMP_INT_STATUS, MPU6050_DMPINT_3_BIT, buffer);
02879 //    return buffer[0];
02880 //}
02881 //bool MPU6050::getDMPInt2Status() {
02882 //    I2Cdev::readBit(devAddr, MPU6050_RA_DMP_INT_STATUS, MPU6050_DMPINT_2_BIT, buffer);
02883 //    return buffer[0];
02884 //}
02885 //bool MPU6050::getDMPInt1Status() {
02886 //    I2Cdev::readBit(devAddr, MPU6050_RA_DMP_INT_STATUS, MPU6050_DMPINT_1_BIT, buffer);
02887 //    return buffer[0];
02888 //}
02889 //bool MPU6050::getDMPInt0Status() {
02890 //    I2Cdev::readBit(devAddr, MPU6050_RA_DMP_INT_STATUS, MPU6050_DMPINT_0_BIT, buffer);
02891 //    return buffer[0];
02892 //}
02893 //
```

```
02895 //
02896 //bool MPU6050::getIntPLLReadyStatus() {
02897 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_STATUS, MPU6050_INTERRUPT_PLL_RDY_INT_BIT, buffer);
02898 //    return buffer[0];
02899 //}
02900 //bool MPU6050::getIntDMPStatus() {
02901 //    I2Cdev::readBit(devAddr, MPU6050_RA_INT_STATUS, MPU6050_INTERRUPT_DMP_INT_BIT, buffer);
02902 //    return buffer[0];
02903 //}
02904 //
02906 //
02907 //bool MPU6050::getDMPEnabled() {
02908 //    I2Cdev::readBit(devAddr, MPU6050_RA_USER_CTRL, MPU6050_USERCTRL_DMP_EN_BIT, buffer);
02909 //    return buffer[0];
02910 //}
02911 //void MPU6050::setDMPEnabled(bool enabled) {
02912 //    I2Cdev::writeBit(devAddr, MPU6050_RA_USER_CTRL, MPU6050_USERCTRL_DMP_EN_BIT, enabled);
02913 //}
02914 //void MPU6050::resetDMP() {
02916 //    I2Cdev::writeBit(devAddr, MPU6050_RA_USER_CTRL, MPU6050_USERCTRL_DMP_RESET_BIT, true);
02917 //    I2Cdev::writeBit(devAddr, MPU6050_RA_USER_CTRL, 0x00, true);
02918 //    I2Cdev::writeBit(devAddr, MPU6050_RA_USER_CTRL, 0x80 | 0x40 | 0x08, true);
02919 //}
02920 //
02922 //
02923 //void MPU6050::setMemoryBank(uint8_t bank, bool prefetchEnabled, bool userBank) {
02924 //    bank &= 0x1F;
02925 //    if (userBank) bank |= 0x20;
02926 //    if (prefetchEnabled) bank |= 0x40;
02927 //    I2Cdev::writeByte(devAddr, MPU6050_RA_BANK_SEL, bank);
02928 //}
02929 //
02931 //
02932 //void MPU6050::setMemoryStartAddress(uint8_t address) {
02933 //    I2Cdev::writeByte(devAddr, MPU6050_RA_MEM_START_ADDR, address);
02934 //}
02935 //
02937 //
02938 //uint8_t MPU6050::readMemoryByte() {
02939 //    I2Cdev::readByte(devAddr, MPU6050_RA_MEM_R_W, buffer);
02940 //    return buffer[0];
02941 //}
02942 //void MPU6050::writeMemoryByte(uint8_t data) {
02943 //    I2Cdev::writeByte(devAddr, MPU6050_RA_MEM_R_W, data);
02944 //}
02945 //void MPU6050::readMemoryBlock(uint8_t *data, uint16_t dataSize, uint8_t bank, uint8_t address) {
02946 //    setMemoryBank(bank);
02947 //    setMemoryStartAddress(address);
02948 //    uint8_t chunkSize;
02949 //    for (uint16_t i = 0; i < dataSize;) {
02950 //        // determine correct chunk size according to bank position and data size
02951 //        chunkSize = MPU6050_DMP_MEMORY_CHUNK_SIZE;
02952 //
02953 //        // make sure we don't go past the data size
02954 //        if (i + chunkSize > dataSize) chunkSize = dataSize - i;
02955 //
02956 //        // make sure this chunk doesn't go past the bank boundary (256 bytes)
02957 //        if (chunkSize > 256 - address) chunkSize = 256 - address;
02958 //
02959 //        // read the chunk of data as specified
02960 //        I2Cdev::readBytes(devAddr, MPU6050_RA_MEM_R_W, chunkSize, data + i);
02961 //
02962 //        // increase byte index by [chunkSize]
02963 //        i += chunkSize;
02964 //
02965 //        // uint8_t automatically wraps to 0 at 256
02966 //        address += chunkSize;
02967 //
02968 //        // if we aren't done, update bank (if necessary) and address
02969 //        if (i < dataSize) {
02970 //            if (address == 0) bank++;
02971 //            setMemoryBank(bank);
02972 //            setMemoryStartAddress(address);
02973 //        }
02974 //    }
02975 //}
02976 //bool MPU6050::writeMemoryBlock(const uint8_t *data, uint16_t dataSize, uint8_t bank, uint8_t address,
      bool verify, bool useProgMem) {
02977 //    setMemoryBank(bank);
02978 //    setMemoryStartAddress(address);
02979 //    uint8_t chunkSize;
02980 //    uint8_t *verifyBuffer;
02981 //    uint8_t *progBuffer;
02982 //    uint16_t i;
02983 //    uint8_t j;
02984 //    if (verify) verifyBuffer = (uint8_t *)malloc(MPU6050_DMP_MEMORY_CHUNK_SIZE);
02985 //    if (useProgMem) progBuffer = (uint8_t *)malloc(MPU6050_DMP_MEMORY_CHUNK_SIZE);
```

```
02986 //    for (i = 0; i < dataSize;) {
02987 //        // determine correct chunk size according to bank position and data size
02988 //        chunkSize = MPU6050_DMP_MEMORY_CHUNK_SIZE;
02989 //
02990 //        // make sure we don't go past the data size
02991 //        if (i + chunkSize > dataSize) chunkSize = dataSize - i;
02992 //
02993 //        // make sure this chunk doesn't go past the bank boundary (256 bytes)
02994 //        if (chunkSize > 256 - address) chunkSize = 256 - address;
02995 //
02996 //        if (useProgMem) {
02997 //            // write the chunk of data as specified
02998 //            for (j = 0; j < chunkSize; j++) progBuffer[j] = pgm_read_byte(data + i + j);
02999 //        } else {
03000 //            // write the chunk of data as specified
03001 //            progBuffer = (uint8_t *)data + i;
03002 //        }
03003 //
03004 //        I2Cdev::writeBytes(devAddr, MPU6050_RA_MEM_R_W, chunkSize, progBuffer);
03005 //
03006 //        // verify data if needed
03007 //        if (verify && verifyBuffer) {
03008 //            setMemoryBank(bank);
03009 //            setMemoryStartAddress(address);
03010 //            I2Cdev::readBytes(devAddr, MPU6050_RA_MEM_R_W, chunkSize, verifyBuffer);
03011 //            if (memcmp(progBuffer, verifyBuffer, chunkSize) != 0) {
03012 //                /*Serial.print("Block write verification error, bank ");
03013 //                Serial.print(bank, DEC);
03014 //                Serial.print(", address ");
03015 //                Serial.print(address, DEC);
03016 //                Serial.print("!\nExpected:");
03017 //                for (j = 0; j < chunkSize; j++) {
03018 //                    Serial.print(" 0x");
03019 //                    if (progBuffer[j] < 16) Serial.print("0");
03020 //                    Serial.print(progBuffer[j], HEX);
03021 //                }
03022 //                Serial.print("\nReceived:");
03023 //                for (uint8_t j = 0; j < chunkSize; j++) {
03024 //                    Serial.print(" 0x");
03025 //                    if (verifyBuffer[i + j] < 16) Serial.print("0");
03026 //                    Serial.print(verifyBuffer[i + j], HEX);
03027 //                }
03028 //                Serial.print("\n");*/
03029 //                free(verifyBuffer);
03030 //                if (useProgMem) free(progBuffer);
03031 //                return false; // uh oh.
03032 //            }
03033 //        }
03034 //
03035 //        // increase byte index by [chunkSize]
03036 //        i += chunkSize;
03037 //
03038 //        // uint8_t automatically wraps to 0 at 256
03039 //        address += chunkSize;
03040 //
03041 //        // if we aren't done, update bank (if necessary) and address
03042 //        if (i < dataSize) {
03043 //            if (address == 0) bank++;
03044 //            setMemoryBank(bank);
03045 //            setMemoryStartAddress(address);
03046 //        }
03047 //    }
03048 //    if (verify) free(verifyBuffer);
03049 //    if (useProgMem) free(progBuffer);
03050 //    return true;
03051 //}
03052 //bool MPU6050::writeProgMemoryBlock(const uint8_t *data, uint16_t dataSize, uint8_t bank, uint8_t address,
       bool verify) {
03053 //    return writeMemoryBlock(data, dataSize, bank, address, verify, true);
03054 //}
03055 //bool MPU6050::writeDMPConfigurationSet(const uint8_t *data, uint16_t dataSize, bool useProgMem) {
03056 //    uint8_t *progBuffer, success, special;
03057 //    uint16_t i, j;
03058 //    if (useProgMem) {
03059 //        progBuffer = (uint8_t *)malloc(8); // assume 8-byte blocks, realloc later if necessary
03060 //    }
03061 //
03062 //    // config set data is a long string of blocks with the following structure:
03063 //    // [bank] [offset] [length] [byte[0], byte[1], ..., byte[length]]
03064 //    uint8_t bank, offset, length;
03065 //    for (i = 0; i < dataSize;) {
03066 //        if (useProgMem) {
03067 //            bank = pgm_read_byte(data + i++);
03068 //            offset = pgm_read_byte(data + i++);
03069 //            length = pgm_read_byte(data + i++);
03070 //        } else {
03071 //            bank = data[i++];
```

