TivaWare™ Sensor Library

USER'S GUIDE

SW-TM4C-SENSORLIB-UG-2.0

Copyright

Texas Instruments
108 Wild Basin, Suite 350
Austin, TX 78746
www.ti.com/tiva-c

Revision Information

This is version 2.0 of this document, last updated on August 29, 2013.

Table of Contents

1 Introduction

The Texas Instruments® Sensor Library is a collection of functions and drivers for interacting with environmental sensors, such as accelerometers, gyroscopes, magnetometers, and so on. These sensors provide information about the environment in which the board is situated, allowing the application to make decisions based on its surroundings.

The Sensor Library consists of the following components:

- An interrupt-driven I2C master driver that handles the sequence of operations required to perform an I2C transfer, as well as provides a request queue to ease sharing of the I2C bus between multiple drivers.
- A set of drivers for I2C connected sensors.
- A set of routines for performing common, sensor-independent operations on sensor data.

The sensor drivers perform the minimal operations required to put the sensor into a sampling mode in its default configuration. Applications can write the data registers of the sensor to configure it into a different mode, such as changing the sampling rate, adjusting data filters, and so on, in order to tailor the sensor's output to match the application's requirements.

Consult the table of contents for a list of sensors supported by the Sensor Library.

1.1 Units

The Sensor Library uses the International System of Units (SI) to represent quantities that are measured by sensors or derived from data obtained by sensors. This system is based on the meter-kilogram-second system, meaning that the fundamental measures are the meter for lengths, the kilogram for weights, and the second for time.

The following table lists the various measurement quantities used by the Sensor Library:

Quantity	Symbol	Description
Length	m	Lengths and distances are measured in meters. Example applications are proximity sensors that measure the distance to an object or the output of a navigation system that tracks a position over time.
Speed	m/s	Speed is measured in meters per second. Example applications are the speed output of a GPS unit or the output of a navigation system that tracks speed over time.
Acceleration	m/s^2	Acceleration is measured in meters per second squared. Example applications are the output of an accelerometer or the output of a navigation system that tracks acceleration over time.

Quantity	Symbol	Description
Rotation Rate	rad/s	The rate of rotation is measured in radians per second. An example application is the output of a gyroscope.
Magnetic field	T	The strength of a magnetic field is measured in teslas. An example application is the output of a magnetometer.
Temperature	C	Temperature is measured in degrees Celsius. An example application is the output of a temperature sensor.
Pressure	Pa	Pressure is measured in pascals. An example application is the output of a barometric pressure sensor.
Illuminance	lx	Illuminance is measured in lux. An example application is the output of an ambient light sensor.
Relative humidity	-	Relative humidity is a unit-less quantity between 0 and 1. An example application is the output of a humidity sensor.

The driver for any sensor that measures one of these quantities converts its sensor output to the associated unit. This protocol allows an application to easily switch from one sensor to another, such as from an accelerometer from one manufacturer to one from another manufacturer, without having to adjust for a different set of units chosen by the device manufacturer. It also allows measurements from multiple sensors, such as temperature, to be easily used because they are always reported in the same units.

1.2 Structure

The components of the Sensor Library are arranged into three layers; the transport layer, the sensor layer, and the processing layer. Each layer uses a different level of abstraction of the sensor data.

At the lowest level is the transport layer, which deals with moving data into and out of the sensors. These transactions consist of a sequence of bytes with no structure or meaning; the transport layer simply moves the bytes into or out of the sensor without any comprehension of what the bytes mean. The I2C master driver is an example of a transport layer driver.

The next level up is the sensor layer, which deals with interpreting data from a sensor. A transport layer driver is used to communicate with the sensor, and the sensor layer driver provides interpretation of the data and conversion into the appropriate standard unit. The TMP006 temperature sensor driver is an example of a sensor layer driver.

The highest level is the processing layer, which deals with sensor data after it has been converted into a standard unit and is therefore sensor agnostic. This layer consists of things like magnetometer compensation algorithms (that correct distortion in magnetometer readings) and attitude estimation algorithms (that combine readings from multiple sensors to determine the orientation of the sensor platform).

Applications can directly use any of the three layers, and can also choose to not use one or more of the layers. For example, an application can use the sensor and transport layers to gather sensor readings and process those readings without the use of the modules in the processing layer. As another example, an application could gather sensor readings via other means and use the processing layer to process the data. The only layer dependency is that the drivers in the sensor layer use a driver from the transport layer to communicate with the sensor.

1.3 Resources

The following is a list of references to articles and papers on the web that can provide a better understand of the sensors and algorithms used within the Sensor Library.

- **International System of Units**
  `en.wikipedia.org/wiki/International_System_of_Units`
  This page describes the origin and derivation of the SI units that are used by the Sensor Library. It also includes links to other articles that discuss specific units.

- **Euclidean Vectors**
  `http://en.wikipedia.org/wiki/Euclidean_vector`
  This page describes Euclidean vectors and their use in mathematics.

- **Rotations**
  `http://en.wikipedia.org/wiki/Rotation`
  `http://en.wikipedia.org/wiki/Rotation_formalisms_in_three_dimensions`
  These pages describe the mathematics of rotations and their application in three-dimensional space.

- **Accelerometer**
  `en.wikipedia.org/wiki/Accelerometer`
  This page describes accelerometers, what they measure, how they work, and so on.

- **Gyroscope**
  `en.wikipedia.org/wiki/Gyroscope`
  This page describes gyroscopes, what they measure, how they work, and so on.

- **Magnetometer**
  `en.wikipedia.org/wiki/Magnetometer`
  This page describes magnetometers, what they measure, how they work, and so on.

- **Hygrometer**
  `en.wikipedia.org/wiki/Hygrometer`
  This page describes hygrometers, what they measure, how they work, and so on.

- **Barometer**
  `en.wikipedia.org/wiki/Barometer`
  This page describes barometers, what they measure, how they work, and so on.

- **Thermometer**
  `en.wikipedia.org/wiki/Thermometer`
  This page describes thermometers, what they measure, how they work, and so on.

- **DCM Tutorial - An Introduction to Orientation Kinematics**
  Starlino Electronics
  `www.starlino.com/wp-content/uploads/data/dcm_tutorial/Starlino_DCM_Tutorial_01.pdf`
  This paper describes the fundamentals of the Direction Cosine Matrix (DCM), how it is used to determine orientation, and how to update the DCM based on sensor readings. The complementary filter DCM algorithm in the Sensor Library is based upon this paper.

- **Computing Euler Angles from Direction Cosines**
  William Premerlani
  `gentlenav.googlecode.com/files/EulerAngles.pdf`
  This paper describes how to compute Euler angles from a DCM.

- **Introduction into quaternions for spacecraft attitude representation**
  Dipl. -Ing. Karsten Großekatthöfer, Dr. -Ing. Zizung Yoon, Technical University of Berlin
  `www.tu-berlin.de/fileadmin/fg169/miscellaneous/Quaternions.pdf`
  This paper describes quaternions, their relationship to Euler angles and the DCM, and how to compute a quaternion from a DCM.

- **Compensating for Tilt, Hard Iron and Soft Iron Effects**
  Christopher Konvalin, Memsense
  `memsense.com/docs/MTD-0802_1.2_Magnetometer_Calibration.pdf`
  This paper describes the effect of iron in the environment on magnetometer readings and how to compensate for those effects.

- **Applications of Magnetoresistive Sensors in Navigation Systems**
  Michael J. Caruso, Honeywell Inc.
  `www51.honeywell.com/aero/common/documents/`
  `myaerospacecatalog-documents/Defense_Brochures-documents/`
  `Magnetic__Literature_Technical_Article-documents/`
  `Applications_of_Magnetoresistive_Sensors_in_Navigation_Systems.pdf`
  This paper describes how to combine tilt (roll/pitch) information from an accelerometer with magnetometer readings in order to compute a magnetic bearing.

2 AK8963 Magnetometer Driver

2.1 Introduction

The AK8963 is a three-axis magnetometer produced by Asahi Kasei Microdevices Corporation. This driver allows the AK8963 to be accessed via the I2C bus.

This driver is contained in `sensorlib/ak8963.c`, with `sensorlib/ak8963.h` containing the API declarations for use by applications.

2.2 API Functions

Functions

- void AK8963DataGetStatus (tAK8963 *psInst, uint_fast8_t *pui8Status1, uint_fast8_t *pui8Status2)
- void AK8963DataMagnetoGetFloat (tAK8963 *psInst, float *pfMagnetoX, float *pfMagnetoY, float *pfMagnetoZ)
- void AK8963DataMagnetoGetRaw (tAK8963 *psInst, uint_fast16_t *pui16MagnetoX, uint_fast16_t *pui16MagnetoY, uint_fast16_t *pui16MagnetoZ)
- uint_fast8_t AK8963DataRead (tAK8963 *psInst, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t AK8963Init (tAK8963 *psInst, tI2CMInstance *psI2CInst, uint_fast8_t ui8I2CAddr, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t AK8963Read (tAK8963 *psInst, uint_fast8_t ui8Reg, uint8_t *pui8Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t AK8963ReadModifyWrite (tAK8963 *psInst, uint_fast8_t ui8Reg, uint_fast8_t ui8Mask, uint_fast8_t ui8Value, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t AK8963Write (tAK8963 *psInst, uint_fast8_t ui8Reg, const uint8_t *pui8Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)

2.2.1 Function Documentation

2.2.1.1 AK8963DataGetStatus

Gets the status registers from the most recent data read.

Prototype:

```
void
AK8963DataGetStatus(tAK8963 *psInst,
```

```
                                uint_fast8_t *pui8Status1,
                                uint_fast8_t *pui8Status2)
```

Parameters:

psInst is a pointer to the AK8963 instance data.

pui8Status1 is a pointer to the value into which the ST1 data is stored.

pui8Status2 is a pointer to the value into which the ST2 data is stored.

Description:

This function returns the magnetometer status registers from the most recent data read. If any of the output data pointers are **NULL**, the corresponding data is not be provided.

Note that the AKM comp routines require ST1 and ST2, so we read them for that reason.

Returns:

None.

2.2.1.2 AK8963DataMagnetoGetFloat

Gets the magnetometer data from the most recent data read.

Prototype:

```
void
AK8963DataMagnetoGetFloat(tAK8963 *psInst,
                          float *pfMagnetoX,
                          float *pfMagnetoY,
                          float *pfMagnetoZ)
```

Parameters:

psInst is a pointer to the AK8963 instance data.

pfMagnetoX is a pointer to the value into which the X-axis magnetometer data is stored.

pfMagnetoY is a pointer to the value into which the Y-axis magnetometer data is stored.

pfMagnetoZ is a pointer to the value into which the Z-axis magnetometer data is stored.

Description:

This function returns the magnetometer data from the most recent data read, converted into tesla. If any of the output data pointers are **NULL**, the corresponding data is not provided.

Returns:

None.

2.2.1.3 AK8963DataMagnetoGetRaw

Gets the raw magnetometer data from the most recent data read.

Prototype:

```
void
AK8963DataMagnetoGetRaw(tAK8963 *psInst,
                        uint_fast16_t *pui16MagnetoX,
                        uint_fast16_t *pui16MagnetoY,
                        uint_fast16_t *pui16MagnetoZ)
```

Parameters:

psInst is a pointer to the AK8963 instance data.

pui16MagnetoX is a pointer to the value into which the raw X-axis magnetometer data is stored.

pui16MagnetoY is a pointer to the value into which the raw Y-axis magnetometer data is stored.

pui16MagnetoZ is a pointer to the value into which the raw Z-axis magnetometer data is stored.

Description:

This function returns the raw magnetometer data from the most recent data read. The data is not manipulated in any way by the driver. If any of the output data pointers are **NULL**, the corresponding data is not provided.

Returns:

None.

2.2.1.4 AK8963DataRead

Reads the magnetometer data from the AK8963.

Prototype:

```
uint_fast8_t
AK8963DataRead(tAK8963 *psInst,
               tSensorCallback *pfnCallback,
               void *pvCallbackData)
```

Parameters:

psInst is a pointer to the AK8963 instance data.

pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initiates a read of the AK8963 data registers. When the read has completed (as indicated by calling the callback function), the new readings can be obtained via:

- AK8963DataMagnetoGetRaw()
- AK8963DataMagnetoGetFloat()

Returns:

Returns 1 if the read was successfully started and 0 if it was not.

2.2.1.5 AK8963Init

Initializes the AK8963 driver.

Prototype:

```
uint_fast8_t
AK8963Init(tAK8963 *psInst,
```

```
                tI2CMInstance *psI2CInst,
                uint_fast8_t ui8I2CAddr,
                tSensorCallback *pfnCallback,
                void *pvCallbackData)
```

Parameters:

psInst is a pointer to the AK8963 instance data.

psI2CInst is a pointer to the I2C master driver instance data.

ui8I2CAddr is the I2C address of the AK8963 device.

pfnCallback is the function to be called when the initialization has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initializes the AK8963 driver, preparing it for operation.

Returns:

Returns 1 if the AK8963 driver was successfully initialized and 0 if it was not.

2.2.1.6 AK8963Read

Reads data from AK8963 registers.

Prototype:

```
uint_fast8_t
AK8963Read(tAK8963 *psInst,
           uint_fast8_t ui8Reg,
           uint8_t *pui8Data,
           uint_fast16_t ui16Count,
           tSensorCallback *pfnCallback,
           void *pvCallbackData)
```

Parameters:

psInst is a pointer to the AK8963 instance data.

ui8Reg is the first register to read.

pui8Data is a pointer to the location to store the data that is read.

ui16Count is the number of data bytes to read.

pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function reads a sequence of data values from consecutive registers in the AK8963.

Returns:

Returns 1 if the read was successfully started and 0 if it was not.

2.2.1.7 AK8963ReadModifyWrite

Performs a read-modify-write of an AK8963 register.

Prototype:

```
uint_fast8_t
AK8963ReadModifyWrite(tAK8963 *psInst,
                      uint_fast8_t ui8Reg,
                      uint_fast8_t ui8Mask,
                      uint_fast8_t ui8Value,
                      tSensorCallback *pfnCallback,
                      void *pvCallbackData)
```

Parameters:

psInst is a pointer to the AK8963 instance data.

ui8Reg is the register to modify.

ui8Mask is the bit mask that is ANDed with the current register value.

ui8Value is the bit mask that is ORed with the result of the AND operation.

pfnCallback is the function to be called when the data has been changed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function changes the value of a register in the AK8963 via a read-modify-write operation, allowing one of the fields to be changed without disturbing the other fields. The *ui8Reg* register is read, ANDed with *ui8Mask*, ORed with *ui8Value*, and then written back to the AK8963.

Returns:

Returns 1 if the read-modify-write was successfully started and 0 if it was not.

2.2.1.8 AK8963Write

Writes data to AK8963 registers.

Prototype:

```
uint_fast8_t
AK8963Write(tAK8963 *psInst,
            uint_fast8_t ui8Reg,
            const uint8_t *pui8Data,
            uint_fast16_t ui16Count,
            tSensorCallback *pfnCallback,
            void *pvCallbackData)
```

Parameters:

psInst is a pointer to the AK8963 instance data.

ui8Reg is the first register to write.

pui8Data is a pointer to the data to write.

ui16Count is the number of data bytes to write.

pfnCallback is the function to be called when the data has been written (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function writes a sequence of data values to consecutive registers in the AK8963. The first byte of the *pui8Data* buffer contains the value to be written into the *ui8Reg* register, the second value contains the data to be written into the next register, and so on.

Returns:

Returns 1 if the write was successfully started and 0 if it was not.

2.3 Programming Example

The following example shows how to initialize the AK8963 and read magnetic field data from it.

```
//
// A boolean that is set when a AK8963 command has completed.
//
volatile bool g_bAK8963Done;

//
// The function that is provided by this example as a callback when AK8963
// transactions have completed.
//
void
AK8963Callback(void *pvCallbackData, uint_fast8_t ui8Status)
{
    //
    // See if an error occurred.
    //
    if(ui8Status != I2CM_STATUS_SUCCESS)
    {
        //
        // An error occurred, so handle it here if required.
        //
    }

    //
    // Indicate that the AK8963 transaction has completed.
    //
    g_bAK8963Done = true;
}

//
// The AK8963 example.
//
void
AK8963Example(void)
{
    float fAccel[3], fGyro[3], fMagneto[3];
    tI2CMInstance sI2CInst;
    tAK8963 sAK8963;

    //
    // Initialize the AK8963.  This codes assumes that the I2C master instance
    // has already been initialized.
    //
    g_bAK8963Done = false;
    AK8963Init(&sAK8963, &sI2CInst, 0x0c, AK8963Callback, 0);
    while(!g_bAK8963Done)
    {
```

```
    }

    //
    // Loop forever reading data from the AK8963.  Typically, this process
    // would be done in the background, but for the purposes of this example,
    // it is shown in an infinite loop.
    //
    while(1)
    {
        //
        // Tell the AK8963 to capture another sample.
        //
        g_bAK8963Done = false;
        AK8963ReadModifyWrite(&sAK8963, AK8963_O_CNTL, 0,
                              AK8963_CNTL_MODE_SINGLE, AK8963Callback, 0);
        while(!g_bAK8963Done)
        {
        }

        //
        // Wait while the sample is being acquired.
        //

        //
        // Read the data from the AK8963.
        //
        g_bAK8963Done = false;
        AK8963DataRead(&sAK8963, AK8963Callback, 0);
        while(!g_bAK8963Done)
        {
        }

        //
        // Get the new magnetometer reading.
        //
        AK8963DataMagnetoGetFloat(&sAK8963, &fMagneto[0], &fMagneto[1],
                                  &fMagneto[2]);

        //
        // Do something with the new magnetometer reading.
        //
    }
}
```

3 AK8975 Magnetometer Driver

3.1 Introduction

The AK8975 is a three-axis magnetometer produced by Asahi Kasei Microdevices Corporation. This driver allows the AK8975 to be accessed via the I2C bus.

This driver is contained in `sensorlib/ak8975.c`, with `sensorlib/ak8975.h` containing the API declarations for use by applications.

3.2 API Functions

Functions

- void AK8975DataGetStatus (tAK8975 *psInst, uint_fast8_t *pui8Status1, uint_fast8_t *pui8Status2)
- void AK8975DataMagnetoGetFloat (tAK8975 *psInst, float *pfMagnetoX, float *pfMagnetoY, float *pfMagnetoZ)
- void AK8975DataMagnetoGetRaw (tAK8975 *psInst, uint_fast16_t *pui16MagnetoX, uint_fast16_t *pui16MagnetoY, uint_fast16_t *pui16MagnetoZ)
- uint_fast8_t AK8975DataRead (tAK8975 *psInst, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t AK8975Init (tAK8975 *psInst, tI2CMInstance *psI2CInst, uint_fast8_t ui8I2CAddr, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t AK8975Read (tAK8975 *psInst, uint_fast8_t ui8Reg, uint8_t *pui8Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t AK8975ReadModifyWrite (tAK8975 *psInst, uint_fast8_t ui8Reg, uint_fast8_t ui8Mask, uint_fast8_t ui8Value, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t AK8975Write (tAK8975 *psInst, uint_fast8_t ui8Reg, uint8_t *pui8Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)

3.2.1 Function Documentation

3.2.1.1 AK8975DataGetStatus

Gets the status registers from the most recent data read.

Prototype:

```
void
AK8975DataGetStatus(tAK8975 *psInst,
```

```
                                uint_fast8_t *pui8Status1,
                                uint_fast8_t *pui8Status2)
```

Parameters:

psInst is a pointer to the AK8975 instance data.

pui8Status1 is a pointer to the value into which the ST1 data is stored.

pui8Status2 is a pointer to the value into which the ST2 data is stored.

Description:

This function returns the magnetometer status registers from the most recent data read. If any of the output data pointers are **NULL**, the corresponding data is not provided.

Note that the AKM comp routines require ST1 and ST2, so we read them for that reason.

Returns:

None.

3.2.1.2 AK8975DataMagnetoGetFloat

Gets the magnetometer data from the most recent data read.

Prototype:

```
void
AK8975DataMagnetoGetFloat(tAK8975 *psInst,
                          float *pfMagnetoX,
                          float *pfMagnetoY,
                          float *pfMagnetoZ)
```

Parameters:

psInst is a pointer to the AK8975 instance data.

pfMagnetoX is a pointer to the value into which the X-axis magnetometer data is stored.

pfMagnetoY is a pointer to the value into which the Y-axis magnetometer data is stored.

pfMagnetoZ is a pointer to the value into which the Z-axis magnetometer data is stored.

