TI Designs

MSP430® Orientation Tracker Design Guide

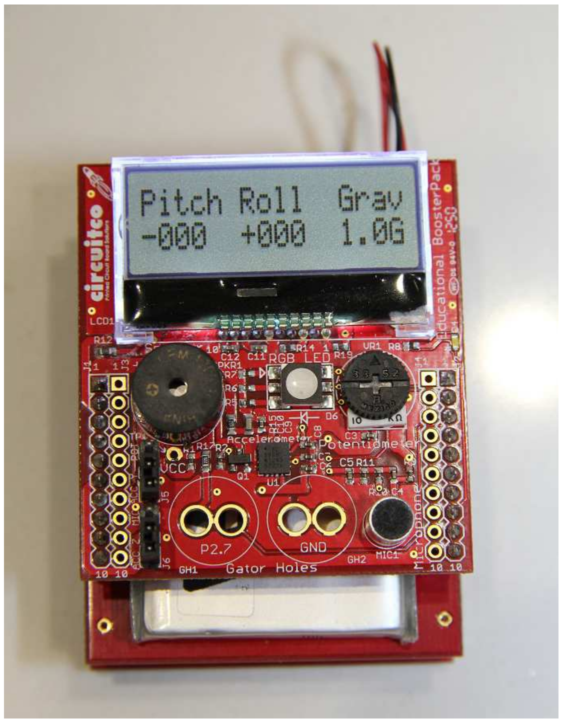
TEXAS INSTRUMENTS

TI Designs

TI Designs provide the foundation that you need including methodology, testing and design files to quickly evaluate and customize and system. TI Designs help you accelerate your time to market.

Design Resources

www.ti.com/tool/TIDU265	Tool Folder Containing Design Files



Design Features

- MSP430 G2 LaunchPad (MSP-EXP430G2) + CircuitCo Educational BoosterPack + Fuel Tank BoosterPack enables orientation tracking and tilt detection
- IQmathLib fixed-point software provides optimized math performance
- The MSP430 samples an accelerometer and interprets the data as x,y,z components of gravity
- Arc Tangent is calculated to determine current orientation, including pitch and roll
- Source code is provided with projects for Code Composer Studio and IAR Embedded Workbench (for the MSP430)

Featured Applications

- Orientation-aware smart devices
- Asset tracking for shipments or items in a large warehouse (can be further enhanced with a wireless add-on)
- Packaging labels, to track shipment orientation and damage due to a drop.
- Toys and devices that track orientation and drop detection, where low cost or low power are at a premium.
- Drop detection for hard drives or other sensitive equipment and electronics.

TI E2E™ Community

ASK Our EP Experts
WebBench Calculator Tools

1 Description

This design guide shows how to combine a value-line MSP430 device, fixed point software math libraries, and an accelerometer for a low cost, ultra-low-power, simple Orientation Tracking solution. The MSP430 samples an accelerometer and performs calculations in real time to determine its orientation and detect when it is in free-fall. This application uses a MSP430G2 LaunchPad Evaluation Kit, QmathLib software libraries and an Educational BoosterPack plug-in module. The source code is provided with projects for Code Composer Studio and IAR Embedded Workbench for MSP430.

This user’s guide will provide steps to run the application and an overview of the hardware and software requirements.

2 Quickstart Guide

2.1 Getting Started

This chapter will briefly cover the steps required to prepare the hardware, load the software and run the application.

2.2 Hardware Setup

The application source code requires a MSP430G2553 LaunchPad and Educational BoosterPack. Optionally a FuelTank BoosterPack can be used to provide battery power. The following jumper settings are required on the LaunchPad and BoosterPack:

- Remove the P1.0 jumper from MSP430 LaunchPad
- Remove the P1.6 jumper from MSP430 LaunchPad
- Place the J5 jumper on "ACC Y" on the Educational BoosterPack
- Place the J6 jumper on "ACC Z" on the Educational BoosterPack

If using the FuelTank BoosterPack to provide battery power to the MSP430 LaunchPad, resistors R11, R12 and R13 need to be removed from the BoosterPack. These connect pins on the BoosterPack header to provide control of the on board BQ24210 Li-Ion battery charger when charging from the MCU, however they directly interfere with the Educational BoosterPack. The BoosterPack is still capable of charging via the USB connector labelled "CHARGE IN". Additionally, The following jumper settings must be applied:

- Remove the "I2C PULLUP" jumper from FuelTank BoosterPack
- Remove the "5V Out" jumper from the FuelTank BoosterPack
- Place the "3.3V Out" jumper on the FuelTank BoosterPack

2.3 Software Setup

The source code includes projects for Code Composer Studio (CCS) and IAR Embedded Workbench for MSP430 (IAR) that can be imported to your IDE of choice. Import the project, build and load the executable onto the MSP430G2 LaunchPad.