```
03072 //            offset = data[i++];
03073 //            length = data[i++];
03074 //        }
03075 //
03076 //        // write data or perform special action
03077 //        if (length > 0) {
03078 //            // regular block of data to write
03079 //            /*Serial.print("Writing config block to bank ");
03080 //            Serial.print(bank);
03081 //            Serial.print(", offset ");
03082 //            Serial.print(offset);
03083 //            Serial.print(", length=");
03084 //            Serial.println(length);*/
03085 //            if (useProgMem) {
03086 //                if (sizeof(progBuffer) < length) progBuffer = (uint8_t *)realloc(progBuffer, length);
03087 //                for (j = 0; j < length; j++) progBuffer[j] = pgm_read_byte(data + i + j);
03088 //            } else {
03089 //                progBuffer = (uint8_t *)data + i;
03090 //            }
03091 //            success = writeMemoryBlock(progBuffer, length, bank, offset, true);
03092 //            i += length;
03093 //        } else {
03094 //            // special instruction
03095 //            // NOTE: this kind of behavior (what and when to do certain things)
03096 //            // is totally undocumented. This code is in here based on observed
03097 //            // behavior only, and exactly why (or even whether) it has to be here
03098 //            // is anybody's guess for now.
03099 //            if (useProgMem) {
03100 //                special = pgm_read_byte(data + i++);
03101 //            } else {
03102 //                special = data[i++];
03103 //            }
03104 //            /*Serial.print("Special command code ");
03105 //            Serial.print(special, HEX);
03106 //            Serial.println(" found...");*/
03107 //            if (special == 0x01) {
03108 //                // enable DMP-related interrupts
03109 //
03110 //                //setIntZeroMotionEnabled(true);
03111 //                //setIntFIFOBufferOverflowEnabled(true);
03112 //                //setIntDMPEnabled(true);
03113 //                I2Cdev::writeByte(devAddr, MPU6050_RA_INT_ENABLE, 0x32);  // single operation
03114 //
03115 //                success = true;
03116 //            } else {
03117 //                // unknown special command
03118 //                success = false;
03119 //            }
03120 //        }
03121 //
03122 //        if (!success) {
03123 //            if (useProgMem) free(progBuffer);
03124 //            return false; // uh oh
03125 //        }
03126 //    }
03127 //    if (useProgMem) free(progBuffer);
03128 //    return true;
03129 //}
03130 //bool MPU6050::writeProgDMPConfigurationSet(const uint8_t *data, uint16_t dataSize) {
03131 //    return writeDMPConfigurationSet(data, dataSize, true);
03132 //}
03133 //
03135 //
03136 //uint8_t MPU6050::getDMPConfig1() {
03137 //    I2Cdev::readByte(devAddr, MPU6050_RA_DMP_CFG_1, buffer);
03138 //    return buffer[0];
03139 //}
03140 //void MPU6050::setDMPConfig1(uint8_t config) {
03141 //    I2Cdev::writeByte(devAddr, MPU6050_RA_DMP_CFG_1, config);
03142 //}
03143 //
03145 //
03146 //uint8_t MPU6050::getDMPConfig2() {
03147 //    I2Cdev::readByte(devAddr, MPU6050_RA_DMP_CFG_2, buffer);
03148 //    return buffer[0];
03149 //}
03150 //void MPU6050::setDMPConfig2(uint8_t config) {
03151 //    I2Cdev::writeByte(devAddr, MPU6050_RA_DMP_CFG_2, config);
03152 //}
03153 //
03154 //
03155 //
03156 //
```

5.7 GyroscopeMPU6050.h File Reference

5.8 GyroscopeMPU6050.h

```
00001 // * Arduino - MPU6050 Gyroscope Driver
00003 // *
00004 // * Concrete implementation of a gyroscope driver for MPU6050 sensor.
00005 // *
00006 // * @author Dalmir da Silva <dalmirdasilva@gmail.com>
00007 // */
00008 //
00009 //#ifndef __ARDUINO_DRIVER_GYROSCOPE_MPU6050_H__
00010 //#define __ARDUINO_DRIVER_GYROSCOPE_MPU6050_H__ 1
00011 //
00013 // *
00014 // */
00015 //class ArduinoGyroscopeMPU6050 {
00016 //
00017 //    /**
00018 //     * Get X rotation.
00019 //     */
00020 //    int getRotationX();
00021 //
00022 //    /**
00023 //     * Get Y rotation.
00024 //     */
00025 //    int getRotationY();
00026 //
00027 //    /**
00028 //     * Get Z rotation.
00029 //     */
00030 //    int getRotationZ();
00031 //};
00032 //
00033 //#endif /* __ARDUINO_DRIVER_GYROSCOPE_MPU6050_H__ */
00034 //
00035 //
00043 //
00047 //
00049 //I2Cdev device library code is placed under the MIT license
00050 //Copyright (c) 2012 Jeff Rowberg
00051 //
00052 //Permission is hereby granted, free of charge, to any person obtaining a copy
00053 //of this software and associated documentation files (the "Software"), to deal
00054 //in the Software without restriction, including without limitation the rights
00055 //to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
00056 //copies of the Software, and to permit persons to whom the Software is
00057 //furnished to do so, subject to the following conditions:
00058 //
00059 //The above copyright notice and this permission notice shall be included in
00060 //all copies or substantial portions of the Software.
00061 //
00062 //THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00063 //IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00064 //FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
00065 //AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
00066 //LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
00067 //OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
00068 //THE SOFTWARE.
00069 //===============================================
00070 //*/
00071 //
00072 //#ifndef _MPU6050_H_
00073 //#define _MPU6050_H_
00074 //
00075 //#include "I2Cdev.h"
00076 //#include "definitions.h"
00077 //#include <avr/pgmspace.h>
00078 //
00079 //
00084 //
00085 //#define MPU6050_RA_XG_OFFS_TC       0x00 //[7] PWR_MODE, [6:1] XG_OFFS_TC, [0] OTP_BNK_VLD
00086 //#define MPU6050_RA_YG_OFFS_TC       0x01 //[7] PWR_MODE, [6:1] YG_OFFS_TC, [0] OTP_BNK_VLD
00087 //#define MPU6050_RA_ZG_OFFS_TC       0x02 //[7] PWR_MODE, [6:1] ZG_OFFS_TC, [0] OTP_BNK_VLD
00088 //#define MPU6050_RA_X_FINE_GAIN      0x03 //[7:0] X_FINE_GAIN
00089 //#define MPU6050_RA_Y_FINE_GAIN      0x04 //[7:0] Y_FINE_GAIN
00090 //#define MPU6050_RA_Z_FINE_GAIN      0x05 //[7:0] Z_FINE_GAIN
00091 //#define MPU6050_RA_XA_OFFS_H        0x06 //[15:0] XA_OFFS
00092 //#define MPU6050_RA_XA_OFFS_L_TC     0x07
00093 //#define MPU6050_RA_YA_OFFS_H        0x08 //[15:0] YA_OFFS
00094 //#define MPU6050_RA_YA_OFFS_L_TC     0x09
00095 //#define MPU6050_RA_ZA_OFFS_H        0x0A //[15:0] ZA_OFFS
00096 //#define MPU6050_RA_ZA_OFFS_L_TC     0x0B
00097 //#define MPU6050_RA_XG_OFFS_USRH     0x13 //[15:0] XG_OFFS_USR
```

```
00098 //#define MPU6050_RA_XG_OFFS_USRL     0x14
00099 //#define MPU6050_RA_YG_OFFS_USRH     0x15 //[15:0] YG_OFFS_USR
00100 //#define MPU6050_RA_YG_OFFS_USRL     0x16
00101 //#define MPU6050_RA_ZG_OFFS_USRH     0x17 //[15:0] ZG_OFFS_USR
00102 //#define MPU6050_RA_ZG_OFFS_USRL     0x18
00103 //#define MPU6050_RA_SMPLRT_DIV       0x19
00104 //#define MPU6050_RA_CONFIG           0x1A
00105 //#define MPU6050_RA_GYRO_CONFIG      0x1B
00106 //#define MPU6050_RA_ACCEL_CONFIG     0x1C
00107 //#define MPU6050_RA_FF_THR           0x1D
00108 //#define MPU6050_RA_FF_DUR           0x1E
00109 //#define MPU6050_RA_MOT_THR          0x1F
00110 //#define MPU6050_RA_MOT_DUR          0x20
00111 //#define MPU6050_RA_ZRMOT_THR        0x21
00112 //#define MPU6050_RA_ZRMOT_DUR        0x22
00113 //#define MPU6050_RA_FIFO_EN          0x23
00114 //#define MPU6050_RA_I2C_MST_CTRL     0x24
00115 //#define MPU6050_RA_I2C_SLV0_ADDR    0x25
00116 //#define MPU6050_RA_I2C_SLV0_REG     0x26
00117 //#define MPU6050_RA_I2C_SLV0_CTRL    0x27
00118 //#define MPU6050_RA_I2C_SLV1_ADDR    0x28
00119 //#define MPU6050_RA_I2C_SLV1_REG     0x29
00120 //#define MPU6050_RA_I2C_SLV1_CTRL    0x2A
00121 //#define MPU6050_RA_I2C_SLV2_ADDR    0x2B
00122 //#define MPU6050_RA_I2C_SLV2_REG     0x2C
00123 //#define MPU6050_RA_I2C_SLV2_CTRL    0x2D
00124 //#define MPU6050_RA_I2C_SLV3_ADDR    0x2E
00125 //#define MPU6050_RA_I2C_SLV3_REG     0x2F
00126 //#define MPU6050_RA_I2C_SLV3_CTRL    0x30
00127 //#define MPU6050_RA_I2C_SLV4_ADDR    0x31
00128 //#define MPU6050_RA_I2C_SLV4_REG     0x32
00129 //#define MPU6050_RA_I2C_SLV4_DO      0x33
00130 //#define MPU6050_RA_I2C_SLV4_CTRL    0x34
00131 //#define MPU6050_RA_I2C_SLV4_DI      0x35
00132 //#define MPU6050_RA_I2C_MST_STATUS   0x36
00133 //#define MPU6050_RA_INT_PIN_CFG      0x37
00134 //#define MPU6050_RA_INT_ENABLE       0x38
00135 //#define MPU6050_RA_DMP_INT_STATUS   0x39
00136 //#define MPU6050_RA_INT_STATUS       0x3A
00137 //#define MPU6050_RA_ACCEL_XOUT_H     0x3B
00138 //#define MPU6050_RA_ACCEL_XOUT_L     0x3C
00139 //#define MPU6050_RA_ACCEL_YOUT_H     0x3D
00140 //#define MPU6050_RA_ACCEL_YOUT_L     0x3E
00141 //#define MPU6050_RA_ACCEL_ZOUT_H     0x3F
00142 //#define MPU6050_RA_ACCEL_ZOUT_L     0x40
00143 //#define MPU6050_RA_TEMP_OUT_H       0x41
00144 //#define MPU6050_RA_TEMP_OUT_L       0x42
00145 //#define MPU6050_RA_GYRO_XOUT_H      0x43
00146 //#define MPU6050_RA_GYRO_XOUT_L      0x44
00147 //#define MPU6050_RA_GYRO_YOUT_H      0x45
00148 //#define MPU6050_RA_GYRO_YOUT_L      0x46
00149 //#define MPU6050_RA_GYRO_ZOUT_H      0x47
00150 //#define MPU6050_RA_GYRO_ZOUT_L      0x48
00151 //#define MPU6050_RA_EXT_SENS_DATA_00 0x49
00152 //#define MPU6050_RA_EXT_SENS_DATA_01 0x4A
00153 //#define MPU6050_RA_EXT_SENS_DATA_02 0x4B
00154 //#define MPU6050_RA_EXT_SENS_DATA_03 0x4C
00155 //#define MPU6050_RA_EXT_SENS_DATA_04 0x4D
00156 //#define MPU6050_RA_EXT_SENS_DATA_05 0x4E
00157 //#define MPU6050_RA_EXT_SENS_DATA_06 0x4F
00158 //#define MPU6050_RA_EXT_SENS_DATA_07 0x50
00159 //#define MPU6050_RA_EXT_SENS_DATA_08 0x51
00160 //#define MPU6050_RA_EXT_SENS_DATA_09 0x52
00161 //#define MPU6050_RA_EXT_SENS_DATA_10 0x53
00162 //#define MPU6050_RA_EXT_SENS_DATA_11 0x54
00163 //#define MPU6050_RA_EXT_SENS_DATA_12 0x55
00164 //#define MPU6050_RA_EXT_SENS_DATA_13 0x56
00165 //#define MPU6050_RA_EXT_SENS_DATA_14 0x57
00166 //#define MPU6050_RA_EXT_SENS_DATA_15 0x58
00167 //#define MPU6050_RA_EXT_SENS_DATA_16 0x59
00168 //#define MPU6050_RA_EXT_SENS_DATA_17 0x5A
00169 //#define MPU6050_RA_EXT_SENS_DATA_18 0x5B
00170 //#define MPU6050_RA_EXT_SENS_DATA_19 0x5C
00171 //#define MPU6050_RA_EXT_SENS_DATA_20 0x5D
00172 //#define MPU6050_RA_EXT_SENS_DATA_21 0x5E
00173 //#define MPU6050_RA_EXT_SENS_DATA_22 0x5F
00174 //#define MPU6050_RA_EXT_SENS_DATA_23 0x60
00175 //#define MPU6050_RA_MOT_DETECT_STATUS    0x61
00176 //#define MPU6050_RA_I2C_SLV0_DO      0x63
00177 //#define MPU6050_RA_I2C_SLV1_DO      0x64
00178 //#define MPU6050_RA_I2C_SLV2_DO      0x65
00179 //#define MPU6050_RA_I2C_SLV3_DO      0x66
00180 //#define MPU6050_RA_I2C_MST_DELAY_CTRL   0x67
00181 //#define MPU6050_RA_SIGNAL_PATH_RESET    0x68
00182 //#define MPU6050_RA_MOT_DETECT_CTRL      0x69
00183 //#define MPU6050_RA_USER_CTRL        0x6A
00184 //#define MPU6050_RA_PWR_MGMT_1       0x6B
```