Description:

This function returns the magnetometer data from the most recent data read, converted into tesla. If any of the output data pointers are **NULL**, the corresponding data is not provided.

Returns:

None.

3.2.1.3 AK8975DataMagnetoGetRaw

Gets the raw magnetometer data from the most recent data read.

Prototype:

```
void
AK8975DataMagnetoGetRaw(tAK8975 *psInst,
                        uint_fast16_t *pui16MagnetoX,
                        uint_fast16_t *pui16MagnetoY,
                        uint_fast16_t *pui16MagnetoZ)
```

Parameters:

psInst is a pointer to the AK8975 instance data.

pui16MagnetoX is a pointer to the value into which the raw X-axis magnetometer data is stored.

pui16MagnetoY is a pointer to the value into which the raw Y-axis magnetometer data is stored.

pui16MagnetoZ is a pointer to the value into which the raw Z-axis magnetometer data is stored.

Description:

This function returns the raw magnetometer data from the most recent data read. The data is not manipulated in any way by the driver. If any of the output data pointers are **NULL**, the corresponding data is not provided.

Returns:

None.

3.2.1.4 AK8975DataRead

Reads the magnetometer data from the AK8975.

Prototype:

```
uint_fast8_t
AK8975DataRead(tAK8975 *psInst,
               tSensorCallback *pfnCallback,
               void *pvCallbackData)
```

Parameters:

psInst is a pointer to the AK8975 instance data.

pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initiates a read of the AK8975 data registers. When the read has completed (as indicated by calling the callback function), the new readings can be obtained via:

- AK8975DataMagnetoGetRaw()
- AK8975DataMagnetoGetFloat()

Returns:

Returns 1 if the read was successfully started and 0 if it was not.

3.2.1.5 AK8975Init

Initializes the AK8975 driver.

Prototype:

```
uint_fast8_t
AK8975Init(tAK8975 *psInst,
```

```
                tI2CMInstance *psI2CInst,
                uint_fast8_t ui8I2CAddr,
                tSensorCallback *pfnCallback,
                void *pvCallbackData)
```

Parameters:

psInst is a pointer to the AK8975 instance data.

psI2CInst is a pointer to the I2C master driver instance data.

ui8I2CAddr is the I2C address of the AK8975 device.

pfnCallback is the function to be called when the initialization has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initializes the AK8975 driver, preparing it for operation.

Returns:

Returns 1 if the AK8975 driver was successfully initialized and 0 if it was not.

3.2.1.6 AK8975Read

Reads data from AK8975 registers.

Prototype:

```
uint_fast8_t
AK8975Read(tAK8975 *psInst,
           uint_fast8_t ui8Reg,
           uint8_t *pui8Data,
           uint_fast16_t ui16Count,
           tSensorCallback *pfnCallback,
           void *pvCallbackData)
```

Parameters:

psInst is a pointer to the AK8975 instance data.

ui8Reg is the first register to read.

pui8Data is a pointer to the location to store the data that is read.

ui16Count is the number of data bytes to read.

pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function reads a sequence of data values from consecutive registers in the AK8975.

Returns:

Returns 1 if the write was successfully started and 0 if it was not.

3.2.1.7 AK8975ReadModifyWrite

Performs a read-modify-write of an AK8975 register.

Prototype:

```
uint_fast8_t
AK8975ReadModifyWrite(tAK8975 *psInst,
                      uint_fast8_t ui8Reg,
                      uint_fast8_t ui8Mask,
                      uint_fast8_t ui8Value,
                      tSensorCallback *pfnCallback,
                      void *pvCallbackData)
```

Parameters:

psInst is a pointer to the AK8975 instance data.

ui8Reg is the register to modify.

ui8Mask is the bit mask that is ANDed with the current register value.

ui8Value is the bit mask that is ORed with the result of the AND operation.

pfnCallback is the function to be called when the data has been changed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function changes the value of a register in the AK8975 via a read-modify-write operation, allowing one of the fields to be changed without disturbing the other fields. The *ui8Reg* register is read, ANDed with *ui8Mask*, ORed with *ui8Value*, and then written back to the AK8975.

Returns:

Returns 1 if the read-modify-write was successfully started and 0 if it was not.

3.2.1.8 AK8975Write

Writes data to AK8975 registers.

Prototype:

```
uint_fast8_t
AK8975Write(tAK8975 *psInst,
            uint_fast8_t ui8Reg,
            uint8_t *pui8Data,
            uint_fast16_t ui16Count,
            tSensorCallback *pfnCallback,
            void *pvCallbackData)
```

Parameters:

psInst is a pointer to the AK8975 instance data.

ui8Reg is the first register to write.

pui8Data is a pointer to the data to write.

ui16Count is the number of data bytes to write.

pfnCallback is the function to be called when the data has been written (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function writes a sequence of data values to consecutive registers in the AK8975. The first byte of the *pui8Data* buffer contains the value to be written into the *ui8Reg* register, the second value contains the data to be written into the next register, and so on.

Returns:

Returns 1 if the write was successfully started and 0 if it was not.

3.3 Programming Example

The following example shows how to initialize the AK8975 and read magnetic field data from it.

```
//
// A boolean that is set when a AK8975 command has completed.
//
volatile bool g_bAK8975Done;

//
// The function that is provided by this example as a callback when AK8975
// transactions have completed.
//
void
AK8975Callback(void *pvCallbackData, uint_fast8_t ui8Status)
{
    //
    // See if an error occurred.
    //
    if(ui8Status != I2CM_STATUS_SUCCESS)
    {
        //
        // An error occurred, so handle it here if required.
        //
    }

    //
    // Indicate that the AK8975 transaction has completed.
    //
    g_bAK8975Done = true;
}

//
// The AK8975 example.
//
void
AK8975Example(void)
{
    float fAccel[3], fGyro[3], fMagneto[3];
    tI2CMInstance sI2CInst;
    tAK8975 sAK8975;

    //
    // Initialize the AK8975.  This code assumes that the I2C master instance
    // has already been initialized.
    //
    g_bAK8975Done = false;
    AK8975Init(&sAK8975, &sI2CInst, 0x0c, AK8975Callback, 0);
    while(!g_bAK8975Done)
    {
```

```
    }

    //
    // Loop forever reading data from the AK8975.  Typically, this process
    // would be done in the background, but for the purposes of this example,
    // it is shown in an infinite loop.
    //
    while(1)
    {
        //
        // Tell the AK8975 to capture another sample.
        //
        g_bAK8975Done = false;
        AK8975ReadModifyWrite(&sAK8975, AK8975_O_CNTL, 0,
                              AK8975_CNTL_MODE_SINGLE, AK8975Callback, 0);
        while(!g_bAK8975Done)
        {
        }

        //
        // Wait while the sample is being acquired.
        //

        //
        // Read the data from the AK8975.
        //
        g_bAK8975Done = false;
        AK8975DataRead(&sAK8975, AK8975Callback, 0);
        while(!g_bAK8975Done)
        {
        }

        //
        // Get the new magnetometer reading.
        //
        AK8975DataMagnetoGetFloat(&sAK8975, &fMagneto[0], &fMagneto[1],
                                  &fMagneto[2]);

        //
        // Do something with the new magnetometer reading.
        //
    }
}
```

4 BMP180 Barometer Driver

4.1 Introduction

The BMP180 is a barometric pressure sensor produced by Bosch Sensortec GmbH. It measures both barometric pressure and temperature, and is capable of providing a temperature-compensated barometric pressure reading (with the compensation occurring on the host processor). This driver allows the BMP180 to be accessed via the I2C bus.

When initialized, a soft reset of the BMP180 is performed, putting it into its default state. The default oversampling rate of 1x is therefore selected.

This driver is contained in `sensorlib/bmp180.c`, with `sensorlib/bmp180.h` containing the API declarations for use by applications.

4.2 API Functions

Functions

- void BMP180DataPressureGetFloat (tBMP180 *psInst, float *pfPressure)
- void BMP180DataPressureGetRaw (tBMP180 *psInst, uint_fast32_t *pui32Pressure)
- uint_fast8_t BMP180DataRead (tBMP180 *psInst, tSensorCallback *pfnCallback, void *pvCallbackData)
- void BMP180DataTemperatureGetFloat (tBMP180 *psInst, float *pfTemperature)
- void BMP180DataTemperatureGetRaw (tBMP180 *psInst, uint_fast16_t *pui16Temperature)
- uint_fast8_t BMP180Init (tBMP180 *psInst, tI2CMInstance *psI2CInst, uint_fast8_t ui8I2CAddr, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t BMP180Read (tBMP180 *psInst, uint_fast8_t ui8Reg, uint8_t *pui8Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t BMP180ReadModifyWrite (tBMP180 *psInst, uint_fast8_t ui8Reg, uint_fast8_t ui8Mask, uint_fast8_t ui8Value, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t BMP180Write (tBMP180 *psInst, uint_fast8_t ui8Reg, uint8_t *pui8Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)

4.2.1 Function Documentation

4.2.1.1 BMP180DataPressureGetFloat

Gets the pressure data from the most recent data read.

Prototype:

```
void
BMP180DataPressureGetFloat(tBMP180 *psInst,
                           float *pfPressure)
```

Parameters:

psInst is a pointer to the BMP180 instance data.

pfPressure is a pointer to the value into which the pressure data is stored.

Description:

This function returns the pressure data from the most recent data read, converted into pascals.

Returns:

None.

4.2.1.2 BMP180DataPressureGetRaw

Gets the raw pressure data from the most recent data read.

Prototype:

```
void
BMP180DataPressureGetRaw(tBMP180 *psInst,
                         uint_fast32_t *pui32Pressure)
```

Parameters:

psInst is a pointer to the BMP180 instance data.

pui32Pressure is a pointer to the value into which the raw pressure data is stored.

Description:

This function returns the raw pressure data from the most recent data read. The data is not manipulated in any way by the driver.

Returns:

None.

4.2.1.3 BMP180DataRead

Reads the pressure data from the BMP180.

Prototype:

```
uint_fast8_t
BMP180DataRead(tBMP180 *psInst,
               tSensorCallback *pfnCallback,
               void *pvCallbackData)
```

Parameters:

psInst is a pointer to the BMP180 instance data.

pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:
This function initiates a read of the BMP180 data registers. When the read has completed (as indicated by calling the callback function), the new temperature and pressure readings can be obtained via:

- BMP180DataPressureGetRaw()
- BMP180DataPressureGetFloat()
- BMP180DataTemperatureGetRaw()
- BMP180DataTemperatureGetFloat()

Returns:
Returns 1 if the read was successfully started and 0 if it was not.

4.2.1.4 BMP180DataTemperatureGetFloat

Gets the temperature data from the most recent data read.

Prototype:
```
void
BMP180DataTemperatureGetFloat(tBMP180 *psInst,
                              float *pfTemperature)
```

Parameters:
psInst is a pointer to the BMP180 instance data.
pfTemperature is a pointer to the value into which the temperature data is stored.

Description:
This function returns the temperature data from the most recent data read, converted into Celsius.

Returns:
None.

4.2.1.5 BMP180DataTemperatureGetRaw

Gets the raw temperature data from the most recent data read.

Prototype:
```
void
BMP180DataTemperatureGetRaw(tBMP180 *psInst,
                            uint_fast16_t *pui16Temperature)
```

Parameters:
psInst is a pointer to the BMP180 instance data.
pui16Temperature is a pointer to the value into which the raw temperature data is stored.

Description:
This function returns the raw temperature data from the most recent data read. The data is not manipulated in any way by the driver.

Returns:
None.

4.2.1.6 BMP180Init

Initializes the BMP180 driver.

Prototype:

```
uint_fast8_t
BMP180Init(tBMP180 *psInst,
           tI2CMInstance *psI2CInst,
           uint_fast8_t ui8I2CAddr,
           tSensorCallback *pfnCallback,
           void *pvCallbackData)
```

Parameters:

psInst is a pointer to the BMP180 instance data.

psI2CInst is a pointer to the I2C master driver instance data.

ui8I2CAddr is the I2C address of the BMP180 device.

pfnCallback is the function to be called when the initialization has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initializes the BMP180 driver, preparing it for operation.

Returns:

Returns 1 if the BMP180 driver was successfully initialized and 0 if it was not.

4.2.1.7 BMP180Read

Reads data from BMP180 registers.

Prototype:

```
uint_fast8_t
BMP180Read(tBMP180 *psInst,
           uint_fast8_t ui8Reg,
           uint8_t *pui8Data,
           uint_fast16_t ui16Count,
           tSensorCallback *pfnCallback,
           void *pvCallbackData)
```

Parameters:

psInst is a pointer to the BMP180 instance data.

ui8Reg is the first register to read.

pui8Data is a pointer to the location to store the data that is read.

ui16Count is the number of data bytes to read.

pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function reads a sequence of data values from consecutive registers in the BMP180.

Returns:

Returns 1 if the write was successfully started and 0 if it was not.

4.2.1.8 BMP180ReadModifyWrite

Performs a read-modify-write of a BMP180 register.

Prototype:

```
uint_fast8_t
BMP180ReadModifyWrite(tBMP180 *psInst,
                      uint_fast8_t ui8Reg,
                      uint_fast8_t ui8Mask,
                      uint_fast8_t ui8Value,
                      tSensorCallback *pfnCallback,
                      void *pvCallbackData)
```

Parameters:

psInst is a pointer to the BMP180 instance data.

ui8Reg is the register to modify.

ui8Mask is the bit mask that is ANDed with the current register value.

ui8Value is the bit mask that is ORed with the result of the AND operation.

pfnCallback is the function to be called when the data has been changed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function changes the value of a register in the BMP180 via a read-modify-write operation, allowing one of the fields to be changed without disturbing the other fields. The *ui8Reg* register is read, ANDed with *ui8Mask*, ORed with *ui8Value*, and then written back to the BMP180.

Returns:

Returns 1 if the read-modify-write was successfully started and 0 if it was not.

4.2.1.9 BMP180Write

Writes data to BMP180 registers.

Prototype:

```
uint_fast8_t
BMP180Write(tBMP180 *psInst,
            uint_fast8_t ui8Reg,
            uint8_t *pui8Data,
            uint_fast16_t ui16Count,
            tSensorCallback *pfnCallback,
            void *pvCallbackData)
```

Parameters:

psInst is a pointer to the BMP180 instance data.

ui8Reg is the first register to write.

pui8Data is a pointer to the data to write.

ui16Count is the number of data bytes to write.

pfnCallback is the function to be called when the data has been written (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function writes a sequence of data values to consecutive registers in the BMP180. The first byte of the *pui8Data* buffer contains the value to be written into the *ui8Reg* register, the second value contains the data to be written into the next register, and so on.

Returns:

Returns 1 if the write was successfully started and 0 if it was not.

4.3 Programming Example

The following example shows how to initialize the BMP180, select 2x oversampling, and read pressure and temperature data from it.

```
//
// A boolean that is set when a BMP180 command has completed.
//
volatile bool g_bBMP180Done;

//
// The function that is provided by this example as a callback when BMP180
// transactions have completed.
//
void
BMP180Callback(void *pvCallbackData, uint_fast8_t ui8Status)
{
    //
    // See if an error occurred.
    //
    if(ui8Status != I2CM_STATUS_SUCCESS)
    {
        //
        // An error occurred, so handle it here if required.
        //
    }

    //
    // Indicate that the BMP180 transaction has completed.
    //
    g_bBMP180Done = true;
}

//
// The BMP180 example.
//
void
BMP180Example(void)
{
    float fTemperature, fPressure;
    tI2CMInstance sI2CInst;
    tBMP180 sBMP180;

    //
    // Initialize the BMP180.  This code assumes that the I2C master instance
```

```
    // has already been initialized.
    //
    g_bBMP180Done = false;
    BMP180Init(&sBMP180, &sI2CInst, 0x77, BMP180Callback, 0);
    while(!g_bBMP180Done)
    {
    }

    //
    // Configure the BMP180 for 2x oversampling.
    //
    g_bBMP180Done = false;
    BMP180ReadModifyWrite(&sBMP180, BMP180_O_CTRL_MEAS,
                          ~BMP180_CTRL_MEAS_OSS_M, BMP180_CTRL_MEAS_OSS_2,
                          BMP180Callback, 0);
    while(!g_bBMP180Done)
    {
    }

    //
    // Loop forever reading data from the BMP180.  Typically, this process
    // would be done in the background, but for the purposes of this example,
    // it is shown in an infinite loop.
    //
    while(1)
    {
        //
        // Request another reading from the BMP180.
        //
        g_bBMP180Done = false;
        BMP180DataRead(&sBMP180, BMP180Callback, 0);
        while(!g_bBMP180Done)
        {
        }

        //
        // Get the new pressure and temperature reading.
        //
        BMP180DataPressureGetFloat(&sBMP180, &fPressure);
        BMP180DataTemperatureGetFloat(&sBMP180, &fTemperature);

        //
        // Do something with the new pressure and temperature readings.
        //
    }
}
```

5 CM3218 Ambient Light Sensor Driver

5.1 Introduction

The CM3218 is an ambient light sensor produced by Capella Microsystems, Inc. This driver allows the CM3218 to be accessed via the I2C bus.

This driver is contained in `sensorlib/cm3218.c`, with `sensorlib/cm3218.h` containing the API declarations for use by applications.

5.2 API Functions

Functions

- void CM3218DataLightVisibleGetFloat (tCM3218 *psInst, float *pfVisibleLight)
- void CM3218DataLightVisibleGetRaw (tCM3218 *psInst, uint16_t *pui16Visible)
- uint_fast8_t CM3218DataRead (tCM3218 *psInst, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t CM3218Init (tCM3218 *psInst, tI2CMInstance *psI2CInst, uint_fast8_t ui8I2CAddr, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t CM3218Read (tCM3218 *psInst, uint_fast8_t ui8Reg, uint16_t *pui16Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t CM3218Write (tCM3218 *psInst, uint_fast8_t ui8Reg, const uint16_t *pui16Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)

5.2.1 Function Documentation

5.2.1.1 CM3218DataLightVisibleGetFloat

Gets the measurement data from the most recent data read.

Prototype:

```
void
CM3218DataLightVisibleGetFloat(tCM3218 *psInst,
                               float *pfVisibleLight)
```

Parameters:

psInst is a pointer to the CM3218 instance data.

pfVisibleLight is a pointer to the value into which the light data is stored as floating point lux.

Description:

This function returns the light data from the most recent data read, converted into lux.

Returns:
None.

5.2.1.2 CM3218DataLightVisibleGetRaw

Gets the raw measurement data from the most recent data read.

Prototype:
```
void
CM3218DataLightVisibleGetRaw(tCM3218 *psInst,
                             uint16_t *pui16Visible)
```

Parameters:
psInst is a pointer to the CM3218 instance data.
pui16Visible is a pointer to the value into which the raw visible light data is stored.

Description:
This function returns the raw measurement data from the most recent data read. The data is not manipulated in any way by the driver.

Returns:
None.

5.2.1.3 CM3218DataRead

Reads the light data from the CM3218.

Prototype:
```
uint_fast8_t
CM3218DataRead(tCM3218 *psInst,
               tSensorCallback *pfnCallback,
               void *pvCallbackData)
```

Parameters:
psInst is a pointer to the CM3218 instance data.
pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).
pvCallbackData is a pointer that is passed to the callback function.

Description:
This function initiates a read of the CM3218 data registers. When the read has completed (as indicated by calling the callback function), the new readings can be obtained via:

- CM3218DataLightVisibleGetRaw()
- CM3218DataLightVisibleGetFloat()

Returns:
Returns 1 if the read was successfully started and 0 if it was not.

5.2.1.4 CM3218Init

Initializes the CM3218 driver.

Prototype:

```
uint_fast8_t
CM3218Init(tCM3218 *psInst,
           tI2CMInstance *psI2CInst,
           uint_fast8_t ui8I2CAddr,
           tSensorCallback *pfnCallback,
           void *pvCallbackData)
```

Parameters:

psInst is a pointer to the CM3218 instance data.

psI2CInst is a pointer to the I2C driver instance data.

ui8I2CAddr is the I2C address of the CM3218 device.

pfnCallback is the function to be called when the initialization has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initializes the CM3218 driver, preparing it for operation.

Returns:

Returns 1 if the CM3218 driver was successfully initialized and 0 if it was not.

5.2.1.5 CM3218Read

Reads data from CM3218 registers.