The LCD on the Educational BoosterPack is hard wired to the 3.3V pin and will turn on when power is applied. The LCD must be configured by the MCU after power is applied to operate normally. When the code is loaded onto the MSP430G2 LaunchPad the LCD configuration will fail since power has already been applied to the LCD. Once the code is loaded, end the debugger session and remove the USB cable supplying power to the board. The device will be ready for operation the next time power is applied by the USB emulator on the LaunchPad or the FuelTank BoosterPack.

When the application is started on a new device the accelerometer must first be calibrated. The software will check for calibration values stored in the INFO-D section of flash. If no values are present the calibration routine will start. If values are already stored, the calibration routine can be manually initiated by holding down S2 on the MSP430G2 LaunchPad when the program is started.

2.4 Calibration Process

The calibration process requires two measurements along each accelerometer axis, one in the positive direction and one in the negative direction. The LCD screen will display an arrow that must point opposite the direction of gravity. Once the LaunchPad is oriented correctly and steady, press the switch S2 on the MSP430G2 LaunchPad to begin calibration. The RGB LED will turn red while samples are being collected and then change back to green to signal the calibration for that direction is finished. The LCD will update with a new calibration direction and the step is repeated. In total this step is repeated six times, twice for each axis of the accelerometer.

Note: The "o" character represents the +Z axis and the LCD screen should face directly up. The "x" character represents the -Z axis and the LCD screen should face directly down.

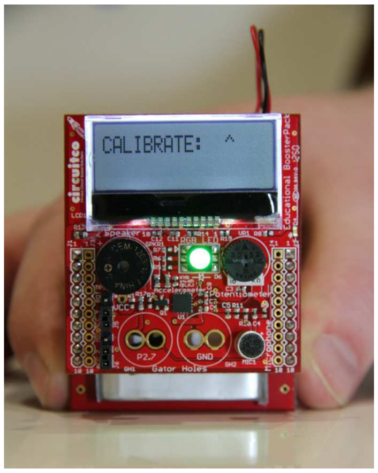


Figure 1. Calibrate the Positive X-axis

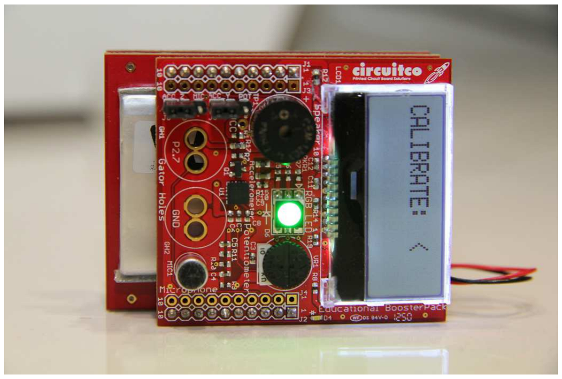


Figure 2. Calibrate the Positive Y-axis

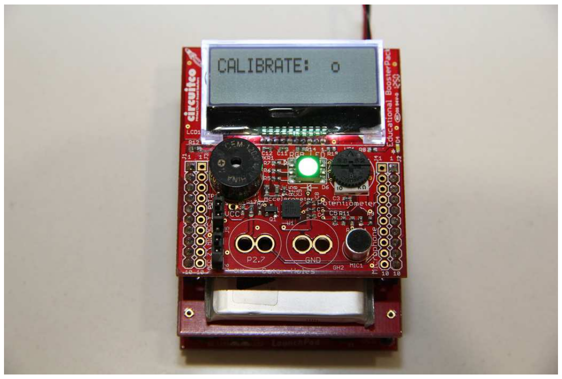


Figure 3. Calibrate the Positive Z-axis

When the calibration process is complete the calibration values will be saved to the INFO-D section of flash and the application will begin displaying the current pitch and roll angles along with the magnitude of gravity to the LCD screen.

The calibration process only needs to be run once per device. If necessary the calibration process can be manually invoked by starting the application with S2 on the MSP430G2 LaunchPad pressed down.