```
00185 //#define MPU6050_RA_PWR_MGMT_2       0x6C
00186 //#define MPU6050_RA_BANK_SEL         0x6D
00187 //#define MPU6050_RA_MEM_START_ADDR   0x6E
00188 //#define MPU6050_RA_MEM_R_W          0x6F
00189 //#define MPU6050_RA_DMP_CFG_1        0x70
00190 //#define MPU6050_RA_DMP_CFG_2        0x71
00191 //#define MPU6050_RA_FIFO_COUNTH      0x72
00192 //#define MPU6050_RA_FIFO_COUNTL      0x73
00193 //#define MPU6050_RA_FIFO_R_W         0x74
00194 //#define MPU6050_RA_WHO_AM_I         0x75
00195 //
00196 //#define MPU6050_TC_PWR_MODE_BIT     7
00197 //#define MPU6050_TC_OFFSET_BIT       6
00198 //#define MPU6050_TC_OFFSET_LENGTH    6
00199 //#define MPU6050_TC_OTP_BNK_VLD_BIT  0
00200 //
00201 //#define MPU6050_VDDIO_LEVEL_VLOGIC  0
00202 //#define MPU6050_VDDIO_LEVEL_VDD     1
00203 //
00204 //#define MPU6050_CFG_EXT_SYNC_SET_BIT    5
00205 //#define MPU6050_CFG_EXT_SYNC_SET_LENGTH 3
00206 //#define MPU6050_CFG_DLPF_CFG_BIT    2
00207 //#define MPU6050_CFG_DLPF_CFG_LENGTH 3
00208 //
00209 //#define MPU6050_EXT_SYNC_DISABLED       0x0
00210 //#define MPU6050_EXT_SYNC_TEMP_OUT_L     0x1
00211 //#define MPU6050_EXT_SYNC_GYRO_XOUT_L    0x2
00212 //#define MPU6050_EXT_SYNC_GYRO_YOUT_L    0x3
00213 //#define MPU6050_EXT_SYNC_GYRO_ZOUT_L    0x4
00214 //#define MPU6050_EXT_SYNC_ACCEL_XOUT_L   0x5
00215 //#define MPU6050_EXT_SYNC_ACCEL_YOUT_L   0x6
00216 //#define MPU6050_EXT_SYNC_ACCEL_ZOUT_L   0x7
00217 //
00218 //#define MPU6050_DLPF_BW_256         0x00
00219 //#define MPU6050_DLPF_BW_188         0x01
00220 //#define MPU6050_DLPF_BW_98          0x02
00221 //#define MPU6050_DLPF_BW_42          0x03
00222 //#define MPU6050_DLPF_BW_20          0x04
00223 //#define MPU6050_DLPF_BW_10          0x05
00224 //#define MPU6050_DLPF_BW_5           0x06
00225 //
00226 //#define MPU6050_GCONFIG_FS_SEL_BIT      4
00227 //#define MPU6050_GCONFIG_FS_SEL_LENGTH   2
00228 //
00229 //#define MPU6050_GYRO_FS_250         0x00
00230 //#define MPU6050_GYRO_FS_500         0x01
00231 //#define MPU6050_GYRO_FS_1000        0x02
00232 //#define MPU6050_GYRO_FS_2000        0x03
00233 //
00234 //#define MPU6050_ACONFIG_XA_ST_BIT           7
00235 //#define MPU6050_ACONFIG_YA_ST_BIT           6
00236 //#define MPU6050_ACONFIG_ZA_ST_BIT           5
00237 //#define MPU6050_ACONFIG_AFS_SEL_BIT         4
00238 //#define MPU6050_ACONFIG_AFS_SEL_LENGTH      2
00239 //#define MPU6050_ACONFIG_ACCEL_HPF_BIT       2
00240 //#define MPU6050_ACONFIG_ACCEL_HPF_LENGTH    3
00241 //
00242 //#define MPU6050_ACCEL_FS_2          0x00
00243 //#define MPU6050_ACCEL_FS_4          0x01
00244 //#define MPU6050_ACCEL_FS_8          0x02
00245 //#define MPU6050_ACCEL_FS_16         0x03
00246 //
00247 //#define MPU6050_DHPF_RESET          0x00
00248 //#define MPU6050_DHPF_5              0x01
00249 //#define MPU6050_DHPF_2P5            0x02
00250 //#define MPU6050_DHPF_1P25           0x03
00251 //#define MPU6050_DHPF_0P63           0x04
00252 //#define MPU6050_DHPF_HOLD           0x07
00253 //
00254 //#define MPU6050_TEMP_FIFO_EN_BIT    7
00255 //#define MPU6050_XG_FIFO_EN_BIT      6
00256 //#define MPU6050_YG_FIFO_EN_BIT      5
00257 //#define MPU6050_ZG_FIFO_EN_BIT      4
00258 //#define MPU6050_ACCEL_FIFO_EN_BIT   3
00259 //#define MPU6050_SLV2_FIFO_EN_BIT    2
00260 //#define MPU6050_SLV1_FIFO_EN_BIT    1
00261 //#define MPU6050_SLV0_FIFO_EN_BIT    0
00262 //
00263 //#define MPU6050_MULT_MST_EN_BIT     7
00264 //#define MPU6050_WAIT_FOR_ES_BIT     6
00265 //#define MPU6050_SLV_3_FIFO_EN_BIT   5
00266 //#define MPU6050_I2C_MST_P_NSR_BIT   4
00267 //#define MPU6050_I2C_MST_CLK_BIT     3
00268 //#define MPU6050_I2C_MST_CLK_LENGTH  4
00269 //
00270 //#define MPU6050_CLOCK_DIV_348       0x0
00271 //#define MPU6050_CLOCK_DIV_333       0x1
```