Prototype:

```
uint_fast8_t
CM3218Read(tCM3218 *psInst,
           uint_fast8_t ui8Reg,
           uint16_t *pui16Data,
           uint_fast16_t ui16Count,
           tSensorCallback *pfnCallback,
           void *pvCallbackData)
```

Parameters:

psInst is a pointer to the CM3218 instance data.

ui8Reg is the first register to read.

pui16Data is a pointer to the location to store the data that is read.

ui16Count the number of register values bytes to read.

pfnCallback is the function to be called when data read is complete (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function reads a sequence of data values from consecutive registers in the CM3218.

Note:
The CM3218 does not auto-increment the register pointer, so reads of more than one value returns the same data.

Returns:
Returns 1 if the write was successfully started and 0 if it was not.

5.2.1.6 CM3218Write

Writes data to CM3218 registers.

Prototype:

```
uint_fast8_t
CM3218Write(tCM3218 *psInst,
            uint_fast8_t ui8Reg,
            const uint16_t *pui16Data,
            uint_fast16_t ui16Count,
            tSensorCallback *pfnCallback,
            void *pvCallbackData)
```

Parameters:
psInst is a pointer to the CM3218 instance data.
ui8Reg is the first register to write.
pui16Data is a pointer to the 16-bit register data to write.
ui16Count is the number of data bytes to write.
pfnCallback is the function to be called when the data has been written (can be **NULL** if a callback is not required).
pvCallbackData is a pointer that is passed to the callback function.

Description:
This function writes a sequence of data values to consecutive registers in the CM3218. The first value in the *pui16Data* buffer contains the data to be written into the *ui8Reg* register, the second value contains the data to be written into the next register, and so on.

Note:
The CM3218 does not auto-increment the register pointer, so writes of more than one register are rejected by the CM3218.

Returns:
Returns 1 if the write was successfully started and 0 if it was not.

5.3 Programming Example

The following example shows how to initialize the CM3218 and read light data from it.

```
//
// A boolean that is set when a CM3218 command has completed.
//
volatile bool g_bCM3218Done;
```

```
//
// The function that is provided by this example as a callback when CM3218
// transactions have completed.
//
void
CM3218Callback(void *pvCallbackData, uint_fast8_t ui8Status)
{
    //
    // See if an error occurred.
    //
    if(ui8Status != I2CM_STATUS_SUCCESS)
    {
        //
        // An error occurred, so handle it here if required.
        //
    }

    //
    // Indicate that the CM3218 transaction has completed.
    //
    g_bCM3218Done = true;
}

//
// The CM3218 example.
//
void
CM3218Example(void)
{
    tI2CMInstance sI2CInst;
    tCM3218 sCM3218;
    float fVisible, fIR;

    //
    // Initialize the CM3218.  This code assumes that the I2C master instance
    // has already been initialized.
    //
    g_bCM3218Done = false;
    CM3218Init(&sCM3218, &sI2CInst, 0x0a, CM3218Callback, 0);
    while(!g_bCM3218Done)
    {
    }

    //
    // Loop forever reading data from the CM3218.  Typically, this process
    // would be done in the background, but for the purposes of this example,
    // it is shown in an infinite loop.
    //
    while(1)
    {
        //
        // Request another reading from the CM3218.
        //
        g_bCM3218Done = false;
        CM3218DataRead(&sCM3218, CM3218Callback, 0);
        while(!g_bCM3218Done)
        {
        }

        //
        // Get the new light reading.
        //
        CM3218DataLightVisibleGetFloat(&sCM3218, &fVisible);

        //
        // Do something with the new light reading.
```

```
                //
            }
        }
```

6 Complementary Filter DCM Module

6.1 Introduction

The complementary filter DCM (Direction Cosine Matrix) module provides an algorithm for fusing accelerometer, gyroscope, and magnetometer readings into an attitude estimation for the sensor platform (body) relative to the world. The DCM is a matrix that contains the cosine of each body axis relative to each world axis, as follows:

$$\begin{bmatrix} cos(xx) & cos(xy) & cos(xz) \\ cos(yx) & cos(yy) & cos(yz) \\ cos(zx) & cos(zy) & cos(zz) \end{bmatrix}$$

The algorithm takes the weighted contribution of the accelerometer, gyroscope, and magnetometer to produce an updated attitude estimation. This approach is a complementary filter algorithm because it combines multiple readings of the same signal (that being the attitude of the sensor platform) using weighting factors that sum to one.

The sensors must be sampled at regular intervals, and those samples provided to the complementary filter DCM algorithm. It is important that the samples are taken at a fixed, regular interval in order for the algorithm to provide accurate results.

After each update, the value of the DCM is recomputed. Whenever required by the application, the attitude can be extracted from the DCM in either Euler angle or quaternion format. The Euler angles are used to convert vectors in the world coordinate system into the body coordinate system by first applying the roll, then the pitch, then the yaw. Alternatively, the Euler angles can be used to convert vectors in the body coordinate system into the world coordinate system by first applying the negative yaw, then the negative pitch, then the negative roll.

This module is contained in `sensorlib/comp_dcm.c`, with `sensorlib/comp_dcm.h` containing the API declarations for use by applications.

6.2 API Functions

Functions

- void CompDCMAccelUpdate (tCompDCM *psDCM, float fAccelX, float fAccelY, float fAccelZ)
- void CompDCMComputeEulers (tCompDCM *psDCM, float *pfRoll, float *pfPitch, float *pfYaw)
- void CompDCMComputeQuaternion (tCompDCM *psDCM, float pfQuaternion[4])
- void CompDCMGyroUpdate (tCompDCM *psDCM, float fGyroX, float fGyroY, float fGyroZ)
- void CompDCMInit (tCompDCM *psDCM, float fDeltaT, float fScaleA, float fScaleG, float fScaleM)

- void CompDCMMagnetoUpdate (tCompDCM *psDCM, float fMagnetoX, float fMagnetoY, float fMagnetoZ)
- void CompDCMMatrixGet (tCompDCM *psDCM, float ppfDCM[3][3])
- void CompDCMStart (tCompDCM *psDCM)
- void CompDCMUpdate (tCompDCM *psDCM)

6.2.1 Function Documentation

6.2.1.1 CompDCMAccelUpdate

Updates the accelerometer reading used by the complementary filter DCM algorithm.

Prototype:

```
void
CompDCMAccelUpdate(tCompDCM *psDCM,
                   float fAccelX,
                   float fAccelY,
                   float fAccelZ)
```

Parameters:

psDCM is a pointer to the DCM state structure.

fAccelX is the accelerometer reading in the X body axis.

fAccelY is the accelerometer reading in the Y body axis.

fAccelZ is the accelerometer reading in the Z body axis.

Description:

This function updates the accelerometer reading used by the complementary filter DCM algorithm. The accelerometer readings provided to this function are used by subsequent calls to CompDCMStart() and CompDCMUpdate() to compute the attitude estimate.

Returns:

None.

6.2.1.2 CompDCMComputeEulers

Computes the Euler angles from the DCM attitude estimation matrix.

Prototype:

```
void
CompDCMComputeEulers(tCompDCM *psDCM,
                     float *pfRoll,
                     float *pfPitch,
                     float *pfYaw)
```

Parameters:

psDCM is a pointer to the DCM state structure.

pfRoll is a pointer to the value into which the roll is stored.

pfPitch is a pointer to the value into which the pitch is stored.

pfYaw is a pointer to the value into which the yaw is stored.

Description:
This function computes the Euler angles that are represented by the DCM attitude estimation matrix. If any of the Euler angles is not required, the corresponding parameter can be **NULL**.

Returns:
None.

6.2.1.3 CompDCMComputeQuaternion

Computes the quaternion from the DCM attitude estimation matrix.

Prototype:
```
void
CompDCMComputeQuaternion(tCompDCM *psDCM,
                         float pfQuaternion[4])
```

Parameters:
psDCM is a pointer to the DCM state structure.
pfQuaternion is an array into which the quaternion is stored.

Description:
This function computes the quaternion that is represented by the DCM attitude estimation matrix.

Returns:
None.

6.2.1.4 CompDCMGyroUpdate

Updates the gyroscope reading used by the complementary filter DCM algorithm.

Prototype:
```
void
CompDCMGyroUpdate(tCompDCM *psDCM,
                  float fGyroX,
                  float fGyroY,
                  float fGyroZ)
```

Parameters:
psDCM is a pointer to the DCM state structure.
fGyroX is the gyroscope reading in the X body axis.
fGyroY is the gyroscope reading in the Y body axis.
fGyroZ is the gyroscope reading in the Z body axis.

Description:
This function updates the gyroscope reading used by the complementary filter DCM algorithm. The gyroscope readings provided to this function are used by subsequent calls to CompDCMUpdate() to compute the attitude estimate.

Returns:
None.

6.2.1.5 CompDCMInit

Initializes the complementary filter DCM attitude estimation state.

Prototype:

```
void
CompDCMInit(tCompDCM *psDCM,
            float fDeltaT,
            float fScaleA,
            float fScaleG,
            float fScaleM)
```

Parameters:

psDCM is a pointer to the DCM state structure.

fDeltaT is the amount of time between DCM updates, in seconds.

fScaleA is the weight of the accelerometer reading in determining the updated attitude estimation.

fScaleG is the weight of the gyroscope reading in determining the updated attitude estimation.

fScaleM is the weight of the magnetometer reading in determining the updated attitude estimation.

Description:

This function initializes the complementary filter DCM attitude estimation state, and must be called prior to performing any attitude estimation.

New readings must be supplied to the complementary filter DCM attitude estimation algorithm at the rate specified by the *fDeltaT* parameter. Failure to provide new readings at this rate results in inaccuracies in the attitude estimation.

The *fScaleA*, *fScaleG*, and *fScaleM* weights must sum to one.

Returns:

None.

6.2.1.6 CompDCMMagnetoUpdate

Updates the magnetometer reading used by the complementary filter DCM algorithm.

Prototype:

```
void
CompDCMMagnetoUpdate(tCompDCM *psDCM,
                     float fMagnetoX,
                     float fMagnetoY,
                     float fMagnetoZ)
```

Parameters:

psDCM is a pointer to the DCM state structure.

fMagnetoX is the magnetometer reading in the X body axis.

fMagnetoY is the magnetometer reading in the Y body axis.

fMagnetoZ is the magnetometer reading in the Z body axis.

Description:
This function updates the magnetometer reading used by the complementary filter DCM algorithm. The magnetometer readings provided to this function are used by subsequent calls to CompDCMStart() and CompDCMUpdate() to compute the attitude estimate.

Returns:
None.

6.2.1.7 CompDCMMatrixGet

Returns the current DCM attitude estimation matrix.

Prototype:
```
void
CompDCMMatrixGet(tCompDCM *psDCM,
                 float ppfDCM[3][3])
```

Parameters:
psDCM is a pointer to the DCM state structure.
ppfDCM is a pointer to the array into which to store the DCM matrix values.

Description:
This function returns the current value of the DCM matrix.

Returns:
None.

6.2.1.8 CompDCMStart

Starts the complementary filter DCM attitude estimation from an initial sensor reading.

Prototype:
```
void
CompDCMStart(tCompDCM *psDCM)
```

Parameters:
psDCM is a pointer to the DCM state structure.

Description:
This function computes the initial complementary filter DCM attitude estimation state based on the initial accelerometer and magnetometer reading. While not necessary for the attitude estimation to converge, using an initial state based on sensor readings results in quicker convergence.

Returns:
None.

6.2.1.9 CompDCMUpdate

Updates the complementary filter DCM attitude estimation based on an updated set of sensor readings.

Prototype:

```
void
CompDCMUpdate(tCompDCM *psDCM)
```

Parameters:

psDCM is a pointer to the DCM state structure.

Description:

This function updates the complementary filter DCM attitude estimation state based on the current sensor readings. This function must be called at the rate specified to CompDCMInit(), with new readings supplied at an appropriate rate (for example, magnetometers typically sample at a much slower rate than accelerometers and gyroscopes).

Returns:

None.

6.3 Programming Example

The following example shows a simple method for feeding sensor readings into the complementary filter DCM algorithm.

```
void
CompDCMExample(void)
{
    float fAccelX, fAccelY, fAccelZ;
    float fGyroX, fGyroY, fGyroZ;
    float fMagnetoX, fMagnetoY, fMagnetoZ;
    float fRoll, fPitch, fYaw;
    tCompDCM sDCM;

    //
    // Initialize the complementary filter DCM algorithm based on a 10ms update
    // rate with a scaling of 2% accelerometer, 96% gyro, and 2% magnetometer.
    //
    CompDCMInit(&sDCM, 0.01, 0.02, 0.96, 0.02);

    //
    // Read the initial accelerometer and magnetometer values.
    //

    //
    // Start the complementary filter DCM algorithm with the initial sensor
    // readings.
    //
    CompDCMAccelUpdate(&sDCM, fAccelX, fAccelY, fAccelZ);
    CompDCMMagnetoUpdate(&sDCM, fMagnetoX, fMagnetoY, fMagnetoZ);
    CompDCMStart(&sDCM);

    //
    // Loop collecting sensor readings.  Typically, this process would done be
    // in the background, but for the purposes of this example, it is shown in
    // an infinite loop.
```

```
    //
    while(1)
    {
        //
        // Read the updated accelerometer, gyro, and magnetometer values.
        //

        //
        // Update the complementary filter DCM algorithm.
        //
        CompDCMAccelUpdate(&sDCM, fAccelX, fAccelY, fAccelZ);
        CompDCMGyroUpdate(&sDCM, fGyroX, fGyroY, fGyroZ);
        CompDCMMagnetoUpdate(&sDCM, fMagnetoX, fMagnetoY, fMagnetoZ);
        CompDCMUpdate(&sDCM);

        //
        // Extract the Euler angles that correspond to the updated DCM state.
        //
        CompDCMComputeEulers(&sDCM, &fRoll, &fPitch, &fYaw);
    }
}
```

7 I2C Master Driver

7.1 Introduction

The I2C master driver is an interrupt-driven state machine that controls transfers across the I2C bus to an external I2C device. There is a queue of I2C requests that are processed in the order they are inserted, allowing several outstanding requests to be handled back-to-back. This process can be used to send several I2C transactions to a single I2C device, send a single I2C transaction (one per entry in the queue) to several I2C devices, or a combination of both.

The state of the I2C master driver is stored in a state structure that must be passed to every API. This protocol allows multiple instances of the I2C master driver to be used with the multiple I2C modules present on most devices. The application must provide the I2C interrupt handler that is called when the I2C interrupt occurs, and it must simply call the I2CMIntHandler() API in order to handle the interrupt (using the state structure that is passed to that API).

The application can supply a callback function with each I2C request, which is then called at the conclusion of the request to inform the application that the transfer has completed. The callback function is called in the context of the I2C interrupt handler, so the actions performed in the callback function must be aware of this context. The just-completed request has already been removed from the request queue when the callback is executed, so there is guaranteed to be at least one space available in the queue for use by the callback function (for example, allowing it to use the result of a read to determine what to do next, and then issue another I2C request in response).

Prior to initializing the I2C master driver, it is the application's responsibility to perform the following actions:

1. Configure the GPIO pins used for the I2C SCL and SDA pins
2. Enable the I2C module
3. Install an interrupt handler for the I2C interrupt that calls the I2CMIntHandler() API (it is recommended to do this at compile time by placing the interrupt handler into the vector table in flash)

I2C requests can be performed in batched and non-batched mode. Non-batched mode uses a data buffer in memory to perform the entire transfer prior to calling the application-supplied callback function. This mode of operation is typical for I2C requests. In batched mode, a much larger transfer can be performed as a sequence of smaller pieces, or batches. On the I2C bus, the transfer is still composed of a start condition, a sequence of byte transfers, and then a stop condition. From an application perspective, the callback function is called multiple times during the transfer, allowing the application to use a much smaller buffer but having to process the data one batch at a time (as opposed to all at once). Batch mode would typically be used for cases where a large transfer needs to be performed but there is not enough memory available to store the entire transfer.

This driver is contained in `sensorlib/i2cm_drv.c`, with `sensorlib/i2cm_drv.h` containing the API declarations for use by applications.

7.2 API Functions

Functions

- uint_fast8_t I2CMCommand (tI2CMInstance *psInst, uint_fast8_t ui8Addr, const uint8_t *pui8WriteData, uint_fast16_t ui16WriteCount, uint_fast16_t ui16WriteBatchSize, uint8_t *pui8ReadData, uint_fast16_t ui16ReadCount, uint_fast16_t ui16ReadBatchSize, tSensorCallback *pfnCallback, void *pvCallbackData)
- void I2CMInit (tI2CMInstance *psInst, uint32_t ui32Base, uint_fast8_t ui8Int, uint_fast8_t ui8TxDMA, uint_fast8_t ui8RxDMA, uint32_t ui32Clock)
- void I2CMIntHandler (tI2CMInstance *psInst)
- uint_fast8_t I2CMRead (tI2CMInstance *psInst, uint_fast8_t ui8Addr, const uint8_t *pui8WriteData, uint_fast16_t ui16WriteCount, uint8_t *pui8ReadData, uint_fast16_t ui16ReadCount, tSensorCallback pfnCallback, void *pvCallbackData)
- uint_fast8_t I2CMRead16BE (tI2CMRead16BE *psInst, tI2CMInstance *psI2CInst, uint_fast8_t ui8Addr, uint_fast8_t ui8Reg, uint16_t *pui16Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t I2CMReadBatched (tI2CMInstance *psInst, uint_fast8_t ui8Addr, const uint8_t *pui8WriteData, uint_fast16_t ui16WriteCount, uint_fast16_t ui16WriteBatchSize, uint8_t *pui8ReadData, uint_fast16_t ui16ReadCount, uint_fast16_t ui16ReadBatchSize, tSensorCallback pfnCallback, void *pvCallbackData)
- uint_fast8_t I2CMReadModifyWrite16BE (tI2CMReadModifyWrite16 *psInst, tI2CMInstance *psI2CInst, uint_fast8_t ui8Addr, uint_fast8_t ui8Reg, uint_fast16_t ui16Mask, uint_fast16_t ui16Value, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t I2CMReadModifyWrite16LE (tI2CMReadModifyWrite16 *psInst, tI2CMInstance *psI2CInst, uint_fast8_t ui8Addr, uint_fast8_t ui8Reg, uint_fast16_t ui16Mask, uint_fast16_t ui16Value, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t I2CMReadModifyWrite8 (tI2CMReadModifyWrite8 *psInst, tI2CMInstance *psI2CInst, uint_fast8_t ui8Addr, uint_fast8_t ui8Reg, uint_fast8_t ui8Mask, uint_fast8_t ui8Value, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t I2CMTransferResume (tI2CMInstance *psInst, uint8_t *pui8Data)
- uint_fast8_t I2CMWrite (tI2CMInstance *psInst, uint_fast8_t ui8Addr, const uint8_t *pui8Data, uint_fast16_t ui16Count, tSensorCallback pfnCallback, void *pvCallbackData)
- uint_fast8_t I2CMWrite16BE (tI2CMWrite16BE *psInst, tI2CMInstance *psI2CInst, uint_fast8_t ui8Addr, uint_fast8_t ui8Reg, const uint16_t *pui16Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t I2CMWrite8 (tI2CMWrite8 *psInst, tI2CMInstance *psI2CInst, uint_fast8_t ui8Addr, uint_fast8_t ui8Reg, const uint8_t *pui8Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t I2CMWriteBatched (tI2CMInstance *psInst, uint_fast8_t ui8Addr, const uint8_t *pui8Data, uint_fast16_t ui16Count, uint_fast16_t ui16BatchSize, tSensorCallback pfnCallback, void *pvCallbackData)

7.2.1 Function Documentation

7.2.1.1 I2CMCommand

Sends a command to an I2C device.

Prototype:

```
uint_fast8_t
I2CMCommand(tI2CMInstance *psInst,
            uint_fast8_t ui8Addr,
            const uint8_t *pui8WriteData,
            uint_fast16_t ui16WriteCount,
            uint_fast16_t ui16WriteBatchSize,
            uint8_t *pui8ReadData,
            uint_fast16_t ui16ReadCount,
            uint_fast16_t ui16ReadBatchSize,
            tSensorCallback *pfnCallback,
            void *pvCallbackData)
```

Parameters:

psInst is a pointer to the I2C master instance data.

ui8Addr is the address of the I2C device to access.

pui8WriteData is a pointer to the data buffer to be written.

ui16WriteCount is the number of bytes to be written.

ui16WriteBatchSize is the number of bytes in each write batch.

pui8ReadData is a pointer to the buffer to be filled with the read data.

ui16ReadCount is the number of bytes to be read.

ui16ReadBatchSize is the number of bytes to be read in each batch.

pfnCallback is the function to be called when the transfer has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function adds an I2C command to the queue of commands to be sent. If successful, the I2C command is then transferred in the background using the interrupt handler. When the transfer is complete, the callback function, if provided, is called in the context of the I2C master interrupt handler.