3 Benchmarks

3.1 Timing Benchmarks

The application performance can be benchmarked by measuring the time consumed to sample the accelerometer and calculate the results. The application code includes calls to benchmark functions and can be turned on by defining *ENABLE_BENCHMARK* in the *HAL_board.h* header file. The benchmark does not include calls to display data to the LCD as this is not typically part of the application and is only included for demonstration purposes.

The MSP430G2553 board file provided in the application source code implements the benchmark functions by controlling GPIO pin P1.1. The pin is driven high at the start of the benchmark and then low when the benchmark completes. This pin can be used to measure the time spent sampling the accelerometer and performing the math calculations.

- Timing benchmarks are run with *MCLK_FREQ* set to *1000000* (1MHz MCLK).
- Timing benchmarks for Code Composer Studio (CCS) are obtained using the TI MSP430 compiler version 4.2.3 with compiler options set to *-O3 -opt_for_speed=0*.
- Timing benchmarks for IAR Embedded Workbench (IAR) are obtained using IAR EW for MSP430 version 5.60.2 with compiler options set to *-Ohz*.

Table 1. Application Timing Benchmarks

IDE	Sample Frequency	Calculations	Duty Cycle
Code Composer Studio	22.230 Hz	5.630 ms	12.515%
IAR Embedded Workbench	22.199 Hz	5.371	11.922%

3.2 Size Benchmarks

This section details the application code, data and constant data sizes required for both the TI and IAR Embedded Workbench compilers. Data size included all global variables, stack, RTS library variables and other data structures. Constant data includes strings for the LCD display functions, lookup tables for the QmathLib and RTS library constant data. Code size is the full application code and is the total used flash size minus the constant data.

The code size and constant data size benchmarks include functions to calibrate the accelerometer and display data to the LCD. These functions exist to demonstrate the application and provide an easy to use interface but do not need to be included in a complete application.

- Sizes for Code Composer Studio are obtained using the TI MSP430 compiler version 4.2.3 with compiler options set to *-O3 -opt_for_speed=0*.
- Sizes for IAR are obtained using IAR EW for MSP430 version 5.60.2 with compiler options set to *-Ohz*.

Table 2. Application Data and Code Sizes

IDE	Data	Constant Data	Code
Code Composer Studio	118 bytes	636 bytes	4138 bytes
IAR Embedded Workbench	117 bytes	640 bytes	3158 bytes

4 API Documentation

4.1 Main

4.1.1 Main Functions

- int main (void)

4.1.2 Main Variables

- Vector3D i16RawData
- _q12 q12Gravity Magnitude
- Vector3D q12 Gravity Vector
- _q12 q12PitchAngle
- _q12 q12RollAngle

4.1.3 Main Detailed Description

The following functions are provided in *main.c* and provide the core application data structures and routine.

4.1.4 Main Function Documentation

main — Main routine of the application.

Prototype:

```
int
main (void)
```

Description:

The main application routine will initialize all of the board and hardware layers for operation. The calibration routine is called and will program calibration data if it is not present.

The application will loop forever, reading data from the accelerometer and calculating the magnitude of gravity and current orientation of the board with pitch and roll angles. This data is output to the LCD every other sample.

Returns:

This function never returns.

4.1.5 Main Variable Documentation

i16RawData —

Definition:

```
struct Vector3D i16RawData
```

Description:

Raw measurement from the 3D-accelerometer.

q12GravityMagnitude —

Definition:

```
 _q12 q12GravityMagnitude
```

Description:

Magnitude of the gravity vector in Q12 format.

q12GravityVector —

Definition:

```
struct Vector3D q12GravityVector
```

Description:

Vector corresponding to gravity in Q12 format.

q12PitchAngle —

Definition:

```
_q12 q12PitchAngle
```

Description:

Pitch angle in Q12 format.

q12RollAngle —

Definition:

```
_q12 q12RollAngle
```

Description:

Roll angle in Q12 format.

4.2 Calculation

4.2.1 Calculation Data Structures

- Vector3D

4.2.2 Calculation Defines

- DROP_THRESHOLD
- RADD_TO_DEG

4.2.3 Calculation Functions

- void calculate_angles (struct Vector3D *q12VInput, _q12 *q12pitch, _q12 *q12roll)
- void calculate_dropDetection (_q12 *q12magnitude)
- void calculate_gravityVector (struct Vector3D *i16VInput, struct Vector3D *q12VOutput)
- void calculate_magnitude (struct Vector3D *q12VInput, _q12 *q12magnitude)

4.2.4 Calculation Detailed Description

The following functions are provided in *calculation.c* and are used to process the accelerometer data and calculate orientation and magnitude.