```
00272 //#define MPU6050_CLOCK_DIV_320       0x2
00273 //#define MPU6050_CLOCK_DIV_308       0x3
00274 //#define MPU6050_CLOCK_DIV_296       0x4
00275 //#define MPU6050_CLOCK_DIV_286       0x5
00276 //#define MPU6050_CLOCK_DIV_276       0x6
00277 //#define MPU6050_CLOCK_DIV_267       0x7
00278 //#define MPU6050_CLOCK_DIV_258       0x8
00279 //#define MPU6050_CLOCK_DIV_500       0x9
00280 //#define MPU6050_CLOCK_DIV_471       0xA
00281 //#define MPU6050_CLOCK_DIV_444       0xB
00282 //#define MPU6050_CLOCK_DIV_421       0xC
00283 //#define MPU6050_CLOCK_DIV_400       0xD
00284 //#define MPU6050_CLOCK_DIV_381       0xE
00285 //#define MPU6050_CLOCK_DIV_364       0xF
00286 //
00287 //#define MPU6050_I2C_SLV_RW_BIT      7
00288 //#define MPU6050_I2C_SLV_ADDR_BIT    6
00289 //#define MPU6050_I2C_SLV_ADDR_LENGTH 7
00290 //#define MPU6050_I2C_SLV_EN_BIT      7
00291 //#define MPU6050_I2C_SLV_BYTE_SW_BIT 6
00292 //#define MPU6050_I2C_SLV_REG_DIS_BIT 5
00293 //#define MPU6050_I2C_SLV_GRP_BIT     4
00294 //#define MPU6050_I2C_SLV_LEN_BIT     3
00295 //#define MPU6050_I2C_SLV_LEN_LENGTH  4
00296 //
00297 //#define MPU6050_I2C_SLV4_RW_BIT         7
00298 //#define MPU6050_I2C_SLV4_ADDR_BIT       6
00299 //#define MPU6050_I2C_SLV4_ADDR_LENGTH    7
00300 //#define MPU6050_I2C_SLV4_EN_BIT         7
00301 //#define MPU6050_I2C_SLV4_INT_EN_BIT     6
00302 //#define MPU6050_I2C_SLV4_REG_DIS_BIT    5
00303 //#define MPU6050_I2C_SLV4_MST_DLY_BIT    4
00304 //#define MPU6050_I2C_SLV4_MST_DLY_LENGTH 5
00305 //
00306 //#define MPU6050_MST_PASS_THROUGH_BIT    7
00307 //#define MPU6050_MST_I2C_SLV4_DONE_BIT   6
00308 //#define MPU6050_MST_I2C_LOST_ARB_BIT    5
00309 //#define MPU6050_MST_I2C_SLV4_NACK_BIT   4
00310 //#define MPU6050_MST_I2C_SLV3_NACK_BIT   3
00311 //#define MPU6050_MST_I2C_SLV2_NACK_BIT   2
00312 //#define MPU6050_MST_I2C_SLV1_NACK_BIT   1
00313 //#define MPU6050_MST_I2C_SLV0_NACK_BIT   0
00314 //
00315 //#define MPU6050_INTCFG_INT_LEVEL_BIT        7
00316 //#define MPU6050_INTCFG_INT_OPEN_BIT         6
00317 //#define MPU6050_INTCFG_LATCH_INT_EN_BIT     5
00318 //#define MPU6050_INTCFG_INT_RD_CLEAR_BIT     4
00319 //#define MPU6050_INTCFG_FSYNC_INT_LEVEL_BIT  3
00320 //#define MPU6050_INTCFG_FSYNC_INT_EN_BIT     2
00321 //#define MPU6050_INTCFG_I2C_BYPASS_EN_BIT    1
00322 //#define MPU6050_INTCFG_CLKOUT_EN_BIT        0
00323 //
00324 //#define MPU6050_INTMODE_ACTIVEHIGH  0x00
00325 //#define MPU6050_INTMODE_ACTIVELOW   0x01
00326 //
00327 //#define MPU6050_INTDRV_PUSHPULL     0x00
00328 //#define MPU6050_INTDRV_OPENDRAIN    0x01
00329 //
00330 //#define MPU6050_INTLATCH_50USPULSE  0x00
00331 //#define MPU6050_INTLATCH_WAITCLEAR  0x01
00332 //
00333 //#define MPU6050_INTCLEAR_STATUSREAD 0x00
00334 //#define MPU6050_INTCLEAR_ANYREAD    0x01
00335 //
00336 //#define MPU6050_INTERRUPT_FF_BIT            7
00337 //#define MPU6050_INTERRUPT_MOT_BIT           6
00338 //#define MPU6050_INTERRUPT_ZMOT_BIT          5
00339 //#define MPU6050_INTERRUPT_FIFO_OFLOW_BIT    4
00340 //#define MPU6050_INTERRUPT_I2C_MST_INT_BIT   3
00341 //#define MPU6050_INTERRUPT_PLL_RDY_INT_BIT   2
00342 //#define MPU6050_INTERRUPT_DMP_INT_BIT       1
00343 //#define MPU6050_INTERRUPT_DATA_RDY_BIT      0
00344 //
00347 //#define MPU6050_DMPINT_5_BIT            5
00348 //#define MPU6050_DMPINT_4_BIT            4
00349 //#define MPU6050_DMPINT_3_BIT            3
00350 //#define MPU6050_DMPINT_2_BIT            2
00351 //#define MPU6050_DMPINT_1_BIT            1
00352 //#define MPU6050_DMPINT_0_BIT            0
00353 //
00354 //#define MPU6050_MOTION_MOT_XNEG_BIT     7
00355 //#define MPU6050_MOTION_MOT_XPOS_BIT     6
00356 //#define MPU6050_MOTION_MOT_YNEG_BIT     5
00357 //#define MPU6050_MOTION_MOT_YPOS_BIT     4
00358 //#define MPU6050_MOTION_MOT_ZNEG_BIT     3
00359 //#define MPU6050_MOTION_MOT_ZPOS_BIT     2
00360 //#define MPU6050_MOTION_MOT_ZRMOT_BIT    0
```

```
00361 //
00362 //#define MPU6050_DELAYCTRL_DELAY_ES_SHADOW_BIT   7
00363 //#define MPU6050_DELAYCTRL_I2C_SLV4_DLY_EN_BIT   4
00364 //#define MPU6050_DELAYCTRL_I2C_SLV3_DLY_EN_BIT   3
00365 //#define MPU6050_DELAYCTRL_I2C_SLV2_DLY_EN_BIT   2
00366 //#define MPU6050_DELAYCTRL_I2C_SLV1_DLY_EN_BIT   1
00367 //#define MPU6050_DELAYCTRL_I2C_SLV0_DLY_EN_BIT   0
00368 //
00369 //#define MPU6050_PATHRESET_GYRO_RESET_BIT    2
00370 //#define MPU6050_PATHRESET_ACCEL_RESET_BIT   1
00371 //#define MPU6050_PATHRESET_TEMP_RESET_BIT    0
00372 //
00373 //#define MPU6050_DETECT_ACCEL_ON_DELAY_BIT       5
00374 //#define MPU6050_DETECT_ACCEL_ON_DELAY_LENGTH    2
00375 //#define MPU6050_DETECT_FF_COUNT_BIT             3
00376 //#define MPU6050_DETECT_FF_COUNT_LENGTH          2
00377 //#define MPU6050_DETECT_MOT_COUNT_BIT            1
00378 //#define MPU6050_DETECT_MOT_COUNT_LENGTH         2
00379 //
00380 //#define MPU6050_DETECT_DECREMENT_RESET  0x0
00381 //#define MPU6050_DETECT_DECREMENT_1      0x1
00382 //#define MPU6050_DETECT_DECREMENT_2      0x2
00383 //#define MPU6050_DETECT_DECREMENT_4      0x3
00384 //
00385 //#define MPU6050_USERCTRL_DMP_EN_BIT             7
00386 //#define MPU6050_USERCTRL_FIFO_EN_BIT            6
00387 //#define MPU6050_USERCTRL_I2C_MST_EN_BIT         5
00388 //#define MPU6050_USERCTRL_I2C_IF_DIS_BIT         4
00389 //#define MPU6050_USERCTRL_DMP_RESET_BIT          3
00390 //#define MPU6050_USERCTRL_FIFO_RESET_BIT         2
00391 //#define MPU6050_USERCTRL_I2C_MST_RESET_BIT      1
00392 //#define MPU6050_USERCTRL_SIG_COND_RESET_BIT     0
00393 //
00394 //#define MPU6050_PWR1_DEVICE_RESET_BIT   7
00395 //#define MPU6050_PWR1_SLEEP_BIT          6
00396 //#define MPU6050_PWR1_CYCLE_BIT          5
00397 //#define MPU6050_PWR1_TEMP_DIS_BIT       3
00398 //#define MPU6050_PWR1_CLKSEL_BIT         2
00399 //#define MPU6050_PWR1_CLKSEL_LENGTH      3
00400 //
00401 //#define MPU6050_CLOCK_INTERNAL          0x00
00402 //#define MPU6050_CLOCK_PLL_XGYRO         0x01
00403 //#define MPU6050_CLOCK_PLL_YGYRO         0x02
00404 //#define MPU6050_CLOCK_PLL_ZGYRO         0x03
00405 //#define MPU6050_CLOCK_PLL_EXT32K        0x04
00406 //#define MPU6050_CLOCK_PLL_EXT19M        0x05
00407 //#define MPU6050_CLOCK_KEEP_RESET        0x07
00408 //
00409 //#define MPU6050_PWR2_LP_WAKE_CTRL_BIT       7
00410 //#define MPU6050_PWR2_LP_WAKE_CTRL_LENGTH    2
00411 //#define MPU6050_PWR2_STBY_XA_BIT            5
00412 //#define MPU6050_PWR2_STBY_YA_BIT            4
00413 //#define MPU6050_PWR2_STBY_ZA_BIT            3
00414 //#define MPU6050_PWR2_STBY_XG_BIT            2
00415 //#define MPU6050_PWR2_STBY_YG_BIT            1
00416 //#define MPU6050_PWR2_STBY_ZG_BIT            0
00417 //
00418 //#define MPU6050_WAKE_FREQ_1P25      0x0
00419 //#define MPU6050_WAKE_FREQ_2P5       0x1
00420 //#define MPU6050_WAKE_FREQ_5         0x2
00421 //#define MPU6050_WAKE_FREQ_10        0x3
00422 //
00423 //#define MPU6050_BANKSEL_PRFTCH_EN_BIT       6
00424 //#define MPU6050_BANKSEL_CFG_USER_BANK_BIT   5
00425 //#define MPU6050_BANKSEL_MEM_SEL_BIT         4
00426 //#define MPU6050_BANKSEL_MEM_SEL_LENGTH      5
00427 //
00428 //#define MPU6050_WHO_AM_I_BIT        6
00429 //#define MPU6050_WHO_AM_I_LENGTH     6
00430 //
00431 //#define MPU6050_DMP_MEMORY_BANKS        8
00432 //#define MPU6050_DMP_MEMORY_BANK_SIZE    256
00433 //#define MPU6050_DMP_MEMORY_CHUNK_SIZE   16
00434 //
00436 //
00437 //class MPU6050 {
00438 //    public:
00439 //        MPU6050();
00440 //        MPU6050(uint8_t address);
00441 //
00442 //        void initialize();
00443 //        bool testConnection();
00444 //
00445 //        // AUX_VDDIO register
00446 //        uint8_t getAuxVDDIOLevel();
00447 //        void setAuxVDDIOLevel(uint8_t level);
00448 //
```