The first byte of *pui8WriteData* contains the I2C address of the device to access, the next *ui16WriteCount* bytes contains the data to be written to the device. The data read from the device is written into the first *ui16ReadCount* bytes of *pui8ReadData*. The *ui16WriteCount* or *ui16ReadCount* parameters can be zero if there are no bytes to be read or written. The write bytes are sent to the device first, and then the read bytes are read from the device afterward.

If *ui16WriteBatchSize* is less than *ui16WriteCount*, the write portion of the transfer is broken up into as many *ui16WriteBatchSize* batches as required to write *ui16WriteCount* bytes. After each batch, the callback function is called with an **I2CM_STATUS_BATCH_DONE** status, and the transfer is paused (with the I2C bus held). The transfer is resumed when I2CMTransferResume() is called. This procedure can be used to perform very large writes without requiring all the data be available at once, at the expense of tying up the I2C bus for the extended duration of the transfer.

If *ui16ReadBatchSize* is less than *ui16ReadCount*, the read portion of the transfer is broken up into as many *ui16ReadBatchSize* batches as required to read *ui16ReadCount* bytes. After each batch, the callback function is called with an **I2CM_STATUS_BATCH_READY** status, and the transfer is paused (with the I2C bus held). The transfer is resumed when I2CMTransferResume() is called. This procedure can be used to perform very large reads without requiring a large SRAM buffer, at the expense of tying up the I2C bus for the extended duration of the transfer.

Returns:
Returns 1 if the command was successfully added to the queue and 0 if it was not.

7.2.1.2 I2CMInit

Initializes the I2C master driver.

Prototype:
```
void
I2CMInit(tI2CMInstance *psInst,
         uint32_t ui32Base,
         uint_fast8_t ui8Int,
         uint_fast8_t ui8TxDMA,
         uint_fast8_t ui8RxDMA,
         uint32_t ui32Clock)
```

Parameters:
psInst is a pointer to the I2C master instance data.
ui32Base is the base address of the I2C module.
ui8Int is the interrupt number for the I2C module.
ui8TxDMA is the uDMA channel number used for transmitting data to the I2C module.
ui8RxDMA is the uDMA channel number used for receiving data from the I2C module.
ui32Clock is the clock frequency of the input clock to the I2C module.

Description:
This function prepares both the I2C master module and driver for operation, and must be the first I2C master driver function called for each I2C master instance. It is assumed that the application has enabled the I2C module, configured the I2C pins, and provided an I2C interrupt handler that calls I2CMIntHandler().

The uDMA module cannot be used at present to transmit/receive data, so the *ui8TxDMA* and *ui8RxDMA* parameters are unused. They are reserved for future use and should be set to 0xff in order to ensure future compatibility.

Returns:
None.

7.2.1.3 I2CMIntHandler

Handles I2C master interrupts.

Prototype:
```
void
I2CMIntHandler(tI2CMInstance *psInst)
```

Parameters:
psInst is a pointer to the I2C master instance data.

Description:
This function performs the processing required in response to an I2C interrupt. The application-supplied interrupt handler should call this function with the correct instance data in response to the I2C interrupt.

Returns:
None.

7.2.1.4 I2CMRead

Reads data from an I2C device.

Prototype:

```
uint_fast8_t
I2CMRead(tI2CMInstance *psInst,
         uint_fast8_t ui8Addr,
         const uint8_t *pui8WriteData,
         uint_fast16_t ui16WriteCount,
         uint8_t *pui8ReadData,
         uint_fast16_t ui16ReadCount,
         tSensorCallback pfnCallback,
         void *pvCallbackData)
```

Parameters:

psInst is a pointer to the I2C master instance data.

ui8Addr is the address of the I2C device to access.

pui8WriteData is a pointer to the data buffer to be written.

ui16WriteCount is the number of bytes to be written.

pui8ReadData is a pointer to the buffer to be filled with the read data.

ui16ReadCount is the number of bytes to be read.

pfnCallback is the function to be called when the transfer has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function adds an I2C read to the queue of commands to be sent. If successful, the I2C read is then performed in the background using the interrupt handler. When the read is complete, the callback function, if provided, is called in the context of the I2C master interrupt handler.

The first byte of *pui8WriteData* contains the I2C address of the device to access, the next *ui16WriteCount* bytes contains the data to be written to the device. The data read from the device is written into the first *ui16ReadCount* bytes of *pui8ReadData*. The *ui16WriteCount* or *ui16ReadCount* parameters can be zero if there are no bytes to be read or written. The write bytes are sent to the device first, and then the read bytes are read from the device afterward.

Returns:

Returns 1 if the command was successfully added to the queue and 0 if it was not.

7.2.1.5 I2CMRead16BE

Performs a read of 16-bit big-endian data from an I2C device.

Prototype:

```
uint_fast8_t
I2CMRead16BE(tI2CMRead16BE *psInst,
             tI2CMInstance *psI2CInst,
             uint_fast8_t ui8Addr,
             uint_fast8_t ui8Reg,
             uint16_t *pui16Data,
             uint_fast16_t ui16Count,
             tSensorCallback *pfnCallback,
             void *pvCallbackData)
```

Parameters:

psInst is a pointer to the 16-bit big-endian read instance data.

psI2CInst is a pointer to the I2C master instance data.

ui8Addr is the address of the I2C device to access.

ui8Reg is the register in the I2C device to access.

pui16Data is a pointer to the buffer to be filled with the register data.

ui16Count is the number of 16-bit register values to be read.

pfnCallback is the function to be called when the read has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initiates a read transaction of 16-bit big-endian data from an I2C device. The data is provided by the device in big-endian format and is byte-swapped as it is read from the I2C device, returning the data in little-endian format.

Returns:

Returns 1 if the command was successfully added to the queue and 0 if it was not.

7.2.1.6 I2CMReadBatched

Reads data in batches from an I2C device.

Prototype:

```
uint_fast8_t
I2CMReadBatched(tI2CMInstance *psInst,
                uint_fast8_t ui8Addr,
                const uint8_t *pui8WriteData,
                uint_fast16_t ui16WriteCount,
                uint_fast16_t ui16WriteBatchSize,
                uint8_t *pui8ReadData,
                uint_fast16_t ui16ReadCount,
                uint_fast16_t ui16ReadBatchSize,
                tSensorCallback pfnCallback,
                void *pvCallbackData)
```

Parameters:

psInst is a pointer to the I2C master instance data.

ui8Addr is the address of the I2C device to access.

pui8WriteData is a pointer to the data buffer to be written.

ui16WriteCount is the number of bytes to be written.

ui16WriteBatchSize is the number of bytes in each write batch.

pui8ReadData is a pointer to the buffer to be filled with the read data.

ui16ReadCount is the number of bytes to be read.

ui16ReadBatchSize is the number of bytes in each read batch.

pfnCallback is the function to be called when the transfer has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function adds an I2C read to the queue of commands to be sent. If successful, the I2C read is then performed in the background using the interrupt handler. When the read is complete, the callback function, if provided, is called in the context of the I2C master interrupt handler.

The first byte of *pui8WriteData* contains the I2C address of the device to access, the next *ui16WriteCount* bytes contains the data to be written to the device. The data read from the device is written into the first *ui16ReadCount* bytes of *pui8ReadData*. The *ui16WriteCount* or *ui16ReadCount* parameters can be zero if there are no bytes to be read or written. The write bytes are sent to the device first, and then the read bytes are read from the device afterward.

The data is written in batches of *ui16WriteBatchSize*. The callback function is called after each batch is written, and I2CMTransferResume() must be called when the next batch should be written.

The data is read in batches of *ui16ReadBatchSize*. The callback function is called after each batch is read, and I2CMTransferResume() must be called when the next batch should be read.

Returns:

Returns 1 if the command was successfully added to the queue and 0 if it was not.

7.2.1.7 I2CMReadModifyWrite16BE

Performs a read-modify-write of 16 bits of big-endian data in an I2C device.

Prototype:

```
uint_fast8_t
I2CMReadModifyWrite16BE(tI2CMReadModifyWrite16 *psInst,
                        tI2CMInstance *psI2CInst,
                        uint_fast8_t ui8Addr,
                        uint_fast8_t ui8Reg,
                        uint_fast16_t ui16Mask,
                        uint_fast16_t ui16Value,
                        tSensorCallback *pfnCallback,
                        void *pvCallbackData) [inline]
```

Parameters:

psInst is a pointer to the read-modify-write instance data.

psI2CInst is a pointer to the I2C master instance data.

ui8Addr is the address of the I2C device to access.

ui8Reg is the register in the I2C device to access.

ui16Mask is the mask indicating the register bits that should be maintained.

ui16Value is the value indicating the new value for the register bits that are not maintained.

pfnCallback is the function to be called when the write has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initiates a read-modify-write transaction of 16 bits of big-endian data in an I2C device. The modify portion of the operation is performed by AND-ing the register value with *ui16Mask* and then OR-ing the result with *ui16Value*. When the read-modify-write is complete, the callback function, if provided, is called in the context of the I2C master interrupt handler.

If the mask (in *ui16Mask*) is zero, then none of the bits in the current register value are maintained. In this case, the read portion of the read-modify-write is bypassed, and the new register value (in *ui16Value*) is directly written to the I2C device.

Returns:

Returns 1 if the command was successfully added to the queue and 0 if it was not.

7.2.1.8 I2CMReadModifyWrite16LE

Performs a read-modify-write of 16 bits of little-endian data in an I2C device.

Prototype:

```
uint_fast8_t
I2CMReadModifyWrite16LE(tI2CMReadModifyWrite16 *psInst,
                        tI2CMInstance *psI2CInst,
                        uint_fast8_t ui8Addr,
                        uint_fast8_t ui8Reg,
                        uint_fast16_t ui16Mask,
                        uint_fast16_t ui16Value,
                        tSensorCallback *pfnCallback,
                        void *pvCallbackData)
```

Parameters:

psInst is a pointer to the read-modify-write instance data.

psI2CInst is a pointer to the I2C master instance data.

ui8Addr is the address of the I2C device to access.

ui8Reg is the register in the I2C device to access.

ui16Mask is the mask indicating the register bits that should be maintained.

ui16Value is the value indicating the new value for the register bits that are not maintained.

pfnCallback is the function to be called when the write has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initiates a read-modify-write transaction of 16 bits of little-endian data in an I2C device. The modify portion of the operation is performed by AND-ing the register value with *ui16Mask* and then OR-ing the result with *ui16Value*. When the read-modify-write is complete, the callback function, if provided, is called in the context of the I2C master interrupt handler.

If the mask (in *ui16Mask*) is zero, then none of the bits in the current register value are maintained. In this case, the read portion of the read-modify-write is bypassed, and the new register value (in *ui16Value*) is directly written to the I2C device.

Returns:

Returns 1 if the command was successfully added to the queue and 0 if it was not.

7.2.1.9 I2CMReadModifyWrite8

Performs a read-modify-write of 8 bits of data in an I2C device.

Prototype:

```
uint_fast8_t
I2CMReadModifyWrite8(tI2CMReadModifyWrite8 *psInst,
                     tI2CMInstance *psI2CInst,
                     uint_fast8_t ui8Addr,
                     uint_fast8_t ui8Reg,
                     uint_fast8_t ui8Mask,
                     uint_fast8_t ui8Value,
                     tSensorCallback *pfnCallback,
                     void *pvCallbackData)
```

Parameters:

psInst is a pointer to the read-modify-write instance data.

psI2CInst is a pointer to the I2C master instance data.

ui8Addr is the address of the I2C device to access.

ui8Reg is the register in the I2C device to access.

ui8Mask is the mask indicating the register bits that should be maintained.

ui8Value is the value indicating the new value for the register bits that are not maintained.

pfnCallback is the function to be called when the write has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initiates a read-modify-write transaction of 8 bits of data in an I2C device. The modify portion of the operation is performed by AND-ing the register value with *ui8Mask* and then OR-ing the result with *ui8Value*. When the read-modify-write is complete, the callback function, if provided, is called in the context of the I2C master interrupt handler.

If the mask (in *ui8Mask*) is zero, then none of the bits in the current register value are maintained. In this case, the read portion of the read-modify-write is bypassed, and the new register value (in *ui8Value*) is directly written to the I2C device.

Returns:

Returns 1 if the command was successfully added to the queue and 0 if it was not.

7.2.1.10 I2CMTransferResume

Resumes an I2C transfer.

Prototype:

```
uint_fast8_t
I2CMTransferResume(tI2CMInstance *psInst,
                   uint8_t *pui8Data)
```

Parameters:

psInst is a pointer to the I2C master instance data.

pui8Data is a pointer to the buffer to be used for the next batch of data.

Description:

This function resumes an I2C transfer that has been paused via the use of the write or read batch size capability.

Returns:

Returns 1 if the transfer was resumed and 0 if there was not a paused transfer to resume.

7.2.1.11 I2CMWrite

Writes data to an I2C device.

Prototype:

```
uint_fast8_t
I2CMWrite(tI2CMInstance *psInst,
          uint_fast8_t ui8Addr,
          const uint8_t *pui8Data,
          uint_fast16_t ui16Count,
          tSensorCallback pfnCallback,
          void *pvCallbackData)
```

Parameters:

psInst is a pointer to the I2C master instance data.

ui8Addr is the address of the I2C device to access.

pui8Data is a pointer to the data buffer to be written.

ui16Count is the number of bytes to be written.

pfnCallback is the function to be called when the write has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function adds an I2C write to the queue of commands to be sent. If successful, the I2C write is then performed in the background using the interrupt handler. When the write is complete, the callback function, if provided, is called in the context of the I2C master interrupt handler.

The first byte of the data buffer contains the I2C address of the device to access, and the remaining *ui16Count* bytes contain the data to be written to the device. The *ui16Count* parameter can be zero if there are no bytes to be written.

Returns:

Returns 1 if the command was successfully added to the queue and 0 if it was not.

7.2.1.12 I2CMWrite16BE

Performs a write of 16-bit big-endian data to an I2C device.

Prototype:

```
uint_fast8_t
I2CMWrite16BE(tI2CMWrite16BE *psInst,
              tI2CMInstance *psI2CInst,
              uint_fast8_t ui8Addr,
              uint_fast8_t ui8Reg,
              const uint16_t *pui16Data,
              uint_fast16_t ui16Count,
              tSensorCallback *pfnCallback,
              void *pvCallbackData)
```

Parameters:

psInst is a pointer to the 16-bit big-endian write instance data.

psI2CInst is a pointer to the I2C master instance data.

ui8Addr is the address of the I2C device to access.

ui8Reg is the register in the I2C device to access.

pui16Data is a pointer to the register data to be written.

ui16Count is the number of 16-bit register values to be written.

pfnCallback is the function to be called when the write has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initiates a write transaction of 16-bit big-endian data to an I2C device. The data in the buffer is provided in little-endian format and is byte-swapped as it is being written to the I2C device.

Returns:

Returns 1 if the command was successfully added to the queue and 0 if it was not.

7.2.1.13 I2CMWrite8

Performs a write of 8-bit data to an I2C device.

Prototype:

```
uint_fast8_t
I2CMWrite8(tI2CMWrite8 *psInst,
           tI2CMInstance *psI2CInst,
           uint_fast8_t ui8Addr,
           uint_fast8_t ui8Reg,
           const uint8_t *pui8Data,
           uint_fast16_t ui16Count,
           tSensorCallback *pfnCallback,
           void *pvCallbackData)
```

Parameters:

psInst is a pointer to the 8-bit write instance data.

psI2CInst is a pointer to the I2C master instance data.

ui8Addr is the address of the I2C device to access.

ui8Reg is the register in the I2C device to access.

pui8Data is a pointer to the register data to be written.

ui16Count is the number of register values to be written.

pfnCallback is the function to be called when the write has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initiates a write transaction of 8-bit data to an I2C device.

Returns:

Returns 1 if the command was successfully added to the queue and 0 if it was not.

7.2.1.14 I2CMWriteBatched

Writes data in batches to an I2C device.

Prototype:

```
uint_fast8_t
I2CMWriteBatched(tI2CMInstance *psInst,
                 uint_fast8_t ui8Addr,
                 const uint8_t *pui8Data,
                 uint_fast16_t ui16Count,
                 uint_fast16_t ui16BatchSize,
                 tSensorCallback pfnCallback,
                 void *pvCallbackData)
```

Parameters:

psInst is a pointer to the I2C master instance data.

ui8Addr is the address of the I2C device to access.

pui8Data is a pointer to the data buffer to be written.

ui16Count is the number of bytes to be written.

ui16BatchSize is the number of bytes in each write batch.

pfnCallback is the function to be called when the transfer has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function adds an I2C write to the queue of commands to be sent. If successful, the I2C write is then performed in the background using the interrupt handler. When the write is complete, the callback function, if provided, is called in the context of the I2C master interrupt handler.

The first byte of the data buffer contains the I2C address of the device to access, and the remaining *ui16Count* bytes contain the data to be written to the device. The *ui16Count* parameter can be zero if there are no bytes to be written.

The data is written in batches of *ui16WriteBatchSize*. The callback function is called after each batch is written, and I2CMTransferResume() must be called when the next batch should be written.

Returns:

Returns 1 if the command was successfully added to the queue and 0 if it was not.

7.3 Programming Example

The following example shows how to perform simple I2C reads and writes to I2C devices.

```
//
// The I2C master driver instance data.
//
tI2CMInstance g_sI2CMSimpleInst;

//
// A boolean that is set when an I2C transaction is completed.
//
volatile bool g_bI2CMSimpleDone = true;

//
// The interrupt handler for the I2C module.
//
void
I2CMSimpleIntHandler(void)
{
    //
    // Call the I2C master driver interrupt handler.
    //
    I2CMIntHandler(&g_sI2CMSimpleInst);
}

//
// The function that is provided by this example as a callback when I2C
// transactions have completed.
//
void
I2CMSimpleCallback(void *pvData, uint_fast8_t ui8Status)
{
    //
    // See if an error occurred.
    //
    if(ui8Status != I2CM_STATUS_SUCCESS)
    {
        //
        // An error occurred, so handle it here if required.
        //
    }

    //
    // Indicate that the I2C transaction has completed.
    //
    g_bI2CMSimpleDone = true;
}

//
// The simple I2C master driver example.
//
void
I2CMSimpleExample(void)
{
    uint8_t pui8Data[4];

    //
    // Initialize the I2C master driver.  It is assumed that the I2C module has
    // already been enabled and the I2C pins have been configured.
    //
    I2CMInit(&g_sI2CMSimpleInst, I2C0_BASE, INT_I2C0, 0xff, 0xff, 80000000);

    //
```

```
    // Write two bytes of data to the I2C device at address 0x22.
    //
    g_bI2CMSimpleDone = false;
    I2CMWrite(&g_sI2CMSimpleInst, 0x22, pui8Data, 2, I2CMSimpleCallback, 0);
    while(!g_bI2CMSimpleDone)
    {
    }

    //
    // Read four bytes of data from the I2C device at address 0x31.
    //
    g_bI2CMSimpleDone = false;
    I2CMRead(&g_sI2CMSimpleInst, 0x31, pui8Data, 1, pui8Data, 4,
             I2CMSimpleCallback, 0);
    while(!g_bI2CMSimpleDone)
    {
    }
}
```

The following example shows how to perform batched reads and writes of large amounts of data using small buffers.

```
//
// The I2C master instance structure.
//
tI2CMInstance g_sI2CMBatchInst;

//
// The data buffer used for the read and write.
//
uint8_t g_pui8Data[256];

//
// A boolean that is set when an I2C transaction is completed.
//
volatile bool g_bI2CMBatchDone = true;

//
// The interrupt handler for the I2C module.
//
void
I2CMBatchIntHandler(void)
{
    //
    // Call the I2C master driver interrupt handler.
    //
    I2CMIntHandler(&g_sI2CMBatchInst);
}

//
// The function that is provided by this example as a callback when batched I2C
// read transactions have completed.
//
void
I2CMBatchReadCallback(void *pvData, uint_fast8_t ui8Status)
{
    //
    // Do something with the data that has been read into g_pui8Data.
    //

    //
    // See if there is more data to be read.
    //
    if(ui8Status == I2CM_STATUS_BATCH_READY)
    {
```

```
        //
        // Resume the batched read.
        //
        I2CMTransferResume(&g_sI2CMBatchInst, g_pui8Data);
    }
    else
    {
        //
        // Indicate that the I2C batched read transaction has completed.
        //
        g_bI2CMBatchDone = true;
    }
}

//
// The function that is provided by this example as a callback when batched I2C
// write transactions have completed.
//
void
I2CMBatchWriteCallback(void *pvData, uint_fast8_t ui8Status)
{
    //
    // See if there is more data to be written.
    //
    if(ui8Status == I2CM_STATUS_BATCH_DONE)
    {
        //
        // Produce new data into g_pui8Data.
        //

        //
        // Resume the batched write.
        //
        I2CMTransferResume(&g_sI2CMBatchInst, g_pui8Data);
    }
    else
    {
        //
        // Indicate that the I2C batched write transaction has completed.
        //
        g_bI2CMBatchDone = true;
    }
}

//
// The batched I2C master driver example.
//
void
I2CMBatchExample(void)
{
    //
    // Initialize the I2C master driver.  It is assumed that the I2C module has
    // already been enabled and the I2C pins have been configured.
    //
    I2CMInit(&g_sI2CMBatchInst, I2C0_BASE, INT_I2C0, 0xff, 0xff, 80000000);

    //
    // Write 8192 bytes of data to the I2C device at address 0x22 in 256 byte
    // batches.
    //
    g_bI2CMBatchDone = false;
    I2CMWriteBatched(&g_sI2CMBatchInst, 0x22, g_pui8Data, 8192, 256,
                     I2CMBatchWriteCallback, 0);
    while(!g_bI2CMBatchDone)
    {
    }
```

```
    //
    // Read 8192 bytes of data from the I2C device at address 0x31 in 256 byte
    // batches.
    //
    g_bI2CMBatchDone = false;
    I2CMReadBatched(&g_sI2CMBatchInst, 0x31, g_pui8Data, 1, 1, g_pui8Data,
                    8192, 256, I2CMBatchReadCallback, 0);
    while(!g_bI2CMBatchDone)
    {
    }
}
```

8 ISL29023 Ambient Light Sensor Driver

8.1 Introduction

The ISL29023 is an ambient and infrared light sensor produced by Intersil. This driver allows the ISL29023 to be accessed via the I2C bus.