4.2.5 Calculation Data Structure Documentation

Vector3D —

Definition:

```
typedef struct
{
int16_t x;
int16_t y;
int16_t z;
}
Vector3D
```

Members:

x

y

z

Description:

Structure definition for a 3D vector.

3D vector structure with x, y and z fields of type int16_t. The int16_t type is equivalent to the _q data types and are interchangeable.

4.2.6 Calculation Define Documentation

DROP_THRESHOLD —

Definition:

```
#define DROP_THRESHOLD
```

Description:

Threshold for drop detection.

When in perfect free fall, the magnitude of gravity will be zero. If any rotation is present when the device is in free fall, the magnitude will be greater than zero, due to the orientation of the sensor with respect to the center of mass. This threshold can be adjusted to fine-tune the sensitivity of the drop detection. Increasing this value will increase the sensitivity of the detection.

RAD_TO_DEG —

Definition:

```
#define RAD_TO_DEG
```

Description:

Constant value for converting radians to degrees.

4.2.7 Calculation Function Documentation

calculate_angles — Calculate pitch and roll angles from a 3D input vector in q12 format.

Prototype:

```
void
calculate_angles(struct Vector3D *q12VInput,
_q12 *q12pitch,
_q12 *q12roll)
```

Description:

Calculate the pitch and roll angles in q12 format using the 3D input vector and the arc tangent function provided by the QmathLib. The roll angle is calculated first and is limited to a range of -PI to +PI. The pitch angle is calculated second and has a range of -PI/2 to +PI/2. The two calculations are shown below.

$$roll = atan(y / z) \tag{1}$$

$$pitch = atan(x/\sqrt{y^2+z^2}\ ) \tag{2}$$

Parameters:

q12VInput – Pointer to gravity vector in Q12 format.

q12pitch – Pointer to write the Q12 format pitch result.

q12roll – Pointer to write the Q12 format roll result.

Returns:

none

calculate_dropDetection — Turn on the buzzer if the magnitude is below the drop threshold.

Prototype:

```
void
calculate_dropDetection(_q12 *q12magnitude)
```

Description:

Compare the q12 magnitude input against the drop threshold. If the magnitude is below the threshold turn on the buzzer.

Parameters:

q12magnitude – Pointer to magnitude in Q12 format.

Returns:

none

calculate_gravityVector — Calculate the gravity vector from raw accelerometer data.

Prototype:

```
void
calculate_gravityVector(struct Vector3D *i16VInput,
struct Vector3D *q12VOutput)
```

Description:

Calculate the 3D gravity vector in q12 format using the stored calibration offset and scale values from the raw accelerometer data. The calibration data is scaled by 16 so the input must be scaled up to calculate the result. This is accomplished by solving the following equations for each axis.

$$result_x = (input_x - offset_x)/scale_x$$
$$result_y = (input_y - offset_y)/scale_y$$
$$result_z = (input_z - offset_z)/scale_z \quad (3)$$

Parameters:

i16VInput – Pointer to raw accelerometer readings.

q16VOutput – Pointer to gravity vector in Q12 format.

Returns:

none

calculate_magnitude — Calculate the magnitude of a 3D input vector in q12 format

Prototype:

```
void
calculate_magnitude(struct Vector3D *q12VInput,
_q12 *q12magnitude)
```

Description:

This function calculates the magnitude of a 3D vector using the following equation:

$$magnitude = \sqrt{x^2 + y^2 + z^2} \quad (4)$$

The QmathLib does not include a three input magnitude function so the function is implemented in the following way with two calls to the QmathLib magnitude function.

$$magnitude = \sqrt{\sqrt{\left(x^2 + y^2\right)}^2 + z^2} \quad (5)$$

Parameters:

q12VInput – Pointer to gravity vector in Q12 format.

q12magnitude – Pointer to write the magnitude result.

Returns:

none

4.3 Calibration

4.3.1 Calibration Functions

- void calibrate_accelerometer (void)

4.3.2 Calibration Variables

- const accelerometer_channel calibrateAxis[3]
- char calibrateDirection[6]
- const int16_t i16CalibrationOffset[3]
- const int16_t i16CalibrationScale[3]
- int16_t i16Offset[3]
- int16_t i16Scale[3]

4.3.3 Calibration Detailed Description

The following functions are provided in *calibration.c* and are used to calibrate the accelerometer readings.