```
00449 //         // SMPLRT_DIV register
00450 //         uint8_t getRate();
00451 //         void setRate(uint8_t rate);
00452 //
00453 //         // CONFIG register
00454 //         uint8_t getExternalFrameSync();
00455 //         void setExternalFrameSync(uint8_t sync);
00456 //         uint8_t getDLPFMode();
00457 //         void setDLPFMode(uint8_t bandwidth);
00458 //
00459 //         // GYRO_CONFIG register
00460 //         uint8_t getFullScaleGyroRange();
00461 //         void setFullScaleGyroRange(uint8_t range);
00462 //
00463 //         // ACCEL_CONFIG register
00464 //         bool getAccelXSelfTest();
00465 //         void setAccelXSelfTest(bool enabled);
00466 //         bool getAccelYSelfTest();
00467 //         void setAccelYSelfTest(bool enabled);
00468 //         bool getAccelZSelfTest();
00469 //         void setAccelZSelfTest(bool enabled);
00470 //         uint8_t getFullScaleAccelRange();
00471 //         void setFullScaleAccelRange(uint8_t range);
00472 //         uint8_t getDHPFMode();
00473 //         void setDHPFMode(uint8_t mode);
00474 //
00475 //         // FF_THR register
00476 //         uint8_t getFreefallDetectionThreshold();
00477 //         void setFreefallDetectionThreshold(uint8_t threshold);
00478 //
00479 //         // FF_DUR register
00480 //         uint8_t getFreefallDetectionDuration();
00481 //         void setFreefallDetectionDuration(uint8_t duration);
00482 //
00483 //         // MOT_THR register
00484 //         uint8_t getMotionDetectionThreshold();
00485 //         void setMotionDetectionThreshold(uint8_t threshold);
00486 //
00487 //         // MOT_DUR register
00488 //         uint8_t getMotionDetectionDuration();
00489 //         void setMotionDetectionDuration(uint8_t duration);
00490 //
00491 //         // ZRMOT_THR register
00492 //         uint8_t getZeroMotionDetectionThreshold();
00493 //         void setZeroMotionDetectionThreshold(uint8_t threshold);
00494 //
00495 //         // ZRMOT_DUR register
00496 //         uint8_t getZeroMotionDetectionDuration();
00497 //         void setZeroMotionDetectionDuration(uint8_t duration);
00498 //
00499 //         // FIFO_EN register
00500 //         bool getTempFIFOEnabled();
00501 //         void setTempFIFOEnabled(bool enabled);
00502 //         bool getXGyroFIFOEnabled();
00503 //         void setXGyroFIFOEnabled(bool enabled);
00504 //         bool getYGyroFIFOEnabled();
00505 //         void setYGyroFIFOEnabled(bool enabled);
00506 //         bool getZGyroFIFOEnabled();
00507 //         void setZGyroFIFOEnabled(bool enabled);
00508 //         bool getAccelFIFOEnabled();
00509 //         void setAccelFIFOEnabled(bool enabled);
00510 //         bool getSlave2FIFOEnabled();
00511 //         void setSlave2FIFOEnabled(bool enabled);
00512 //         bool getSlave1FIFOEnabled();
00513 //         void setSlave1FIFOEnabled(bool enabled);
00514 //         bool getSlave0FIFOEnabled();
00515 //         void setSlave0FIFOEnabled(bool enabled);
00516 //
00517 //         // I2C_MST_CTRL register
00518 //         bool getMultiMasterEnabled();
00519 //         void setMultiMasterEnabled(bool enabled);
00520 //         bool getWaitForExternalSensorEnabled();
00521 //         void setWaitForExternalSensorEnabled(bool enabled);
00522 //         bool getSlave3FIFOEnabled();
00523 //         void setSlave3FIFOEnabled(bool enabled);
00524 //         bool getSlaveReadWriteTransitionEnabled();
00525 //         void setSlaveReadWriteTransitionEnabled(bool enabled);
00526 //         uint8_t getMasterClockSpeed();
00527 //         void setMasterClockSpeed(uint8_t speed);
00528 //
00529 //         // I2C_SLV* registers (Slave 0-3)
00530 //         uint8_t getSlaveAddress(uint8_t num);
00531 //         void setSlaveAddress(uint8_t num, uint8_t address);
00532 //         uint8_t getSlaveRegister(uint8_t num);
00533 //         void setSlaveRegister(uint8_t num, uint8_t reg);
00534 //         bool getSlaveEnabled(uint8_t num);
00535 //         void setSlaveEnabled(uint8_t num, bool enabled);
```

```
00536 //        bool getSlaveWordByteSwap(uint8_t num);
00537 //        void setSlaveWordByteSwap(uint8_t num, bool enabled);
00538 //        bool getSlaveWriteMode(uint8_t num);
00539 //        void setSlaveWriteMode(uint8_t num, bool mode);
00540 //        bool getSlaveWordGroupOffset(uint8_t num);
00541 //        void setSlaveWordGroupOffset(uint8_t num, bool enabled);
00542 //        uint8_t getSlaveDataLength(uint8_t num);
00543 //        void setSlaveDataLength(uint8_t num, uint8_t length);
00544 //
00545 //        // I2C_SLV* registers (Slave 4)
00546 //        uint8_t getSlave4Address();
00547 //        void setSlave4Address(uint8_t address);
00548 //        uint8_t getSlave4Register();
00549 //        void setSlave4Register(uint8_t reg);
00550 //        void setSlave4OutputByte(uint8_t data);
00551 //        bool getSlave4Enabled();
00552 //        void setSlave4Enabled(bool enabled);
00553 //        bool getSlave4InterruptEnabled();
00554 //        void setSlave4InterruptEnabled(bool enabled);
00555 //        bool getSlave4WriteMode();
00556 //        void setSlave4WriteMode(bool mode);
00557 //        uint8_t getSlave4MasterDelay();
00558 //        void setSlave4MasterDelay(uint8_t delay);
00559 //        uint8_t getSlate4InputByte();
00560 //
00561 //        // I2C_MST_STATUS register
00562 //        bool getPassthroughStatus();
00563 //        bool getSlave4IsDone();
00564 //        bool getLostArbitration();
00565 //        bool getSlave4Nack();
00566 //        bool getSlave3Nack();
00567 //        bool getSlave2Nack();
00568 //        bool getSlave1Nack();
00569 //        bool getSlave0Nack();
00570 //
00571 //        // INT_PIN_CFG register
00572 //        bool getInterruptMode();
00573 //        void setInterruptMode(bool mode);
00574 //        bool getInterruptDrive();
00575 //        void setInterruptDrive(bool drive);
00576 //        bool getInterruptLatch();
00577 //        void setInterruptLatch(bool latch);
00578 //        bool getInterruptLatchClear();
00579 //        void setInterruptLatchClear(bool clear);
00580 //        bool getFSyncInterruptLevel();
00581 //        void setFSyncInterruptLevel(bool level);
00582 //        bool getFSyncInterruptEnabled();
00583 //        void setFSyncInterruptEnabled(bool enabled);
00584 //        bool getI2CBypassEnabled();
00585 //        void setI2CBypassEnabled(bool enabled);
00586 //        bool getClockOutputEnabled();
00587 //        void setClockOutputEnabled(bool enabled);
00588 //
00589 //        // INT_ENABLE register
00590 //        uint8_t getIntEnabled();
00591 //        void setIntEnabled(uint8_t enabled);
00592 //        bool getIntFreefallEnabled();
00593 //        void setIntFreefallEnabled(bool enabled);
00594 //        bool getIntMotionEnabled();
00595 //        void setIntMotionEnabled(bool enabled);
00596 //        bool getIntZeroMotionEnabled();
00597 //        void setIntZeroMotionEnabled(bool enabled);
00598 //        bool getIntFIFOBufferOverflowEnabled();
00599 //        void setIntFIFOBufferOverflowEnabled(bool enabled);
00600 //        bool getIntI2CMasterEnabled();
00601 //        void setIntI2CMasterEnabled(bool enabled);
00602 //        bool getIntDataReadyEnabled();
00603 //        void setIntDataReadyEnabled(bool enabled);
00604 //
00605 //        // INT_STATUS register
00606 //        uint8_t getIntStatus();
00607 //        bool getIntFreefallStatus();
00608 //        bool getIntMotionStatus();
00609 //        bool getIntZeroMotionStatus();
00610 //        bool getIntFIFOBufferOverflowStatus();
00611 //        bool getIntI2CMasterStatus();
00612 //        bool getIntDataReadyStatus();
00613 //
00614 //        // ACCEL_*OUT_* registers
00615 //        void getMotion9(int16_t* ax, int16_t* ay, int16_t* az, int16_t* gx, int16_t* gy, int16_t* gz,
      int16_t* mx, int16_t* my, int16_t* mz);
00616 //        void getMotion6(int16_t* ax, int16_t* ay, int16_t* az, int16_t* gx, int16_t* gy, int16_t* gz);
00617 //        void getAcceleration(int16_t* x, int16_t* y, int16_t* z);
00618 //        int16_t getAccelerationX();
00619 //        int16_t getAccelerationY();
00620 //        int16_t getAccelerationZ();
00621 //
```

```
00622 //         // TEMP_OUT_* registers
00623 //         int16_t getTemperature();
00624 //
00625 //         // GYRO_*OUT_* registers
00626 //         void getRotation(int16_t* x, int16_t* y, int16_t* z);
00627 //         void getRotationXY(int16_t* x, int16_t* y);
00628 //         int16_t getRotationX();
00629 //         int16_t getRotationY();
00630 //         int16_t getRotationZ();
00631 //
00632 //         // EXT_SENS_DATA_* registers
00633 //         uint8_t getExternalSensorByte(int position);
00634 //         uint16_t getExternalSensorWord(int position);
00635 //         uint32_t getExternalSensorDWord(int position);
00636 //
00637 //         // MOT_DETECT_STATUS register
00638 //         bool getXNegMotionDetected();
00639 //         bool getXPosMotionDetected();
00640 //         bool getYNegMotionDetected();
00641 //         bool getYPosMotionDetected();
00642 //         bool getZNegMotionDetected();
00643 //         bool getZPosMotionDetected();
00644 //         bool getZeroMotionDetected();
00645 //
00646 //         // I2C_SLV*_DO register
00647 //         void setSlaveOutputByte(uint8_t num, uint8_t data);
00648 //
00649 //         // I2C_MST_DELAY_CTRL register
00650 //         bool getExternalShadowDelayEnabled();
00651 //         void setExternalShadowDelayEnabled(bool enabled);
00652 //         bool getSlaveDelayEnabled(uint8_t num);
00653 //         void setSlaveDelayEnabled(uint8_t num, bool enabled);
00654 //
00655 //         // SIGNAL_PATH_RESET register
00656 //         void resetGyroscopePath();
00657 //         void resetAccelerometerPath();
00658 //         void resetTemperaturePath();
00659 //
00660 //         // MOT_DETECT_CTRL register
00661 //         uint8_t getAccelerometerPowerOnDelay();
00662 //         void setAccelerometerPowerOnDelay(uint8_t delay);
00663 //         uint8_t getFreefallDetectionCounterDecrement();
00664 //         void setFreefallDetectionCounterDecrement(uint8_t decrement);
00665 //         uint8_t getMotionDetectionCounterDecrement();
00666 //         void setMotionDetectionCounterDecrement(uint8_t decrement);
00667 //
00668 //         // USER_CTRL register
00669 //         bool getFIFOEnabled();
00670 //         void setFIFOEnabled(bool enabled);
00671 //         bool getI2CMasterModeEnabled();
00672 //         void setI2CMasterModeEnabled(bool enabled);
00673 //         void switchSPIEnabled(bool enabled);
00674 //         void resetFIFO();
00675 //         void resetI2CMaster();
00676 //         void resetSensors();
00677 //
00678 //         // PWR_MGMT_1 register
00679 //         void reset();
00680 //         bool getSleepEnabled();
00681 //         void setSleepEnabled(bool enabled);
00682 //         bool getWakeCycleEnabled();
00683 //         void setWakeCycleEnabled(bool enabled);
00684 //         bool getTempSensorEnabled();
00685 //         void setTempSensorEnabled(bool enabled);
00686 //         uint8_t getClockSource();
00687 //         void setClockSource(uint8_t source);
00688 //
00689 //         // PWR_MGMT_2 register
00690 //         uint8_t getWakeFrequency();
00691 //         void setWakeFrequency(uint8_t frequency);
00692 //         bool getStandbyXAccelEnabled();
00693 //         void setStandbyXAccelEnabled(bool enabled);
00694 //         bool getStandbyYAccelEnabled();
00695 //         void setStandbyYAccelEnabled(bool enabled);
00696 //         bool getStandbyZAccelEnabled();
00697 //         void setStandbyZAccelEnabled(bool enabled);
00698 //         bool getStandbyXGyroEnabled();
00699 //         void setStandbyXGyroEnabled(bool enabled);
00700 //         bool getStandbyYGyroEnabled();
00701 //         void setStandbyYGyroEnabled(bool enabled);
00702 //         bool getStandbyZGyroEnabled();
00703 //         void setStandbyZGyroEnabled(bool enabled);
00704 //
00705 //         // FIFO_COUNT_* registers
00706 //         uint16_t getFIFOCount();
00707 //
00708 //         // FIFO_R_W register
```