This driver is contained in `sensorlib/isl29023.c`, with `sensorlib/isl29023.h` containing the API declarations for use by applications.

8.2 API Functions

Functions

- void ISL29023DataLightIRGetFloat (tISL29023 *psInst, float *pfIR)
- void ISL29023DataLightIRGetRaw (tISL29023 *psInst, uint16_t *pui16IR)
- void ISL29023DataLightVisibleGetFloat (tISL29023 *psInst, float *pfVisibleLight)
- void ISL29023DataLightVisibleGetRaw (tISL29023 *psInst, uint16_t *pui16Visible)
- uint_fast8_t ISL29023DataRead (tISL29023 *psInst, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t ISL29023Init (tISL29023 *psInst, tI2CMInstance *psI2CInst, uint_fast8_t ui8I2CAddr, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t ISL29023Read (tISL29023 *psInst, uint_fast8_t ui8Reg, uint8_t *pui8Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t ISL29023ReadModifyWrite (tISL29023 *psInst, uint_fast8_t ui8Reg, uint8_t ui8Mask, uint8_t ui8Value, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t ISL29023Write (tISL29023 *psInst, uint_fast8_t ui8Reg, uint8_t *pui8Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)

8.2.1 Function Documentation

8.2.1.1 ISL29023DataLightIRGetFloat

Gets the measurement data from the most recent data read.

Prototype:

```
void
ISL29023DataLightIRGetFloat(tISL29023 *psInst,
                            float *pfIR)
```

Parameters:

psInst is a pointer to the ISL29023 instance data.

pfIR is a pointer to the value into which the IR data is stored as floating point lux.

Description:

This function returns the IR data from the most recent data read, converted into lux.

Returns:

None.

8.2.1.2 ISL29023DataLightIRGetRaw

Gets the raw measurement data from the most recent data read.

Prototype:

```
void
ISL29023DataLightIRGetRaw(tISL29023 *psInst,
                          uint16_t *pui16IR)
```

Parameters:

psInst is a pointer to the ISL29023 instance data.

pui16IR is a pointer to the value into which the raw IR data is stored.

Description:

This function returns the raw measurement data from the most recent data read. The data is not manipulated in any way by the driver.

Returns:

None.

8.2.1.3 ISL29023DataLightVisibleGetFloat

Gets the measurement data from the most recent data read.

Prototype:

```
void
ISL29023DataLightVisibleGetFloat(tISL29023 *psInst,
                                 float *pfVisibleLight)
```

Parameters:

psInst is a pointer to the ISL29023 instance data.

pfVisibleLight is a pointer to the value into which the light data is stored as floating point lux.

Description:

This function returns the light data from the most recent data read, converted into lux.

Returns:

None.

8.2.1.4 ISL29023DataLightVisibleGetRaw

Gets the raw measurement data from the most recent data read.

Prototype:

```
void
ISL29023DataLightVisibleGetRaw(tISL29023 *psInst,
                               uint16_t *pui16Visible)
```

Parameters:

psInst is a pointer to the ISL29023 instance data.

pui16Visible is a pointer to the value into which the raw light data is stored.

Description:

This function returns the raw measurement data from the most recent data read. The data is not manipulated in any way by the driver.

Returns:

None.

8.2.1.5 ISL29023DataRead

Reads the light data from the ISL29023.

Prototype:

```
uint_fast8_t
ISL29023DataRead(tISL29023 *psInst,
                 tSensorCallback *pfnCallback,
                 void *pvCallbackData)
```

Parameters:

psInst is a pointer to the ISL29023 instance data.

pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initiates a read of the ISL29023 data registers. When the read has completed (as indicated by calling the callback function), the new readings can be obtained via:

- ISL29023DataLightVisibleGetRaw()
- ISL29023DataLightVisibleGetFloat()
- ISL29023DataLightIRGetRaw()
- ISL29023DataLightIRGetFloat()

Returns:

Returns 1 if the read was successfully started and 0 if it was not.

8.2.1.6 ISL29023Init

Initializes the ISL29023 driver.

Prototype:

```
uint_fast8_t
ISL29023Init(tISL29023 *psInst,
             tI2CMInstance *psI2CInst,
             uint_fast8_t ui8I2CAddr,
             tSensorCallback *pfnCallback,
             void *pvCallbackData)
```

Parameters:

psInst is a pointer to the ISL29023 instance data.

psI2CInst is a pointer to the I2C driver instance data.

ui8I2CAddr is the I2C address of the ISL29023 device.

pfnCallback is the function to be called when the initialization has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initializes the ISL29023 driver, preparing it for operation. This function also asserts a reset signal to the ISL29023 to clear any previous configuration data.

Returns:

Returns 1 if the ISL29023 driver was successfully initialized and 0 if it was not.

8.2.1.7 ISL29023Read

Reads data from ISL29023 registers.

Prototype:

```
uint_fast8_t
ISL29023Read(tISL29023 *psInst,
             uint_fast8_t ui8Reg,
             uint8_t *pui8Data,
             uint_fast16_t ui16Count,
             tSensorCallback *pfnCallback,
             void *pvCallbackData)
```

Parameters:

psInst is a pointer to the ISL29023 instance data.

ui8Reg is the first register to read.

pui8Data is a pointer to the location to store the data that is read.

ui16Count is the number of data bytes to read.

pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function reads a sequence of data values from consecutive registers in the ISL29023.

Returns:

Returns 1 if the write was successfully started and 0 if it was not.

8.2.1.8 ISL29023ReadModifyWrite

Performs a read-modify-write of an ISL29023 register.

Prototype:

```
uint_fast8_t
ISL29023ReadModifyWrite(tISL29023 *psInst,
                        uint_fast8_t ui8Reg,
                        uint8_t ui8Mask,
                        uint8_t ui8Value,
                        tSensorCallback *pfnCallback,
                        void *pvCallbackData)
```

Parameters:

psInst is a pointer to the ISL29023 instance data.

ui8Reg is the register to modify.

ui8Mask is the bit mask that is ANDed with the current register value.

ui8Value is the bit mask that is ORed with the result of the AND operation.

pfnCallback is the function to be called when the data has been changed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function changes the value of a register in the ISL29023 via a read-modify-write operation, allowing one of the fields to be changed without disturbing the other fields. The *ui8Reg* register is read, ANDed with *ui8Mask*, ORed with *ui8Value*, and then written back to the ISL29023.

Returns:

Returns 1 if the read-modify-write was successfully started and 0 if it was not.

8.2.1.9 ISL29023Write

Write register data to the ISL29023.

Prototype:

```
uint_fast8_t
ISL29023Write(tISL29023 *psInst,
              uint_fast8_t ui8Reg,
              uint8_t *pui8Data,
              uint_fast16_t ui16Count,
              tSensorCallback *pfnCallback,
              void *pvCallbackData)
```

Parameters:

psInst is a pointer to the ISL29023 instance data.

ui8Reg is the first register to write.

pui8Data is a pointer to the data to write.

ui16Count is the number of data bytes to write.

pfnCallback is the function to be called when the data has been written (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function writes a sequence of data values to consecutive registers in the ISL29023. The first byte of the *pui8Data* buffer contains the value to be written into the *ui8Reg* register, the second value contains the data to be written into the next register, and so on.

Returns:

Returns 1 if the write was successfully started and 0 if it was not.

8.3 Programming Example

The following example shows how to initialize the ISL29023 and read light data from it.

```
//
// A boolean that is set when a ISL29023 command has completed.
//
volatile bool g_bISL29023Done;

//
// The function that is provided by this example as a callback when ISL29023
// transactions have completed.
//
void
ISL29023Callback(void *pvCallbackData, uint_fast8_t ui8Status)
{
    //
    // See if an error occurred.
    //
    if(ui8Status != I2CM_STATUS_SUCCESS)
    {
        //
        // An error occurred, so handle it here if required.
        //
    }

    //
    // Indicate that the ISL29023 transaction has completed.
    //
    g_bISL29023Done = true;
}

//
// The ISL29023 example.
//
void
ISL29023Example(void)
{
    tI2CMInstance sI2CInst;
    tISL29023 sISL29023;
    float fVisible, fIR;

    //
    // Initialize the ISL29023.  This code assumes that the I2C master instance
    // has already been initialized.
    //
```

```
    g_bISL29023Done = false;
    ISL29023Init(&sISL29023, &sI2CInst, 0x44, ISL29023Callback, 0);
    while(!g_bISL29023Done)
    {
    }

    //
    // Loop forever reading data from the ISL29023.  Typically, this process
    // would be done in the background, but for the purposes of this example,
    // it is shown in an infinite loop.
    //
    while(1)
    {
        //
        // Request another reading from the ISL29023.
        //
        g_bISL29023Done = false;
        ISL29023DataRead(&sISL29023, ISL29023Callback, 0);
        while(!g_bISL29023Done)
        {
        }

        //
        // Get the new light readings.
        //
        ISL29023DataLightVisibleGetFloat(&sISL29023, &fVisible);
        ISL29023DataLightIRGetFloat(&sISL29023, &fIR);

        //
        // Do something with the new light readings.
        //
    }
}
```

9 Magnetometer Module

9.1 Introduction

The magnetometer module provides a set of magnetometer-independent functions for performing common operations on the data captured by a magnetometer. These functions operate on the data obtained from any magnetometer and are unit-agnostic because they depend only upon the relative magnitude of the readings.

The magnetometer compensation routines provide hard- and soft-iron compensation of the magnetometer reads. Iron in the proximity of the magnetometer causes local disturbances in the earth's magnetic field; these disturbances can be measured with the magnetometer and canceled via the compensation routines (MagnetoCompensateInit() and MagnetoCompensate()).

The magnetometer reading can be combined with tilt information from an accelerometer to produce a tilt-compensated compass. The MagnetoHeadingCompute() function performs this operation; which should be done after hard- and soft-iron compensation for improved accuracy.

This module is contained in `sensorlib/magneto.c`, with `sensorlib/magneto.h` containing the API declarations for use by applications.

9.2 API Functions

Functions

- void MagnetoCompensate (tMagnetoCompensation *psInst, float *pfMagnetoX, float *pfMagnetoY, float *pfMagnetoZ)
- void MagnetoCompensateInit (tMagnetoCompensation *psInst, float fXOffset, float fYOffset, float fZOffset, float fXYAngle, float fYRatio, float fXZAngle, float fZRatio)
- float MagnetoHeadingCompute (float fMagnetoX, float fMagnetoY, float fMagnetoZ, float fRoll, float fPitch)

9.2.1 Function Documentation

9.2.1.1 MagnetoCompensate

Performs hard- and soft-iron compensation on magnetometer readings.

Prototype:

```
void
MagnetoCompensate(tMagnetoCompensation *psInst,
                  float *pfMagnetoX,
```

```
                        float *pfMagnetoY,
                        float *pfMagnetoZ)
```

Parameters:

psInst is a pointer to the magnetometer compensation state structure.

pfMagnetoX is a pointer to the magnetometer X-axis reading.

pfMagnetoY is a pointer to the magnetometer Y-axis reading.

pfMagnetoZ is a pointer to the magnetometer Z-axis reading.

Description:

This function performs hard- and soft-iron compensation on the given magnetometer reading. Hard-iron distortions cause a fixed offset in the reading, regardless of orientation. Hard-iron compensation is performed by negating this fixed offset.

Soft-iron distortion is more complicated, causing an offset that varies as the sensor rotates, which results in the sensor returning an ellipse as it rotates instead of a circle. Performing soft-iron compensation requires rotating the sensor reading such that the major axis of the ellipse is aligned with one of the magnetometer axes, scaling one of the axes, then rotating the scaled sensor reading back. This operation is performed two times; once to scale the Y axis to the same scale as the X axis, and once again to scale the Z axis to the same scale as the X axis.

Hard-iron compensation is performed prior to soft-iron compensation.

Returns:

None.

9.2.1.2 MagnetoCompensateInit

Initializes the magnetometer hard- and soft-iron compensation state.

Prototype:

```
void
MagnetoCompensateInit(tMagnetoCompensation *psInst,
                      float fXOffset,
                      float fYOffset,
                      float fZOffset,
                      float fXYAngle,
                      float fYRatio,
                      float fXZAngle,
                      float fZRatio)
```

Parameters:

psInst is a pointer to the magnetometer compensation state structure.

fXOffset is the hard-iron compensation for the X axis.

fYOffset is the hard-iron compensation for the Y axis.

fZOffset is the hard-iron compensation for the Z axis.

fXYAngle is the amount to rotate around the Z axis prior to scaling the Y axis reading, in radians.

fYRatio is the amount to scale the Y axis reading.

fXZAngle is the amount to rotate around the Y axis prior to scaling the Z axis reading, in radians.

fZRatio is the amount to scale the Z axis reading.

Description:
This function initializes the magnetometer compensation state structure with the values that are used to perform hard- and soft-iron compensation of magnetometer readings.

Returns:
None.

9.2.1.3 MagnetoHeadingCompute

Computes the compass heading from magnetometer data and roll/pitch.

Prototype:

```
float
MagnetoHeadingCompute(float fMagnetoX,
                      float fMagnetoY,
                      float fMagnetoZ,
                      float fRoll,
                      float fPitch)
```

Parameters:
fMagnetoX is the X component of the magnetometer reading.
fMagnetoY is the Y component of the magnetometer reading.
fMagnetoZ is the Z component of the magnetometer reading.
fRoll is the roll angle, in radians.
fPitch is the pitch angle, in radians.

Description:
This function computes the compass heading by performing tilt compensation on the magnetometer reading.

Returns:
Returns the compass heading, in radians.

9.3 Programming Example

The following example shows how to perform hard- and soft-iron compensation on magnetometer readings.

```
void
MagnetoCompensateExample(void)
{
    float fMagnetoX, fMagnetoY, fMagnetoZ;
    tMagnetoCompensation sMagComp;

    //
    // Initialize the magnetometer compensation.  In this example, measurements
    // have shown a hard-iron distortion of (0.05, -0.03, 0.025) and a
    // soft-iron distortion of 0.9 in the Y axis after a 52 degree rotation and
    // 0.89 in the Z axis after a -33 degree rotation.
```

```
    //
    MagnetoCompensateInit(&sMagComp, 0.05, -0.03, 0.025, (52.0 * M_PI) / 180.0,
                          0.9, (-33.0 * M_PI) / 180.0, 0.9);

    //
    // Loop reading magnetometer values.  Typically, this process would be done
    // in the background, but for the purposes of this example, it is shown in
    // an infinite loop.
    //
    while(1)
    {
        //
        // Read the updated magnetometer value.
        //

        //
        // Perform hard- and soft-iron compensation on the magnetometer
        // reading.
        //
        MagnetoCompensate(&sMagComp, &fMagnetoX, &fMagnetoY, &fMagnetoZ);
    }
}
```

The following example shows how to compute the compass heading from magnetometer readings. For the purposes of this example, a fixed 5 degree roll and 7.5 degree pitch is assumed for the sensor platform. Normally an accelerometer or a fusion algorithm would be used to dynamically compute the roll and pitch of the sensor platform.

```
void
MagnetoCompassExample(void)
{
    float fMagnetoX, fMagnetoY, fMagnetoZ, fHeading;

    //
    // Loop reading magnetometer values.  Typically, this process would be done
    // in the background, but for the purposes of this example, it is shown in
    // an infinite loop.
    //
    while(1)
    {
        //
        // Read the updated magnetometer value.
        //

        //
        // Compute the compass heading from the magnetometer reading.
        //
        fHeading = MagnetoHeadingCompute(fMagnetoX, fMagnetoY, fMagnetoZ,
                                         (5.0 * M_PI) / 180.0,
                                         (7.5 * M_PI) / 180.0);
    }
}
```

10 MPU6050 Accelerometer and Gyroscope Driver

10.1 Introduction

The MPU6050 is an integrated three-axis accelerometer and gyroscope produced by Invensense Inc. In addition to an internal accelerometer and gyroscope, this device has an auxiliary I2C bus that can be used to automatically sample external sensors. This driver allows the MPU6050 to be accessed via the I2C bus.

When initialized, a soft reset of the MPU6050 is performed, putting it into its default state. The default accelerometer range of +/- 2 g and gyroscope range of 250 degrees/second are therefore selected.

This driver is contained in `sensorlib/mpu6050.c`, with `sensorlib/mpu6050.h` containing the API declarations for use by applications.

10.2 API Functions

Functions

- void MPU6050DataAccelGetFloat (tMPU6050 *psInst, float *pfAccelX, float *pfAccelY, float *pfAccelZ)
- void MPU6050DataAccelGetRaw (tMPU6050 *psInst, uint_fast16_t *pui16AccelX, uint_fast16_t *pui16AccelY, uint_fast16_t *pui16AccelZ)
- void MPU6050DataGyroGetFloat (tMPU6050 *psInst, float *pfGyroX, float *pfGyroY, float *pfGyroZ)
- void MPU6050DataGyroGetRaw (tMPU6050 *psInst, uint_fast16_t *pui16GyroX, uint_fast16_t *pui16GyroY, uint_fast16_t *pui16GyroZ)
- uint_fast8_t MPU6050DataRead (tMPU6050 *psInst, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t MPU6050Init (tMPU6050 *psInst, tI2CMInstance *psI2CInst, uint_fast8_t ui8I2CAddr, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t MPU6050Read (tMPU6050 *psInst, uint_fast8_t ui8Reg, uint8_t *pui8Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t MPU6050ReadModifyWrite (tMPU6050 *psInst, uint_fast8_t ui8Reg, uint_fast8_t ui8Mask, uint_fast8_t ui8Value, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t MPU6050Write (tMPU6050 *psInst, uint_fast8_t ui8Reg, const uint8_t *pui8Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)

10.2.1 Function Documentation

10.2.1.1 MPU6050DataAccelGetFloat

Gets the accelerometer data from the most recent data read.