4.3.4 Calibration Function Documentation

calibrate_accelerometer — Calibrate the accelerometer with measurements along all six axis

Prototype:

```
void
calibrate_accelerometer(void)
```

Description:

Calibrate the accelerometer with measurements along all six axis.

Returns:

none

4.3.5 Calibration Variable Documentation

calibrateAxis —

Definition:

```
const accelerometer_channel calibrateAxis[3]
```

Description:

Accelerometer channels to use for calibration readings.

calibrateDirection —

Definition:

```
char calibrateDirection[6]
```

Decription:

Characters to display orientation during calibration process.

These characters represent the direction to align the sensor during the calibration process. The direction pointed should be opposite the direction or gravity.

The calibration characters in order: +X, -X, +Y, -Y, +Z, -Z

i16CalibrationOffset —

Definition:

```
const int16_t i16CalibrationOffset[3]
```

Calibration offset data for accelerometer readings.

Calibration offset data for the accelerometer readings. The data is scaled by by 16 to retain accuracy. These values will need to be stored in information memory and are dependant on the implementation of the HAL_flash layer.

i16CalibrationScale —

Definition:

```
const int16_t i16CalibrationScale[3]
```

Description:

Calibration scale data for accelerometer readings.

Calibration scale data for the accelerometer readings. The data is scaled by 16 to retain accuracy. These values will need to be stored in information memory and are dependant on the implementation of the HAL_flash layer.

i16Scale —

Definition:

```
int16_t i16Scale[3]
```

Description:

Scale measurements to store calibration data before writing to flash.

4.4 Display

4.4.1 Display Functions

- void display_update (_q12 *q12Pitch, _q12 *q12Roll, _q12 *q12GravityMagnitude)

4.4.2 Display Detailed Description

The following functions are provided in *display.c* and provide methods for displaying information on the LCD.

4.4.3 Display Function Documentation

display_update — Write the measurements to the display.

Prototype:

```
void
display_update(_q12 *q12Pitch,
_q12 *q12Roll,
_q12 *q12GravityMagnitude)
```

Description:

Update the LCD screen with the pitch, roll and magnitude calculations in q12 format. The angles are converted to q6 format and degrees and then to ASCII characters with three integer digits and sign. The magnitude is converted to ASCII characters with a single integer and single fractional digit.

Parameters:

q12Pitch – Pointer to pitch angle measurement.

q12Roll – Pointer to roll angle measurement.

q12GravityMagnitude – Pointer to gravity magnitude measurement.

Returns:

none

4.5 Board

4.5.1 Board Defines

- __delay_ms(n)
- __delay_us(n)
- board_benchmarkStart()
- board_benchmarkStop()
- board_buttonPressed()
- ENABLE_BENCHMARK
- MCLK_FREQ

4.5.2 Board Functions

- void board_gotoSleep (void)
- void board_init (void)
- __interrupt void board_watchdogISR (void)

4.5.3 Board Detailed Description

The following functions are provided in HAL_board.c and provide methods for initialize the LaunchPad MCU.

4.5.4 Board Define Documentation

__delay_ms — Delay by n milliseconds.

Definition:

```
#define __delay_ms(n)
```

Description:

Delay by n milliseconds

Parameters:

n – Number of milliseconds to delay by.

Returns:

none

__delay_us — Delay by n microseconds.

Definition:

```
#define __delay_us(n)
```

Description:

Delay by n microseconds.

Parameters:

n – Number of microseconds to delay by.

Returns:

none

board_benchmarkStart — Set P1.1 high to start the benchmark.

Definition:

```
#define board_benchmarkStart()
```

Description:

Set P1.1 high to start the benchmark.

Returns:

none

board_benchmarkStop — Set P1.1 low to end the benchmark.

Definition:

```
#define board_benchmarkStop()
```

Description:

Set P1.1 low to end the benchmark.

Returns:

none

board_buttonPressed — Return the state of the button switch.

Definition:

```
#define board_buttonPressed()
```

Description:

Return the state of the button switch.

Returns:

True, if the button is pressed.

ENABLE_BENCHMARK —

Definition:

```
#define ENABLE_BENCHMARK
```

Description:

Allow calls to the benchmark functions

MCLK_FREQ —

Definition:

```
#define MCLK_FREQ
```

Description:

MCLK frequency to run at.