```
00709 //        uint8_t getFIFOByte();
00710 //        void setFIFOByte(uint8_t data);
00711 //        void getFIFOBytes(uint8_t *data, uint8_t length);
00712 //
00713 //        // WHO_AM_I register
00714 //        uint8_t getDeviceID();
00715 //        void setDeviceID(uint8_t id);
00716 //
00717 //        // ======== UNDOCUMENTED/DMP REGISTERS/METHODS ========
00718 //
00719 //        // XG_OFFS_TC register
00720 //        uint8_t getOTPBankValid();
00721 //        void setOTPBankValid(bool enabled);
00722 //        int8_t getXGyroOffset();
00723 //        void setXGyroOffset(int8_t offset);
00724 //
00725 //        // YG_OFFS_TC register
00726 //        int8_t getYGyroOffset();
00727 //        void setYGyroOffset(int8_t offset);
00728 //
00729 //        // ZG_OFFS_TC register
00730 //        int8_t getZGyroOffset();
00731 //        void setZGyroOffset(int8_t offset);
00732 //
00733 //        // X_FINE_GAIN register
00734 //        int8_t getXFineGain();
00735 //        void setXFineGain(int8_t gain);
00736 //
00737 //        // Y_FINE_GAIN register
00738 //        int8_t getYFineGain();
00739 //        void setYFineGain(int8_t gain);
00740 //
00741 //        // Z_FINE_GAIN register
00742 //        int8_t getZFineGain();
00743 //        void setZFineGain(int8_t gain);
00744 //
00745 //        // XA_OFFS_* registers
00746 //        int16_t getXAccelOffset();
00747 //        void setXAccelOffset(int16_t offset);
00748 //
00749 //        // YA_OFFS_* register
00750 //        int16_t getYAccelOffset();
00751 //        void setYAccelOffset(int16_t offset);
00752 //
00753 //        // ZA_OFFS_* register
00754 //        int16_t getZAccelOffset();
00755 //        void setZAccelOffset(int16_t offset);
00756 //
00757 //        // XG_OFFS_USR* registers
00758 //        int16_t getXGyroOffsetUser();
00759 //        void setXGyroOffsetUser(int16_t offset);
00760 //
00761 //        // YG_OFFS_USR* register
00762 //        int16_t getYGyroOffsetUser();
00763 //        void setYGyroOffsetUser(int16_t offset);
00764 //
00765 //        // ZG_OFFS_USR* register
00766 //        int16_t getZGyroOffsetUser();
00767 //        void setZGyroOffsetUser(int16_t offset);
00768 //
00769 //        // INT_ENABLE register (DMP functions)
00770 //        bool getIntPLLReadyEnabled();
00771 //        void setIntPLLReadyEnabled(bool enabled);
00772 //        bool getIntDMPEnabled();
00773 //        void setIntDMPEnabled(bool enabled);
00774 //
00775 //        // DMP_INT_STATUS
00776 //        bool getDMPInt5Status();
00777 //        bool getDMPInt4Status();
00778 //        bool getDMPInt3Status();
00779 //        bool getDMPInt2Status();
00780 //        bool getDMPInt1Status();
00781 //        bool getDMPInt0Status();
00782 //
00783 //        // INT_STATUS register (DMP functions)
00784 //        bool getIntPLLReadyStatus();
00785 //        bool getIntDMPStatus();
00786 //
00787 //        // USER_CTRL register (DMP functions)
00788 //        bool getDMPEnabled();
00789 //        void setDMPEnabled(bool enabled);
00790 //        void resetDMP();
00791 //
00792 //        // BANK_SEL register
00793 //        void setMemoryBank(uint8_t bank, bool prefetchEnabled=false, bool userBank=false);
00794 //
00795 //        // MEM_START_ADDR register
```

```
00796 //        void setMemoryStartAddress(uint8_t address);
00797 //
00798 //        // MEM_R_W register
00799 //        uint8_t readMemoryByte();
00800 //        void writeMemoryByte(uint8_t data);
00801 //        void readMemoryBlock(uint8_t *data, uint16_t dataSize, uint8_t bank=0, uint8_t address=0);
00802 //        bool writeMemoryBlock(const uint8_t *data, uint16_t dataSize, uint8_t bank=0, uint8_t address=0,
      bool verify=true, bool useProgMem=false);
00803 //        bool writeProgMemoryBlock(const uint8_t *data, uint16_t dataSize, uint8_t bank=0, uint8_t
      address=0, bool verify=true);
00804 //
00805 //        bool writeDMPConfigurationSet(const uint8_t *data, uint16_t dataSize, bool useProgMem=false);
00806 //        bool writeProgDMPConfigurationSet(const uint8_t *data, uint16_t dataSize);
00807 //
00808 //        // DMP_CFG_1 register
00809 //        uint8_t getDMPConfig1();
00810 //        void setDMPConfig1(uint8_t config);
00811 //
00812 //        // DMP_CFG_2 register
00813 //        uint8_t getDMPConfig2();
00814 //        void setDMPConfig2(uint8_t config);
00815 //
00816 //
00817 //        // special methods for MotionApps 2.0 implementation
00818 //        #ifdef MPU6050_INCLUDE_DMP_MOTIONAPPS20
00819 //            uint8_t *dmpPacketBuffer;
00820 //            uint16_t dmpPacketSize;
00821 //
00822 //            uint8_t dmpInitialize();
00823 //            bool dmpPacketAvailable();
00824 //
00825 //            uint8_t dmpSetFIFORate(uint8_t fifoRate);
00826 //            uint8_t dmpGetFIFORate();
00827 //            uint8_t dmpGetSampleStepSizeMS();
00828 //            uint8_t dmpGetSampleFrequency();
00829 //            int32_t dmpDecodeTemperature(int8_t tempReg);
00830 //
00831 //            // Register callbacks after a packet of FIFO data is processed
00832 //            //uint8_t dmpRegisterFIFORateProcess(inv_obj_func func, int16_t priority);
00833 //            //uint8_t dmpUnregisterFIFORateProcess(inv_obj_func func);
00834 //            uint8_t dmpRunFIFORateProcesses();
00835 //
00836 //            // Setup FIFO for various output
00837 //            uint8_t dmpSendQuaternion(uint_fast16_t accuracy);
00838 //            uint8_t dmpSendGyro(uint_fast16_t elements, uint_fast16_t accuracy);
00839 //            uint8_t dmpSendAccel(uint_fast16_t elements, uint_fast16_t accuracy);
00840 //            uint8_t dmpSendLinearAccel(uint_fast16_t elements, uint_fast16_t accuracy);
00841 //            uint8_t dmpSendLinearAccelInWorld(uint_fast16_t elements, uint_fast16_t accuracy);
00842 //            uint8_t dmpSendControlData(uint_fast16_t elements, uint_fast16_t accuracy);
00843 //            uint8_t dmpSendSensorData(uint_fast16_t elements, uint_fast16_t accuracy);
00844 //            uint8_t dmpSendExternalSensorData(uint_fast16_t elements, uint_fast16_t accuracy);
00845 //            uint8_t dmpSendGravity(uint_fast16_t elements, uint_fast16_t accuracy);
00846 //            uint8_t dmpSendPacketNumber(uint_fast16_t accuracy);
00847 //            uint8_t dmpSendQuantizedAccel(uint_fast16_t elements, uint_fast16_t accuracy);
00848 //            uint8_t dmpSendEIS(uint_fast16_t elements, uint_fast16_t accuracy);
00849 //
00850 //            // Get Fixed Point data from FIFO
00851 //            uint8_t dmpGetAccel(int32_t *data, const uint8_t* packet=0);
00852 //            uint8_t dmpGetAccel(int16_t *data, const uint8_t* packet=0);
00853 //            uint8_t dmpGetAccel(VectorInt16 *v, const uint8_t* packet=0);
00854 //            uint8_t dmpGetQuaternion(int32_t *data, const uint8_t* packet=0);
00855 //            uint8_t dmpGetQuaternion(int16_t *data, const uint8_t* packet=0);
00856 //            uint8_t dmpGetQuaternion(Quaternion *q, const uint8_t* packet=0);
00857 //            uint8_t dmpGet6AxisQuaternion(int32_t *data, const uint8_t* packet=0);
00858 //            uint8_t dmpGet6AxisQuaternion(int16_t *data, const uint8_t* packet=0);
00859 //            uint8_t dmpGet6AxisQuaternion(Quaternion *q, const uint8_t* packet=0);
00860 //            uint8_t dmpGetRelativeQuaternion(int32_t *data, const uint8_t* packet=0);
00861 //            uint8_t dmpGetRelativeQuaternion(int16_t *data, const uint8_t* packet=0);
00862 //            uint8_t dmpGetRelativeQuaternion(Quaternion *data, const uint8_t* packet=0);
00863 //            uint8_t dmpGetGyro(int32_t *data, const uint8_t* packet=0);
00864 //            uint8_t dmpGetGyro(int16_t *data, const uint8_t* packet=0);
00865 //            uint8_t dmpGetGyro(VectorInt16 *v, const uint8_t* packet=0);
00866 //            uint8_t dmpSetLinearAccelFilterCoefficient(float coef);
00867 //            uint8_t dmpGetLinearAccel(int32_t *data, const uint8_t* packet=0);
00868 //            uint8_t dmpGetLinearAccel(int16_t *data, const uint8_t* packet=0);
00869 //            uint8_t dmpGetLinearAccel(VectorInt16 *v, const uint8_t* packet=0);
00870 //            uint8_t dmpGetLinearAccel(VectorInt16 *v, VectorInt16 *vRaw, VectorFloat *gravity);
00871 //            uint8_t dmpGetLinearAccelInWorld(int32_t *data, const uint8_t* packet=0);
00872 //            uint8_t dmpGetLinearAccelInWorld(int16_t *data, const uint8_t* packet=0);
00873 //            uint8_t dmpGetLinearAccelInWorld(VectorInt16 *v, const uint8_t* packet=0);
00874 //            uint8_t dmpGetLinearAccelInWorld(VectorInt16 *v, VectorInt16 *vReal, Quaternion *q);
00875 //            uint8_t dmpGetGyroAndAccelSensor(int32_t *data, const uint8_t* packet=0);
00876 //            uint8_t dmpGetGyroAndAccelSensor(int16_t *data, const uint8_t* packet=0);
00877 //            uint8_t dmpGetGyroAndAccelSensor(VectorInt16 *g, VectorInt16 *a, const uint8_t* packet=0);
00878 //            uint8_t dmpGetGyroSensor(int32_t *data, const uint8_t* packet=0);
00879 //            uint8_t dmpGetGyroSensor(int16_t *data, const uint8_t* packet=0);
00880 //            uint8_t dmpGetGyroSensor(VectorInt16 *v, const uint8_t* packet=0);
```