Prototype:

```
void
MPU6050DataAccelGetFloat(tMPU6050 *psInst,
                         float *pfAccelX,
                         float *pfAccelY,
                         float *pfAccelZ)
```

Parameters:

psInst is a pointer to the MPU6050 instance data.

pfAccelX is a pointer to the value into which the X-axis accelerometer data is stored.

pfAccelY is a pointer to the value into which the Y-axis accelerometer data is stored.

pfAccelZ is a pointer to the value into which the Z-axis accelerometer data is stored.

Description:

This function returns the accelerometer data from the most recent data read, converted into meters per second squared (m/s^2). If any of the output data pointers are **NULL**, the corresponding data is not provided.

Returns:

None.

10.2.1.2 MPU6050DataAccelGetRaw

Gets the raw accelerometer data from the most recent data read.

Prototype:

```
void
MPU6050DataAccelGetRaw(tMPU6050 *psInst,
                       uint_fast16_t *pui16AccelX,
                       uint_fast16_t *pui16AccelY,
                       uint_fast16_t *pui16AccelZ)
```

Parameters:

psInst is a pointer to the MPU6050 instance data.

pui16AccelX is a pointer to the value into which the raw X-axis accelerometer data is stored.

pui16AccelY is a pointer to the value into which the raw Y-axis accelerometer data is stored.

pui16AccelZ is a pointer to the value into which the raw Z-axis accelerometer data is stored.

Description:

This function returns the raw accelerometer data from the most recent data read. The data is not manipulated in any way by the driver. If any of the output data pointers are **NULL**, the corresponding data is not provided.

Returns:

None.

10.2.1.3 MPU6050DataGyroGetFloat

Gets the gyroscope data from the most recent data read.

Prototype:

```
void
MPU6050DataGyroGetFloat(tMPU6050 *psInst,
                        float *pfGyroX,
                        float *pfGyroY,
                        float *pfGyroZ)
```

Parameters:

psInst is a pointer to the MPU6050 instance data.

pfGyroX is a pointer to the value into which the X-axis gyroscope data is stored.

pfGyroY is a pointer to the value into which the Y-axis gyroscope data is stored.

pfGyroZ is a pointer to the value into which the Z-axis gyroscope data is stored.

Description:

This function returns the gyroscope data from the most recent data read, converted into radians per second. If any of the output data pointers are **NULL**, the corresponding data is not provided.

Returns:

None.

10.2.1.4 MPU6050DataGyroGetRaw

Gets the raw gyroscope data from the most recent data read.

Prototype:

```
void
MPU6050DataGyroGetRaw(tMPU6050 *psInst,
                      uint_fast16_t *pui16GyroX,
                      uint_fast16_t *pui16GyroY,
                      uint_fast16_t *pui16GyroZ)
```

Parameters:

psInst is a pointer to the MPU6050 instance data.

pui16GyroX is a pointer to the value into which the raw X-axis gyroscope data is stored.

pui16GyroY is a pointer to the value into which the raw Y-axis gyroscope data is stored.

pui16GyroZ is a pointer to the value into which the raw Z-axis gyroscope data is stored.

Description:

This function returns the raw gyroscope data from the most recent data read. The data is not manipulated in any way by the driver. If any of the output data pointers are **NULL**, the corresponding data is not provided.

Returns:

None.

10.2.1.5 MPU6050DataRead

Reads the accelerometer and gyroscope data from the MPU6050.

Prototype:

```
uint_fast8_t
MPU6050DataRead(tMPU6050 *psInst,
                tSensorCallback *pfnCallback,
                void *pvCallbackData)
```

Parameters:

psInst is a pointer to the MPU6050 instance data.

pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initiates a read of the MPU6050 data registers. When the read has completed (as indicated by calling the callback function), the new readings can be obtained via:

- MPU6050DataAccelGetRaw()
- MPU6050DataAccelGetFloat()
- MPU6050DataGyroGetRaw()
- MPU6050DataGyroGetFloat()

Returns:

Returns 1 if the read was successfully started and 0 if it was not.

10.2.1.6 MPU6050Init

Initializes the MPU6050 driver.

Prototype:

```
uint_fast8_t
MPU6050Init(tMPU6050 *psInst,
            tI2CMInstance *psI2CInst,
            uint_fast8_t ui8I2CAddr,
            tSensorCallback *pfnCallback,
            void *pvCallbackData)
```

Parameters:

psInst is a pointer to the MPU6050 instance data.

psI2CInst is a pointer to the I2C master driver instance data.

ui8I2CAddr is the I2C address of the MPU6050 device.

pfnCallback is the function to be called when the initialization has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initializes the MPU6050 driver, preparing it for operation.

Returns:

Returns 1 if the MPU6050 driver was successfully initialized and 0 if it was not.

10.2.1.7 MPU6050Read

Reads data from MPU6050 registers.

Prototype:

```
uint_fast8_t
MPU6050Read(tMPU6050 *psInst,
            uint_fast8_t ui8Reg,
            uint8_t *pui8Data,
            uint_fast16_t ui16Count,
            tSensorCallback *pfnCallback,
            void *pvCallbackData)
```

Parameters:

psInst is a pointer to the MPU6050 instance data.

ui8Reg is the first register to read.

pui8Data is a pointer to the location to store the data that is read.

ui16Count is the number of data bytes to read.

pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function reads a sequence of data values from consecutive registers in the MPU6050.

Returns:

Returns 1 if the write was successfully started and 0 if it was not.

10.2.1.8 MPU6050ReadModifyWrite

Performs a read-modify-write of a MPU6050 register.

Prototype:

```
uint_fast8_t
MPU6050ReadModifyWrite(tMPU6050 *psInst,
                       uint_fast8_t ui8Reg,
                       uint_fast8_t ui8Mask,
                       uint_fast8_t ui8Value,
                       tSensorCallback *pfnCallback,
                       void *pvCallbackData)
```

Parameters:

psInst is a pointer to the MPU6050 instance data.

ui8Reg is the register to modify.

ui8Mask is the bit mask that is ANDed with the current register value.

ui8Value is the bit mask that is ORed with the result of the AND operation.

pfnCallback is the function to be called when the data has been changed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function changes the value of a register in the MPU6050 via a read-modify-write operation, allowing one of the fields to be changed without disturbing the other fields. The *ui8Reg* register is read, ANDed with *ui8Mask*, ORed with *ui8Value*, and then written back to the MPU6050.

Returns:

Returns 1 if the read-modify-write was successfully started and 0 if it was not.

10.2.1.9 MPU6050Write

Writes data to MPU6050 registers.

Prototype:

```
uint_fast8_t
MPU6050Write(tMPU6050 *psInst,
             uint_fast8_t ui8Reg,
             const uint8_t *pui8Data,
             uint_fast16_t ui16Count,
             tSensorCallback *pfnCallback,
             void *pvCallbackData)
```

Parameters:

psInst is a pointer to the MPU6050 instance data.

ui8Reg is the first register to write.

pui8Data is a pointer to the data to write.

ui16Count is the number of data bytes to write.

pfnCallback is the function to be called when the data has been written (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function writes a sequence of data values to consecutive registers in the MPU6050. The first byte of the *pui8Data* buffer contains the value to be written into the *ui8Reg* register, the second value contains the data to be written into the next register, and so on.

Returns:

Returns 1 if the write was successfully started and 0 if it was not.

10.3 Programming Example

The following example shows how to initialize the MPU6050, select the +/- 4 g range for the accelerometer, and read acceleration and rotation data from it.

```
//
// A boolean that is set when a MPU6050 command has completed.
//
volatile bool g_bMPU6050Done;

//
// The function that is provided by this example as a callback when MPU6050
```

```
// transactions have completed.
//
void
MPU6050Callback(void *pvCallbackData, uint_fast8_t ui8Status)
{
    //
    // See if an error occurred.
    //
    if(ui8Status != I2CM_STATUS_SUCCESS)
    {
        //
        // An error occurred, so handle it here if required.
        //
    }

    //
    // Indicate that the MPU6050 transaction has completed.
    //
    g_bMPU6050Done = true;
}

//
// The MPU6050 example.
//
void
MPU6050Example(void)
{
    float fAccel[3], fGyro[3];
    tI2CMInstance sI2CInst;
    tMPU6050 sMPU6050;

    //
    // Initialize the MPU6050.  This code assumes that the I2C master instance
    // has already been initialized.
    //
    g_bMPU6050Done = false;
    MPU6050Init(&sMPU6050, &sI2CInst, 0x68, MPU6050Callback, 0);
    while(!g_bMPU6050Done)
    {
    }

    //
    // Configure the MPU6050 for +/- 4 g accelerometer range.
    //
    g_bMPU6050Done = false;
    MPU6050ReadModifyWrite(&sMPU6050, MPU6050_O_ACCEL_CONFIG,
                           ~MPU6050_ACCEL_CONFIG_AFS_SEL_M,
                           MPU6050_ACCEL_CONFIG_AFS_SEL_4G, MPU6050Callback,
                           0);
    while(!g_bMPU6050Done)
    {
    }

    //
    // Loop forever reading data from the MPU6050.  Typically, this process
    // would be done in the background, but for the purposes of this example,
    // it is shown in an infinite loop.
    //
    while(1)
    {
        //
        // Request another reading from the MPU6050.
        //
        g_bMPU6050Done = false;
        MPU6050DataRead(&sMPU6050, MPU6050Callback, 0);
        while(!g_bMPU6050Done)
```

```
            {
            }

            //
            // Get the new accelerometer and gyroscope readings.
            //
            MPU6050DataAccelGetFloat(&sMPU6050, &fAccel[0], &fAccel[1],
                                     &fAccel[2]);
            MPU6050DataGyroGetFloat(&sMPU6050, &fGyro[0], &fGyro[1], &fGyro[2]);

            //
            // Do something with the new accelerometer and gyroscope readings.
            //
        }
    }
```

11 MPU9150 Accelerometer, Gyroscope, and Magnetometer Driver

11.1 Introduction

The MPU9150 is an integrated three-axis accelerometer, gyroscope, and magnetometer produced by Invensense Inc. In addition to an internal accelerometer, gyroscope, and magnetometer, this device has an auxiliary I2C bus that can be used to automatically sample external sensors. This driver allows the MPU9150 to be accessed via the I2C bus.

When initialized, a soft reset of the MPU9150 is performed, putting it into its default state. The default accelerometer range of +/- 2 g and gyroscope range of 250 degrees/second are therefore selected (the magnetometer does not have a configurable measurement range).

This driver is contained in `sensorlib/mpu9150.c`, with `sensorlib/mpu9150.h` containing the API declarations for use by applications.

11.2 API Functions

Functions

- void MPU9150DataAccelGetFloat (tMPU9150 *psInst, float *pfAccelX, float *pfAccelY, float *pfAccelZ)
- void MPU9150DataAccelGetRaw (tMPU9150 *psInst, uint_fast16_t *pui16AccelX, uint_fast16_t *pui16AccelY, uint_fast16_t *pui16AccelZ)
- void MPU9150DataGyroGetFloat (tMPU9150 *psInst, float *pfGyroX, float *pfGyroY, float *pfGyroZ)
- void MPU9150DataGyroGetRaw (tMPU9150 *psInst, uint_fast16_t *pui16GyroX, uint_fast16_t *pui16GyroY, uint_fast16_t *pui16GyroZ)
- void MPU9150DataMagnetoGetFloat (tMPU9150 *psInst, float *pfMagnetoX, float *pfMagnetoY, float *pfMagnetoZ)
- void MPU9150DataMagnetoGetRaw (tMPU9150 *psInst, uint_fast16_t *pui16MagnetoX, uint_fast16_t *pui16MagnetoY, uint_fast16_t *pui16MagnetoZ)
- uint_fast8_t MPU9150DataRead (tMPU9150 *psInst, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t MPU9150Init (tMPU9150 *psInst, tI2CMInstance *psI2CInst, uint_fast8_t ui8I2CAddr, tSensorCallback *pfnCallback, void *pvCallbackData)
- tAK8975 * MPU9150MagnetoInstGet (tMPU9150 *psInst)
- uint_fast8_t MPU9150Read (tMPU9150 *psInst, uint_fast8_t ui8Reg, uint8_t *pui8Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t MPU9150ReadModifyWrite (tMPU9150 *psInst, uint_fast8_t ui8Reg, uint_fast8_t ui8Mask, uint_fast8_t ui8Value, tSensorCallback *pfnCallback, void *pvCallbackData)

- uint_fast8_t MPU9150Write (tMPU9150 *psInst, uint_fast8_t ui8Reg, const uint8_t *pui8Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)

11.2.1 Function Documentation

11.2.1.1 MPU9150DataAccelGetFloat

Gets the accelerometer data from the most recent data read.

Prototype:

```
void
MPU9150DataAccelGetFloat(tMPU9150 *psInst,
                         float *pfAccelX,
                         float *pfAccelY,
                         float *pfAccelZ)
```

Parameters:

psInst is a pointer to the MPU9150 instance data.
pfAccelX is a pointer to the value into which the X-axis accelerometer data is stored.
pfAccelY is a pointer to the value into which the Y-axis accelerometer data is stored.
pfAccelZ is a pointer to the value into which the Z-axis accelerometer data is stored.

Description:

This function returns the accelerometer data from the most recent data read, converted into meters per second squared (m/s^2). If any of the output data pointers are **NULL**, the corresponding data is not provided.

Returns:

None.

11.2.1.2 MPU9150DataAccelGetRaw

Gets the raw accelerometer data from the most recent data read.

Prototype:

```
void
MPU9150DataAccelGetRaw(tMPU9150 *psInst,
                       uint_fast16_t *pui16AccelX,
                       uint_fast16_t *pui16AccelY,
                       uint_fast16_t *pui16AccelZ)
```

Parameters:

psInst is a pointer to the MPU9150 instance data.
pui16AccelX is a pointer to the value into which the raw X-axis accelerometer data is stored.
pui16AccelY is a pointer to the value into which the raw Y-axis accelerometer data is stored.
pui16AccelZ is a pointer to the value into which the raw Z-axis accelerometer data is stored.

Description:

This function returns the raw accelerometer data from the most recent data read. The data is not manipulated in any way by the driver. If any of the output data pointers are **NULL**, the corresponding data is not provided.

Returns:
None.

11.2.1.3 MPU9150DataGyroGetFloat

Gets the gyroscope data from the most recent data read.

Prototype:
```
void
MPU9150DataGyroGetFloat(tMPU9150 *psInst,
                        float *pfGyroX,
                        float *pfGyroY,
                        float *pfGyroZ)
```

Parameters:
psInst is a pointer to the MPU9150 instance data.
pfGyroX is a pointer to the value into which the X-axis gyroscope data is stored.
pfGyroY is a pointer to the value into which the Y-axis gyroscope data is stored.
pfGyroZ is a pointer to the value into which the Z-axis gyroscope data is stored.

Description:
This function returns the gyroscope data from the most recent data read, converted into radians per second. If any of the output data pointers are **NULL**, the corresponding data is not provided.

Returns:
None.

11.2.1.4 MPU9150DataGyroGetRaw

Gets the raw gyroscope data from the most recent data read.

Prototype:
```
void
MPU9150DataGyroGetRaw(tMPU9150 *psInst,
                      uint_fast16_t *pui16GyroX,
                      uint_fast16_t *pui16GyroY,
                      uint_fast16_t *pui16GyroZ)
```

Parameters:
psInst is a pointer to the MPU9150 instance data.
pui16GyroX is a pointer to the value into which the raw X-axis gyroscope data is stored.
pui16GyroY is a pointer to the value into which the raw Y-axis gyroscope data is stored.
pui16GyroZ is a pointer to the value into which the raw Z-axis gyroscope data is stored.

Description:
This function returns the raw gyroscope data from the most recent data read. The data is not manipulated in any way by the driver. If any of the output data pointers are **NULL**, the corresponding data is not provided.

Returns:
None.

11.2.1.5 MPU9150DataMagnetoGetFloat

Gets the magnetometer data from the most recent data read.

Prototype:

```
void
MPU9150DataMagnetoGetFloat(tMPU9150 *psInst,
                           float *pfMagnetoX,
                           float *pfMagnetoY,
                           float *pfMagnetoZ)
```

Parameters:

psInst is a pointer to the MPU9150 instance data.

pfMagnetoX is a pointer to the value into which the X-axis magnetometer data is stored.

pfMagnetoY is a pointer to the value into which the Y-axis magnetometer data is stored.

pfMagnetoZ is a pointer to the value into which the Z-axis magnetometer data is stored.

Description:

This function returns the magnetometer data from the most recent data read, converted into tesla. If any of the output data pointers are **NULL**, the corresponding data is not provided.

Returns:

None.

11.2.1.6 MPU9150DataMagnetoGetRaw

Gets the raw magnetometer data from the most recent data read.

Prototype:

```
void
MPU9150DataMagnetoGetRaw(tMPU9150 *psInst,
                         uint_fast16_t *pui16MagnetoX,
                         uint_fast16_t *pui16MagnetoY,
                         uint_fast16_t *pui16MagnetoZ)
```

Parameters:

psInst is a pointer to the MPU9150 instance data.

pui16MagnetoX is a pointer to the value into which the raw X-axis magnetometer data is stored.

pui16MagnetoY is a pointer to the value into which the raw Y-axis magnetometer data is stored.

pui16MagnetoZ is a pointer to the value into which the raw Z-axis magnetometer data is stored.

Description:

This function returns the raw magnetometer data from the most recent data read. The data is not manipulated in any way by the driver. If any of the output data pointers are **NULL**, the corresponding data is not provided.

Returns:

None.

11.2.1.7 MPU9150DataRead

Reads the accelerometer and gyroscope data from the MPU9150 and the magnetometer data from the on-chip aK8975.

Prototype:

```
uint_fast8_t
MPU9150DataRead(tMPU9150 *psInst,
                tSensorCallback *pfnCallback,
                void *pvCallbackData)
```

Parameters:

psInst is a pointer to the MPU9150 instance data.

pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initiates a read of the MPU9150 data registers. When the read has completed (as indicated by calling the callback function), the new readings can be obtained via:

- MPU9150DataAccelGetRaw()
- MPU9150DataAccelGetFloat()
- MPU9150DataGyroGetRaw()
- MPU9150DataGyroGetFloat()
- MPU9150DataMagnetoGetRaw()
- MPU9150DataMagnetoGetFloat()

Returns:

Returns 1 if the read was successfully started and 0 if it was not.

11.2.1.8 MPU9150Init

Initializes the MPU9150 driver.

Prototype:

```
uint_fast8_t
MPU9150Init(tMPU9150 *psInst,
            tI2CMInstance *psI2CInst,
            uint_fast8_t ui8I2CAddr,
            tSensorCallback *pfnCallback,
            void *pvCallbackData)
```

Parameters:

psInst is a pointer to the MPU9150 instance data.

psI2CInst is a pointer to the I2C master driver instance data.

ui8I2CAddr is the I2C address of the MPU9150 device.

pfnCallback is the function to be called when the initialization has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:
This function initializes the MPU9150 driver, preparing it for operation.

Returns:
Returns 1 if the MPU9150 driver was successfully initialized and 0 if it was not.

11.2.1.9 MPU9150MagnetoInstGet

Returns the pointer to the tAK8975 object

Prototype:
```
tAK8975 *
MPU9150MagnetoInstGet(tMPU9150 *psInst)
```

Parameters:
psInst is a pointer to the MPU9150 instance data.

Description:
The MPU9150 contains in internal AK8975 magnetometer. To access data from that sensor, application should use this function to get a pointer to the tAK8975 object, and then use the AK8975 APIs.

Returns:
Returns the pointer to the tAK8975 object

11.2.1.10 MPU9150Read

Reads data from MPU9150 registers.

Prototype:
```
uint_fast8_t
MPU9150Read(tMPU9150 *psInst,
            uint_fast8_t ui8Reg,
            uint8_t *pui8Data,
            uint_fast16_t ui16Count,
            tSensorCallback *pfnCallback,
            void *pvCallbackData)
```

Parameters:
psInst is a pointer to the MPU9150 instance data.
ui8Reg is the first register to read.
pui8Data is a pointer to the location to store the data that is read.
ui16Count is the number of data bytes to read.
pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).
pvCallbackData is a pointer that is passed to the callback function.

Description:
This function reads a sequence of data values from consecutive registers in the MPU9150.

Returns:
Returns 1 if the write was successfully started and 0 if it was not.

11.2.1.11 MPU9150ReadModifyWrite

Performs a read-modify-write of a MPU9150 register.