MCLK will be configured with the calibration values stored in INFO flash. Acceptable values are 1000000, 8000000, 12000000 and 16000000.

4.5.5 Board Function Documentation

board_gotoSleep — Enter LPM3 and return, when the WDT wakes the device up.

Prototype:

```
void
board_gotoSleep(void)
```

Description:

Enter LPM3 and return when the WDT wakes the device up.

Returns:

none

board_init — Initialize the Launchpad for operation.

Prototype:

```
void
board_init(void)
```

Description:

Initialize the Launchpad for operation.

Returns:

none

board_watchdogISR — WDT interrupt to wake up the CPU at fixed sample intervals.

Prototype:

```
__interrupt void
board_watchdogISR(void)
```

Description:

WDT interrupt to wake up the CPU at fixed sample intervals.

Returns:

none

4.6 Flash

4.6.1 Flash Functions

- void flash_init (void)
- bool flash_isCalibrated (void)
- void flash_writeCalibration (int16_t *i16Scale, int16_t *i16Offset)

4.6.2 Flash Variables

- const int16_t i16CalibrationOffset[3]
- const int16_t i16CalibrationScale[3]
- const uint16_t ui16CalibrationPassword

4.6.3 Flash Detailed Description

The following functions are provided in *HAL_flash.c* and provide methods for initializing flash and storing calibration data.

4.6.4 Flash Function Documentation

flash_init — Initialize flash controller.

Prototype:

```
void
flash_init(void)
```

Description:

Initialize flash controller.

Returns:

none

flash_isCalibrated — State of the flash calibration data.

Prototype:

```
bool
flash_isCalibrated(void)
```

Description:

State of the flash calibration data.

Returns:

True if the flash has been calibrated.

flash_writeCalibration — Write calibration data to flash.

Prototype:

```
void
flash_writeCalibration(int16_t *i16Scale,
int16_t *i16Offset)
```

Description:

Write calibration data to flash.

Parameters:

i16Scale – Pointer to the calibration scale data to copy to flash.

i16Offset – Pointer to the calibration offset data to copy to flash.

Returns:

none

4.6.5 Flash Variable Documentation

i16CalibrationOffset —

Definition:

```
const int16_t i16CalibrationOffset[3]
```

Description:

Calibration offset data for accelerometer readings.

Calibration offset data for the accelerometer readings. The data is scaled by by 16 to retain accuracy. These values will need to be stored in information memory and are dependant on the implementation of the HAL_flash layer.

i16CalibrationScale —

Definition:

```
const int16_t i16CalibrationScale[3]
```

Description:

Calibration scale data for accelerometer readings.

Calibration scale data for the accelerometer readings. The data is scaled by 16 to retain accuracy. These values will need to be stored in information memory and are dependant on the implementation of the HAL_flash layer.

ui16CalibrationPassword —

Definition:

```
const uint16_t ui16CalibrationPassword
```

Description:

Calibration password to check if data is already stored.

Use a unique key to check if info memory has already been calibrated for this device. The first time the program is loaded the calibration routine will need to be run to add data.

4.7 Accelerometer

4.7.1 Accelerometer Enumerations

- accelerometer_channel

4.7.2 Accelerometer Functions

- void accelerometer_init (void)
- int16_t accelerometer_read (accelerometer_channel channel)

4.7.3 Accelerometer Detailed Description

The following functions are provided in *HAL_accelerometer.c* and provide methods for reading data from the accelerometer.

4.7.4 Accelerometer Enumeration Documentation

accelerometer_init — Initialize the accelerometer for operation.

Prototype:

```
void
accelerometer_init(void)
```

Description:

Initialize the accelerometer for operation. The boosterpack shares pins with the button so the ADC10 will be initialized even time the accelerometer is read and then turned off.

Returns:

none

accelerometer_read — Read a single channel from the accelerometer.

Prototype:

```
int16_t
accelerometer_read(accelerometer_channel channel)
```

Description:

Read a channel from the accelerometer corresponding to an axis passed in as an argument.

Parameters:

channel – Which channel of the accelerometer to read

Returns:

raw ADC reading of the accelerometer channel requested

4.8 Buzzer Functions

4.8.1 Buzzer Defines

- BUZZER_FREQ

4.8.2 Buzzer Functions

- void buzzer_init (void)
- void buzzer_off (void)
- void buzzer_on (void)

4.8.3 Buzzer Detailed Description

The following functions are provided in *HAL_buzzer.c* and provide methods for using the buzzer.