```
00881 //            uint8_t dmpGetControlData(int32_t *data, const uint8_t* packet=0);
00882 //            uint8_t dmpGetTemperature(int32_t *data, const uint8_t* packet=0);
00883 //            uint8_t dmpGetGravity(int32_t *data, const uint8_t* packet=0);
00884 //            uint8_t dmpGetGravity(int16_t *data, const uint8_t* packet=0);
00885 //            uint8_t dmpGetGravity(VectorInt16 *v, const uint8_t* packet=0);
00886 //            uint8_t dmpGetGravity(VectorFloat *v, Quaternion *q);
00887 //            uint8_t dmpGetUnquantizedAccel(int32_t *data, const uint8_t* packet=0);
00888 //            uint8_t dmpGetUnquantizedAccel(int16_t *data, const uint8_t* packet=0);
00889 //            uint8_t dmpGetUnquantizedAccel(VectorInt16 *v, const uint8_t* packet=0);
00890 //            uint8_t dmpGetQuantizedAccel(int32_t *data, const uint8_t* packet=0);
00891 //            uint8_t dmpGetQuantizedAccel(int16_t *data, const uint8_t* packet=0);
00892 //            uint8_t dmpGetQuantizedAccel(VectorInt16 *v, const uint8_t* packet=0);
00893 //            uint8_t dmpGetExternalSensorData(int32_t *data, uint16_t size, const uint8_t* packet=0);
00894 //            uint8_t dmpGetEIS(int32_t *data, const uint8_t* packet=0);
00895 //
00896 //            uint8_t dmpGetEuler(float *data, Quaternion *q);
00897 //            uint8_t dmpGetYawPitchRoll(float *data, Quaternion *q, VectorFloat *gravity);
00898 //
00899 //            // Get Floating Point data from FIFO
00900 //            uint8_t dmpGetAccelFloat(float *data, const uint8_t* packet=0);
00901 //            uint8_t dmpGetQuaternionFloat(float *data, const uint8_t* packet=0);
00902 //
00903 //            uint8_t dmpProcessFIFOPacket(const unsigned char *dmpData);
00904 //            uint8_t dmpReadAndProcessFIFOPacket(uint8_t numPackets, uint8_t *processed=NULL);
00905 //
00906 //            uint8_t dmpSetFIFOProcessedCallback(void (*func) (void));
00907 //
00908 //            uint8_t dmpInitFIFOParam();
00909 //            uint8_t dmpCloseFIFO();
00910 //            uint8_t dmpSetGyroDataSource(uint8_t source);
00911 //            uint8_t dmpDecodeQuantizedAccel();
00912 //            uint32_t dmpGetGyroSumOfSquare();
00913 //            uint32_t dmpGetAccelSumOfSquare();
00914 //            void dmpOverrideQuaternion(long *q);
00915 //            uint16_t dmpGetFIFOPacketSize();
00916 //
00917 //        #endif
00918 //
00919 //        // special methods for MotionApps 4.1 implementation
00920 //        #ifdef MPU6050_INCLUDE_DMP_MOTIONAPPS41
00921 //            uint8_t *dmpPacketBuffer;
00922 //            uint16_t dmpPacketSize;
00923 //
00924 //            uint8_t dmpInitialize();
00925 //            bool dmpPacketAvailable();
00926 //
00927 //            uint8_t dmpSetFIFORate(uint8_t fifoRate);
00928 //            uint8_t dmpGetFIFORate();
00929 //            uint8_t dmpGetSampleStepSizeMS();
00930 //            uint8_t dmpGetSampleFrequency();
00931 //            int32_t dmpDecodeTemperature(int8_t tempReg);
00932 //
00933 //            // Register callbacks after a packet of FIFO data is processed
00934 //            //uint8_t dmpRegisterFIFORateProcess(inv_obj_func func, int16_t priority);
00935 //            //uint8_t dmpUnregisterFIFORateProcess(inv_obj_func func);
00936 //            uint8_t dmpRunFIFORateProcesses();
00937 //
00938 //            // Setup FIFO for various output
00939 //            uint8_t dmpSendQuaternion(uint_fast16_t accuracy);
00940 //            uint8_t dmpSendGyro(uint_fast16_t elements, uint_fast16_t accuracy);
00941 //            uint8_t dmpSendAccel(uint_fast16_t elements, uint_fast16_t accuracy);
00942 //            uint8_t dmpSendLinearAccel(uint_fast16_t elements, uint_fast16_t accuracy);
00943 //            uint8_t dmpSendLinearAccelInWorld(uint_fast16_t elements, uint_fast16_t accuracy);
00944 //            uint8_t dmpSendControlData(uint_fast16_t elements, uint_fast16_t accuracy);
00945 //            uint8_t dmpSendSensorData(uint_fast16_t elements, uint_fast16_t accuracy);
00946 //            uint8_t dmpSendExternalSensorData(uint_fast16_t elements, uint_fast16_t accuracy);
00947 //            uint8_t dmpSendGravity(uint_fast16_t elements, uint_fast16_t accuracy);
00948 //            uint8_t dmpSendPacketNumber(uint_fast16_t accuracy);
00949 //            uint8_t dmpSendQuantizedAccel(uint_fast16_t elements, uint_fast16_t accuracy);
00950 //            uint8_t dmpSendEIS(uint_fast16_t elements, uint_fast16_t accuracy);
00951 //
00952 //            // Get Fixed Point data from FIFO
00953 //            uint8_t dmpGetAccel(int32_t *data, const uint8_t* packet=0);
00954 //            uint8_t dmpGetAccel(int16_t *data, const uint8_t* packet=0);
00955 //            uint8_t dmpGetAccel(VectorInt16 *v, const uint8_t* packet=0);
00956 //            uint8_t dmpGetQuaternion(int32_t *data, const uint8_t* packet=0);
00957 //            uint8_t dmpGetQuaternion(int16_t *data, const uint8_t* packet=0);
00958 //            uint8_t dmpGetQuaternion(Quaternion *q, const uint8_t* packet=0);
00959 //            uint8_t dmpGet6AxisQuaternion(int32_t *data, const uint8_t* packet=0);
00960 //            uint8_t dmpGet6AxisQuaternion(int16_t *data, const uint8_t* packet=0);
00961 //            uint8_t dmpGet6AxisQuaternion(Quaternion *q, const uint8_t* packet=0);
00962 //            uint8_t dmpGetRelativeQuaternion(int32_t *data, const uint8_t* packet=0);
00963 //            uint8_t dmpGetRelativeQuaternion(int16_t *data, const uint8_t* packet=0);
00964 //            uint8_t dmpGetRelativeQuaternion(Quaternion *data, const uint8_t* packet=0);
00965 //            uint8_t dmpGetGyro(int32_t *data, const uint8_t* packet=0);
00966 //            uint8_t dmpGetGyro(int16_t *data, const uint8_t* packet=0);
00967 //            uint8_t dmpGetGyro(VectorInt16 *v, const uint8_t* packet=0);
```

```
00968 //            uint8_t dmpGetMag(int16_t *data, const uint8_t* packet=0);
00969 //            uint8_t dmpSetLinearAccelFilterCoefficient(float coef);
00970 //            uint8_t dmpGetLinearAccel(int32_t *data, const uint8_t* packet=0);
00971 //            uint8_t dmpGetLinearAccel(int16_t *data, const uint8_t* packet=0);
00972 //            uint8_t dmpGetLinearAccel(VectorInt16 *v, const uint8_t* packet=0);
00973 //            uint8_t dmpGetLinearAccel(VectorInt16 *v, VectorInt16 *vRaw, VectorFloat *gravity);
00974 //            uint8_t dmpGetLinearAccelInWorld(int32_t *data, const uint8_t* packet=0);
00975 //            uint8_t dmpGetLinearAccelInWorld(int16_t *data, const uint8_t* packet=0);
00976 //            uint8_t dmpGetLinearAccelInWorld(VectorInt16 *v, const uint8_t* packet=0);
00977 //            uint8_t dmpGetLinearAccelInWorld(VectorInt16 *v, VectorInt16 *vReal, Quaternion *q);
00978 //            uint8_t dmpGetGyroAndAccelSensor(int32_t *data, const uint8_t* packet=0);
00979 //            uint8_t dmpGetGyroAndAccelSensor(int16_t *data, const uint8_t* packet=0);
00980 //            uint8_t dmpGetGyroAndAccelSensor(VectorInt16 *g, VectorInt16 *a, const uint8_t* packet=0);
00981 //            uint8_t dmpGetGyroSensor(int32_t *data, const uint8_t* packet=0);
00982 //            uint8_t dmpGetGyroSensor(int16_t *data, const uint8_t* packet=0);
00983 //            uint8_t dmpGetGyroSensor(VectorInt16 *v, const uint8_t* packet=0);
00984 //            uint8_t dmpGetControlData(int32_t *data, const uint8_t* packet=0);
00985 //            uint8_t dmpGetTemperature(int32_t *data, const uint8_t* packet=0);
00986 //            uint8_t dmpGetGravity(int32_t *data, const uint8_t* packet=0);
00987 //            uint8_t dmpGetGravity(int16_t *data, const uint8_t* packet=0);
00988 //            uint8_t dmpGetGravity(VectorInt16 *v, const uint8_t* packet=0);
00989 //            uint8_t dmpGetGravity(VectorFloat *v, Quaternion *q);
00990 //            uint8_t dmpGetUnquantizedAccel(int32_t *data, const uint8_t* packet=0);
00991 //            uint8_t dmpGetUnquantizedAccel(int16_t *data, const uint8_t* packet=0);
00992 //            uint8_t dmpGetUnquantizedAccel(VectorInt16 *v, const uint8_t* packet=0);
00993 //            uint8_t dmpGetQuantizedAccel(int32_t *data, const uint8_t* packet=0);
00994 //            uint8_t dmpGetQuantizedAccel(int16_t *data, const uint8_t* packet=0);
00995 //            uint8_t dmpGetQuantizedAccel(VectorInt16 *v, const uint8_t* packet=0);
00996 //            uint8_t dmpGetExternalSensorData(int32_t *data, uint16_t size, const uint8_t* packet=0);
00997 //            uint8_t dmpGetEIS(int32_t *data, const uint8_t* packet=0);
00998 //
00999 //            uint8_t dmpGetEuler(float *data, Quaternion *q);
01000 //            uint8_t dmpGetYawPitchRoll(float *data, Quaternion *q, VectorFloat *gravity);
01001 //
01002 //            // Get Floating Point data from FIFO
01003 //            uint8_t dmpGetAccelFloat(float *data, const uint8_t* packet=0);
01004 //            uint8_t dmpGetQuaternionFloat(float *data, const uint8_t* packet=0);
01005 //
01006 //            uint8_t dmpProcessFIFOPacket(const unsigned char *dmpData);
01007 //            uint8_t dmpReadAndProcessFIFOPacket(uint8_t numPackets, uint8_t *processed=NULL);
01008 //
01009 //            uint8_t dmpSetFIFOProcessedCallback(void (*func) (void));
01010 //
01011 //            uint8_t dmpInitFIFOParam();
01012 //            uint8_t dmpCloseFIFO();
01013 //            uint8_t dmpSetGyroDataSource(uint8_t source);
01014 //            uint8_t dmpDecodeQuantizedAccel();
01015 //            uint32_t dmpGetGyroSumOfSquare();
01016 //            uint32_t dmpGetAccelSumOfSquare();
01017 //            void dmpOverrideQuaternion(long *q);
01018 //            uint16_t dmpGetFIFOPacketSize();
01019 //        #endif
01020 //
01021 //    private:
01022 //        uint8_t devAddr;
01023 //        uint8_t buffer[14];
01024 //};
01025 //
01026 //#endif /* _MPU6050_H_ */
```

5.9 GyroscopeMPU9250.cpp File Reference

`#include "GyroscopeMPU9250.h"`

Include dependency graph for GyroscopeMPU9250.cpp:

5.10 GyroscopeMPU9250.cpp