Prototype:

```
uint_fast8_t
MPU9150ReadModifyWrite(tMPU9150 *psInst,
                       uint_fast8_t ui8Reg,
                       uint_fast8_t ui8Mask,
                       uint_fast8_t ui8Value,
                       tSensorCallback *pfnCallback,
                       void *pvCallbackData)
```

Parameters:

psInst is a pointer to the MPU9150 instance data.

ui8Reg is the register to modify.

ui8Mask is the bit mask that is ANDed with the current register value.

ui8Value is the bit mask that is ORed with the result of the AND operation.

pfnCallback is the function to be called when the data has been changed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function changes the value of a register in the MPU9150 via a read-modify-write operation, allowing one of the fields to be changed without disturbing the other fields. The *ui8Reg* register is read, ANDed with *ui8Mask*, ORed with *ui8Value*, and then written back to the MPU9150.

Returns:

Returns 1 if the read-modify-write was successfully started and 0 if it was not.

11.2.1.12 MPU9150Write

Writes data to MPU9150 registers.

Prototype:

```
uint_fast8_t
MPU9150Write(tMPU9150 *psInst,
             uint_fast8_t ui8Reg,
             const uint8_t *pui8Data,
             uint_fast16_t ui16Count,
             tSensorCallback *pfnCallback,
             void *pvCallbackData)
```

Parameters:

psInst is a pointer to the MPU9150 instance data.

ui8Reg is the first register to write.

pui8Data is a pointer to the data to write.

ui16Count is the number of data bytes to write.

pfnCallback is the function to be called when the data has been written (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function writes a sequence of data values to consecutive registers in the MPU9150. The first byte of the *pui8Data* buffer contains the value to be written into the *ui8Reg* register, the second value contains the data to be written into the next register, and so on.

Returns:

Returns 1 if the write was successfully started and 0 if it was not.

11.3 Programming Example

The following example shows how to initialize the MPU9150, select the +/- 4 g range for the accelerometer, and read acceleration, rotation, and magnetic field data from it.

```
//
// A boolean that is set when a MPU9150 command has completed.
//
volatile bool g_bMPU9150Done;

//
// The function that is provided by this example as a callback when MPU9150
// transactions have completed.
//
void
MPU9150Callback(void *pvCallbackData, uint_fast8_t ui8Status)
{
    //
    // See if an error occurred.
    //
    if(ui8Status != I2CM_STATUS_SUCCESS)
    {
        //
        // An error occurred, so handle it here if required.
        //
    }

    //
    // Indicate that the MPU9150 transaction has completed.
    //
    g_bMPU9150Done = true;
}

//
// The MPU9150 example.
//
void
MPU9150Example(void)
{
    float fAccel[3], fGyro[3], fMagneto[3];
    tI2CMInstance sI2CInst;
    tMPU9150 sMPU9150;

    //
    // Initialize the MPU9150.  This code assumes that the I2C master instance
    // has already been initialized.
    //
    g_bMPU9150Done = false;
    MPU9150Init(&sMPU9150, &sI2CInst, 0x68, MPU9150Callback, 0);
    while(!g_bMPU9150Done)
```

```
    {
    }

    //
    // Configure the MPU9150 for +/- 4 g accelerometer range.
    //
    g_bMPU9150Done = false;
    MPU9150ReadModifyWrite(&sMPU9150, MPU9150_O_ACCEL_CONFIG,
                           ~MPU9150_ACCEL_CONFIG_AFS_SEL_M,
                           MPU9150_ACCEL_CONFIG_AFS_SEL_4G, MPU9150Callback,
                           0);
    while(!g_bMPU9150Done)
    {
    }

    //
    // Loop forever reading data from the MPU9150.  Typically, this process
    // would be done in the background, but for the purposes of this example,
    // it is shown in an infinite loop.
    //
    while(1)
    {
        //
        // Request another reading from the MPU9150.
        //
        g_bMPU9150Done = false;
        MPU9150DataRead(&sMPU9150, MPU9150Callback, 0);
        while(!g_bMPU9150Done)
        {
        }

        //
        // Get the new accelerometer, gyroscope, and magnetometer readings.
        //
        MPU9150DataAccelGetFloat(&sMPU9150, &fAccel[0], &fAccel[1],
                                 &fAccel[2]);
        MPU9150DataGyroGetFloat(&sMPU9150, &fGyro[0], &fGyro[1], &fGyro[2]);
        MPU9150DataMagnetoGetFloat(&sMPU9150, &fMagneto[0], &fMagneto[1],
                                   &fMagneto[2]);

        //
        // Do something with the new accelerometer, gyroscope, and magnetometer
        // readings.
        //
    }
}
```

12 Quaternion Math Module

12.1 Introduction

The quaternion math module provides a set of functions for performing common math operations on quaternions. Quaternion multiplication, inversion, calculating the magnitude of a quaternion, finding the angles between two quaternions, and generating a quaternion from Euler angles are supported.

For functions that produce new quaternions, the first argument is the output quaternion and the remaining arguments are the inputs. This protocol allows the code to be written similar to how it would be written mathematically. For example:

$$C = A * B$$

Where A, B, and C are quaternions, would be written:

```
QuaternionMult(C, A, B);
```

This module is contained in `sensorlib/quaternion.c`, with `sensorlib/quaternion.h` containing the API declarations for use by applications.

12.2 API Functions

Functions

- float QuaternionAngle (float pfQIn1[4], float pfQIn2[4])
- void QuaternionFromEuler (float pfQOut[4], float fRollDeg, float fPitchDeg, float fYawDeg)
- void QuaternionInverse (float pfQOut[4], float pfQIn[4])
- float QuaternionMagnitude (float pfQIn[4])
- void QuaternionMult (float pfQOut[4], float pfQIn1[4], float pfQIn2[4])

12.2.1 Function Documentation

12.2.1.1 QuaternionAngle

Computes the angle between two quaternions

Prototype:

```
float
QuaternionAngle(float pfQIn1[4],
                float pfQIn2[4])
```

Parameters:

pfQIn1 is a source quaternion in W, X, Y, Z form

pfQIn2 is a source quaternion in W, X, Y, Z form

Description:

This function computes the angle between two quaternions.

Returns:

Returns the angle, in radians, between the two quaternions.

12.2.1.2 QuaternionFromEuler

Computes a quaternion from a set of eueler angles specified in degrees

Prototype:

```
void
QuaternionFromEuler(float pfQOut[4],
                    float fRollDeg,
                    float fPitchDeg,
                    float fYawDeg)
```

Parameters:

pfQOut is the inverted quaternion in W, X, Y, Z form

fRollDeg is roll in degrees

fPitchDeg is pitch in degrees

fYawDeg is yaw in degrees

Description:

This function computes a quaternion from a set of euler angles specified in degrees

Returns:

Returns a quaternion representing the provided eulers

12.2.1.3 QuaternionInverse

Computes the inverse of a quaternion.

Prototype:

```
void
QuaternionInverse(float pfQOut[4],
                  float pfQIn[4])
```

Parameters:

pfQOut is the inverted quaternion in W, X, Y, Z form

pfQIn is the source quaternion in W, X, Y, Z form

Description:

This function computes the inverse of a quaternion. The inverse of a quaternion produces a rotation opposite to the source quaternion. This can be achieved by simply changing the signs of the imaginary components of a quaternion when the quatnerion is a unit quaternion.

Returns:
Returns the inverse of a quaternion.

12.2.1.4 QuaternionMagnitude

Computes the magnitude of a quaternion.

Prototype:
```
float
QuaternionMagnitude(float pfQIn[4])
```

Parameters:
pfQIn is the source quaternion in W, X, Y, Z form

Description:
This function computes the magnitude of a quaternion by summing the square of each of the quatnerion components.

Returns:
Returns the scalar magnitude of the quaternion

12.2.1.5 QuaternionMult

Computes the product of two quaternions.

Prototype:
```
void
QuaternionMult(float pfQOut[4],
               float pfQIn1[4],
               float pfQIn2[4])
```

Parameters:
pfQOut is the product of In1 X In2
pfQIn1 is the source quaternion in W, X, Y, Z form
pfQIn2 is the source quaternion in W, X, Y, Z form

Description:
This function computes the cross product of two quaternions.

Returns:
Returns the cross product of the two quaternions.

12.3 Programming Example

The following example shows how to calculate the angle between two quaternions.

```
void
QuaternionExample(void)
{
    float pfQa[4], pfQb[4];
    float fAngle, fMag;

    //
    // Generate quaternion A from a set of Euler angles
    //
    QuatnerionFromEuler(pfQa, 10.0f, 20.0f, 30.0f);

    //
    // Generate quaternion B from another set of Euler angles
    //
    QuaternionFromEuler(pfQb, 10.0f, 20.0f, 42.0f);

    //
    // QuaternionFromEuler() should return unit quaternions.  See if
    // the magnitude is approximately 1.0
    //
    fMag = QuaternionMagnitude(pfQa);

    //
    // Calculate the angle between A and B
    //
    fAngle = QuaternionAngle(pfQa, pfQb);
}
```

13 SHT21 Humidity and Temperature Sensor Driver

13.1 Introduction

The SHT21 is a humidity and temperature sensor produced by Sensirion. The sensor measures the relative humidity over liquid water. This driver allows the SHT21 to be accessed via the I2C bus.

When initialized, a soft reset of the SHT21 is performed, putting it into its default state. The default humidity measurement resolution of 12 bits and temperature measurement resolution of 14 bits are therefore selected.

This driver is contained in `sensorlib/sht21.c`, with `sensorlib/sht21.h` containing the API declarations for use by applications.

13.2 API Functions

Functions

- void SHT21DataHumidityGetFloat (tSHT21 *psInst, float *pfHumidity)
- void SHT21DataHumidityGetRaw (tSHT21 *psInst, uint16_t *pui16Humidity)
- uint_fast8_t SHT21DataRead (tSHT21 *psInst, tSensorCallback *pfnCallback, void *pvCallbackData)
- void SHT21DataTemperatureGetFloat (tSHT21 *psInst, float *pfTemperature)
- void SHT21DataTemperatureGetRaw (tSHT21 *psInst, uint16_t *pui16Temperature)
- uint_fast8_t SHT21Init (tSHT21 *psInst, tI2CMInstance *psI2CInst, uint_fast8_t ui8I2CAddr, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t SHT21Read (tSHT21 *psInst, uint_fast8_t ui8Reg, uint8_t *pui8Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t SHT21ReadModifyWrite (tSHT21 *psInst, uint_fast8_t ui8Reg, uint_fast8_t ui8Mask, uint_fast8_t ui8Value, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t SHT21Write (tSHT21 *psInst, uint_fast8_t ui8Reg, const uint8_t *pui8Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)

13.2.1 Function Documentation

13.2.1.1 SHT21DataHumidityGetFloat

Returns the relative humidity measurement as a floating point percentage.

Prototype:

```
void
SHT21DataHumidityGetFloat(tSHT21 *psInst,
                          float *pfHumidity)
```

Parameters:

psInst pointer to the SHT21 instance data.

pfHumidity is a pointer to the value into which the humidity data is stored.

Description:

This function converts the raw humidity measurement to floating-point-percentage relative humidity over water. For more information on the conversion algorithm see the SHT21 datasheet section 6.1.

Returns:

None.

13.2.1.2 SHT21DataHumidityGetRaw

Returns the raw humidity measurement from the SHT21.

Prototype:

```
void
SHT21DataHumidityGetRaw(tSHT21 *psInst,
                        uint16_t *pui16Humidity)
```

Parameters:

psInst is a pointer to the SHT21 instance data.

pui16Humidity is a pointer to the value into which the raw humidity data is stored.

Description:

This function returns the raw humidity data from the most recent data read. The data is not manipulated in any way by the driver.

Returns:

None.

13.2.1.3 SHT21DataRead

Reads the temperature and humidity data from the SHT21.

Prototype:

```
uint_fast8_t
SHT21DataRead(tSHT21 *psInst,
              tSensorCallback *pfnCallback,
              void *pvCallbackData)
```

Parameters:

psInst is a pointer to the SHT21 instance data

pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initiates a read of the SHT21 data registers. The user must first initiate a measurement by using the SHT21Write() function configured to write the command for a humidity or temperature measurement. In the case of a measurement with I2C bus hold, this function is not needed. When the read has completed (as indicated by callback function), the new readings can be obtained via:

- SHT21DataTemperatureGetRaw()
- SHT21DataTemperatureGetFloat()
- SHT21DataHumidityGetRaw()
- SHT21DataHumidityGetFloat()

Returns:

Returns 1 if the read was successfully started and 0 if it was not.

13.2.1.4 SHT21DataTemperatureGetFloat

Returns the most recent temperature measurement in floating point degrees Celsius.

Prototype:

```
void
SHT21DataTemperatureGetFloat(tSHT21 *psInst,
                             float *pfTemperature)
```

Parameters:

psInst is a pointer to the SHT21 instance data.

pfTemperature is a pointer to the value into which the temperature data is stored.

Description:

This function converts the raw temperature measurement data into floating point degrees Celsius and returns the result. See the SHT21 datasheet section 6.2 for more information about the conversion formula used.

Returns:

None.

13.2.1.5 SHT21DataTemperatureGetRaw

Returns the raw temperature measurement as received from the SHT21.

Prototype:

```
void
SHT21DataTemperatureGetRaw(tSHT21 *psInst,
                           uint16_t *pui16Temperature)
```

Parameters:

psInst is a pointer to the SHT21 instance data.

pui16Temperature is a pointer to the value into which the raw temperature data is stored.

Description:
This function returns the raw temperature data from the most recent data read. The data is not manipulated in any way by the driver.

Returns:
None.

13.2.1.6 SHT21Init

Initializes the SHT21 driver.

Prototype:
```
uint_fast8_t
SHT21Init(tSHT21 *psInst,
          tI2CMInstance *psI2CInst,
          uint_fast8_t ui8I2CAddr,
          tSensorCallback *pfnCallback,
          void *pvCallbackData)
```

Parameters:
psInst is a pointer to the SHT21 instance data.
psI2CInst is a pointer to the I2C driver instance data.
ui8I2CAddr is the I2C address of the SHT21 device.
pfnCallback is the function to be called when the initialization has completed (can be **NULL** if a callback is not required).
pvCallbackData is a pointer that is passed to the callback function.

Description:
This function initializes the SHT21 driver, preparing it for operation, and initiates a reset of the SHT21 device, clearing any previous configuration data.

Returns:
Returns 1 if the SHT21 driver was successfully initialized and 0 if it was not.

13.2.1.7 SHT21Read

Reads data from SHT21 registers.

Prototype:
```
uint_fast8_t
SHT21Read(tSHT21 *psInst,
          uint_fast8_t ui8Reg,
          uint8_t *pui8Data,
          uint_fast16_t ui16Count,
          tSensorCallback *pfnCallback,
          void *pvCallbackData)
```

Parameters:
psInst is a pointer to the SHT21 instance data.
ui8Reg is the first register to read.

pui8Data is a pointer to the location to store the data that is read.

ui16Count the number of data bytes to read.

pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).

pvCallbackData pointer that is passed to the callback function.

Description:

This function reads a sequence of data values from consecutive registers in the SHT21.

Returns:

Returns 1 if the read was successfully started and 0 if it was not.

13.2.1.8 SHT21ReadModifyWrite

Performs a read-modify-write of a SHT21 register.

Prototype:

```
uint_fast8_t
SHT21ReadModifyWrite(tSHT21 *psInst,
                     uint_fast8_t ui8Reg,
                     uint_fast8_t ui8Mask,
                     uint_fast8_t ui8Value,
                     tSensorCallback *pfnCallback,
                     void *pvCallbackData)
```

Parameters:

psInst is a pointer to the SHT21 instance data.

ui8Reg is the register to modify.

ui8Mask is the bit mask that is ANDed with the current register value.

ui8Value is the bit mask that is ORed with the result of the AND operation.

pfnCallback is the function to be called when the data has been changed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function changes the value of a register in the SHT21 via a read-modify-write operation, allowing one of the fields to be changed without disturbing the other fields. The *ui8Reg* register is read, ANDed with *ui8Mask*, ORed with *ui8Value*, and then written back to the SHT21.

Returns:

Returns 1 if the read-modify-write was successfully started and 0 if it was not.

13.2.1.9 SHT21Write

Writes data to SHT21 registers.

Prototype:

```
uint_fast8_t
SHT21Write(tSHT21 *psInst,
```

```
                uint_fast8_t ui8Reg,
                const uint8_t *pui8Data,
                uint_fast16_t ui16Count,
                tSensorCallback *pfnCallback,
                void *pvCallbackData)
```

Parameters:

psInst is a pointer to the SHT21 instance data.

ui8Reg is the register offset to be written.

pui8Data is the data buffer bytes to write.

ui16Count is the number of bytes to write.

pfnCallback is the function to be called when the data has been written (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function writes a sequence of data values to consecutive registers in the SHT21. The first byte of the *pui8Data* buffer contains the value to be written into the *ui8Reg* register, the second value contains the data to be written into the next register, and so on.

Returns:

Returns 1 if the write was successfully started and 0 if it was not.

13.3 Programming Example

The following example shows how to initialize the SHT21, configure the measurement resolution to 11 bits for both relative humidity and temperature, and read relative humidity and temperature data from it.

```
//
// A boolean that is set when a SHT21 command has completed.
//
volatile bool g_bSHT21Done;

//
// The function that is provided by this example as a callback when SHT21
// transactions have completed.
//
void
SHT21Callback(void *pvCallbackData, uint_fast8_t ui8Status)
{
    //
    // See if an error occurred.
    //
    if(ui8Status != I2CM_STATUS_SUCCESS)
    {
        //
        // An error occurred, so handle it here if required.
        //
    }

    //
    // Indicate that the SHT21 transaction has completed.
    //
    g_bSHT21Done = true;
```

```
}

//
// The SHT21 example.
//
void
SHT21Example(void)
{
    float fHumidity, fTemperature;
    tI2CMInstance sI2CInst;
    tSHT21 sSHT21;

    //
    // Initialize the SHT21.  This code assumes that the I2C master instance
    // has already been initialized.
    //
    g_bSHT21Done = false;
    SHT21Init(&sSHT21, &sI2CInst, 0x40, SHT21Callback, 0);
    while(!g_bSHT21Done)
    {
    }

    //
    // Configure the SHT21 for 11 bits sampling size for relative humidity and
    // temperature.
    //
    g_bSHT21Done = false;
    SHT21ReadModifyWrite(&sSHT21, SHT21_CMD_WRITE_CONFIG, 0,
                         (SHT21_CONFIG_RES_11 |
                          SHT21_CONFIG_OTP_RELOAD_DISABLE), SHT21Callback, 0);
    while(!g_bSHT21Done)
    {
    }

    //
    // Loop forever reading data from the SHT21.  Typically, this process would
    // be done in the background, but for the purposes of this example, it is
    // shown in an infinite loop.
    //
    while(1)
    {
        //
        // Request another reading from the SHT21.
        //
        g_bSHT21Done = false;
        SHT21DataRead(&sSHT21, SHT21Callback, 0);
        while(!g_bSHT21Done)
        {
        }

        //
        // Get the new relative humidity and temperature reading.
        //
        SHT21DataHumidityGetFloat(&sSHT21, &fHumidity);
        SHT21DataTemperatureGetFloat(&sSHT21, &fTemperature);

        //
        // Do something with the new relative humidity and temperature reading.
        //
    }
}
```

14 TMP006 Temperature Sensor Driver

14.1 Introduction

The TMP006 is an infrared thermopile temperature sensor produced by Texas Instruments Incorporated. This driver allows the TMP006 to be accessed via the I2C bus.

When initialized, a soft reset of the TMP006 is performed, putting it into its default state. The default conversion rate of one sample per second is therefore selected.

This driver is contained in `sensorlib/tmp006.c`, with `sensorlib/tmp006.h` containing the API declarations for use by applications.

14.2 API Functions

Functions

- uint_fast8_t TMP006DataRead (tTMP006 *psInst, tSensorCallback *pfnCallback, void *pvCallbackData)
- void TMP006DataTemperatureGetFloat (tTMP006 *psInst, float *pfAmbient, float *pfObject)
- void TMP006DataTemperatureGetRaw (tTMP006 *psInst, int16_t *pi16Ambient, int16_t *pi16Object)
- uint_fast8_t TMP006Init (tTMP006 *psInst, tI2CMInstance *psI2CInst, uint_fast8_t ui8I2CAddr, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t TMP006Read (tTMP006 *psInst, uint_fast8_t ui8Reg, uint16_t *pui16Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t TMP006ReadModifyWrite (tTMP006 *psInst, uint_fast8_t ui8Reg, uint_fast16_t ui16Mask, uint_fast16_t ui16Value, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t TMP006Write (tTMP006 *psInst, uint_fast8_t ui8Reg, const uint16_t *pui16Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)

14.2.1 Function Documentation

14.2.1.1 TMP006DataRead

Reads the temperature data from the TMP006.