4.8.4 Buzzer Define Documentation

BUZZER_FREQ —

Definition:

```
#define BUZZER_FREQ
```

Description:

Buzzer PWM frequency.

4.8.5 Buzzer Function Documentation

buzzer_init — Setup buzzer for TA0 PWM output.

Prototype:

```
void
buzzer_init(void)
```

Description:

Setup buzzer for TA0 PWM output.

Returns:

none

buzzer_off — Turn off the buzzer

Prototype:

```
void
buzzer_off(void)
```

Description:

Turn off the buzzer.

Returns:

none

buzzer_on — Turn on the buzzer using TA0 and CCR1 for PWM.

Prototype:

```
void
buzzer_on(void)
```

Description:

Turn on the buzzer using TA0 and CCR1 for PWM.

Returns:

none

4.9 LCD

4.9.1 LCD Defines

- SPI_FREQ

4.9.2 LCD Functions

- void LCD_clear (void)
- void LCD_init (void)
- void LCD_setPosition (uint8_t ui8Row, uint8_t ui8Column)
- void LCD_writeData (char *pcData, uint16_t ui16Count)
- void LCD_writeString (char *pcData)

4.9.3 LCD Detailed Description

The following functions are provided in *HAL_LCD.c* and provide methods for writing to the LCD.

4.9.4 LCD Define Documentation

SPI_FREQ — Definition:

```
#define SPI_FREQ
```

Description:

SPI clock frequency.

4.9.5 LCD Function Documentation

LCD_clear — Clear the display of characters.

Prototype:

```
void
LCD_clear(void)
```

Description:

Clear the display of characters.

Returns:

none

LCD_init — Initialize the LCD screen.

Prototype:

```
void
LCD_init(void)
```

Description:

Initialize the pins and peripherals for LCD operation. Perform basic setup of the LCD required for operation.

Returns:

none

LCD_setPosition — Set LCD to a row and column position.

Prototype:

```
void
LCD_setPosition(uint8_t ui8Row,
uint8_t ui8Column)
```

Description:

Set LCD to a row and column position.

Parameters:

ui8Row – Row to set the LCD position, 0-1

ui8Column – Column to set the LCD position, 0-39

Returns:

none

LCD_writeString — Send a string of characters to the LCD display.

prototype:

```
void
LCD_writeString(char *pcData)
```

Description:

Send a string of characters to the LCD display.

Parameters:

pcData – Pointer to the string to write to the LCD.

Returns:

none

4.10 RGB LED

4.10.1 RGB LED Enumerations

- RGB_LED_channel

4.10.2 RGB LED Functions

- void RGB_LED_init (void)
- void RGB_LED_off (RGB_LED_channel channel)
- void RGB_LED_on (RGB_LED_channel channel)
- void RGB_LED_set (RGB_LED_channel channel)

4.10.3 RGB LED Detailed Description

The following functions are provided in *HAL_RGB_LED.c* and provide methods for controlling the RGB LED.

4.10.4 RGB LED Enumeration Documentation

RGB_LED_channel — Description:

Enum type for acceptable RGB channels.

Enumerators:

RGB_LED_none – Off

RGB_LED_red – Red channel

RGB_LED_green – Green channel

RGB_LED_blue – Blue channel

RGB_LED_white – White channel

RGB_LED_all – All channels

4.10.5 RGB LED Function Documentation

RGB_LED_init — Initialize the pins for RGB LED operation

Prototype:

```
void
RGB_LED_init(void)
```

Description:

Initialize the pins for RGB LED operation.

Returns:

none

RGB_LED_off — Turn off an LED color channel.

Prototype:

```
void
RGB_LED_off(RGB_LED_channel channel)
```

Description:

Turn off an LED color channel.

Parameters:

channel – RGB channel to turn off.

Returns:

none

RGB_LED_on — Turn on an LED color channel.

Prototype:

```
void
   RGB_LED_on(RGB_LED_channel channel)
```

Description:

Turn on an LED color channel.

Parameters:

channel – RGB channel to turn on.

Returns:

none

RBG_LED_set — Set the LED state.

Prototype:

```
void
   RGB_LED_set(RGB_LED_channel channel)
```

Description:

Set the LED state.

Parameters:

channel – RGB channel to set the state to.

Returns:

none