```
00001 #include "GyroscopeMPU9250.h"
00002
00003 GyroscopeMPU9250::GyroscopeMPU9250(bool ad0)
00004         : RegisterBasedWiredDevice(MPU9250_ADDRESS | (ad0 & 0x01)) {
00005     config.value = 0x00;
00006 }
00007
00008 float GyroscopeMPU9250::getRotationX() {
00009     return readAxisRotation(GYRO_XOUT_H);
00010 }
```

```
00011
00012 float GyroscopeMPU9250::getRotationY() {
00013     return readAxisRotation(GYRO_YOUT_H);
00014 }
00015
00016 float GyroscopeMPU9250::getRotationZ() {
00017     return readAxisRotation(GYRO_ZOUT_H);
00018 }
00019
00020 unsigned char GyroscopeMPU9250::readXYZ(unsigned char *buf) {
00021     return readRegisterBlock(GYRO_XOUT_H, buf, 6);
00022 }
00023
00024 float GyroscopeMPU9250::readAxisRotation(unsigned char axisRegister) {
00025     unsigned char buf[2];
00026     if (readRegisterBlock(axisRegister, buf, 2) != 2) {
00027         return 0.0;
00028     }
00029     return convertToDegreePerSeconds(buf);
00030 }
00031
00032 void GyroscopeMPU9250::setFullScaleRange(
      FullScaleRange fsr) {
00033     configureRegisterBits(GYRO_CONFIG, GYRO_CONFIG_GYRO_FS_SEL, (unsigned
       char) fsr);
00034     config.GYRO_FS_SEL = fsr >> 3;
00035 }
00036
00037 void GyroscopeMPU9250::selectClock(ClockSelection cs) {
00038     configureRegisterBits(PWR_MGMT_1, PWR_MGMT_1_CLKSEL, (unsigned char) cs);
00039 }
00040
00041 void GyroscopeMPU9250::reset() {
00042     configureRegisterBits(PWR_MGMT_1, PWR_MGMT_1_H_RESET, 0xff);
00043 }
00044
00045 void GyroscopeMPU9250::sleep() {
00046     configureRegisterBits(PWR_MGMT_1, PWR_MGMT_1_SLEEP, 0xff);
00047 }
00048
00049 void GyroscopeMPU9250::awake() {
00050     configureRegisterBits(PWR_MGMT_1, PWR_MGMT_1_SLEEP, 0x00);
00051 }
00052
00053 void GyroscopeMPU9250::enableAxis(Axis axis) {
00054     configureRegisterBits(PWR_MGMT_2, PWR_MGMT_2_DISABLE_G, ~axis);
00055 }
00056
00057 float GyroscopeMPU9250::convertToDegreePerSeconds(unsigned char
      buf[2]) {
00058     int raw = (buf[0] << 8) | buf[1];
00059     float counts[] = { 131.0, 65.5, 32.8, 16.4 };
00060     return raw / counts[config.GYRO_FS_SEL];
00061 }
```

5.11 GyroscopeMPU9250.h File Reference

```
#include <Gyroscope.h>
#include <RegisterBasedWiredDevice.h>
```

Include dependency graph for GyroscopeMPU9250.h: This graph shows which files directly or indirectly include this file:

Classes

- class GyroscopeMPU9250
- union GyroscopeMPU9250::GYRO_CONFIGbits
- union GyroscopeMPU9250::PWR_MGMT_1bits
- union GyroscopeMPU9250::PWR_MGMT_2bits

Macros

- #define MPU9250_ADDRESS 0x68

5.11.1 Macro Definition Documentation

5.11.1.1 #define MPU9250_ADDRESS 0x68

Arduino - Gyroscope Driver.

Implementation for MPU9250.

Author

Dalmir da Silva dalmirdasilva@gmail.com

Definition at line 15 of file GyroscopeMPU9250.h.

5.12 GyroscopeMPU9250.h

```
00001
00009 #ifndef __ARDUINO_DRIVER_GYROSCOPE_MPU9250_H__
00010 #define __ARDUINO_DRIVER_GYROSCOPE_MPU9250_H__ 1
00011
00012 #include <Gyroscope.h>
00013 #include <RegisterBasedWiredDevice.h>
00014
00015 #define MPU9250_ADDRESS 0x68
00016
00027 class GyroscopeMPU9250 : public Gyroscope, public RegisterBasedWiredDevice {
00028
00029 public:
00030
00031     enum Register {
00032         PWR_MGMT_1 = 0x6b,
00033         PWR_MGMT_2 = 0x6c,
00034         GYRO_CONFIG = 0x1b,
00035         GYRO_XOUT_H = 0x43,
00036         GYRO_XOUT_L = 0x44,
00037         GYRO_YOUT_H = 0x45,
00038         GYRO_YOUT_L = 0x46,
00039         GYRO_ZOUT_H = 0x47,
00040         GYRO_ZOUT_L = 0x48,
00041     };
00042
00054     enum FullScaleRange {
00055         FS_SEL_250DPS = 0x00,
00056         FS_SEL_500DPS = 0x08,
00057         FS_SEL_1000DPS = 0x10,
00058         FS_SEL_2000DPS = 0x18
00059     };
00060
00061     enum ClockSelection {
00062         INTERNAL_20MHZ_OSCILLATOR = 0x00,
00063         BEST_AVAILABLE_SOURCE = 0x01,
00064         STOPS_CLOCK_KEEPS_TIMING = 0x07,
00065     };
00066
00070     enum Axis {
00071         AXIS_NONE = 0x00,
00072         AXIS_X = 0x04,
00073         AXIS_Y = 0x02,
00074         AXIS_Z = 0x01,
00075         AXIS_XY = AXIS_X | AXIS_Y,
00076         AXIS_XZ = AXIS_X | AXIS_Z,
00077         AXIS_YZ = AXIS_Y | AXIS_Z,
00078         AXIS_XYZ = AXIS_X | AXIS_Y | AXIS_Z
00079     };
00080
00084     enum Mask {
00085         GYRO_CONFIG_GYRO_FS_SEL = 0x18,
00086         PWR_MGMT_2_DISABLE_G = 0x07,
00087         PWR_MGMT_1_H_RESET = 0x80,
00088         PWR_MGMT_1_SLEEP = 0x40,
00089         PWR_MGMT_1_CYCLE = 0x20,
00090         PWR_MGMT_1_GYRO_STANDBY = 0x10,
00091         PWR_MGMT_1_CLKSEL = 0x07
00092     };
```

```
00093
00099     union GYRO_CONFIGbits {
00100         struct {
00101             unsigned char FCHOICE_B :2;
00102             unsigned char :1;
00103             unsigned char GYRO_FS_SEL :2;
00104             unsigned char ZGYRO_CTEN :1;
00105             unsigned char YGYRO_CTEN :1;
00106             unsigned char XGYRO_Cten :1;
00107         };
00108         unsigned char value;
00109     };
00110
00116     union PWR_MGMT_1bits {
00117         struct {
00118             unsigned char CLKSEL :3;
00119             unsigned char PD_PTAT :1;
00120             unsigned char GYRO_STANDBY :1;
00121             unsigned char CYCLE :1;
00122             unsigned char SLEEP :1;
00123             unsigned char H_RESET :1;
00124         };
00125         unsigned char value;
00126     };
00127
00133     union PWR_MGMT_2bits {
00134         struct {
00135             unsigned char DISABLE_ZG :1;
00136             unsigned char DISABLE_YG :1;
00137             unsigned char DISABLE_XG :1;
00138             unsigned char DISABLE_ZA :1;
00139             unsigned char DISABLE_YA :1;
00140             unsigned char DISABLE_XA :1;
00141             unsigned char :2;
00142         };
00143         struct {
00144             unsigned char DISABLE_G :3;
00145             unsigned char DISABLE_A :3;
00146             unsigned char :2;
00147         };
00148         unsigned char value;
00149     };
00150
00156     GyroscopeMPU9250(bool ad0);
00157
00158     float getRotationX();
00159
00160     float getRotationY();
00161
00162     float getRotationZ();
00163
00164     unsigned char readXYZ(unsigned char *buf);
00165
00166     float readAxisRotation(unsigned char axisRegister);
00167
00168     void setFullScaleRange(FullScaleRange fsr);
00169
00170     void selectClock(ClockSelection cs);
00171
00172     void reset();
00173
00174     void sleep();
00175
00176     void awake();
00177
00178     void enableAxis(Axis axis);
00179
00180     float convertToDegreePerSeconds(unsigned char buf[2]);
00181
00182 private:
00183
00184     GYRO_CONFIGbits config;
00185 };
00186
00187 #endif /* __ARDUINO_DRIVER_GYROSCOPE_MPU9250_H__ */
```

Index