Prototype:

```
uint_fast8_t
TMP006DataRead(tTMP006 *psInst,
               tSensorCallback *pfnCallback,
               void *pvCallbackData)
```

Parameters:

psInst is a pointer to the TMP006 instance data.

pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initiates a read of the TMP006 data registers. When the read has completed (as indicated by calling the callback function), the new readings can be obtained via:

- TMP006DataTemperatureGetRaw()
- TMP006DataTemperatureGetFloat()

Returns:

Returns 1 if the read was successfully started and 0 if it was not.

14.2.1.2 TMP006DataTemperatureGetFloat

Gets the measurement data from the most recent data read.

Prototype:

```
void
TMP006DataTemperatureGetFloat(tTMP006 *psInst,
                              float *pfAmbient,
                              float *pfObject)
```

Parameters:

psInst is a pointer to the TMP006 instance data.

pfAmbient is a pointer to the value into which the ambient temperature data is stored as floating point degrees Celsius.

pfObject is a pointer to the value into which the object temperature data is stored as floating point degrees Celsius.

Description:

This function returns the temperature data from the most recent data read, converted into Celsius.

Returns:

None.

14.2.1.3 TMP006DataTemperatureGetRaw

Gets the raw measurement data from the most recent data read.

Prototype:

```
void
TMP006DataTemperatureGetRaw(tTMP006 *psInst,
                            int16_t *pi16Ambient,
                            int16_t *pi16Object)
```

Parameters:

psInst is a pointer to the TMP006 instance data.

pi16Ambient is a pointer to the value into which the raw ambient temperature data is stored.

pi16Object is a pointer to the value into which the raw object temperature data is stored.

Description:

This function returns the raw measurement data from the most recent data read. The data is not manipulated in any way by the driver.

Returns:

None.

14.2.1.4 TMP006Init

Initializes the TMP006 driver.

Prototype:

```
uint_fast8_t
TMP006Init(tTMP006 *psInst,
           tI2CMInstance *psI2CInst,
           uint_fast8_t ui8I2CAddr,
           tSensorCallback *pfnCallback,
           void *pvCallbackData)
```

Parameters:

psInst is a pointer to the TMP006 instance data.

psI2CInst is a pointer to the I2C driver instance data.

ui8I2CAddr is the I2C address of the TMP006 device.

pfnCallback is the function to be called when the initialization has completed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initializes the TMP006 driver, preparing it for operation, and initiates a reset of the TMP006 device, clearing any previous configuration data.

Returns:

Returns 1 if the TMP006 driver was successfully initialized and 0 if it was not.

14.2.1.5 TMP006Read

Reads data from TMP006 registers.

Prototype:

```
uint_fast8_t
TMP006Read(tTMP006 *psInst,
           uint_fast8_t ui8Reg,
           uint16_t *pui16Data,
           uint_fast16_t ui16Count,
           tSensorCallback *pfnCallback,
           void *pvCallbackData)
```

Parameters:

psInst is a pointer to the TMP006 instance data.

ui8Reg is the first register to read.

pui16Data is a pointer to the location to store the data that is read.

ui16Count the number of register values to read.

pfnCallback is the function to be called when data read is complete (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function reads a sequence of data values from consecutive registers in the TMP006.

Note:

The TMP006 does not auto-increment the register pointer, so reads of more than one value returns garbage for the subsequent values.

Returns:

Returns 1 if the write was successfully started and 0 if it was not.

14.2.1.6 TMP006ReadModifyWrite

Performs a read-modify-write of a TMP006 register.

Prototype:

```
uint_fast8_t
TMP006ReadModifyWrite(tTMP006 *psInst,
                      uint_fast8_t ui8Reg,
                      uint_fast16_t ui16Mask,
                      uint_fast16_t ui16Value,
                      tSensorCallback *pfnCallback,
                      void *pvCallbackData)
```

Parameters:

psInst is a pointer to the TMP006 instance data.

ui8Reg is the register offset to read modify and write

ui16Mask is the bit mask that is ANDed with the current register value.

ui16Value is the bit mask that is ORed with the result of the AND operation.

pfnCallback is the function to be called when the data has been changed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function changes the value of a register in the TMP006 via a read-modify-write operation, allowing one of the fields to be changed without disturbing the other fields. The *ui8Reg* register is read, ANDed with *ui16Mask*, ORed with *ui16Value*, and then written back to the TMP006.

Returns:

Returns 1 if the read-modify-write was successfully started and 0 if it was not.

14.2.1.7 TMP006Write

Writes data to TMP006 registers.

Prototype:

```
uint_fast8_t
TMP006Write(tTMP006 *psInst,
            uint_fast8_t ui8Reg,
            const uint16_t *pui16Data,
            uint_fast16_t ui16Count,
            tSensorCallback *pfnCallback,
            void *pvCallbackData)
```

Parameters:

psInst is a pointer to the TMP006 instance data.

ui8Reg is the first register to write.

pui16Data is a pointer to the 16-bit register data to write.

ui16Count is the number of 16-bit registers to write.

pfnCallback is the function to be called when the data has been written (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function writes a sequence of data values to consecutive registers in the TMP006. The first value in the *pui16Data* buffer contains the data to be written into the *ui8Reg* register, the second value contains the data to be written into the next register, and so on.

Note:

The TMP006 does not auto-increment the register pointer, so writes of more than one register are rejected by the TMP006.

Returns:

Returns 1 if the write was successfully started and 0 if it was not.

14.3 Programming Example

The following example shows how to initialize the TMP006, select two samples per second conversion rate, and read temperature data from it.

```
//
// A boolean that is set when a TMP006 command has completed.
//
volatile bool g_bTMP006Done;

//
// The function that is provided by this example as a callback when TMP006
// transactions have completed.
//
void
TMP006Callback(void *pvCallbackData, uint_fast8_t ui8Status)
{
    //
    // See if an error occurred.
```

```
    //
    if(ui8Status != I2CM_STATUS_SUCCESS)
    {
        //
        // An error occurred, so handle it here if required.
        //
    }

    //
    // Indicate that the TMP006 transaction has completed.
    //
    g_bTMP006Done = true;
}

//
// The TMP006 example.
//
void
TMP006Example(void)
{
    float fAmbient, fObject;
    tI2CMInstance sI2CInst;
    tTMP006 sTMP006;

    //
    // Initialize the TMP006.  This code assumes that the I2C master instance
    // has already been initialized.
    //
    g_bTMP006Done = false;
    TMP006Init(&sTMP006, &sI2CInst, 0x41, TMP006Callback, 0);
    while(!g_bTMP006Done)
    {
    }

    //
    // Configure the TMP006 for two samples per second conversion rate.
    //
    g_bTMP006Done = false;
    TMP006ReadModifyWrite(&sTMP006, TMP006_O_CONFIG, ~TMP006_CONFIG_CR_M,
                          TMP006_CONFIG_CR_2, TMP006Callback, 0);
    while(!g_bTMP006Done)
    {
    }

    //
    // Loop forever reading data from the TMP006.  Typically, this process
    // would be done in the background, but for the purposes of this example it
    // is shown in an infinite loop.
    //
    while(1)
    {
        //
        // Request another reading from the TMP006.
        //
        g_bTMP006Done = false;
        TMP006DataRead(&sTMP006, TMP006Callback, 0);
        while(!g_bTMP006Done)
        {
        }

        //
        // Get the new temperature reading.
        //
        TMP006DataTemperatureGetFloat(&sTMP006, &fAmbient, &fObject);

        //
```

```
            // Do something with the new temperature reading.
            //
        }
    }
```

15 TMP100 Temperature Sensor Driver

15.1 Introduction

The TMP100 is a digital temperature sensor produced by Texas Instruments Incorporated. This driver allows the TMP100 to be accessed via the I2C bus.

When initialized, the configuration register of the TMP100 written to its default value, putting it into its default state.

This driver is contained in `sensorlib/tmp100.c`, with `sensorlib/tmp100.h` containing the API declarations for use by applications.

15.2 API Functions

Functions

- uint_fast8_t TMP100DataRead (tTMP100 *psInst, tSensorCallback *pfnCallback, void *pvCallbackData)
- void TMP100DataTemperatureGetFloat (tTMP100 *psInst, float *pfTemperature)
- void TMP100DataTemperatureGetRaw (tTMP100 *psInst, int16_t *pi16Temperature)
- uint_fast8_t TMP100Init (tTMP100 *psInst, tI2CMInstance *psI2CInst, uint_fast8_t ui8I2CAddr, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t TMP100Read (tTMP100 *psInst, uint_fast8_t ui8Reg, uint16_t *pui16Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t TMP100ReadModifyWrite (tTMP100 *psInst, uint_fast8_t ui8Reg, uint_fast16_t ui16Mask, uint_fast16_t ui16Value, tSensorCallback *pfnCallback, void *pvCallbackData)
- uint_fast8_t TMP100Write (tTMP100 *psInst, uint_fast8_t ui8Reg, const uint16_t *pui16Data, uint_fast16_t ui16Count, tSensorCallback *pfnCallback, void *pvCallbackData)

15.2.1 Function Documentation

15.2.1.1 TMP100DataRead

Reads the temperature data from the TMP100.

Prototype:

```
uint_fast8_t
TMP100DataRead(tTMP100 *psInst,
               tSensorCallback *pfnCallback,
               void *pvCallbackData)
```

Parameters:

psInst is a pointer to the TMP100 instance data.

pfnCallback is the function to be called when the data has been read (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function initiates a read of the TMP100 data registers. When the read has completed (as indicated by calling the callback function), the new readings can be obtained via:

- TMP100DataTemperatureGetRaw()
- TMP100DataTemperatureGetFloat()

Returns:

Returns 1 if the read was successfully started and 0 if it was not.

15.2.1.2 TMP100DataTemperatureGetFloat

Gets the measurement data from the most recent data read.

Prototype:

```
void
TMP100DataTemperatureGetFloat(tTMP100 *psInst,
                              float *pfTemperature)
```

Parameters:

psInst is a pointer to the TMP100 instance data.

pfTemperature is a pointer to the value into which the temperature data is stored as floating point degrees Celsius.

Description:

This function returns the temperature data from the most recent data read, converted into Celsius.

Returns:

None.

15.2.1.3 TMP100DataTemperatureGetRaw

Gets the raw measurement data from the most recent data read.

Prototype:

```
void
TMP100DataTemperatureGetRaw(tTMP100 *psInst,
                            int16_t *pi16Temperature)
```

Parameters:

psInst is a pointer to the TMP100 instance data.

pi16Temperature is a pointer to the value into which the raw temperature data is stored.

Description:
This function returns the raw measurement data from the most recent data read. The data is not manipulated in any way by the driver.

Returns:
None.

15.2.1.4 TMP100Init

Initializes the TMP100 driver.

Prototype:
```
uint_fast8_t
TMP100Init(tTMP100 *psInst,
           tI2CMInstance *psI2CInst,
           uint_fast8_t ui8I2CAddr,
           tSensorCallback *pfnCallback,
           void *pvCallbackData)
```

Parameters:
psInst is a pointer to the TMP100 instance data.
psI2CInst is a pointer to the I2C driver instance data.
ui8I2CAddr is the I2C address of the TMP100 device.
pfnCallback is the function to be called when the initialization has completed (can be **NULL** if a callback is not required).
pvCallbackData is a pointer that is passed to the callback function.

Description:
This function initializes the TMP100 driver, preparing it for operation, and initiates a reset of the TMP100 device, clearing any previous configuration data.

Returns:
Returns 1 if the TMP100 driver was successfully initialized and 0 if it was not.

15.2.1.5 TMP100Read

Reads data from TMP100 registers.

Prototype:
```
uint_fast8_t
TMP100Read(tTMP100 *psInst,
           uint_fast8_t ui8Reg,
           uint16_t *pui16Data,
           uint_fast16_t ui16Count,
           tSensorCallback *pfnCallback,
           void *pvCallbackData)
```

Parameters:
psInst is a pointer to the TMP100 instance data.
ui8Reg is the first register to read.

pui16Data is a pointer to the location to store the data that is read.

ui16Count the number of register values to read.

pfnCallback is the function to be called when data read is complete (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function reads a sequence of data values from consecutive registers in the TMP100.

Note:

The TMP100 does not auto-increment the register pointer, so reads of more than one value returns garbage for the subsequent values.

Returns:

Returns 1 if the write was successfully started and 0 if it was not.

15.2.1.6 TMP100ReadModifyWrite

Performs a read-modify-write of a TMP100 register.

Prototype:

```
uint_fast8_t
TMP100ReadModifyWrite(tTMP100 *psInst,
                      uint_fast8_t ui8Reg,
                      uint_fast16_t ui16Mask,
                      uint_fast16_t ui16Value,
                      tSensorCallback *pfnCallback,
                      void *pvCallbackData)
```

Parameters:

psInst is a pointer to the TMP100 instance data.

ui8Reg is the register offset to read modify and write

ui16Mask is the bit mask that is ANDed with the current register value.

ui16Value is the bit mask that is ORed with the result of the AND operation.

pfnCallback is the function to be called when the data has been changed (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function changes the value of a register in the TMP100 via a read-modify-write operation, allowing one of the fields to be changed without disturbing the other fields. The *ui8Reg* register is read, ANDed with *ui16Mask*, ORed with *ui16Value*, and then written back to the TMP100.

Returns:

Returns 1 if the read-modify-write was successfully started and 0 if it was not.

15.2.1.7 TMP100Write

Writes data to TMP100 registers.

Prototype:

```
uint_fast8_t
TMP100Write(tTMP100 *psInst,
            uint_fast8_t ui8Reg,
            const uint16_t *pui16Data,
            uint_fast16_t ui16Count,
            tSensorCallback *pfnCallback,
            void *pvCallbackData)
```

Parameters:

psInst is a pointer to the TMP100 instance data.

ui8Reg is the first register to write.

pui16Data is a pointer to the 16-bit register data to write.

ui16Count is the number of 16-bit registers to write.

pfnCallback is the function to be called when the data has been written (can be **NULL** if a callback is not required).

pvCallbackData is a pointer that is passed to the callback function.

Description:

This function writes a sequence of data values to consecutive registers in the TMP100. The first value in the *pui16Data* buffer contains the data to be written into the *ui8Reg* register, the second value contains the data to be written into the next register, and so on.

Note:

The TMP100 does not auto-increment the register pointer, so writes of more than one register are rejected by the TMP100.

Returns:

Returns 1 if the write was successfully started and 0 if it was not.

15.3 Programming Example

The following example shows how to initialize the TMP100, select 12-bit resolution, and read temperature data from it.

```
//
// A boolean that is set when a TMP100 command has completed.
//
volatile bool g_bTMP100Done;

//
// The function that is provided by this example as a callback when TMP100
// transactions have completed.
//
void
TMP100Callback(void *pvCallbackData, uint_fast8_t ui8Status)
{
    //
    // See if an error occurred.
    //
    if(ui8Status != I2CM_STATUS_SUCCESS)
    {
        //
        // An error occurred, so handle it here if required.
```

```
        //
    }

    //
    // Indicate that the TMP100 transaction has completed.
    //
    g_bTMP100Done = true;
}

//
// The TMP100 example.
//
void
TMP100Example(void)
{
    tI2CMInstance sI2CInst;
    float fTemperature;
    tTMP100 sTMP100;

    //
    // Initialize the TMP100.  This code assumes that the I2C master instance
    // has already been initialized.
    //
    g_bTMP100Done = false;
    TMP100Init(&sTMP100, &sI2CInst, 0x4a, TMP100Callback, 0);
    while(!g_bTMP100Done)
    {
    }

    //
    // Configure the TMP100 for 12-bit conversion resolution.
    //
    g_bTMP100Done = false;
    TMP100ReadModifyWrite(&sTMP100, TMP100_O_CONFIG, ~TMP100_CONFIG_RES_M,
                          TMP100_CONFIG_RES_12BIT, TMP100Callback, 0);
    while(!g_bTMP100Done)
    {
    }

    //
    // Loop forever reading data from the TMP100.  Typically, this process
    // would be done in the background, but for the purposes of this example it
    // is shown in an infinite loop.
    //
    while(1)
    {
        //
        // Request another reading from the TMP100.
        //
        g_bTMP100Done = false;
        TMP100DataRead(&sTMP100, TMP100Callback, 0);
        while(!g_bTMP100Done)
        {
        }

        //
        // Get the new temperature reading.
        //
        TMP100DataTemperatureGetFloat(&sTMP100, &fTemperature);

        //
        // Do something with the new temperature reading.
        //
    }
}
```

16 Vector Math Module

16.1 Introduction

The vector math module provides a set of functions for performing common math operations on three-dimensional vectors. Vector addition, scaling, dot products, and cross products are supported. Vector operations can be combined to perform other useful vector operations; for example, the square root of the dot product of a vector with itself computes the vector magnitude, and scaling a vector by that the inverse of that magnitude converts the vector into a unit vector.

For functions that produce new vectors, the first argument is the output vector and the remaining arguments are the inputs. This protocol allows the code to be written similar to how it would be written mathematically. For example:

$$C = A + B$$

Where A, B, and C are vectors, would be written:

```
VectorAdd(C, A, B);
```

This module is contained in `sensorlib/vector.c`, with `sensorlib/vector.h` containing the API declarations for use by applications.

16.2 API Functions

Functions

- void VectorAdd (float pfVectorOut[3], float pfVectorIn1[3], float pfVectorIn2[3])
- void VectorCrossProduct (float pfVectorOut[3], float pfVectorIn1[3], float pfVectorIn2[3])
- float VectorDotProduct (float pfVectorIn1[3], float pfVectorIn2[3])
- void VectorScale (float pfVectorOut[3], float pfVectorIn[3], float fScale)

16.2.1 Function Documentation

16.2.1.1 VectorAdd

Adds two vectors.

Prototype:

```
void
VectorAdd(float pfVectorOut[3],
```

```
                float pfVectorIn1[3],
                float pfVectorIn2[3])
```

Parameters:

pfVectorOut is the output vector.

pfVectorIn1 is the first vector.

pfVectorIn2 is the second vector.

Description:

This function adds two 3-dimensional vectors.

Returns:

None.

16.2.1.2 VectorCrossProduct

Computes the cross product of two vectors.

Prototype:

```
void
VectorCrossProduct(float pfVectorOut[3],
                   float pfVectorIn1[3],
                   float pfVectorIn2[3])
```

Parameters:

pfVectorOut is the output vector.

pfVectorIn1 is the first vector.

pfVectorIn2 is the second vector.

Description:

This function computes the cross product of two 3-dimensional vectors.

Returns:

None.

16.2.1.3 VectorDotProduct

Computes the dot product of two vectors.

Prototype:

```
float
VectorDotProduct(float pfVectorIn1[3],
                 float pfVectorIn2[3])
```

Parameters:

pfVectorIn1 is the first vector.

pfVectorIn2 is the second vector.

Description:

This function computes the dot product of two 3-dimensional vector.

Returns:
Returns the dot product of the two vectors.

16.2.1.4 VectorScale

Scales a vector.

Prototype:

```
void
VectorScale(float pfVectorOut[3],
            float pfVectorIn[3],
            float fScale)
```

Parameters:
pfVectorOut is the output vector.
pfVectorIn is the input vector.
fScale is the scale factor.

Description:
This function scales a 3-dimensional vector by multiplying each of its components by the scale factor.

Returns:
None.

16.3 Programming Example

The following example shows how to manipulate vectors.

```
void
VectorExample(void)
{
    float pfA[3], pfB[3], pfC[3];

    //
    // Add A and B, placing the result into C.
    //
    VectorAdd(pfC, pfA, pfB);

    //
    // Scale C by 0.5.
    //
    VectorScale(pfC, pfC, 0.5);

    //
    // Compute the cross product of A and B, placing the result into C.
    //
    VectorCrossProduct(pfC, pfA, pfB);

    //
    // Use the dot product of C with itself to normalize C.
    //
    VectorScale(pfC, pfC, 1.0 / sqrtf(VectorDotProduct(pfC, pfC)));
}
```

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI's terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265